Vitis Unified Software Platform Documentation

Embedded Software Development

UG1400 (v2019.2) March 18, 2020

XILINX

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
03/18/2020 Version 2019.2	
Using HSM Mode	Added PPK inputs to Stage 0 elements
BSP, DTS, and Application Generation in Vivado	Removed section
02/28/2020 Version 2019.2	
FreeRTOS Analysis using STM	Updated flow.
TCF Profiling	Updated flow.
System Performance Modeling Using a User-Defined Flow	Updated flow.
12/18/2019 Version 2019.2	
Creating a Platform Project	Corrected flow.
Migrating an SDK Project to the Vitis Software Platform	Added new use case.
Packaging Drivers for Vitis	Added new section.
11/11/2019 Version 2019.2	
Vitis 2019.2 Software Platform Release Notes	Updated content.
Installation Requirements	Updated content and removed licensing information.
Vitis Software Platform Installation	Updated content.
10/30/2019 Version 2019.2	
Initial release.	N/A

Table of Contents

Revision History 2

Section I: Getting Started 10

Chapter 1: Vitis 2019.2 Software Platform Release Notes 11
- What's in the Vitis Software Platform 11
- Supported Platforms 12
- Changed Behavior 12
- Known Issues 13

Chapter 2: Installation Requirements 14
- Install Required CentOS/RHEL Packages 14

Chapter 3: Vitis Software Platform Installation 16
- Install the Vitis Software Platform 16
- Installing Xilinx Runtime 17
- Installing Embedded Platforms 18
- Setting Up the Environment to Run the Vitis Software Platform 19

Chapter 4: Vitis Unified Software Platform Overview 20
- Key Concepts 20
- Document Scope and Audience 21

Chapter 5: Migrating to the Vitis Software Platform from Xilinx SDK 22
- Workspace Structure in the Vitis Software Platform and SDK 22
- Workflows in the Vitis Software Platform and SDK 24

Chapter 6: Quick Start 26
- Create a Platform 26
- Create a Hello World Application 28

Section II: Using the Vitis IDE 30

Chapter 7: Develop **31**
Target Platform 31
Applications 36
Using Custom Libraries in Application Projects 56

Chapter 8: Run, Debug, and Optimize **57**
Run Application Project 57
Debug Application Project 67
Cross-Triggering 86
Profile/Analyze 93
Optimize 102
Packaging the System/Utilities 126

Chapter 9: Vitis Utilities **129**
Xilinx Software Command-Line Tool 129
Program FPGA 129
Dump/Restore Data File 131
Launch Shell 131
Import 131
Export 132
Generating Device Tree 133

Chapter 10: Embedded Software Development Use Cases in the Vitis Software Platform **135**
Creating a Platform Project 135
Customizing a Pre-Built Platform 137
Adding Domains to a Platform Project 138
Creating Applications from Domains in a Platform 139
Managing Multiple Applications in a System Project 142
Creating and Building Applications for XSA Exported from the Vivado Design Suite 144
Switching FSBL Targeting Processor 146
Creating Multiple Domains for a Single Hardware 147
Changing a Referenced Domain 149
Changing and Updating the Hardware Specification 150
Debugging the Application on Hardware 151
Running and Debugging Applications under a System Project Together 152

Creating a Bootable Image 152
Flash Programming 154
Generating Device Tree 155
Debugging an Application using the User-Modified/Custom FSBL 155
Modifying the Domain Sources (Driver and Library Code) 156
Migrating an SDK Project to the Vitis Software Platform 160

Section III: Bootgen Tool 162

Chapter 11: Introduction 163
Installing Bootgen 164
Boot Time Security 164

Chapter 12: Boot Image Layout 166
Zynq-7000 SoC Boot and Configuration 166
Zynq UltraScale+ MPSoC Boot and Configuration 175

Chapter 13: Creating Boot Images 188
Boot Image Format (BIF) 188
BIF Syntax and Supported File Types 189
Attributes 191

Chapter 14: Using Bootgen Interfaces 197
Bootgen GUI Options 197
Using Bootgen on the Command Line 198
Commands and Descriptions 199

Chapter 15: Boot Time Security 202
Using Encryption 203
Using Authentication 211
Using HSM Mode 222

Chapter 16: FPGA Support 238
Encryption and Authentication 238
HSM Mode 239

Chapter 17: Use Cases and Examples 242
Zynq MPSoC Use Cases 242

Chapter 18: BIF Attribute Reference **252**
aarch32_mode 252
aeskeyfile 252
alignment 254
auth_params 255
authentication 257
big_endian 257
bh_keyfile 258
bh_key_iv 258
bhsignature 259
blocks 259
boot_device 260
bootimage 261
bootloader 262
bootvectors 263
checksum 263
core 264
destination_cpu 264
destination_device 265
early_handoff 266
encryption 266
exception_level 267
familykey 268
fsbl_config 268
headersignature 269
hivec 270
init 270
keysrc_encryption 271
load 272
offset 272
partition_owner 273
pid 273
pmufw_image 274
ppkfile 274
presign 275
pskfile 275
puf_file 276

reserve....277
split....277
spkfile....278
spksignature....279
spk_select....280
sskfile....281
startup....281
trustzone....282
udf_bh....282
udf_data....283
xip_mode....284

Chapter 19: Command Reference....285

arch....285
bif_help....285
dual_qspi_mode....286
efuseppkbits....286
encrypt....287
encryption_dump....287
fill....288
generate_hashes....288
generate_keys....289
image....291
log....291
nonbooting....292
o....292
p....293
padimageheader....293
process_bitstream....294
read....294
spksignature....295
split....295
verify....296
verify_kdf....296
w....297
zynqmpes1....297
Initialization Pairs and INT File Attribute....298

Chapter 20: Bootgen Utility **300**

Section IV: Xilinx Software Command-Line Tool **302**

Chapter 21: Xilinx Software Command-Line Tool **303**
System Requirements 304

Chapter 22: XSCT Commands **306**
Target Connection Management 307
Target Registers 310
Program Execution 312
Target Memory 324
Target Download FPGA/BINARY 330
Target Reset 333
Target Breakpoints/Watchpoints 334
JTAG UART 339
Miscellaneous 341
JTAG Access 349
Target File System 357
SVF Operations 364
Device Configuration System 369
Vitis Projects 370

Chapter 23: XSCT Use Cases **412**
Changing Compiler Options of an Application Project 413
Creating an Application Project Using an Application Template (Zynq UltraScale+ MPSoC FSBL) 413
Creating a Bootable Image and Program the Flash 413
Debugging a Program Already Running on the Target 414
Debugging Applications on Zynq UltraScale+ MPSoC 416
Modifying BSP Settings 419
Performing Standalone Application Debug 419
Generating SVF Files 422
Running an Application in Non-Interactive Mode 423
Running Tcl Scripts 423
Switching Between XSCT and Vitis Integrated Development Environment 424
Using JTAG UART 425
Working with Libraries 426

Editing FSBL/PMUFW Source File....427
Editing FSBL/PMUFW Settings....427

Chapter 24: Packaging Drivers for Vitis....428
XSCT Interface Examples....428
Microprocessor Software Specification (MSS)....442
Microprocessor Library Definition (MLD)....449
Microprocessor Driver Definition (MDD)....461
Microprocessor Application Definition (MAD)....473
Tcl Commands Listed Alphabetically....476

Section V: Embedded Design Tutorials....575

Section VI: Embedded Drivers and Libraries....576

Appendix A: Additional Resources and Legal Notices....577
Xilinx Resources....577
Documentation Navigator and Design Hubs....577
Please Read: Important Legal Notices....578

Section I

Getting Started

This section provides a brief overview of the Vitis™ Software Platform and describes the installation requirements and procedures to install and run the tool.

Chapter 1

Vitis 2019.2 Software Platform Release Notes

This section contains information regarding the features and updates of the Vitis software platform in this release.

What's in the Vitis Software Platform

Hardware-Accelerated Application Development Flow

For FPGA-based acceleration, the Vitis™ core development kit lets you build a software application using an API, such as the OpenCL™ API, to run hardware (HW) kernels on accelerator cards, like the Xilinx® Alveo™ Data Center acceleration cards. The Vitis core development kit also supports running the software application on an embedded processor platform running Linux, such as on Zynq UltraScale+ MPSoC devices. For the embedded processor platform, the Vitis core development kit execution model also uses the OpenCL API and the Linux-based Xilinx Runtime (XRT) to schedule the HW kernels and control data movement.

The Vitis core development kit tools support the Alveo U50, U200, U250, and U280 Data Center accelerator cards, as well as the zcu102_base, zcu104_base, zc702_base, and zc706_base embedded processor platforms. In addition to these off-the-shelf platforms, custom platforms are also supported.

The Vitis software platform allows you to migrate data center applications to embedded platforms. The Vitis core development kit includes the `v++` compiler for the hardware kernel on all platforms, the `g++` compiler for compiling the application to run on an x86 host, and an Arm® compiler for cross-compiling the application to run on the embedded processor of a Xilinx device.

Embedded Software Development Flow

The Vitis IDE is designed to be used for the development of embedded software applications targeted towards Xilinx embedded processors. The Vitis IDE works with hardware designs created with Vivado Design Suite. Embedded platforms for the hardware acceleration flow can also be used in embedded software development.

Supported Platforms

Embedded Platforms

IMPORTANT! *Artix®-7, Kintex®-7, Virtex®-7, along with bare metal and RTOS platforms, are not supported for acceleration. For non-accelerated based designs, see Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400).*

Embedded platforms available for use with the Vitis core development kit can be found at https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/edge-platforms.html. The supported platforms are listed below:

- `zcu102_base`
- `zcu102_base_dfx`
- `zcu104_base`
- `zc702_base`
- `zc706_base`

Refer to Installing Embedded Platforms for information on installing and setting up embedded processor platforms.

Changed Behavior

The Vitis unified software platform inherits the features of SDAccel, SDSoC, and SDK, and integrates them into one unified software platform. This section provides an overview of migrating projects SDAccel, SDSoC, and SDK to the Vitis unified software platform.

Migrating from SDAccel

The following table specifies differences between the SDAccel development environment and the Vitis software platform.

Table 1: **Migration Summary**

Area	SDAccel Behavior	Vitis Behavior
Configuring the Alveo accelerator card	`xbutil flash`	`xbmgmt flash` **IMPORTANT!** *The `xbutil` utility previously performed the `flash` function, however, that sub-command has moved to the `xbmgmt` utility, which is intended for use by system administrators and requires `sudo` privilege.*
Compilation and Linking	`xocc`	`v++`
Design Analysis	For the command line flow, you use the SDAccel IDE to view reports generated during the build process.	For the command line flow, you use the Vitis analyzer to view the reports generated during the build process.

Migrating from SDSoC

For details about migrating your project from the SDSoC Development Environment to the Vitis unified software platform, refer to the Migrating Embedded Processor Applications from SDSoC to Vitis section in *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

Migrating from SDK

For details about migrating your project from the SDK to the Vitis unified software platform, refer to Migrating to the Vitis Software Platform from Xilinx SDK.

Known Issues

Known Issues for the Vitis software platform are available in AR#72773.

Known Issues for Xilinx Runtime are available in AR#71752.

Chapter 2

Installation Requirements

The Vitis software platform consists of an integrated development environment (IDE) for interactive project development, and command-line tools for scripted or manual application development. The Vitis software platform also includes the Vivado® Design Suite for implementing the kernel on the target device, and for developing custom hardware platforms.

***Note*:** Windows OS support is limited to the Vitis embedded software development flow. The acceleration features in the Vitis software platform are not supported.

Some requirements listed here are only *required* for software acceleration features, but not for embedded software development features. Xilinx recommends installing all the required packages to have the best experience with the Vitis software platform.

To install and run on a computer, your system must meet the following minimum requirements.

Table 2: **Embedded Software Development Flow Minimum System Requirements**

Component	Requirement
Operating System	Linux, 64-bit: • Ubuntu 16.04.5 LTS, 16.04.6 LTS, 18.04.1 LTS, 18.04.2 LTS • CentOS 7.4, 7.5, 7.6 • RHEL 7.4, 7.5, 7.6 • SUSE Enterprise Linux 12.4
	Windows, 64-bit: • Windows 7 Professional (with SP1) • Windows 10 Professional (1809 update, 1903 Pre-release)
System Memory	32 GB (64 GB is recommended)
Internet Connection	Required for downloading drivers and utilities.
Hard disk space	100 GB

Install Required CentOS/RHEL Packages

Before installing the Vitis software platform on CentOS or RedHat, you must install the Extra Packages for Enterprise Linux (EPEL), and ensure you have the proper kernel-headers and kernel-devel packages installed. The initial setup commands depend on your operating system. For more information, see https://fedoraproject.org/wiki/EPEL.

IMPORTANT! *For Linux, GLX version 1.3 or higher is required. For more information, see AR#73017.*

Note: Ubuntu does not require additional packages.

1. Install EPEL.

   On RedHat:

   To enable an additional repository on your system and install the packages, open a terminal window, and enter the following command:

   ```
   $ sudo yum-config-manager --enable rhel-7-server-optional-rpms

   $ sudo yum install -y https://dl.fedoraproject.org/pub/epel/
   epel-release-latest-7.noarch.rpm
   ```

   On CentOS:

   Open a terminal window, and enter the following command:

   ```
   sudo yum install epel-release
   ```

2. To install kernel headers and kernel development packages, run the following commands:

   ```
   $ sudo yum install kernel-headers-`uname -r`
   $ sudo yum install kernel-devel-`uname -r`
   ```

   Note: Ensure that `uname` is surrounded by backticks (`) and *not* single quotes (').

3. Cold reboot your system.

Chapter 3

Vitis Software Platform Installation

1. Install the Vitis Software Platform
2. Follow the instructions in Installing Xilinx Runtime.
3. Install the embedded platform, as described in Installing Embedded Platforms
4. Follow the instructions in Setting Up the Environment to Run the Vitis Software Platform.

Note: To install and use XRT on CentOS/RedHat, ensure that you have already installed all required packages and recommended libraries, as described in Install Required CentOS/RHEL Packages.

Install 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 Xilinx Downloads Website.
2. Download the installer for your operating system.
3. Run the installer, which opens the Xilinx Unified 2019.2 Installer.
4. Click **Next**.
5. Enter your Xilinx user account credentials, and then select **Download and Install Now**.
6. Click **Next**.
7. Accept the terms and conditions by clicking each **I Agree** check box.
8. Click **Next**.
9. Select **Vitis**, and then click **Next**.
10. Optionally, customize your installation by selecting design tools and devices, and then click **Next**.

    Note: Both the Vitis tools and Vivado Design Suite are installed. You do not need to separately install Vivado tools. You can also install System Generator and Model Composer if needed.

11. Select the installation directory, optional shortcut and file association options, and then click **Next**.

12. Review the installation summary, which shows the options and locations you have selected.
13. To proceed with the installation of the Vitis software platform, click **Install**.

After a successful installation, a confirmation message is displayed.

Installing Xilinx Runtime

Xilinx Runtime (XRT) is implemented as a combination of user-space and kernel driver components. XRT supports Alveo PCIe-based cards, as well as Zynq UltraScale+ MPSoC-based embedded system platforms, and provides a software interface to Xilinx programmable logic devices.

You only need to install XRT once, regardless of how many platforms you may be installing.

IMPORTANT! *XRT installation uses standard Linux RPM and Linux DEB distribution files, and root access is required for all software and firmware installations.*

`<rpm-dir>` or `<deb-dir>` is the directory where you downloaded the packages to install.

To download and install the XRT package for your operating system, do the following.

CentOS/RedHat

1. To download the RPM file, click this link.
2. To install the package, enter the following command.

```
sudo yum install <rpm-dir>/<xrt_filename>.rpm
```

Ubuntu

1. To download the DEB file, click one of the following:
   - Ubuntu 16.04
   - Ubuntu 18.04
2. To install the package, enter the following command.

```
sudo apt install <deb-dir>/<xrt_filename_OS>.deb
```

Note: `<OS>` represents the Ubuntu operating system version (16.04 or 18.04) you are using.

IMPORTANT! *When installing XRT on Ubuntu, if the 2015 version of `pyopencl` is installed on your system, you must uninstall it. The XRT installation will install the 2019 version of `pyopencl` and will return an error if the 2015 version is installed. For more information, see AR#73055.*

Installing Embedded Platforms

To support the Vitis application acceleration development flow, embedded platforms must run Linux and XRT. For the Vitis embedded software development flow, you can use embedded platforms with Linux, standalone/bare metal, or RTOS domains. Embedded platform download files can be found at https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/edge-platforms.html.

The supported platforms are listed below:

- **zcu102_base:** Provides the platform definition, XRT drivers, and shared libraries for the ZCU102 Zynq UltraScale+ MPSoC based platform.
- **zcu102_base_dfx:** Provides the platform definition, XRT drivers, and shared libraries for the ZCU102 Zynq UltraScale+ MPSoC based platform with dynamic function exchange.
- **zcu104_base:** Provides the platform definition, XRT drivers, and shared libraries for the ZCU104 Zynq UltraScale+ MPSoC based platform.
- **zc702_base:** Provides the platform definition, XRT drivers, and shared libraries for the ZC702 Zynq®-7000 SoC based platform.
- **zc706_base:** Provides the platform definition, XRT drivers, and shared libraries for the ZC706 Zynq®-7000 SoC based platform.

To install the platform, download the appropriate Zip file and extract it into `/opt/xilinx/platforms`, or specify the `PLATFORM_REPO_PATHS` environment variable to specify the path to installed platforms.

IMPORTANT! *For use in the Vitis application acceleration development flow, embedded platforms require the installation of XRT as described in Installing Xilinx Runtime.*

When installing the files for an embedded platform, you must also install the cross-compilation Sysroot, that can be found on the downloads page. The Sysroot is required for compiling the host application using the GNU Arm cross-compiler in the Vitis tool flow. The `sysroots_scripts.tar.gz` download file contains an `sdk.sh` script for both Zynq®-7000 SoC and Zynq UltraScale+ MPSoC. Running the `sdk.sh` script sets up the Sysroot headers, libs, and include files for compilation of applications running on the embedded platforms.

You can add the Sysroot to a makefile for your command line project, or add it to your application project in the Vitis IDE. For example, in your makefile point `<SYSROOT>` to `/<install_path>/aarch64-xilinx-linux`, which is generated when running `sdk.sh`.

TIP: *The embedded processor platform downloads page also includes third-party source files that can be downloaded for your review or use in customizing platforms.*

Setting Up the Environment to Run the Vitis Software Platform

To configure the environment to run the Vitis software platform, run the following scripts, which set up the environment to run in a specific command shell.

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/Vitis/2019.2/settings64.sh
#setup XILINX_XRT
source /opt/xilinx/xrt/setup.sh
```

TIP: *`.csh` scripts are also provided.*

To specify the location of any platforms you have installed as directed in Installing Embedded Platforms, set the following environment variable:

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

Windows

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

- Launch from a desktop icon or Start menu command.
- From a Windows command shell, use `settings64.bat`:

  ```
  C:> <VITIS_INSTALL_DIR>\VITIS\2019.2\settings64.bat
  ```

  And launch: `vitis`.

Chapter 4

Vitis Unified Software Platform Overview

The Vitis™ integrated development environment (IDE) is part of the Vitis unified software platform. The Vitis IDE is designed to be used for the development of embedded software applications targeted towards Xilinx® embedded processors. The Vitis IDE works with hardware designs created with Vivado® Design Suite. The Vitis IDE is based on the Eclipse open source standard. The features for software developers include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Integrated environment for seamless debugging and profiling of embedded targets
- Source code version control
- System-level performance analysis
- Focused special tools to configure FPGA
- Bootable image creation
- Flash programming
- Script-based command-line tool

Key Concepts

The concepts listed below are key to understanding the Vitis embedded software development flow.

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

- **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).
- **Platform Project:** A platform project is customizable for adding domains and modifying domain settings. A platform project can be created by importing an XSA, or by importing an existing 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 Xilinx 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 the 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.

Document Scope and Audience

The purpose of this content is to familiarize software application developers and system software developers with the Vitis software platform and help them get started with using the tool. This content provides an overview of the Vitis software platform and the features listed here.

- Creating a platform project
- Creating an application project
- Vitis integrated development environment (IDE) extensions

Chapter 5

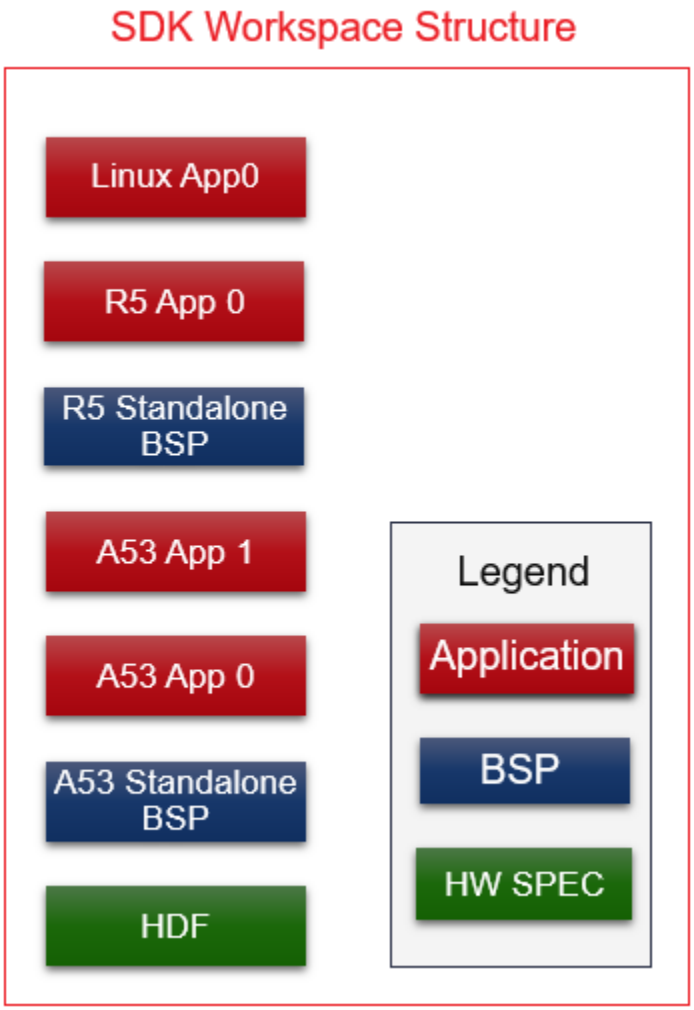
Migrating to the Vitis Software Platform from Xilinx SDK

If you are a Xilinx® Software Development Kit (SDK) user and are migrating to the Vitis™ software platform, the Embedded Software Development Use Cases in the Vitis Software Platform section lists a set of use cases that show you how to perform some of the regular tasks like working with platforms, applications, domains, debugging, flash programming, and so on. The steps required for Migrating an SDK Project to the Vitis Software Platform are also explained.

Workspace Structure in the Vitis Software Platform and SDK

In the Vitis software platform, two new concepts are introduced for better managing components in the workspace: *platform project* and *system project*. In the SDK workspace, the hardware specification, software board support package (BSP), and application all live at the top level.

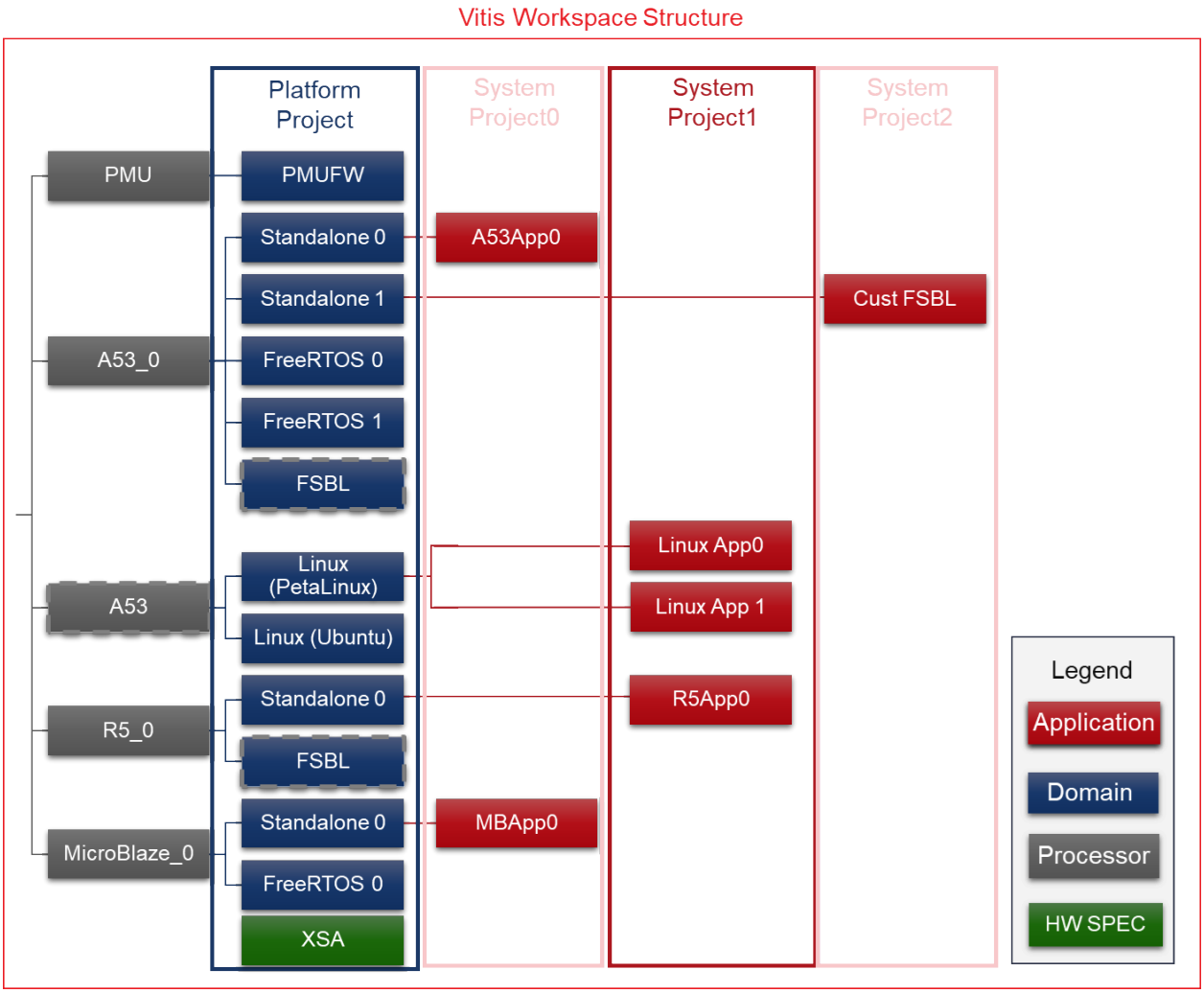
Figure 1: **SDK Structure**



The SDK BSP concept is upgraded to a *domain* in the Vitis software platform. A domain can refer to the settings and files of a standalone BSP, a Linux OS, a third party OS/BSP like FreeRTOS, or a component like the device tree generator.

In the Vitis software platform, a platform project groups hardware and domains together. Boot components like FSBL and PMUFW are automatically generated in platform projects. A system project groups together applications that run simultaneously on the device.

Figure 2: **Vitis Software Platform Structure**



Workflows in the Vitis Software Platform and SDK

The following table compares the key concepts and flows in the Vitis software platform covered in this document with their equivalents in SDK, if applicable.

Table 3: **Vitis Software Platform and SDK Comparison**

Vitis Software Platform	SDK
Creating a Platform Project	Import hardware specification and create a BSP
Adding Domains to a Platform Project	Create a BSP

Table 3: **Vitis Software Platform and SDK Comparison** *(cont'd)*

Vitis Software Platform	SDK
Creating Applications from Domains in a Platform	Create applications from BSP
Customizing a Pre-Built Platform	There is no corresponding concept in SDK.
Managing Multiple Applications in a System Project	There is no corresponding concept in SDK.
Creating and Building Applications for XSA Exported from the Vivado Design Suite	Create an application from XSA exported from Vivado Design Suite.
Creating Multiple Domains for a Single Hardware	Create multiple BSPs for a single hardware configuration
Changing a Referenced Domain	Change referenced BSP
Changing and Updating the Hardware Specification	The concept is the same, but the details of the workflow might have some minor changes.
Debugging the Application on Hardware	The concept is the same, but the details of the workflow might have some minor changes.
Running and Debugging Applications under a System Project Together	There is no corresponding concept in SDK.
Creating a Bootable Image	The concept is the same, but the details of the workflow might have some minor changes.
Flash Programming	The concept is the same, but the details of the workflow might have some minor changes.
Generating Device Tree	The concept is the same, but the details of the workflow might have some minor changes.
Debugging an Application using the User-Modified/Custom FSBL	The concept is the same, but the details of the workflow might have some minor changes.
Modifying the Domain Sources (Driver and Library Code)	The concept is the same, but the details of the workflow might have some minor changes.

Chapter 6

Quick Start

Get started with the Vitis™ software platform by creating a platform and a Hello World application.

Create a Platform

A platform project is the container for the hardware platform, runtime library, the settings for each processor, and the bootloader for the device. It can be as simple as a standalone board support package for a Cortex™-A53, or a combination of different kinds of runtime configurations for Cortex-A53, Cortex-R5F and MicroBlaze™ processors. This section explains how to create a hardware design and use that hardware design to create an application platform.

Create a Hardware Design (XSA File)

To create a hardware design, create a Vivado® project, customize the Zynq® UltraScale+™ MPSoC settings, connect to the PL peripherals, and generate the block design. For information on how to create a Vivado project, see *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209).

Note: A `bd.tcl` should be prepared in the `hw_src` directory. If you have cloned this repository, the Vivado block diagram design can be recreated in Vivado.

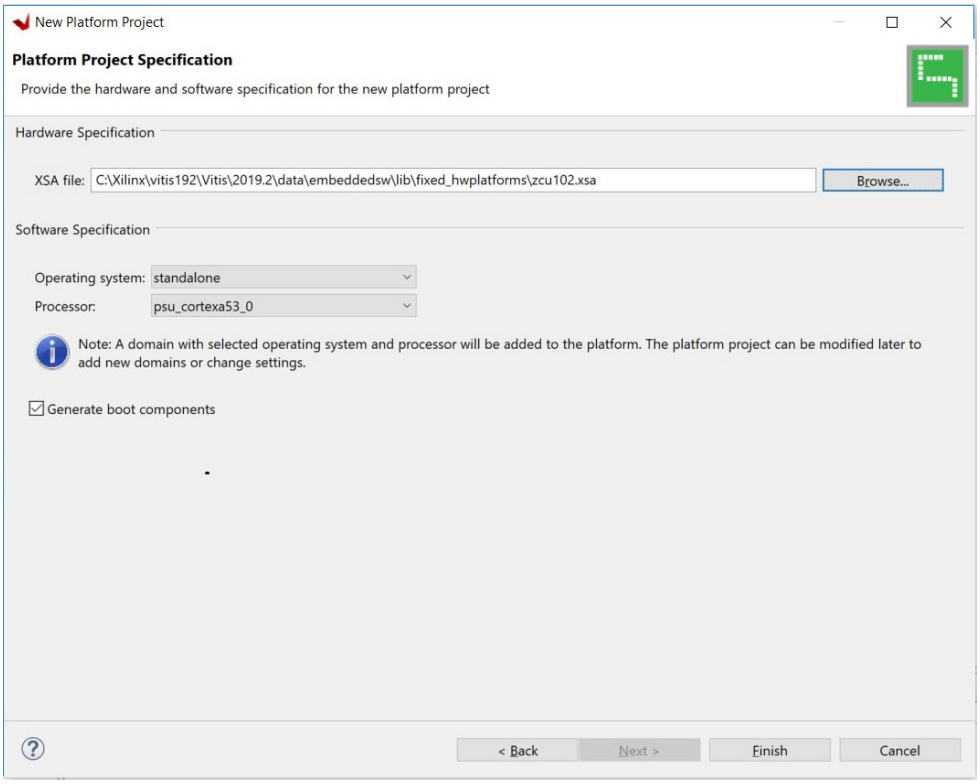
1. Create a Vivado block diagram design using the command `source hw_src/design_1.tcl`.
2. From IP integrator, run **Generate Block Design**. The block design is generated.
3. Select **File → Export → Export Hardware** to export the hardware platform from Vivado.
4. On the Export Hardware dialog, click **OK**.

Create a Platform Project

After exporting a hardware design, you can create a platform project. To create a platform project, do the following.

1. Launch the Vitis™ software platform.

2. Select **File → New → Platform Project**. The New Platform Project wizard opens.
3. In the Project name field, enter a name for your platform project and click **Next**.
4. Select **Create from hardware specification (XSA)** and click **Next**.
5. In the XSA file field, browse and select the XSA file that you exported from the Vivado Design Suite.



6. Use the dropdown menus to select **standalone** as the operating system and **psu_cortexa53_0** as the processor. The Generate boot components checkbox is selected. You can deselect this if you do not need boot components (FSBL in case of Zynq and PMU firmware in case of Zynq UltraScale+ MPSoC).
7. Click **Finish**. The platform is generated with multiple domains, one that you specified and the rest are the ones required for the boot components. It can later be modified to add new domains.
8. Double-click **platform.spr** in the Explorer view. This opens the platform tab for viewing and modification. You can modify settings for FSBL, standalone domains, and PMU firmware.
9. To generate the platform, right-click the platform project and select **Build Project**. Alternatively, you can also click .

Create a Hello World Application

After installing the Vitis™ software platform, the next step is to create a software application project. Software application projects are the final application containers. The project directory that is created contains (or links to) your C/C++ source files, executable output file, and associated utility files, such as the Makefiles used to build the project.

***Note*:** The Vitis software platform automatically creates a system project for you. A system project is a top-level container project that holds all of the applications that can run in a system at the same time. This is useful if you have many processors in your system, especially if they communicate with one another, because you can debug, launch, and profile applications as a set instead of as individual items.

Build a Sample Application

This section describes how to create a sample Hello World application using an existing template.

1. Launch the Vitis software platform.
2. Select a workspace directory for your first project.
3. Click **Launch**. The welcome page appears.
4. Close the welcome page. The development perspective opens.
5. Select **File → New → Application Project**.
6. Enter a name in the Project name field and click **Next**. The Select platform tab opens. You should choose a platform for your project. You can either use a pre-supplied platform (from Xilinx or another vendor), a previously created custom platform, or you can create one automatically from an exported Vivado® hardware project.
7. On the **Select platform** tab, click the platform you just created and click **Next**. To use your own hardware platform, click the icon and add your platform to the list.
8. Select the system configuration for your project and click **Next**. The Templates window opens.
9. Select **Hello World** and click **Next**. Your workspace opens with the Explorer pane showing the `hello_world_system` system project and the `zcu102` platform project.
10. Right-click the system project and select **Build Project**. You have now built your application and the Console tab shows the details of the file and application size.

Debug and Run the Application

Now that you have generated the executable binary, you can test it on a board. To run the application on the board, perform the following preliminary steps:

- Connect a JTAG cable to the computer.

- Set the Boot Mode switch of the board to JTAG mode.
- Connect a USB UART cable and setup your UART console.
- Power up the board.

1. Expand the system project and choose the application project you want to debug. Right-click the application and select **Debug As → Launch on Hardware (Single Application Debug)**.
2. On the Confirm Perspective Switch dialog, click **Yes**. The Vitis IDE switches to the Debug perspective and the debugger stops at the entry to your `main()` function.
3. Using the commands in the toolbar, step through the application. After you step through the `print()` function, `Hello World` appears in the UART console.

Section II

Using the Vitis IDE

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

Chapter 7

Develop

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

Target Platform

In the Vitis software platform, hardware is referred to as the target platform. The target platform is a combination of hardware components (XSA) and software components (domains, boot components like FS and so on).

A platform project is a customizable target platform in a workspace. In a platform project, you can add, modify, or remove domains. You can also as enable, disable, and modify boot components. It can be as simple as a standalone BSP for Cortex™-A53, or it can be a combination of different kinds of runtime configurations for Cortex-A53, Cortex-R5, and MicroBlaze processors. This section explains how to create a hardware design and use that hardware design to create an application platform.

Creating a Platform Project

To create a new platform project for application development in the Vitis integrated design environment (IDE), see Create a Platform Project.

Adding a Domain to an Existing Platform

Adding a Standalone Domain

The application needs to be linked to a domain. Before creating the application project, make sure that the target domain software environment is available. If not, add the required domain to your platform.

1. Double-click the `platform.spr` file in the Vitis Explorer view.

   Note: If you have not yet created a platform file, refer to Creating a Platform Project.

2. Click the button.

3. Select the OS as **Standalone** and a Processor of your choice.
4. Click **OK**.

Adding a Linux Domain

1. Double-click the `platform.spr` file in the Vitis Explorer view.

   Note: If you have not yet created a platform file, refer to Creating a Platform Project.

2. Click the button.
3. Select the Operating System as **Linux** and a Processor of your choice.
4. Click **Finish**. This creates a platform project and the Platform Overview page opens.
5. Click **Click here** to configure the Linux domain.
6. Use pre-built software components to fill in the required fields. Specifically, use a BIF file to generate the boot.bin, and then use the directory that contains the boot components to which the BIF file refers. The Linux image directory is optional, and contains image.ub.

   Note: BIF and boot components can be generated by PetaLinux.

7. Click **OK**. The Linux domain is configured.
8. Click the icon to generate or build the platform. The Explorer view shows the generated image files in the platform project.

Adding a FreeRTOS Domain

1. Double-click the `platform.spr` file in the Vitis Explorer view.

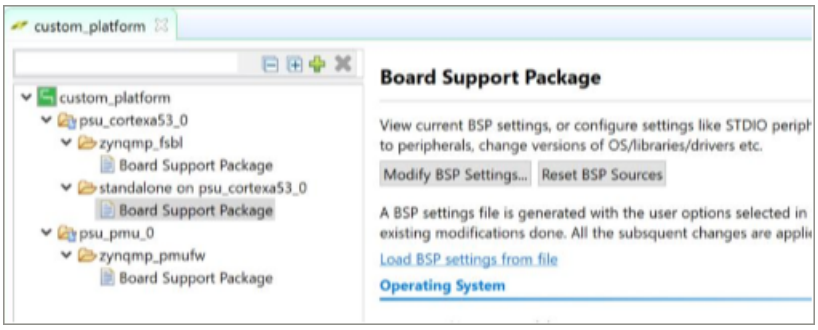
   Note: If you have not yet created a platform file, refer to Creating a Platform Project.

2. Click the button.
3. Select the OS as **FreeRTOS** and a Processor of your choice.
4. Click **OK**.

Configuring a Standalone BSP Domain

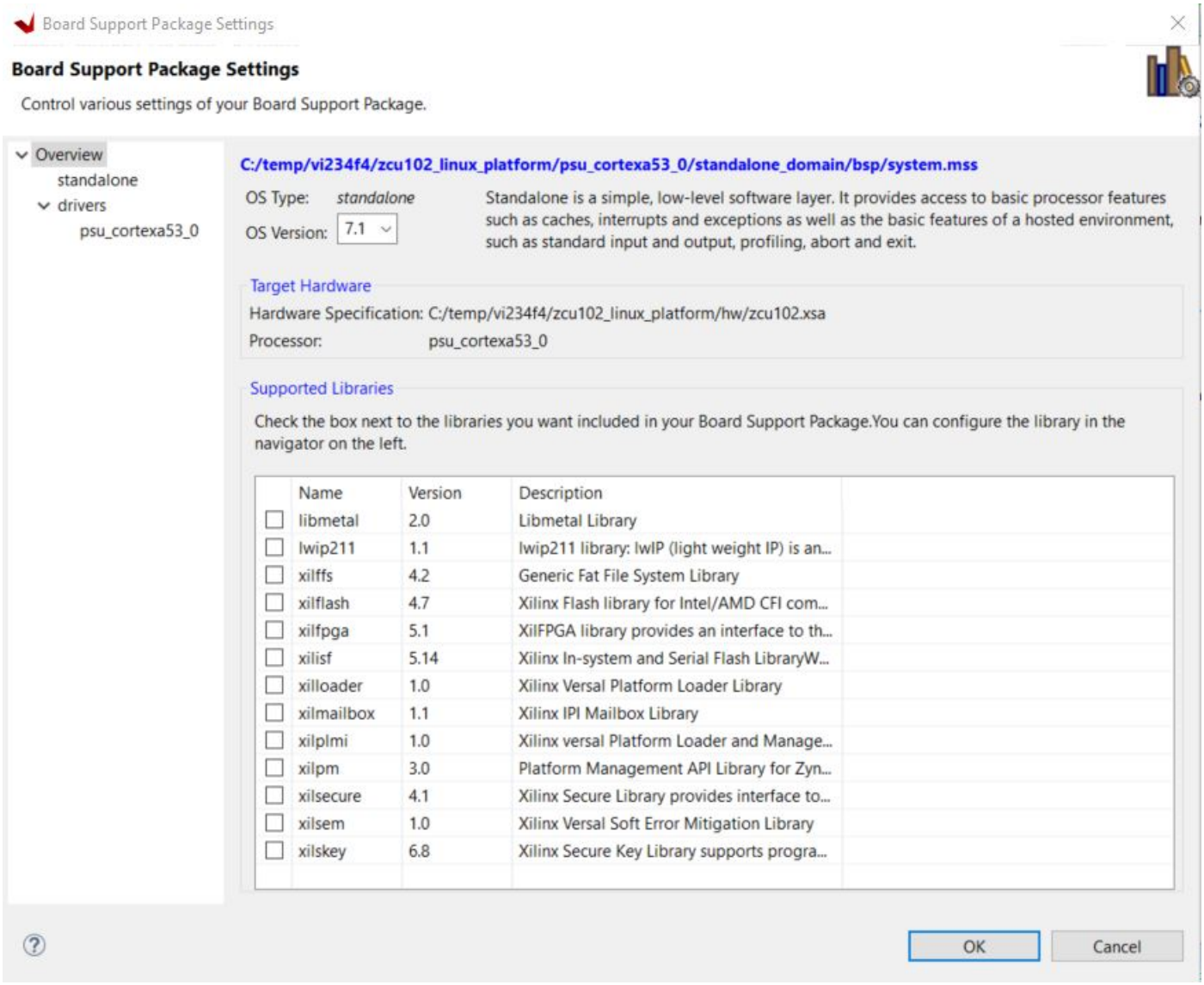
There are several kinds of domains. The standalone BSP is the most commonly used domain. To configure a standalone BSP, perform the following steps.

1. From the Explorer, double-click `platform.spr` file and select the appropriate domain/board support package. The overview page opens.
2. In the overview page, click **Modify BSP Settings**.



Using the Board Support Package Overview Page

Using the Overview page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.



***Note*:** You cannot change the OS choice in this page, because the OS type was determined during the software platform creation.

Using the Board Support Package Settings Page

The Board Support Package settings page enables you to configure parameters of the OS and its constituent libraries.

***Note*:** Options for only the libraries that you enabled in the Overview page will be visible. Options for the OS/standalone supported peripherals, that are present in the hardware platform, are also shown on the page.

Using the Board Support Package Drivers Page

The Drivers page 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. If you want to remove a driver for a peripheral, assign the driver to **none**.

Some device drivers export parameters that you can configure. If a device in the driver list has parameters, it is listed in navigation pane on the left and you can access them by clicking on the device name.

Using the Board Support Package Settings Driver Configuration Page

The Driver Configuration page lists all of the configurable driver parameters for the device selected under the **drivers** entry on the left. To change a parameter, click on the corresponding **Value** field and type the new setting.

When you finish with all the settings you want to make, click **OK**. The Vitis software platform re-generates the domain/BSP sources.

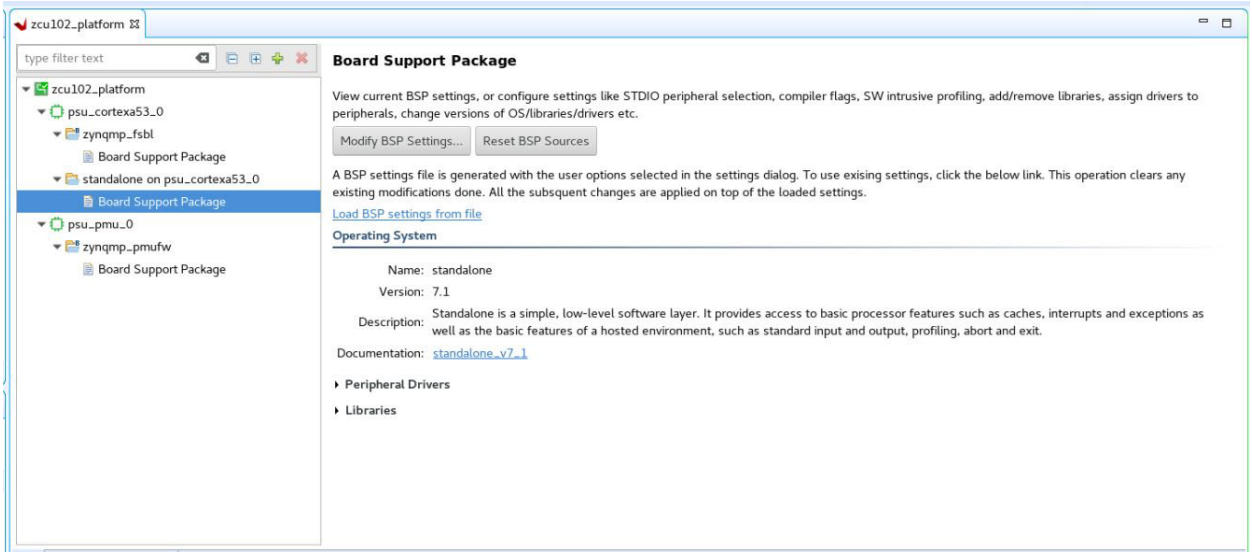
If the Build All option is selected in the **Project** menu, Vitis software platform automatically rebuilds your target platform with your new settings applied.

***Note*:** The exact list of software components appearing in the Board Support Package Settings dialog box depends on the components available in your Vitis software platform install, as well as the list of components found in any software repositories that are set up in your workspace. For more information about how repositories work, see Vitis Utilities.

Generating a Platform

To generate a platform, follow these steps.

1. Double click **platform.spr** in the Explorer view. This opens the platform tab for viewing and modification. You can modify settings for FSBL, standalone domains, and PMUFW.



2. In the platform view, click to generate the platform. The Generation Successful message pops up.

Modifying Source Code for FSBL and PMU Firmware

1. To modify the source code for FSBL or PMU firmware, go to Explorer view and expand the corresponding platform.
2. Expand the boot domain folder and modify the source files inside.
3. Save your changes and click the icon. This will build the boot components with the new changes.

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

Re-targeting a Platform for a New Hardware Specification

The Vitis software platform workspace supports multiple hardware projects and multiple domains per hardware project. A single software project can be portable across these hardware platforms and board support platforms. At any given time, the software domain project and the software applications associated with that domain project are targeted at (or referenced to) a single hardware project. Therefore, when running or debugging a software project on another hardware project, you must re-target your software domain project to another hardware project.

To re-target your platform project to another hardware specification:

1. Right-click the platform project that your software application currently references.
2. Select **Update Hardware Specification**.

3. Browse and locate the new hardware specification file and click **OK**.
4. In the Explorer view, right click the platform and select **Build Project**.

   This new platform is re-targeted to the hardware specification that you selected.

Resetting BSP Sources for a Domain

This feature allows you to reset the source files of a domain's BSP. To reset:

1. Click the `platform.spr` file in the Explorer view and select the appropriate domain.
2. Click **Reset BSP Sources**.
3. Click **Yes**. 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.

Applications

Creating a Standalone Application Project

You can create a C or C++ standalone application project by using the New Application Project wizard.

To create a project:

1. Click **File → New → Application Project**. The New Application Project wizard appears.

   Note: This is equivalent to clicking on **File → New → Project** to open the New Project wizard, selecting **Xilinx → Application Project**, and clicking **Next**.
2. Type a project name into the **Project Name** field.
3. Select the location for the project. You can use the default location as displayed in the Location field by leaving the **Use default location** check box selected. Otherwise, click the check box and type or browse to the directory location.
4. Select **Create a new platform from hardware (XSA)**. The Vitis IDE lists the all the available pre-defined hardware designs.
5. Select any one hardware design from the list and click **Next**.
6. From the CPU drop-down list, select the processor for which you want to build the application. This is an important step when there are multiple processors in your design. In this case you can either select **psu_cortexa53_0** or **psu_cortexr5_0**.
7. Select your preferred language: **C** or **C++**.
8. Select an OS for the targeted application.

9. Click **Next** to advance to the Templates screen.
10. The Vitis software platform provides useful sample applications listed in the Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.
11. Select the desired template. If you want to create a blank project, select **Empty Application**. You can then add C files to the project, after the project is created.
12. Click **Finish** to create your application project and board support package (if it does not exist).

    Note: Xilinx recommends that you use the managed make flow rather than standard make C/C++ unless you are comfortable working with Makefiles.

Creating a Linux Application Project

You can create a C or C++ Linux application project by using the New Application Project wizard.

To create a project:

1. Click **File → New → Application Project**. The New Application Project dialog box appears.
2. Type a project name into the **Project Name** field.
3. Select the location for the project. You can use the default location as displayed in the Location field by leaving the **Use default location** check box selected. Otherwise, click the check box and type or browse to the directory location.
4. Select **Next**.
5. On the Select platform tab, select the Platform that has a Linux domain and click **Next**.

   Note: If your platform does not yet have a Linux domain, refer to Adding a Linux Domain.
6. On the Domain window, select the domain from the Domain drop-down.
7. Select your preferred language: **C** or **C++**.
8. Optionally, select **Linux System Root** to specify the Linux sysroot path and select **Linux Toolchain** to specify the Linux toolchain path.
9. Click **Next** to move to the **Templates** screen.
10. The Vitis software platform provides useful sample applications listed in the Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.
11. Select the desired template. If you want to create a blank project, select **Empty Application**. You can then add C files to the project, after the project is created.
12. Click **Finish** to create your Linux application project.
13. Click the icon to generate or build the application project.

Creating a User Application Template

The Vitis software platform and XSCT support creation of user-defined application templates using the repository functionality. To create a standalone or Linux application template:

1. A great way to start creating an user-defined application template is to look at an existing template for the directory structure and files that needs to be defined along with the source files.
   a. Sample standalone OS application template files are available at `<Vitis software platform installation directory>\data\embeddedsw\lib\sw_apps\lwip_echo_server`.
   b. Sample Linux OS application template files are available at `<Vitis software platform installation directory>\data\embeddedsw\lib\sw_apps_linux\linux_hello_world`.
   c. Observe the folder name. Also note that the file names are the same as the application template names, excluding the file extensions.
   d. Decide on your application template name and OS.
   e. Create an application Tcl file. The Tcl file name should be same as the application template name.
   f. Add the following functions to the Tcl file:
      i. `swapp_get_name`: This function returns the application template name. The return value should be same as the application template name.

```
proc swapp_get_name {} {
    return "lwIP Echo Server";
}
```

      ii. `swapp_get_description`: This function returns the description of the application template in the Vitis IDE. You can customize the description according to the application details.

```
proc swapp_get_description {} {
return "The lwIP Echo Server application provides a simple
demonstration of
how to use the light-weight IP stack (lwIP). This application sets
up the board
to use IP address 192.168.1.10, with MAC address
00:0a:35:00:01:02. The server listens
for input at port 7 and simply echoes back whatever data is sent
to that port."
}
```

      iii. `swapp_is_supported_sw`: This function checks for the required software libraries for the application project. For example, the `lwip_echo_server` application template requires the `lwip` library in the domain.

```
proc swapp_is_supported_sw {} {
    # make sure we are using standalone OS
    check_standalone_os;

    # check for stdout being set
    check_stdout_sw;

    # make sure lwip141 is available
    set librarylist [hsi::get_libs -filter "NAME==lwip141"];

    if { [llength $librarylist] == 0 } {
        error "This application requires lwIP library in the Board Support Package.";
    } elseif { [llength $librarylist] > 1} {
        error "Multiple lwIP libraries present in the Board Support Package."
    }

    return 1;
}
```

iv. `swapp_is_supported_hw`: This function checks if the application is supported for a particular design or not. For example, `lwip` is not supported for MicroBlaze™ processors.

```
proc swapp_is_supported_hw {} {
    # Check if Ethernet IP in the system
    check_emac_hw;

    # check for stdout being set
    check_stdout_hw;

    # do processor specific checks
    set proc  [hsi::get_sw_processor];
     set proc_type [common::get_property IP_NAME [hsi::get_cells -hier $proc]]
    if { $proc_type == "microblaze"} {
        # make sure there is a timer (if this is a MB)
        set timerlist [hsi::get_cells -hier -filter { ip_name == "xps_timer" }];
        if { [llength $timerlist] <= 0 } {
            set timerlist [hsi::get_cells -hier -filter { ip_name == "axi_timer" }];
            if { [llength $timerlist] <= 0 } {
                error "There seems to be no timer peripheral in the hardware. lwIP requires an xps_timer for TCP
                operations.";
            }
        }
    }

    # require about 1M of memory
    require_memory "1000000";

    return 1;
}
```

v. `swapp_get_linker_constraints`: This function is used to generate the linker script. If this function returns `lscript no`, the linkerscript is copied from the application template. For example, the FSBL application does not generate a linker script. There exists a default linker script in the `src` folder that is used to create an application.

```
proc swapp_get_linker_constraints {} {
    # don't generate a linker script. fsbl has its own linker
script
    return "lscript no";
}
```

vi. `swapp_get_supported_processors`: This function checks the supported processors for the application template. For example, the `linux_hello_world` project supports the `ps7_cortexa9`, `psu_cortexa53`, and `microblaze` processors.

```
proc swapp_get_supported_processors {} {
    return "ps7_cortexa9 psu_cortexa53 microblaze";
}
```

vii. `proc swap_get_supported_os`: This function checks the OS supported by the application template.

```
proc swapp_get_supported_os {} {
    return "linux";
}
```

2. Create an application MSS file to provide specific driver libraries to the application template. The MSS file name should be similar to the application template name.
3. Provide the `OS` and `LIBRARY` parameter details.

```
 PARAMETER VERSION = 2.2.0


BEGIN OS
 PARAMETER OS_NAME = standalone
 PARAMETER STDIN =  *
 PARAMETER STDOUT = *
END

BEGIN LIBRARY
 PARAMETER LIBRARY_NAME = lwip141
 PARAMETER API_MODE = RAW_API
 PARAMETER dhcp_does_arp_check = true
 PARAMETER lwip_dhcp = true
END
```

4. Copy the newly created TCL and MSS files to the `data` folder.
5. Create your source source files and save them in the `src` folder. Copy the `lscript.ld` file to the `src` folder, if required.
6. Move the `data` and `src` folders to a newly created folder. For example:
   - For standalone application templates, create a folder `sw_apps` and move the `data` and `src` folders to the newly created folder. The Vitis software platform considers the applications created in the `sw_apps` folder as standalone applications.
   - For Linux application templates, create a folder `sw_apps_linux` and move the `data` and `src` folders to the newly created folder. The Vitis software platform considers the applications created in the `sw_apps_linux` folder as Linux applications.

Accessing User Application Templates

You can access the user template applications in the Vitis IDE or using the XSCT. To access the user application templates:

1. Using the Vitis IDE:
   a. Launch the Vitis IDE.
   b. Select **Xilinx → Repositories → Add**.

c. Select the repository folder, from the dialog box that appears.

   Note: For standalone applications, the parent folder that contains the applications should be `sw_apps`. Example: `C:\temp\repo\sw_apps\custom_app_name`. For Linux applications, the parent folder that contains the applications should be `sw_apps_linux`. Example: `C:\temp\repo\sw_apps_linux\custom_app_name`.

d. Select **File → New → Application Project**. The **New Project** wizard page appears.

e. Specify a project name.

f. Select your platform or create a new platform from the hardware (xsa).

g. Choose the desired OS and processor type to match with the user application template.

h. Select the user application template, from the **Available Templates** list and click **Finish** to create an application based on the selected user application template.

2. Using XSCT:

   a. Execute the following commands at the XSCT prompt:

```
setws {c:\temp\workspace}
repo -set {C:\temp\repo}
app create -name custom_app -hw zc702 -os standalone -proc -template
{custom_app_name}
app build -name custom_app
```

Working with Projects

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 Configurations

Software developers typically build different versions of executables, with different settings used to build those executables. For example, an application that is built for debugging uses a certain set of options (such as compiler flags and macro definitions), while the same application is built with a different set of options for eventual release to customers. The Vitis software platform makes it easier to maintain these different profiles using the concept of build configurations.

A build configuration is a named collection of build tools options. The set of options in a given build configuration causes the build tools to generate a final binary with specific characteristics. When the wizard completes its process, it generates launch configurations with names that follow the pattern `<projectname>`, where `<projectname>` represents the name of the project.

Each build configuration can customize:

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

By default, the Vitis software platform provides three build configurations, as listed in the following table:

Table 4: **Build Configurations**

Configuration Type	Compiler Flags
Debug	`-O0 -g`
Release	`-O2`
Profile	`-O2 -g -pg`

Changing the Build Configuration

Use the **Tool Settings** properties tab to customize the tools and tool options used in your build configuration. Follow these steps to change build settings:

1. Select the project for which you want to modify the build settings in the **Project Explorer** view.
2. Select **Project** → **Properties**. The **Properties for <project>** window appears. The left panel of the window has a properties list. This list shows the build properties that apply to the current project.
3. Expand the **C/C++ Build** property.
4. Select **Settings**.
5. Use the **Configuration** list to select the configuration that needs to be modified.
6. Click the **Tool Settings** tab.

7. Select the tool and change the settings as per your requirement.
8. Click **Apply** to save the settings.
9. When you finish updating the tools and their settings, click **OK** to save and close the **Properties for <project>** window.

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:

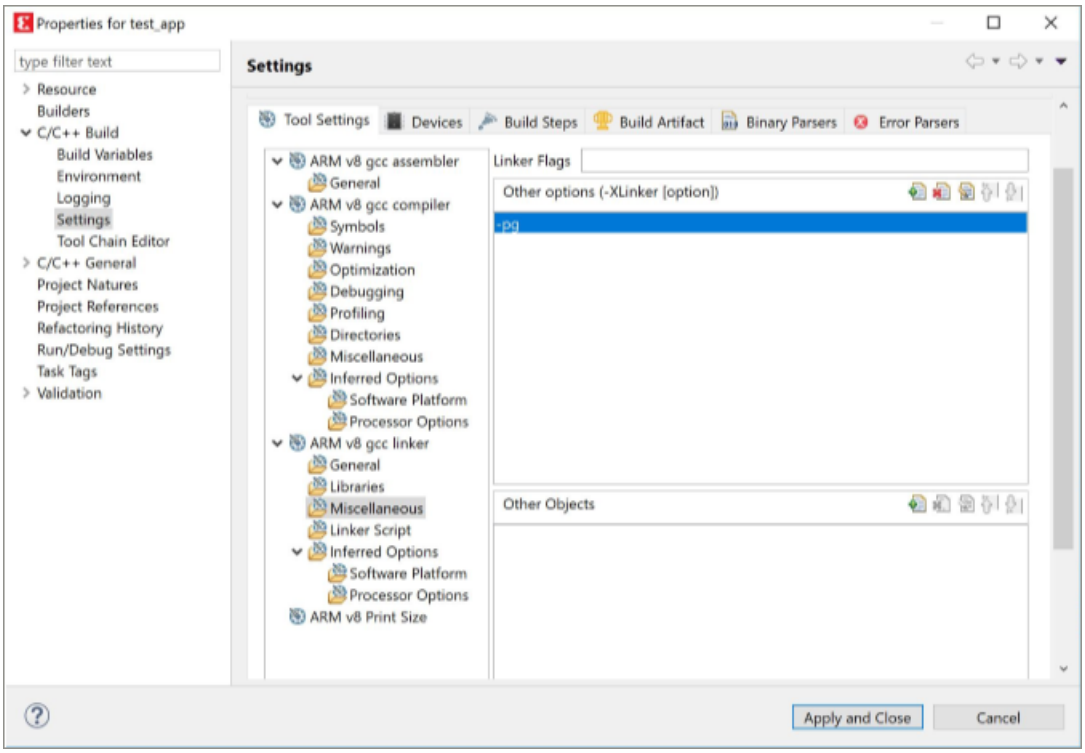
1. Right-click your Application project and select **C/C++ Build Settings**. Alternatively, select **Properties** and navigate to **C/C++ Build > Settings**.
2. Expand the target linker section and select the libraries to which you want to add the custom library path and library name.

Specifying the Linker Options

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

To set properties for your project:

1. Right-click your managed make project and select **C/C++ Build Settings**. Alternatively, select **Properties** and navigate to **C/C++ Build → Settings**.
2. Under the Tool Settings tab, expand the target linker section.
3. Select **Miscellaneous**.
4. Specify linker options in the Linker Flags field by clicking the **Add** button. Options can be deleted using the **Delete** button, or modified using the **Edit** button.

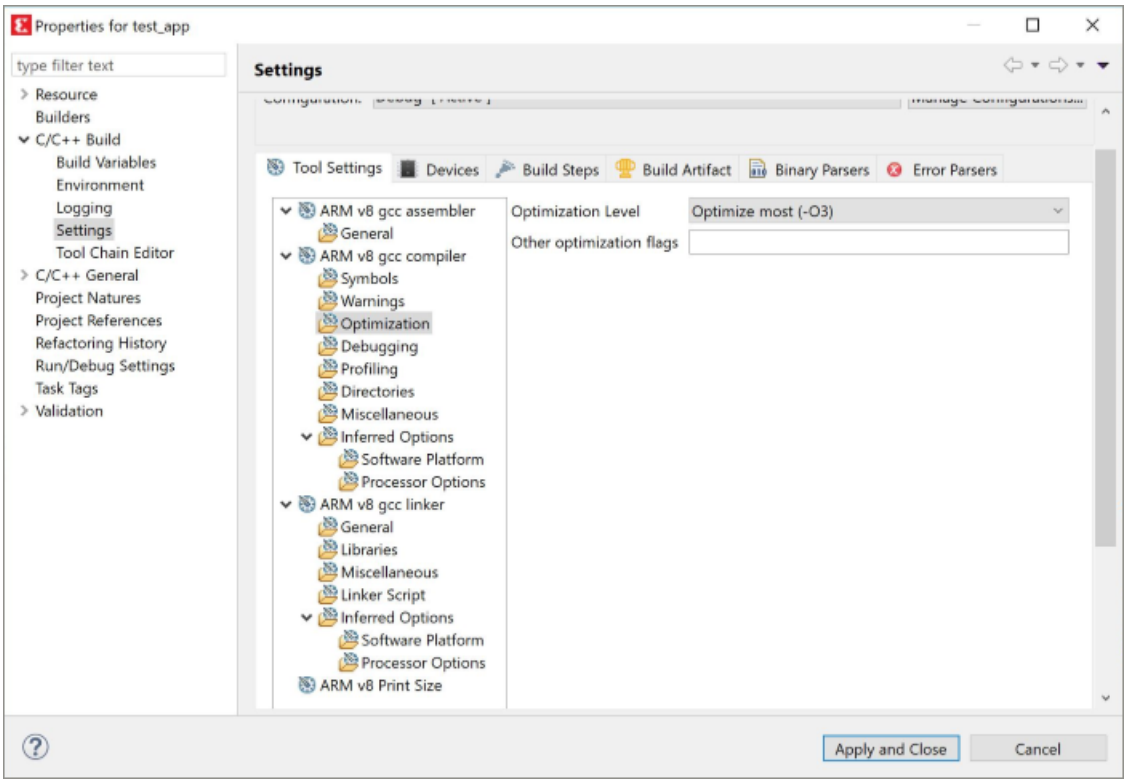


Specifying Debug and Optimization Compiler Flags

Based on the build configuration selected, the Vitis software platform assigns a default optimization level and debug flags for compilation. You can change the default value for your project.

To set properties for your project:

1. Right-click your managed make project.
2. Select **Properties**. Alternatively, to set properties for a specific source file in your project, right-click a source file within your standard make project and select **Properties** to open the properties dialog box.
3. Expand the list under **C/C++ Build**.
4. Click on **Settings**.
5. Under the Tool Settings tab, expand the **gcc compiler** list.
6. Select **Optimization** to change the optimization level and **Debugging** to change the debugging level.

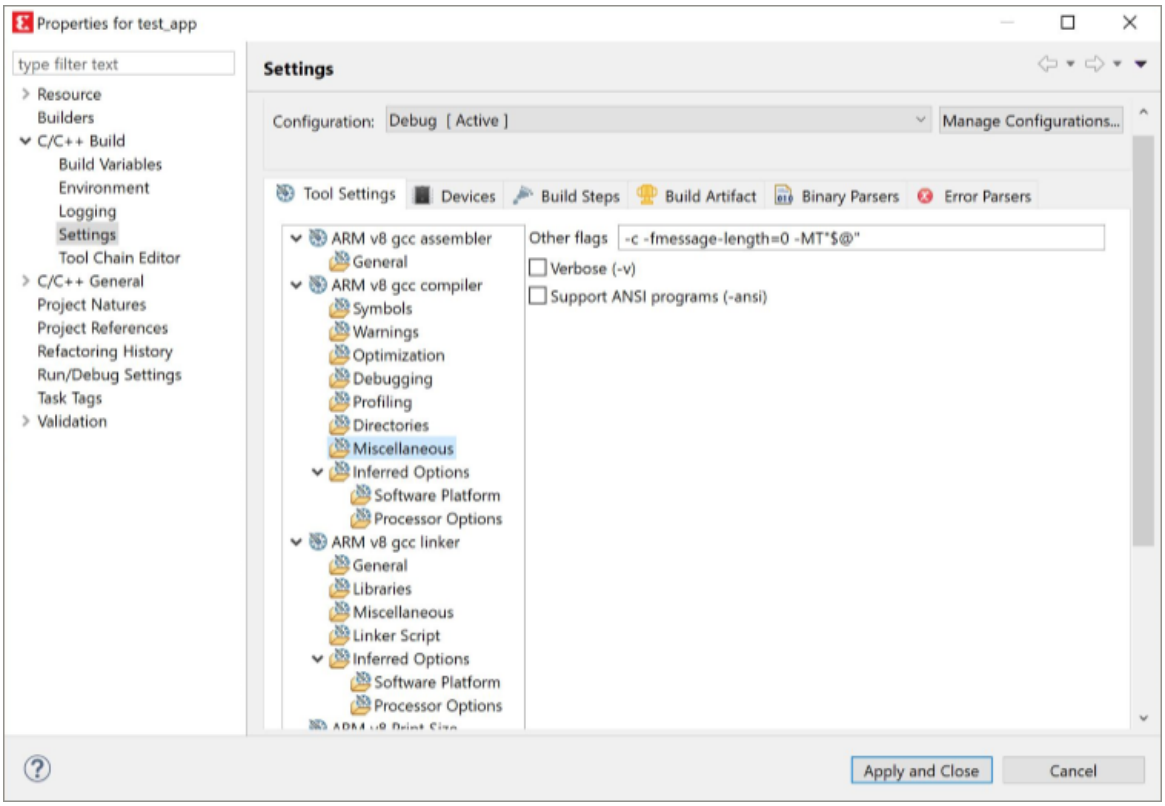


Specifying Miscellaneous Compiler Flags

You can specify any other compiler flags not covered in the **Tool Settings** for program compilation.

To set properties for your project:

1. Right-click your managed make project and select **Properties**. Alternatively, to set properties for a specific source file in your project, right-click a source file within your standard make project and select **Properties**.
2. Click **C/C++ Build** to expand the list and click on **Settings**.
3. In the Tool Settings tab, expand the **gcc compiler** list.
4. Select **Miscellaneous**.
5. In the Other flags field, specify compiler flags.



Restoring Build Configuration

Follow these steps to restore the build properties to have a factory-default configuration, or to revert to a last-known working build configuration:

1. Select the project for which you want to modify the build settings in the Project Explorer view.
2. Select **Project → Properties**. The **Properties for <project>** window appears. The left panel of the window has a properties list. This list shows the build properties that apply to the current project.
3. Click the **Restore Defaults** button.
4. When you finish restoring the build settings, click **OK** to save and close the **Properties for <project>** window.

Makefiles

Compilation of source files into object files is controlled using Makefiles. With the Vitis software platform, there are two possible options for Makefiles:

- **Managed Make:** For managed make projects, the Vitis software platform automatically creates Makefiles. Makefiles created by the Vitis software platform typically compile the sources into object files, and finally link the different object files into an executable. In most cases, managed make eliminates the job of writing Makefiles. This is the suggested option.

- **Standard Make:** If you want ultimate control over the compilation process, use standard make projects. In this case, you must manually write a Makefile with steps to compile and link an application. Using the standard make flow hides a number of dependencies from the Vitis software platform. You must follow manual steps for other tasks such as debugging or running the application from within the Vitis software platform. Therefore, the standard make flow is not recommended for general use.

Debugging Projects

The debugger in the Vitis software platform enables you to see what is happening to a program while it executes. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system.

The debugger supports debugging through Xilinx System Debugger and GNU Debugger (GDB). Xilinx System Debugger is derived from open-source tools and is integrated with the Vitis software platform.

Hardware Debug Target

The Vitis software platform supports debugging of a program on a processor running on an FPGA or a Zynq-7000 SoC device. All processor architectures (MicroBlaze and Arm® Cortex A9 processors) are supported. The Vitis software platform communicates to the processor on the FPGA or Zynq-7000 SoC device.

Before you debug the processor on the FPGA, configure the FPGA with the appropriate system bitstream.

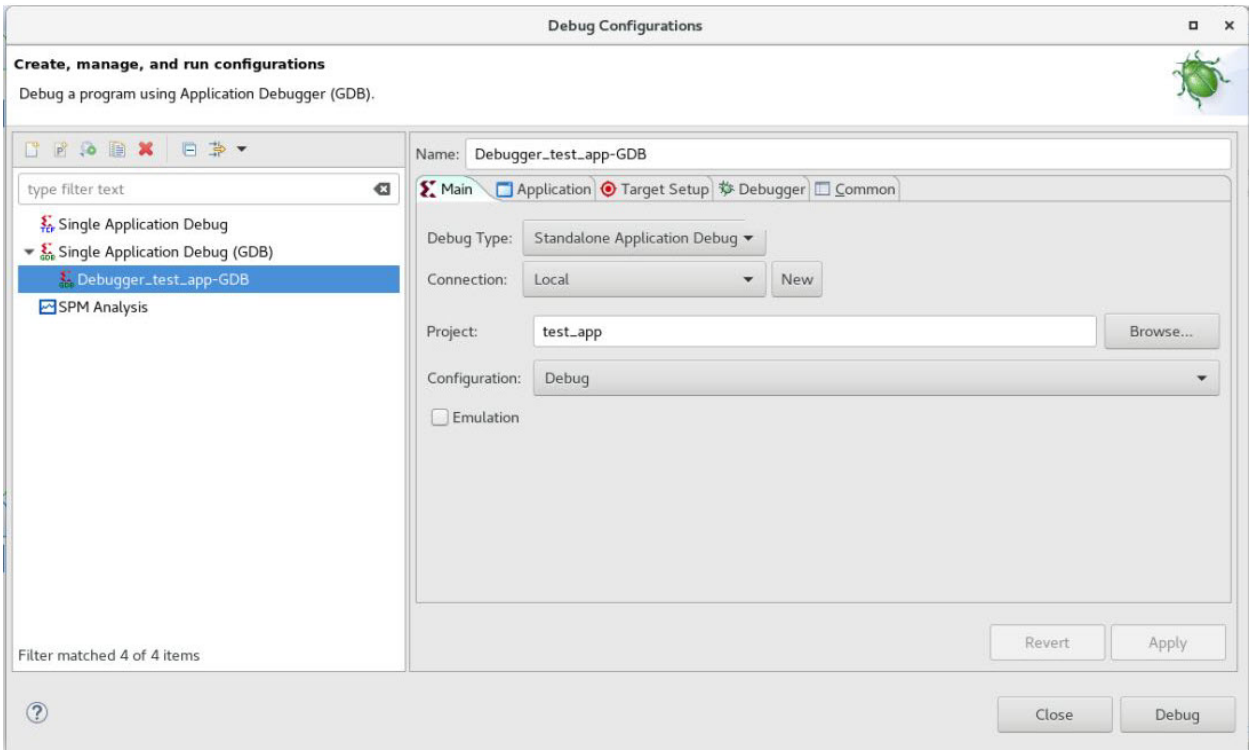
The debug logic for each processor enables program debugging by controlling the processor execution. The debug logic on soft MicroBlaze processor cores is configurable and can be enabled or disabled by the hardware designer when building the embedded hardware. Enabling the debug logic on MicroBlaze processors provides advanced debugging capabilities such as hardware breakpoints, read/write memory watchpoints, safe-mode debugging, and more visibility into MicroBlaze processors. This is the recommended method for debugging MicroBlaze software.

Working with GDB

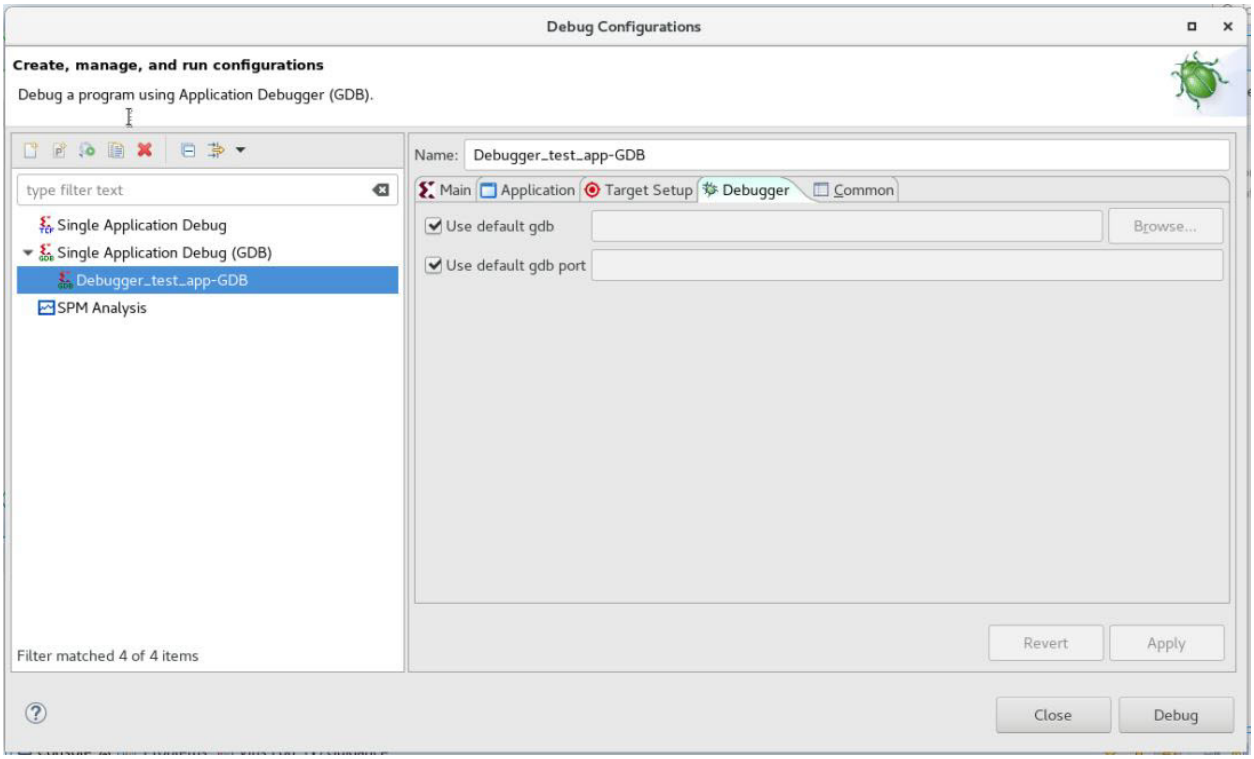
This topic describes how to use GDB to debug bare-metal applications.

To debug bare-metal applications:

1. Create a sample Hello World project.
2. Select the application and click **Run → Debug As → Single Application Debug (GDB)**. The Debug Configuration window opens with the Main tab selected.



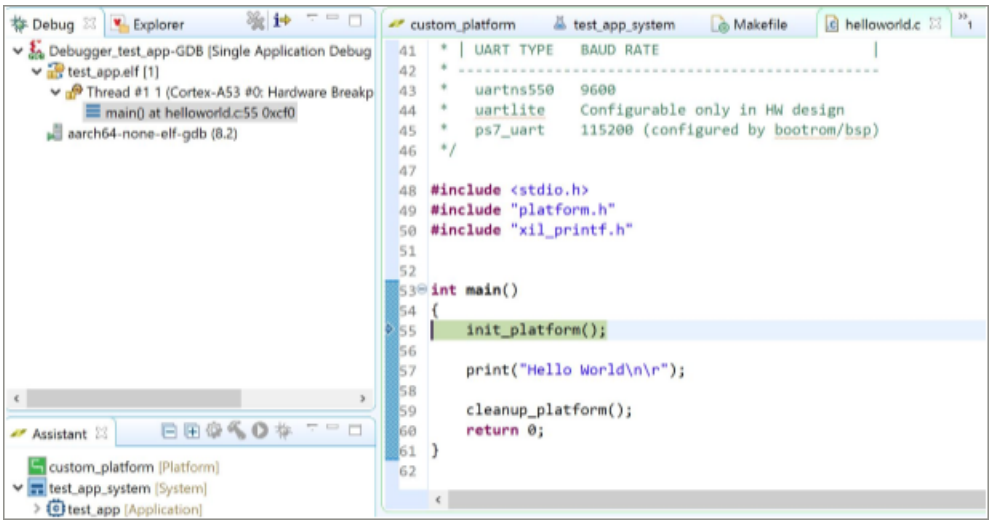
3. By default, the GDB shipped within the Vitis software platform is used with the default port, but you can specify the GDB and the port in the Debugger tab in Debug Configurations.



Note: Default ports used by the GDB server for different architectures are as follows:

- Arm: 3000
- Arm: AArch64
- MicroBlaze: 3002

4. Click the **Debug** button to start debugging the application.



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.

The Vitis software platform provides a linker script generator to simplify the task of creating a linker script for GCC. The linker script generator GUI examines the target hardware platform and determines the available memory sections. The only action required by you is to assign the different code and data sections in the ELF file to different memory regions.

Note:

- For multi-processor 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 address available in memory. If you are creating an application under a given hardware/domain project, the memory will overlap for the applications.

Generating a Linker Script for an Application

To generate a linker script for an application, do the following:

1. Select the application project in the **Project Navigator** view.

2. Right-click **Generate Linker Script**. Alternatively, you can click **Xilinx → Generate Linker Script**. The left side of the dialog box is read-only, except for the Output Script name and project build settings in the **Modify project build settings as follows** field. This region shows all the available memory areas for the design. You have two choices of how to allocate memory: using the Basic tab or the Advanced tab. Both perform the same tasks; however, the Basic tab is less granular and treats all types of data as "data" and all types of instructions as "code". This is often sufficient to accomplish most tasks. Use the **Advanced** tab for precise allocation of software blocks into various types of memory.
3. Click **OK**.

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

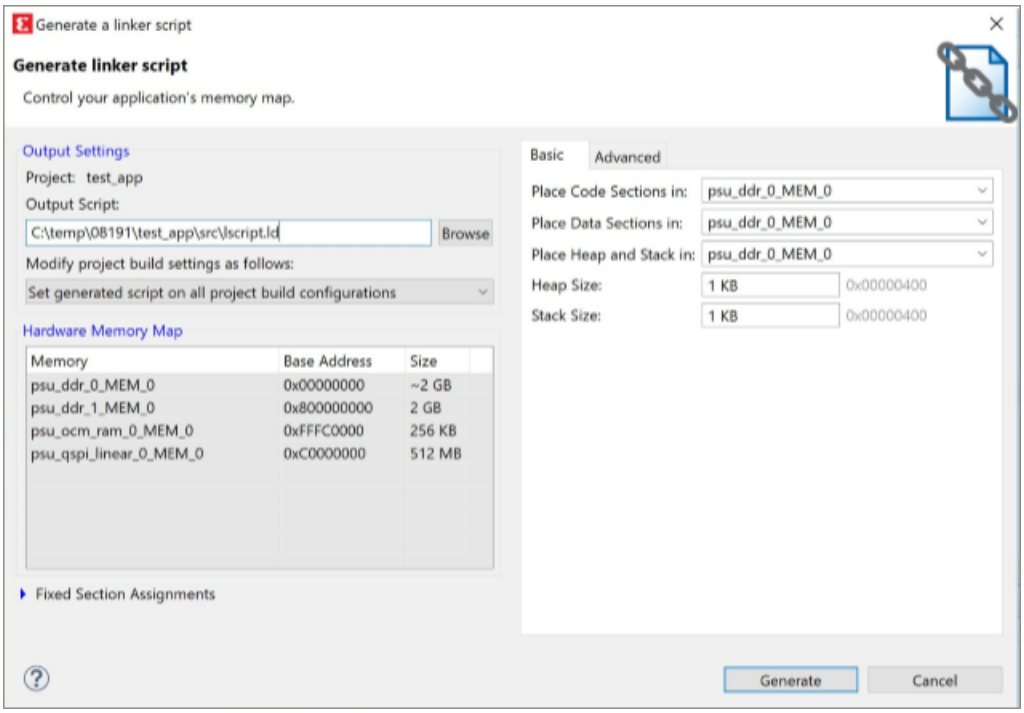
   Note: If the linker script already exists, a message window appears, asking if you want to overwrite the file. Click **OK** to overwrite the file or **Cancel** to cancel the overwrite.

   The Vitis software platform automatically adds the linker script to the linker settings for a managed make project based on the options selected in **Modify project build settings** as follows.

Basic Tab

Configure the following sections of the Linker Script Generator dialog box Basic tab. Placing these key sections into the appropriate memory region can improve performance. Use the drop-down menu next to the code, data, and heap or stack sections to select the region and type of memory that you want these blocks to reside in.

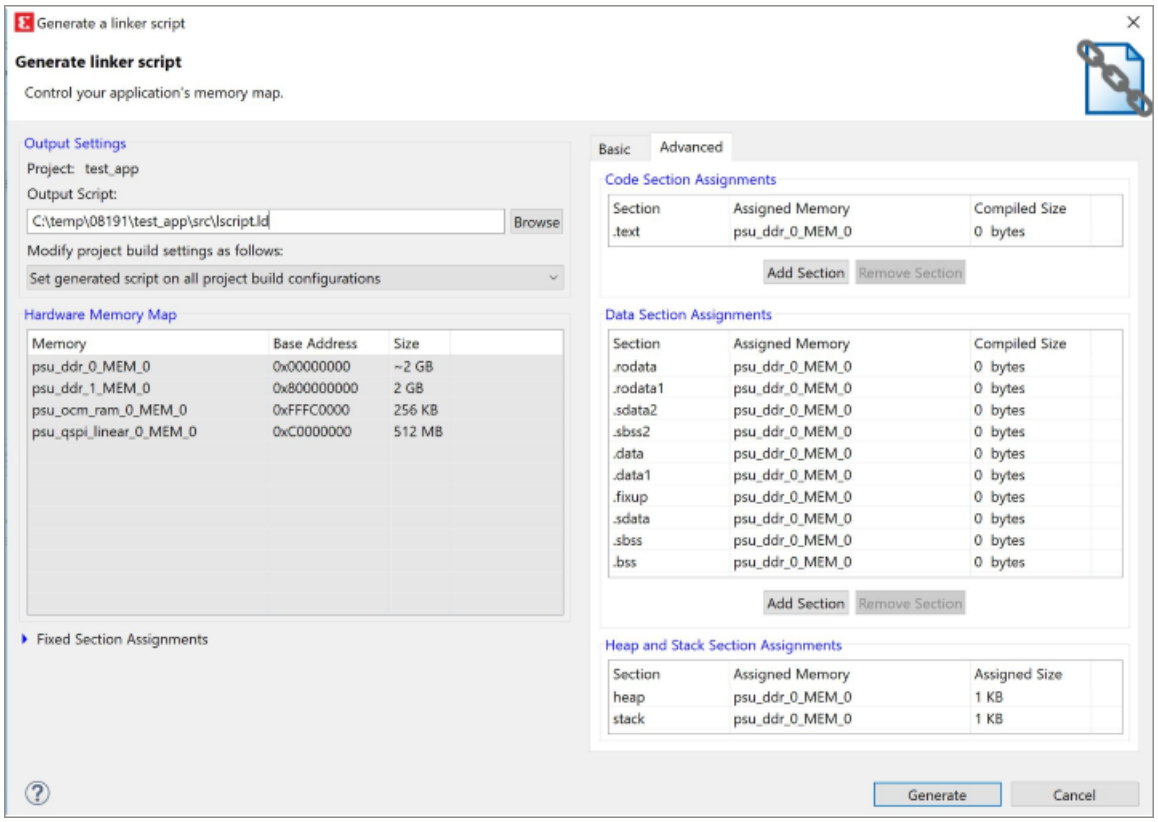
- **Code Sections:** This is used to store the executable code (instructions). Typically DDR memory is used for this task. Sometimes interrupt handlers or frequently used functions are built into separate sections and can be mapped to lower latency memory such as BRAM or OCM.
- **Data Sections:** Place initialized and uninitialized data in this region. Often DDR memory is used; however, if the data size requirements are small, OCM or BRAM can be used to improve performance.
- **Heap and Stack:** Heap is accessed through dynamic memory allocation calls such as `malloc()`. These sections are typically left in DDR unless they are known to be small, in which case they can be placed in OCM or BRAM. If the stack is lightly used, no significant performance loss will occur if left in DDR.
- **Heap Size:** Specify the heap size. Even if a programmer does not use dynamic memory allocation explicitly, there are some functions that use the heap such as `printf()`. It is a good idea to allocate a few KB for such functions, as a precaution.
- **Stack Size:** Specify the stack size. Remember that the stack size grows down in memory and could overrun the heap without warning. Make certain that you allocate enough memory, especially if you use recursive functions or deep hierarchies.



Advanced Tab

If you require more control over the definition of memory sections and assignments to them, use the LinkerScript Generator dialog box Advanced tab.

- **Code Section Assignments:** Typically there is only one code section, `.text`, unless you specifically created other code sections. All the code sections appear in this region.
- **Data Sections Assignments:** The compilers automatically generate a number of different types of data sections including read-only data (`.rodata`), initialized data (`.data`), and uninitialized data (`.bss`).
- **Heap and Stack Section Assignments:** Use this area to map the heap and stack onto memory and define their sizes.
- **Heap Size:** Specify the heap size. Even if a programmer does not use dynamic memory allocation explicitly, there are some functions that use the heap such as `printf()`. It is a good idea to allocate a few KB for such functions, as a precaution.
- **Stack Size:** Specify the stack size. Remember that the stack size grows down in memory and could overrun the heap without warning. Make certain that you allocate enough memory, especially if you use recursive functions or deep hierarchies.



Manually Adding the Linker Script

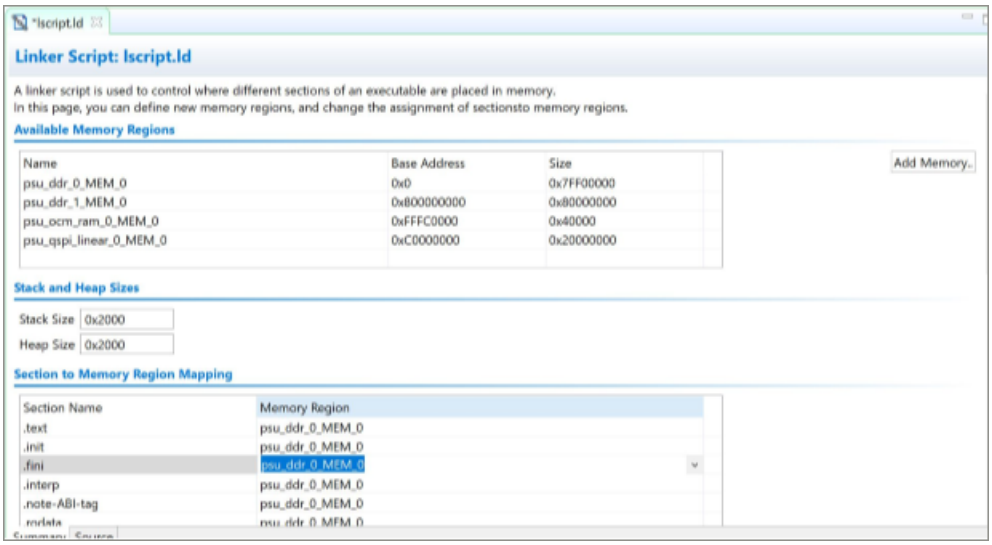
If you want to manually add the linker script for a managed make flow, do the following:

1. Right-click your managed make project and select **C/C++ Build Settings**.
2. Click the linker corresponding to your target processor, for example **ARM v8 gcc linker**.
3. Select **Linker Script** to add the linker script.
4. For standard make projects, add the linker script manually to your Makefile linker options.

Modifying a Linker Script

When you generate a linker script, there are multiple ways in which you can update it.

1. Open the linker script using a text editor, and directly edit the linker script. Right-click on the linker script and select **Open With → Text Editor**.
2. Regenerate the linker script with different settings using the linker script generator.
3. Use the linker script editor to make modifications. To do this, double-click on the linker script. The custom linker script editor displays relevant sections of the linker script.



The linker script editor provides the following functionality.

Table 5: **Linker Script Editor Functionality**

Name	Function
Available 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.

Creating a Library Project

You can create a managed make library project by using the New Library Project wizard.

To create a library project:

1. Click **File → New → Library Project**.
2. Click **Next**. The **New Library Project** wizard appears.
3. Type a project name into the **Project Name** field.
4. Select the location for the project. You can use the default location as displayed in the **Location** field by leaving the **Use default location** check box selected. Otherwise, click the check box and type or browse to the directory location.
5. Select a **Shared Library** or a **Static Library** project as required. The following table lists the flags set by the tool during the library project creation.

Table 6: **Library Project Creation Flags**

		Standalone			Linux			
		Static Library			Static Library		Shared Library	
Processor	**Toolchain**	**Extra Compiler Flags**	**Archiver Flags**	**Extra Linker Flags**	**Extra Compiler Flags**	**Archiver Flags**	**Extra Compiler Flags**	**Extra Linker Flags**
A9	Linaro	"-mcpu=cortex-A9 -mfpu=vfpv3 -mfloat-abi=hard"	None	None	"--static"	None	"-fPIC"	"-shared"
A9	Code Sourcery	None	None	None	"--static"	None	"-fPIC"	"-shared"
A53	Linaro	None	None	None	"--static"	None	"-fPIC"	"-shared"
A53-32 Bit	Linaro	"-march=armv7-a"	None	None	"--static"	None	"-fPIC"	"-shared"
R5	Linaro	"-mcpu=cortex-r5"	None	None	NA	NA	NA	NA
MicroBlaze	Xilinx	"-mcpu=v9.5 -mlittle-endian -mno-xl-soft-mul -mxl-barrel-shift -mxl-pattern-compare"	"-mlittle-endian"	None	"--static"	None	"-fPIC"	"-shared"

6. In platform selection page, select the platform to create your library project with the desired library type. Click **Next** to proceed.
7. Select your preferred domain and language (**C** or **C++**).
8. Click **Finish** to create your library project.

   Note: For MicroBlaze processor based projects, the Vitis software platform does not support any option to specify hardware. Specify the compiler options in the **Extra Compiler Flags** based on your hardware.

9. You can now add your source files, write exposed APIs in the header files, update compiler settings, and build the project to generate a library. Depending on the library type selected during the creation of the library project, a shared library (`<library_name>.so`) or a static library (`<library_name>.a`) is generated.

Creating a New Zynq UltraScale+ MPSoC FSBL Application Project

To create a new Zynq UltraScale+ MPSoC FSBL application in the Vitis software platform, do the following:

1. Click **File → New → Application Project**.

   The New Application Project dialog box appears.
2. In the Project Name field, type a name for the new project.
3. Select the location for the project. To use the default location as displayed in the Location field, leave the **Use default location** check box selected. Otherwise, click to deselect the check box, then type or browse to the directory location.
4. Select **Create a new platform from hardware (XSA)**. The Vitis IDE lists the all the available pre-defined hardware designs.
5. Select any one hardware design from the list and click **Next**.
6. From the CPU drop-down list, select the processor for which you want to build the application. This is an important step when there are multiple processors in your design. In this case you can either select **psu_cortexa53_0** or **psu_cortexr5_0**.
7. Select your preferred language: **C**.
8. Select an OS for the targeted application.
9. Click **Next.**
10. In the Templates dialog box, select the Zynq UltraScale+ MPSoC FSBL template.
11. Click **Finish** to create your application project and board support package (if it does not exist).

User Makefile Flow

The Vitis IDE supports the import of a user Makefile. See the following steps for details.

1. Create an empty project for the platform.
2. Import the sources and Makefile(s) of the command line application.
3. Right-click the project, and select **C/C++ Build Settings**.
4. Deselect the **Generate Makefiles automatically** checkbox. In the Build directory field, enter the directory where you imported the Makefile.
5. The default build command is `make`. You can customize this build command by deselecting the **Use default build command** checkbox and then entering your custom build command in the text entry field.
6. Select the **Behavior** tab. Update the build, incremental build, and clean commands as required.

Using Custom Libraries in Application Projects

You can create custom libraries for common utilities and use them in the application projects. To use the custom libraries in an application project, do the following:

1. Create a custom library using the New Library Project wizard. For more details, see Creating a Library Project.
2. Select the project for which you want to include the custom library, in the Project Explorer view.
3. Select **Project** → **Properties**. The **Properties for <project>** window appears. The left panel of the window has a properties list. This list shows the build properties that apply to the current project.
4. Expand the **C/C++ Build** property.
5. Click on **Settings**.
6. Under **Tool Settings** tab, expand the **gcc compiler** list.
7. Select **Directories** to change the add the library header file path. You can now include the required header files from the library project to the application.
8. Expand the **gcc linker** list.
9. Select **Libraries** to add the custom library and the library path to the application project.
10. Click **Apply** to save the settings.
11. When you finish updating the tools and their settings, click **OK** to save and close the **Properties for <project>** window.

Chapter 8

Run, Debug, and Optimize

Run Application Project

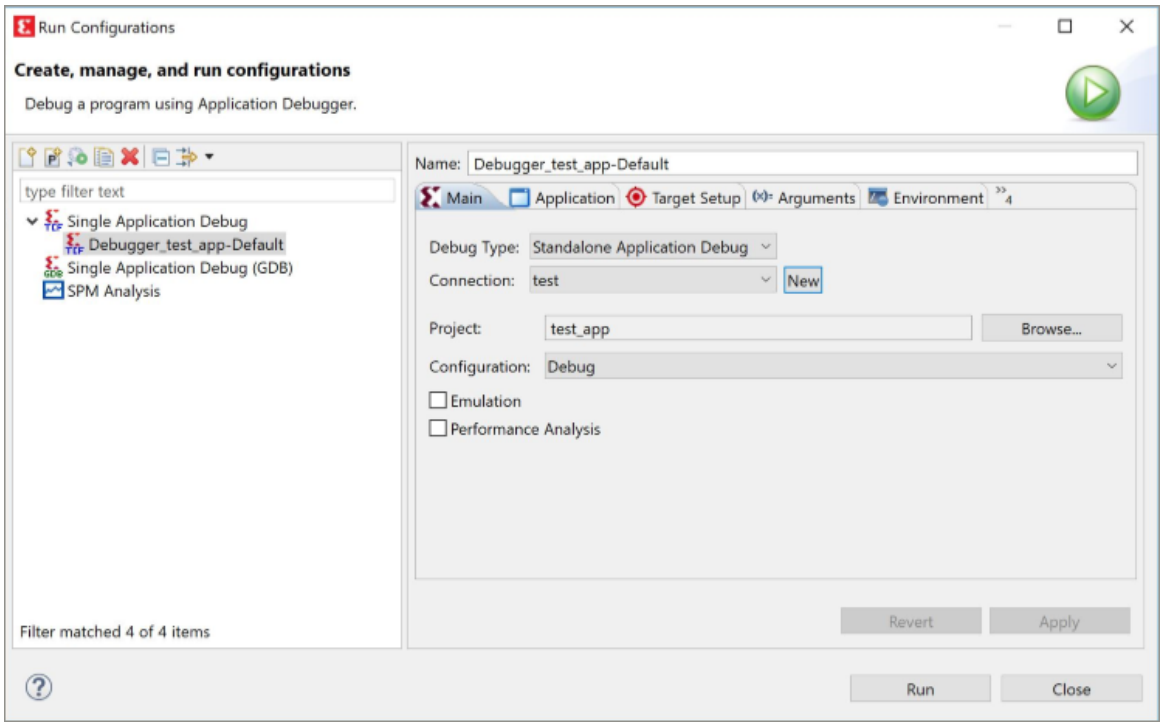
Launch Configurations

To debug, run, and profile an application, you must create a launch configuration that captures the settings for executing the application. To do this, right-click on the application project and select **Run As → Run Configurations ...**. The Run configuration window opens. Double click the **Single Application Debug** to create a Run Configuration. The Run Configuration window opens with the Main tab.

Main Tab

The main tab has the following options:

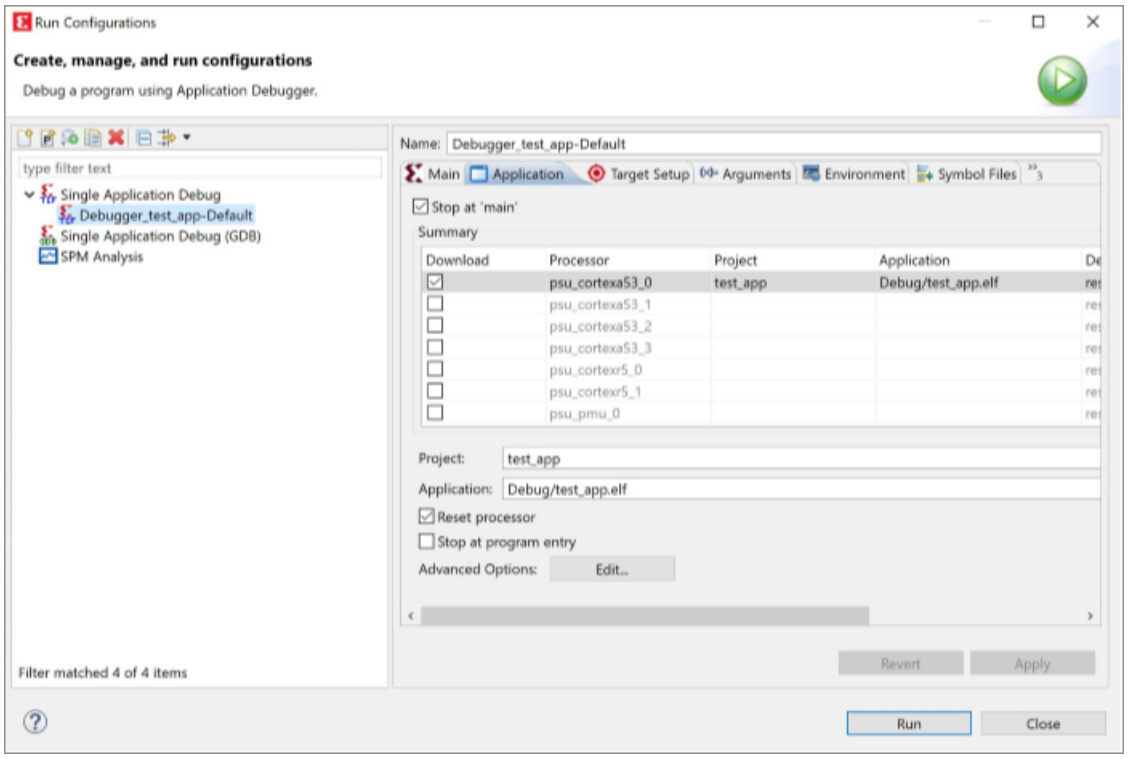
- **Debug Type:** You can choose from Standalone Application Debug, Linux Application Debug, or Attach to running target.
- **Connection:** In the connection field, you can create a target connection by clicking **New**.



***Note*:** The other options will populate automatically to run the application.

Application Tab

In the Application tab, set up the details for your application project and select the ELF file.



- **Stop At Main:** Used to stop the debugger at `main()` function.
- **Stop at Program Entry:** Used to stop the debugger at program entry.
- **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.
- **Advanced Options:** These options are used for profiling an application. Click **Edit** to see the options. The options to select are **This is a self relocating application** and **Profiling Options**.

Target Setup Tab

Provide a unique name for your configuration. Next, in the **Target Setup** tab, set up the following details:

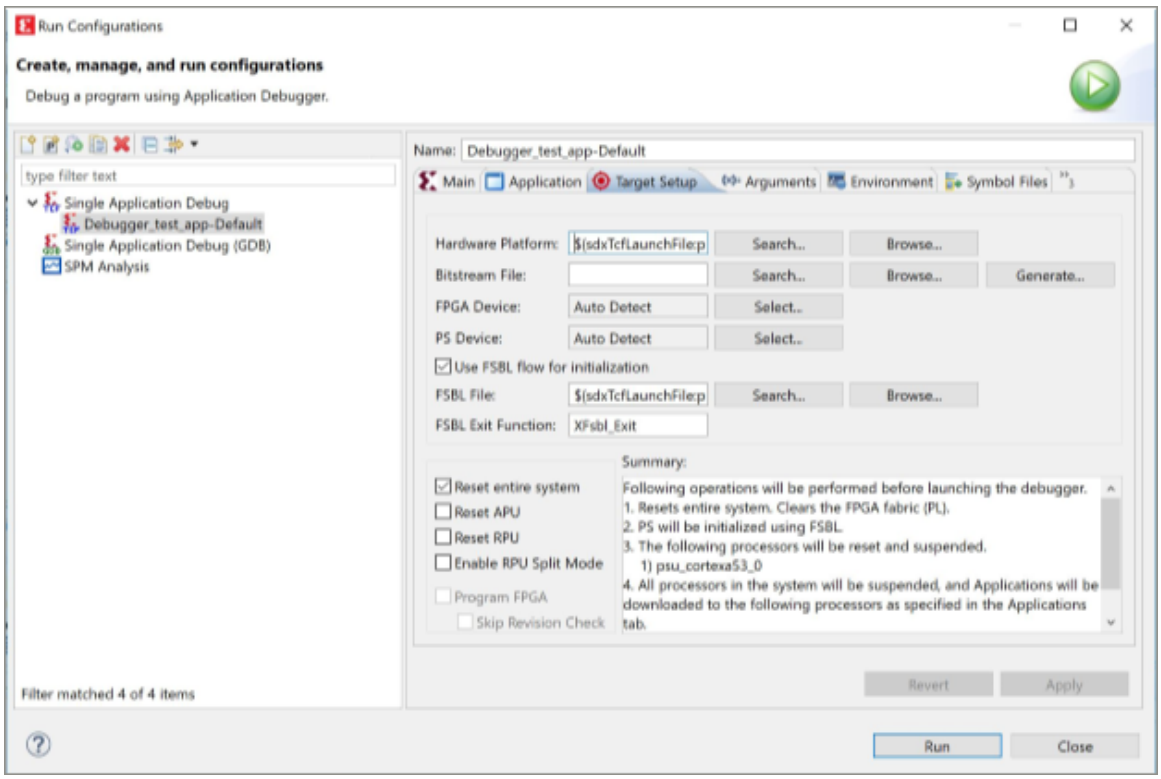
- Debug Type
- Connection: Local or Remote

Select **Local** for running the program on a target that is connected to local host.

Create a remote connection by clicking **New**, and select the same for running the program on a target connected to the remote host.

- **FPGA Device:** This is automatically selected for you.

- **PS Device:** This is automatically selected for you.
- **Hardware Platform:** Select the hardware platform for your design.
- **Bitstream file:** Search or browse to your Bitstream file.
- **FSBL File or Initialization File:** Selects either the FSBL file or Initialization file based on whether the checkbox is selected. By default, the **Use FSBL Flow for Initialization** check-box is checked.
- **Reset Entire System:** Perform a system reset if there is only one processor in the system.
- **Initialize Using FSBL file:** Initialize PS using FSBL file.
- **Reset APU:** Reset all the APU processor cores.
- **Reset RPU:** Reset all the RPU processor cores.
- **Enable RPU Split Mode:** Put RPU cores in split mode so that they can be used independent of each other.
- **Program FPGA:** To program the bit file.
- **Skip Revision Check:** Enabling this option will skip the device revision while programming bit stream.



Profiler

The Vitis™ unified software platform provides capability to profile your software application. Use the Profiler tab to specify options for the profiler. Refer to Profile/Analyze for more information.

Creating or Editing a Launch Configuration

You can launch Run, Debug and Profile tasks directly with a set of default configurations. Right-click on the desired application and select **Run As**, or **Debug As**. Select **Launch on Hardware (Single Application Debug)** from the context menu.

Customizing Launch Configurations

The Launch Configurations preferences page allows you set filtering options that are used throughout the workbench to limit the exposure of certain kinds of launch configurations. These filtering setting affect the launch dialog, launch histories and the workbench.

Table 7: **Launch Configuration Options**

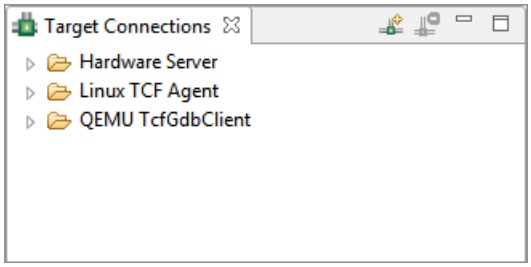
Option	Description	Default
Filter configurations in closed projects	Filter out configurations that are associated with a project that is currently closed	On
Filter configurations in deleted or missing projects	Filter out configurations that are associated with a project that has been deleted or are simply no longer available	On
Apply windows working set	Applies the filtering from any working sets currently active to the visibility of configurations associated with resources in the active working sets. That is to say, if project P has two configurations associated with it, but is not in the currently active working set, the configurations do not appear in the UI, much like P does not.	On
Filter checked launch configuration types	Filter all configurations of the selected type regardless of the other filtering options. The checked options are not displayed in the Run/Debug Configurations dialog box. ***Note*:** To avoid confusion, only configurations that are supported by the Vitis software platform are available by default.	On
Delete configurations when associated project is deleted	Any launch configurations associated with a project being deleted are also deleted if this option is enabled. After they have been deleted, the configurations are not recoverable.	On
Migrate	As new features are added to the launching framework, there sometimes exists the need to make changes to launch configurations. Some of these changes are made automatically, but those that are not (nonreversible ones) are left up to the end user. The migration section allows you to self-migrate any launch configurations that require it. Upon pressing the **Migrate...** button, if there are any configurations requiring migration, they are presented to you, and you can select the ones that you want to migrate.	

Target Connections

The **Target Connections** view 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 Hardware Server agent. In order to connect to remote targets, the hardware server agent must be running on the remote host, which is connected to the target.

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



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:

1. In the Target Connections window of the Vitis IDE, click the **Add Target Connection** button ( ).
2. The Target Connection Details dialog box opens.
3. In the Target Name field, type a name for the new remote connection.
4. Check the **Set as default target** checkbox to set this target as default. The Vitis software platform uses the default target for all the future interactions with the board.
5. 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.
6. In the Port field, type the port number on which the hw_server is running. By default, the hw_server runs on port 3121.
7. Select **Use Symbol Server**, if the hardware server is running on a remote host.
8. Click **OK** to create a new target connection.

Setting Custom JTAG Frequency

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

To set a custom JTAG frequency:

1. In the **Target Connections** view, click the **Add Target Connection** button ( ). The Target Connection Details dialog box opens.
2. Specify the name of the new remote target connection, for example **test**.
3. Check the **Set as default target** checkbox to set this target as default. The Vitis software platform uses the default target for all the future interactions with the board.
4. Specify 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.
5. Specify the port number on which the hw_server is running. By default, the hw_server runs on port 3121. Select **Use Symbol Server**, if the hardware server is running on a remote host.
6. Click **Advanced** to view the JTAG device chain details.
7. Select the JTAG device chain and click **Frequency** to open the **Set JTAG Frequency** dialog box.
8. 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.
9. Click **OK** to save the configuration and close the **Set JTAG Frequency** dialog box. The selected frequency is saved in the workspace and is used to set the frequency before executing a connect command for the selected device.
10. Click **OK** to create a new target connection.

    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 Connections** 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) `/opt/Xilinx/ 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**.

Viewing Memory Contents

The Memory view lets you monitor and modify your process memory. The process memory is presented as a list called memory monitors. Each monitor represents a section of memory specified by its location called base address. Each memory monitor can be displayed in different predefined data formats known as memory renderings.

The Memory view contains these two panes:

- **Monitors** panel - Displays the list of memory monitors added to the debug session currently selected in the **Debug** view.
- **Renderings** panel - Displays memory renderings. The content of this panel is controlled by the selection in the **Monitors** panel.

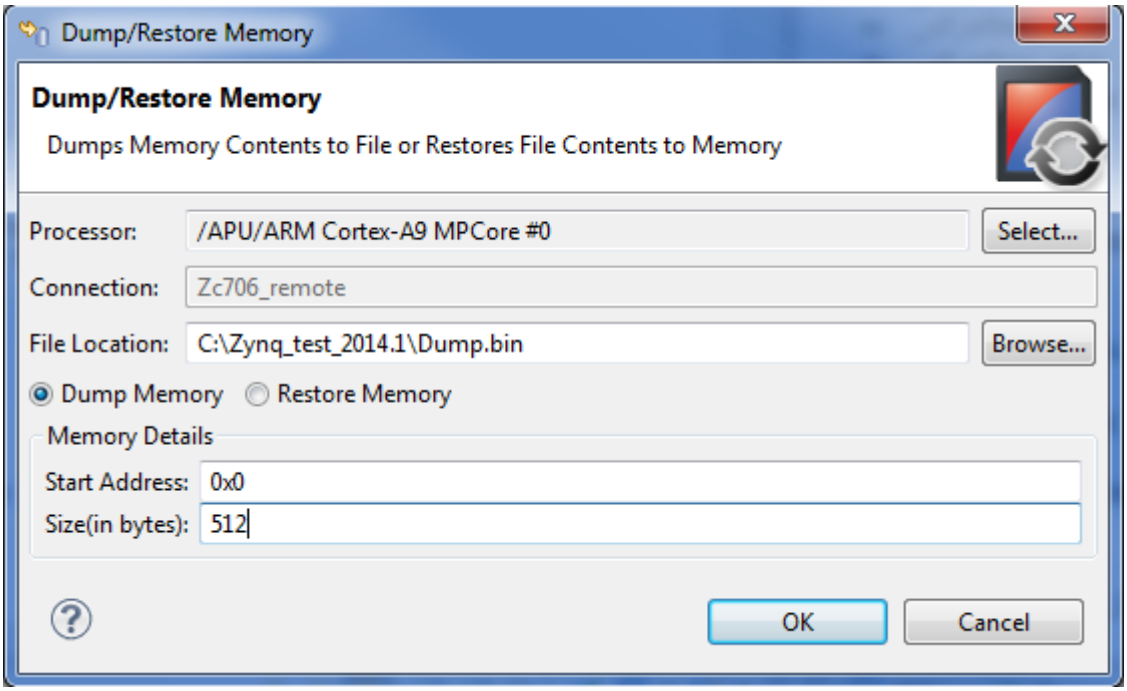
To open the **Memory** view, click the **Memory** tab of the **Debug** perspective. Alternatively, from the IDE menu bar, select **Window → Show View → Memory**.

Dump/Restore Memory

The Memory window does not have the ability to load or dump memory contents from or to a file.

You can use the Dump/Restore Memory function to copy the memory file contents to a data file and restore data file contents back to memory. To do this:

1. Launch the hardware server, if it is not already running.
2. Select **Xilinx → Dump/Restore Memory**.
3. The Dump/Restore Memory dialog box opens.



4. Click **Select** to select a Processor from the Select Peer and Context window. The Vitis software platform creates peers based on available target connections. For this example, the Vitis software platform creates a peer called Zc706_remote.
5. Select the peer corresponding to your Target connection from the Peers list (in this case, Zc706_remote), and then select the related processor, **ARM Cortex-A9 MPCore #0**, from the APU Context.

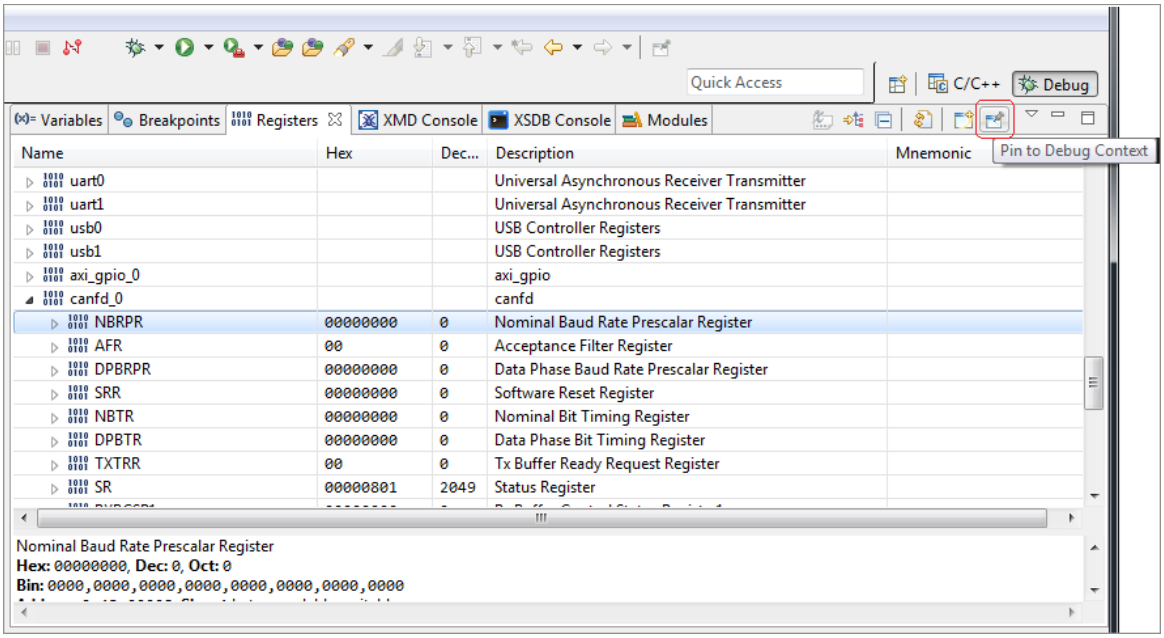
   Note: Select the processor context, not the device context. In the example here, the processor context is APU.
6. Click **OK** to select the processor.
7. Set the location of the data file to restore from or dump to.
8. Select either the **Restore Memory** or **Dump Memory** option button.
9. In the Start field, specify the starting memory address from which you want to dump or restore memory.
10. In the Size (in bytes) field, specify the number of bytes to be dumped or restored.
11. Click **OK**. The Vitis software platform dumps or restores data from the starting address specified.

Viewing Target Registers

The Registers view lists all registers, including general purpose registers and system registers. As an example, for Zynq® devices, the Registers view shows 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.

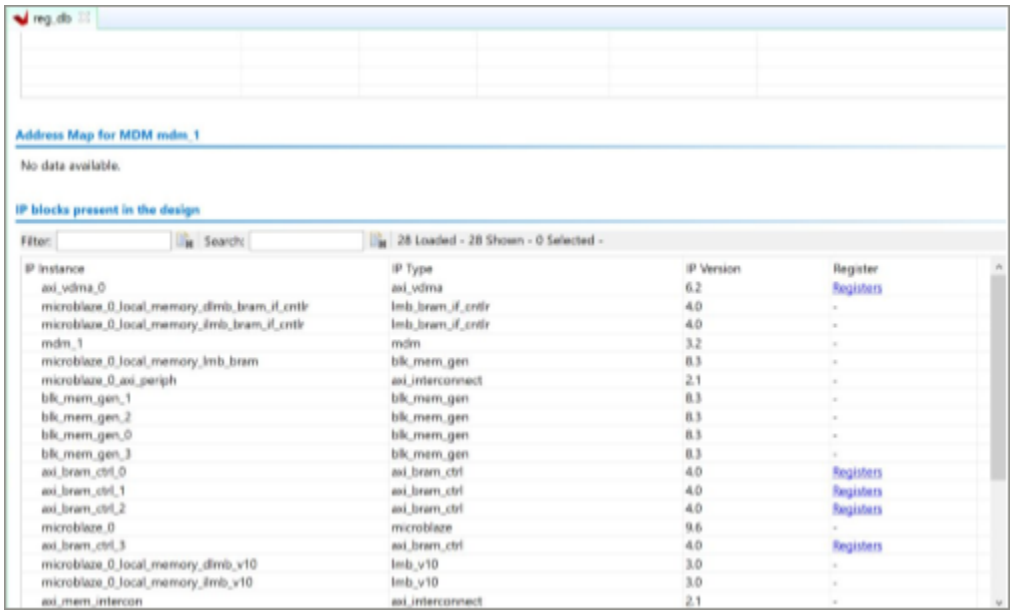
To open the Registers view, click the **Registers** tab of the Debug perspective. Alternatively, from the IDE menu bar, select **Window → Show View → Registers**.

You can modify editable field values, during debug. You can also pin the Registers view using the Pin to Debug Context toolbar icon, as shown in the figure below.

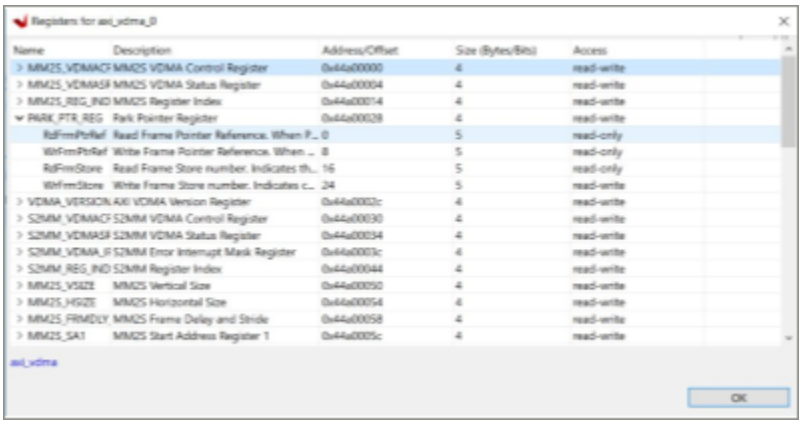


Viewing IP Register Details

The Vitis software platform now supports viewing of IP register details, using either the Hardware (`system.xsa`) view or during debug using the Registers view. After successful platform project creation, the `system.xsa` file in the **Hardware Specification** view is opened. The file now displays cross-references to the registers of IP blocks present in the design.



To view register details, click on the **Registers** link on the Hardware Specification view.



Debug Application Project

System Debugger Supported Design Flows

Standalone Application Debug Using Xilinx System Debugger

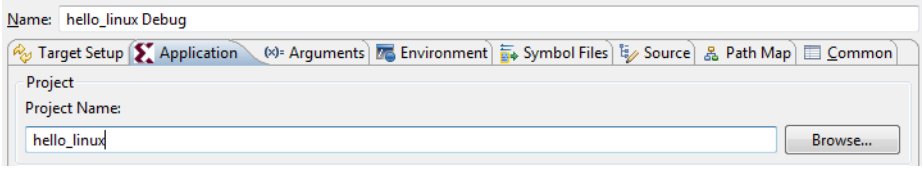
This topic describes how to use the Xilinx System Debugger to debug bare-metal applications.

1. Create a sample Hello World project.

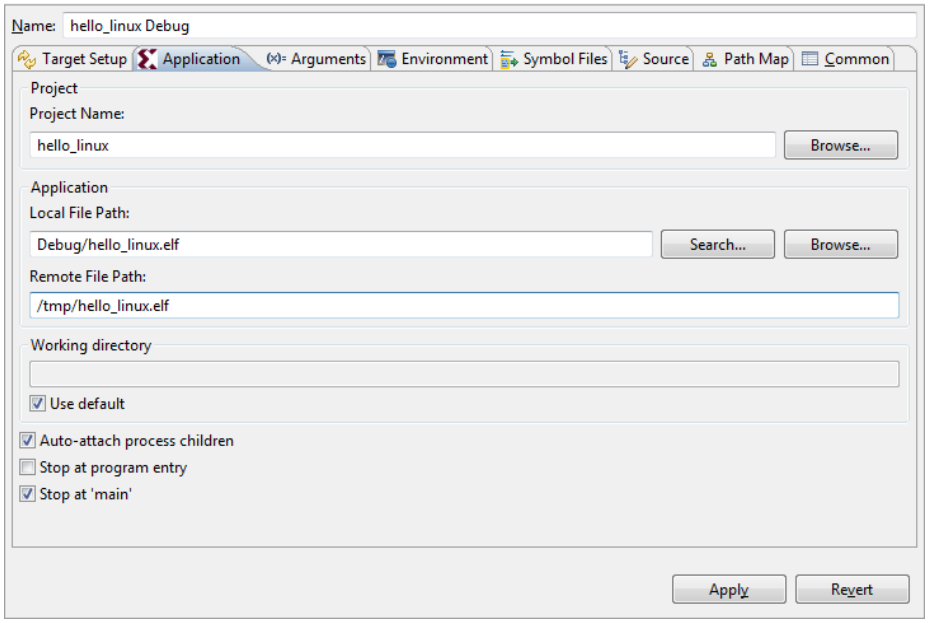
2. Select the application and click **Run → Debug As → Launch On Hardware (Single Application Debug)**.

Linux Application Debugging with System Debugger

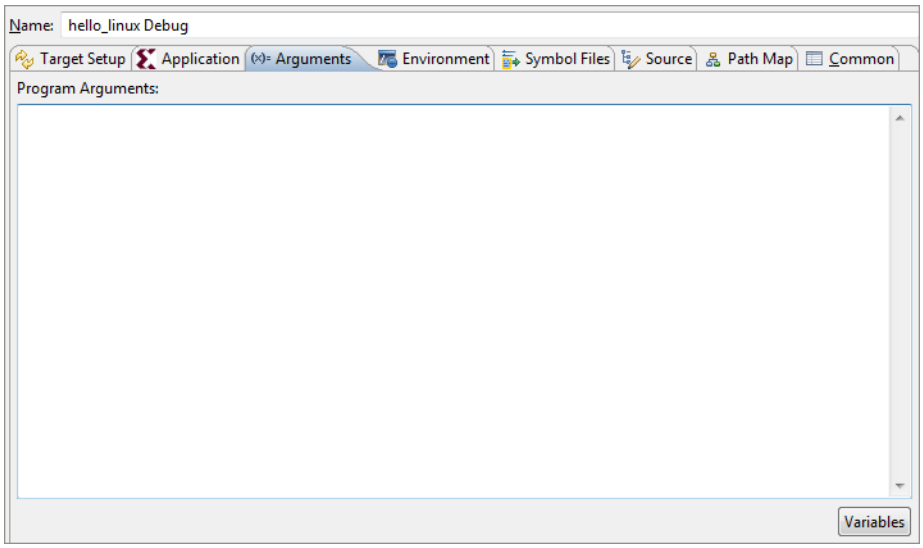
1. Launch the Vitis software platform.
2. Create a Linux application.
3. Select the application you want to debug.
4. Right-click on the application and select **Debug As → Debug Configuration**.
5. Click **Launch on Hardware (Single Application Debug)** to create a new configuration.
6. In the Debug Configuration window:
   a. Click the **Target Setup** tab.
   b. From the Debug Type drop-down list, select **Linux Application Debug**.
   c. Provide the Linux host name or IP address in the Host Name field.
   d. By default, tcf-agent runs on the 1534 port on the Linux. If you are running tcf-agent on a different port, update the **Port** field with the correct port number.
   e. In the Application Tab, click **Browse** and select the project name. The Vitis software platform automatically fills the information in the application.



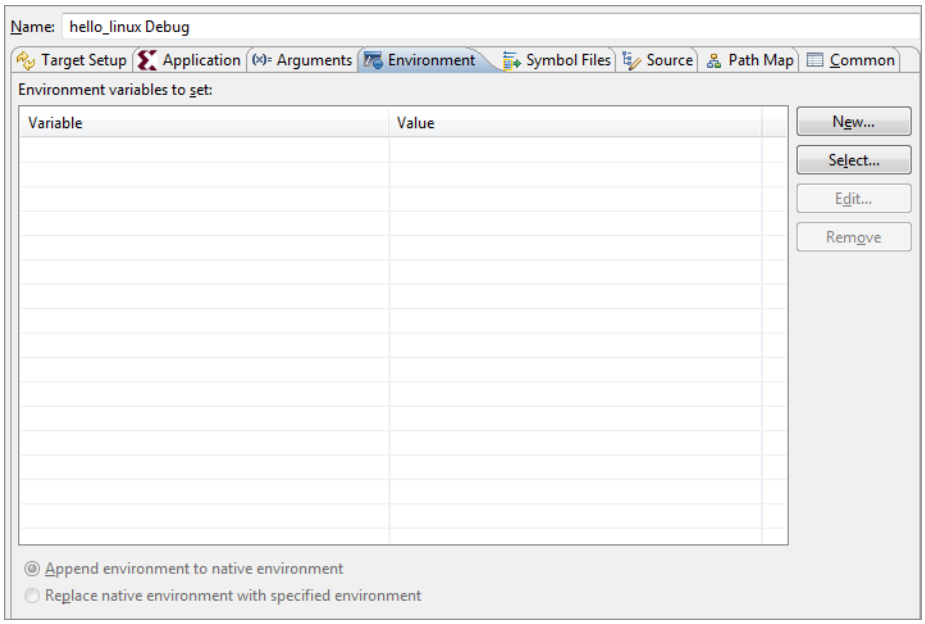
   f. In the Remote File Path field, specify the path where you want to download the application in Linux.



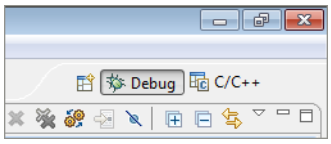
g. If your application is expecting some arguments, specify them in the Arguments tab.

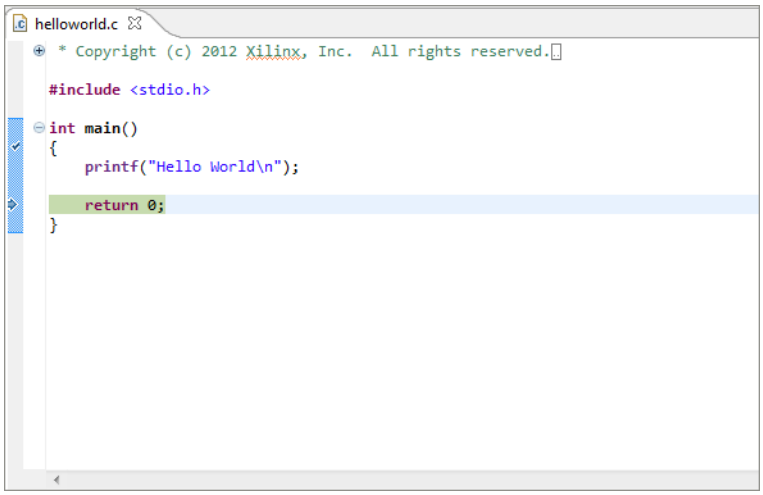


h. If your application is expecting to set some environment variables, specify them in the Environments tab.



i. Click the **Debug** button. A separate console automatically opens for process standard I/O operations.





j. Click the **Terminate** button to terminate the application.

Troubleshooting

My application already exists in the Linux target. How can I tell System Debugger to use my existing application, instead of downloading the application?

1. In the Application tab of System Debugger, leave the Project Name and Local File Path fields empty.

2. In the Remote File Path field, specify the remote application path and click the **Debug** button. System debugger loads the specified application.

Attach and Debug using Xilinx System Debugger

It is possible to debug the Linux kernel using Xilinx System Debugger. Follow the steps below to attach to the Linux kernel running on the target and to debug the source code.

1. Compile the kernel source using the following configuration options:

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
```

2. Launch the Vitis software platform.
3. Click **Window → Open Perspective → Debug**.
4. Right-click on the application and select **Debug As → Debug Configuration**.
5. In the Debug Configurations dialog box, select **Launch on Hardware (Single Application Debug)** and click the **New** button ( ).
6. Name the configuration **Zynq_Linux_Kernel_Debug**.
7. Debugging begins, with the processors in the running state.
8. Click the **Pause** button to suspend the processor: . Debug starts in the Disassembly mode.
9. Add vmlinux symbol files to both processor cores:
   a. Right-click on **ARM Cortex-A9 MPCore#0** and select **Symbol Files**.
   b. Click **add** and add vmlinux symbol files.
   c. Click **OK**.
   d. Right-click on **ARM Cortex-A9 MPCore#1** and select **Symbol Files**.
   e. Click **add** and add vmlinux symbol files.
   f. Click **OK**.
10. You must set up Source Lookup if you built the code on a Linux machine and try to run the debugger on Windows.
11. Select the debug configuration **Zynq_Linux_Kernel_Debug**, then right-click it and select **Edit Source Lookup**.
12. Click **Add**.
13. Select **Path Mapping** from the **Add Source** dialog box.
14. Add the Compilation path and local file system path by clicking **Add**.
15. Successful source lookup takes you to the source code debug.
16. You can add function breakpoints using the Breakpoints view toolbar.

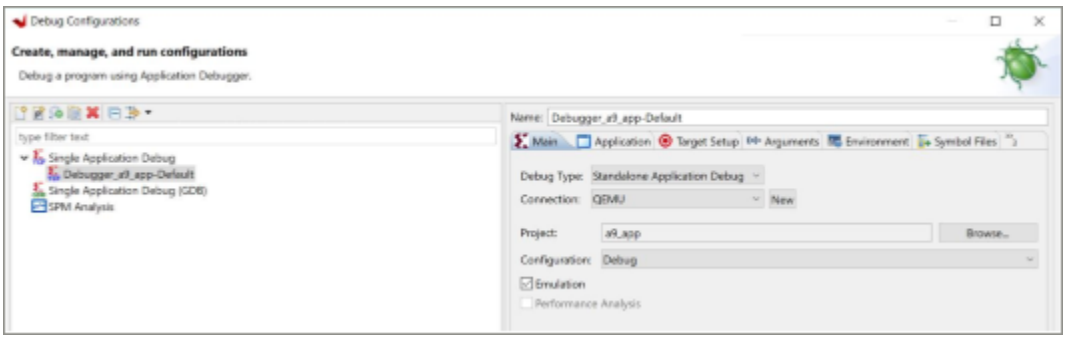
17. Add a breakpoint at the `start_kernel` function.
18. Click the reset button. The Zynq-7000 SoC processor boots from the SD card and stops at the beginning of the kernel initialization.

    Note: The Linux kernel is always compiled with full optimizations and in-lining enabled. Therefore, stepping through code might not work as expected due to the possible reordering of some instructions. Furthermore, some variables might be optimized out by the compiler and consequently might not be available for the debugger.

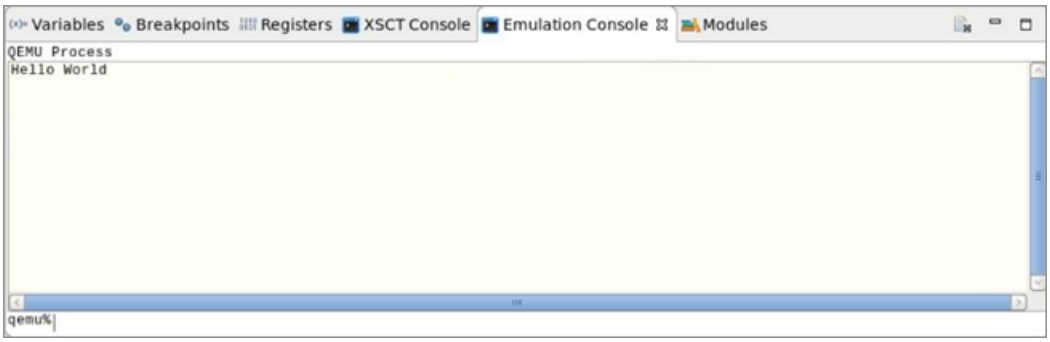
Standalone Application Debug using System Debugger on QEMU

1. Launch the Vitis software platform.
2. Create a standalone application project. Alternatively, you can also select an existing project.
3. Select **Debug As → Debug Configurations**.
4. Double-click **Launch on Emulator (Single Application Debug)** and select the **Emulation** check box on the Main Tab to create a new configuration.

   Note: Only hardware platforms based on Zynq UltraScale+ MPSoC can be selected for standalone application debugging.



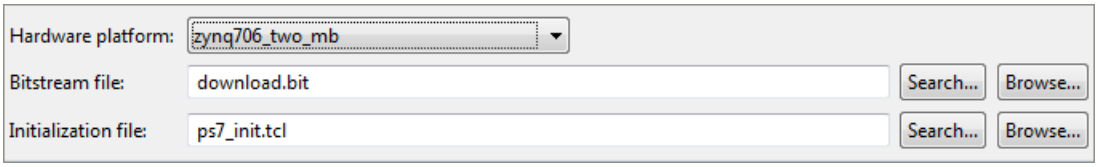
5. In the Debug Configuration dialog box:
   a. If your application is expecting some arguments, specify them in the Arguments tab page.
   b. If your application is expecting to set some environment variables, specify them in the Environments tab page.
6. Click **Debug**.
7. You can also launch the Emulation Console by selecting **Window → Show View → Other**. The Emulation Console can be used to interact with the program running on QEMU. The STDIN can be provided in the input box at the `qemu%` prompt. Output is displayed in the area above the input text.



Multi-Processor Debugging with System Debugger

You can debug multiple processors simultaneously with a single System Debugger debug configuration.

1. Create application projects for the processors included in the design.
2. Select any application and click **Debug As → Debug Configurations**.
3. In the **Debug Configurations** window, left panel, select the configuration type (Single Application Debug) and click the **New** button: .
4. Name the configuration **Multi_Processor_ZC706_Debug**.
5. In the Target Setup tab, select the appropriate setup.
6. Select the **Standalone Application Debug** from the **Debug Type** dropdown list.
7. Select the target you want to connect. With this selection, no resets or initializations are performed on the target before launching the debugger.
8. To automatically populate bitstream and initialization files, from the Hardware platform drop-down list, select the appropriate hardware platform. Use the **Browse** buttons if you wish to select different bitstream and initialization files.



9. If you want to reset the entire system, enable the **Reset entire system** checkbox in the Debug Configurations window.
10. If you want to program the bitstream after system reset, enable the **Program FPGA** checkbox.
11. Enable the **Run ps7_init** checkbox to run the PS initialization file.

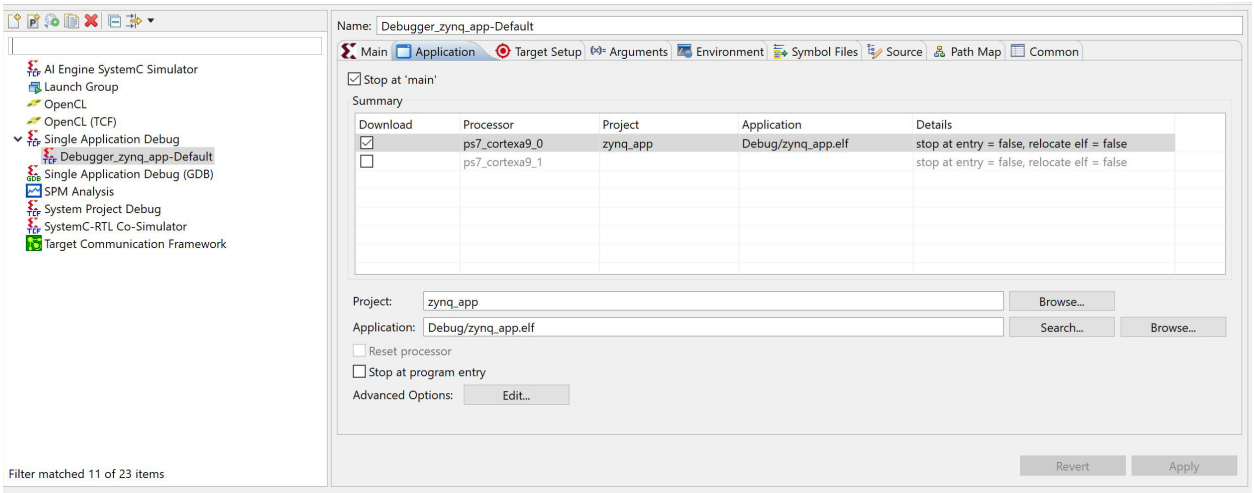
    Note: The Summary window displays a summary of System Debugger operations.

12. Select the **Applications** tab to display all processors available to the selected hardware platform.

13. Select the **Download Application** checkbox if you want to download the application to the selected processor.

    Note: If a single project exists for the processor, the Project Name and Application Name fields populate automatically when you select the **Download Application** checkbox. If more than one project exists for the processor, you must make the **Project Name** selection manually.

14. Select the **Stop at program entry** checkbox if you want to stop the processor before application main().



15. Click the **Debug** button to launch multi-processor debugging.

Using a Remote Host with System Debugger

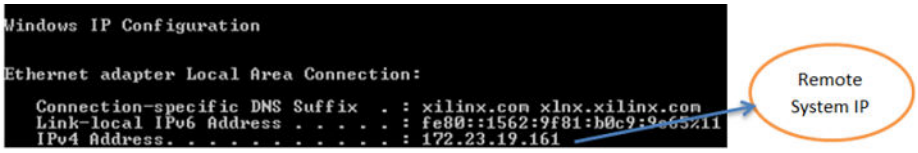
1. Setting Up the Remote System Environment

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

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

   b. Make sure your board is correctly connected.

   c. In a cmd window of the host machine, check the IP Address:



2. Setting Up the Local System for Remote Debug:

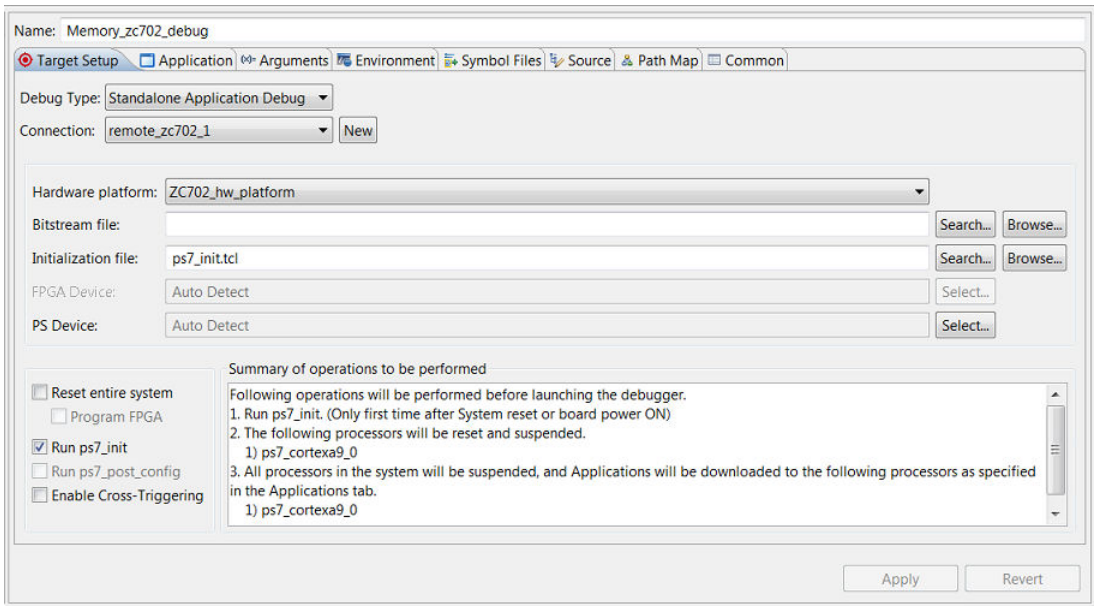
   a. Launch the Vitis software platform.

   b. Select the application to debug remotely.

   c. Select **Debug As → Debug Configurations**.

d. Create a new system debugger configuration.

e. In the Target Setup tab, click **New** to create a new target connection.



f. In the New Target Connection dialog box, add the required details for the remote host that is connected to the target.

g. Target Name: Type a name for the target.

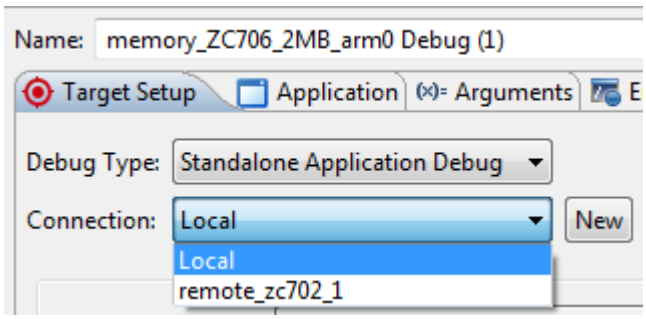
h. Host: IP address or name of the host machine.

i. Port: Port on which the hardware server was launched, such as 3121.

j. 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 Vitis software platform.

k. Click **OK**.

l. Now you can see that there are two available connections. In this case, remote_zc702_1 is the remote connection.



m. Select or add the remaining debug configuration details and click **Debug**.

OS Aware Debugging

OS aware debug over JTAG helps in visualizing OS specific information such as processes or threads that are currently running, process or thread specific stack trace, registers, variables view. By enabling the OS awareness, you can debug the OS running on the processor cores and the processes or the threads running on the OS simultaneously.

Enabling OS Aware Debug

This section describes setting up OS aware debug for a Zynq board running Linux from an SD card, using the Vitis IDE. It is assumed that users are aware of setting up a Jtag connection to the board, building Linux kernel and booting it from an SD card. For details on how to set up the kernel debug, refer to Attach and Debug using Xilinx System Debugger.

1. Compile the kernel source using the following configuration options:

   ```
   CONFIG_DEBUG_KERNEL=y
   CONFIG_DEBUG_INFO=y
   ```

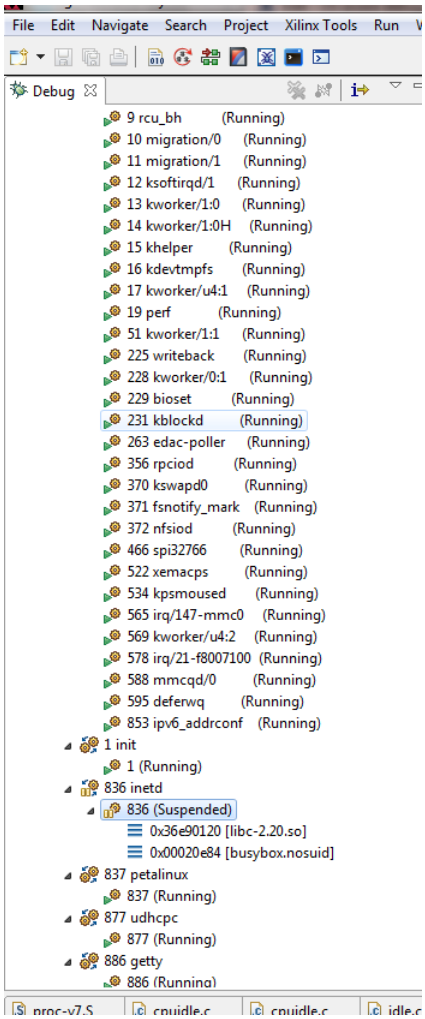
2. Launch the Vitis software platform.
3. Click **Window → Open Perspective → Debug**.
4. Click **Debug As → Debug Configurations**.
5. In the Debug Configurations dialog box, select **Single Application Debug** and click the **New** button ( ).
6. Click **Debug**.
7. Debugging begins, with the processors in the running state.
8. Select the **Enable Linux OS Awareness** option from the Debug view in the processor context.
9. You can also perform the following actions from the menu that appears.
   - **Refresh OSA Processes**: Select this option to refresh the list of running processes.
   - **Auto refresh on exec**: When selected, all the running processes are refreshed and seen in the **Debug** view. When not selected, new processes are not visible in the debug view.
   - **Auto refresh on suspend**: When selected, all the processes will be re-synced whenever the processor suspends. When not selected, only the current process is re-synced.
   - **Linux OSA File Selection**: Select this option to change the symbol file.
10. Alternatively, OS aware debugging can also be enabled using the `-osa` command in the Xilinx System Debugger (XSDB) command-line console.

    ```
    osa -file <symbol-file> -fast-step -fast-exec
    ```

Process/Thread Level Debugging

The Debug view is updated with the list of processes running on the Linux kernel, when the OS aware debugging is enabled. For details on how to enable OS aware debugging, refer to Enabling OS Aware Debug. The processes list is updated for the first time when the processor core is halted and is updated dynamically thereafter (new processes are added to the list and terminated processes are removed).

A process context can be expanded to see the threads that are part of the process.



Symbol files can be added for a process context to enable source level debugging and see stack trace variables. Source level breakpoints can also be set. Alternatively, the source level debugging can be enabled by setting the Path Map. The debugger uses the Path Map setting to search and load symbols files for all executable files and shared libraries in the system.

Note: Path Map is used for both symbols and source lookups.

Debugging a Process from main()

To debug a new process from `main()`, a global breakpoint (not against any particular target/context) should be set, before starting the process. Symbol files are loaded based on path map settings, so there should be a corresponding entry for the new process before starting it.

To debug a process from `main()`:

1. Select a project in the Project Explorer view.
2. Select **Debug As → Debug Configurations**. The Debug Configurations window appears.
3. Click the **Path Map** tab to set the path mappings for the selected debug configuration. Path maps help enable source level debugging. The debugger uses Path Map setting to search and load symbols files for all executable files and shared libraries in the system.
4. Set either the line breakpoint in the source file of the Linux application or function breakpoint at `main()`. Every time a new process starts, the debugger checks symbols of the process and plants the breakpoint in the process if the source file or the `main()` function is found in the symbols.
5. Run the application from the terminal.
6. As soon as the control hits a breakpoint, the **Debug** view is updated with the information of the process.
7. The Debug view also shows the file, function and the line information of the breakpoint hit. A thread label includes the name of the CPU core, if the thread is currently running on a core.
8. Source level debugging such as stepping in, stepping out, watching variables, stack trace can be performed. The target side path for a binary file does not include a mount point path. This is a known limitation. For example, when the process is located on the SD card, which is mounted at `/mnt`, the debugger shows the file as `<filename>` and not as the expected `/mnt/<filename>`.

Debugging a Loadable Kernel Module

To debug a kernel module, set path mapping to map the module name to symbol file of the module. To see loaded modules, select **Kernel** in the **Debug** view, and look at the **Modules** view. Kernel modules are listed by name and not by the file path.

To debug a kernel module:

1. Select a project in the **Project Explorer** view.
2. Select **Debug As → Debug Configurations**. The Debug Configurations window appears.
3. Click the **Path Map** tab to set the path mappings for the selected debug configuration.
4. Click **Add** to insert the kernel module.
5. Insert a function or line breakpoint and run the core. As soon as the breakpoint is hit, the debug view is updated with all the information.

6. Similar to any other process or thread level debugging, you can insert breakpoints, step in, step out, watch variables, stack trace or perform other source level debugging tasks.

Xen Aware Debugging

Xen aware debug helps users in visualizing the hypervisor specific information such as different domains (Dom-0 and Dom-Us), virtual processors (VCPUs) on each domain.

This feature enables debugging following Xen components:

- Hypervisor
- Dom-0/Dom-U kernel
- Dom-0/Dom-U user space processes
- Dom-U standalone applications

Enabling Xen Awareness

This section describes setting up the Xen aware debug for Zynq UltraScale+ MPSoC devices running Linux from SD card, using the Vitis IDE. It is assumed that the following prerequisites have been satisfied:

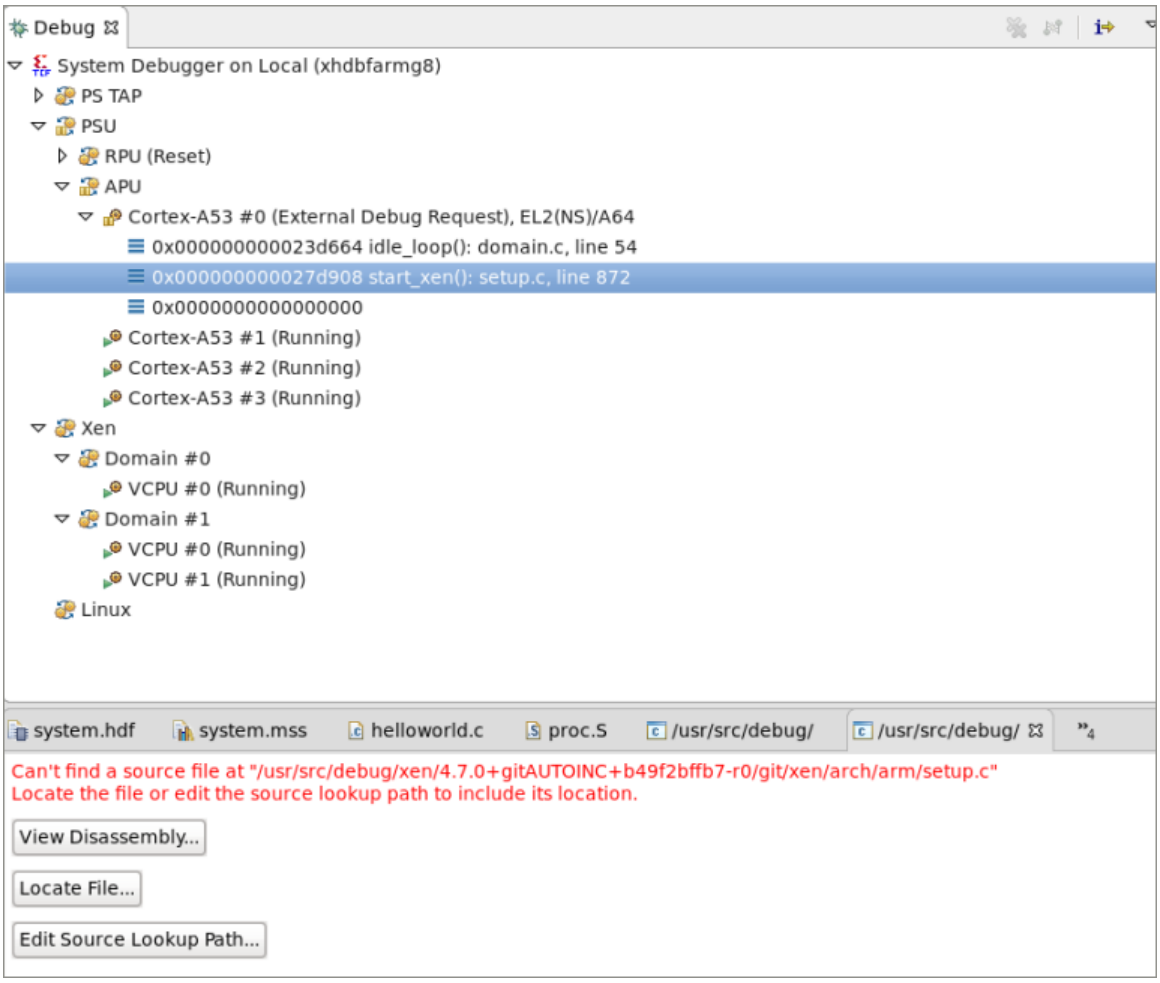
- You have the ZCU102 board running a Xen and Dom-0.
- You have the Xen symbol file (`xen-syms`).

For details on how to boot Xen and Dom-0, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).

1. Launch the Vitis IDE.
2. Select **Window → Open Perspective → Debug**.
3. Select **Debug As → Debug Configurations**.
4. In the Debug Configurations dialog box, select **Launch on Hardware (Single Application Debug)**.
5. Click **New** ( ).
6. Select **Attach to running target** debug type and click **Debug**. Debugging begins with the processors in the running state.
7. Right click **Cortex-A53 #0 target** and select **Symbol Files**.
8. Select the symbol file (**xen-syms**).
9. Select the **OS awareness** checkbox.

Debugging Hypervisor

1. Boot Xen and Dom-0. For details on how to boot Xen and Dom-0, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).
2. Enable Xen awareness by enabling OS aware debug for Xen symbol file. Symbol files are added to a process context to enable source level debugging. For details on how to enable Xen awareness, refer to Enabling Xen Awareness.
3. The Debug view is updated with the list of processes running on the Linux kernel when OS-aware debugging is enabled. The processes list is updated for the first time when the processor core is halted, and is updated dynamically thereafter (new processes are added to the list and terminated processes are removed).
4. Click **Edit Source Lookup Path** to set the path mappings for the selected debug configuration. The debugger uses path map to search and load symbols files for all executable files and shared libraries in the system.



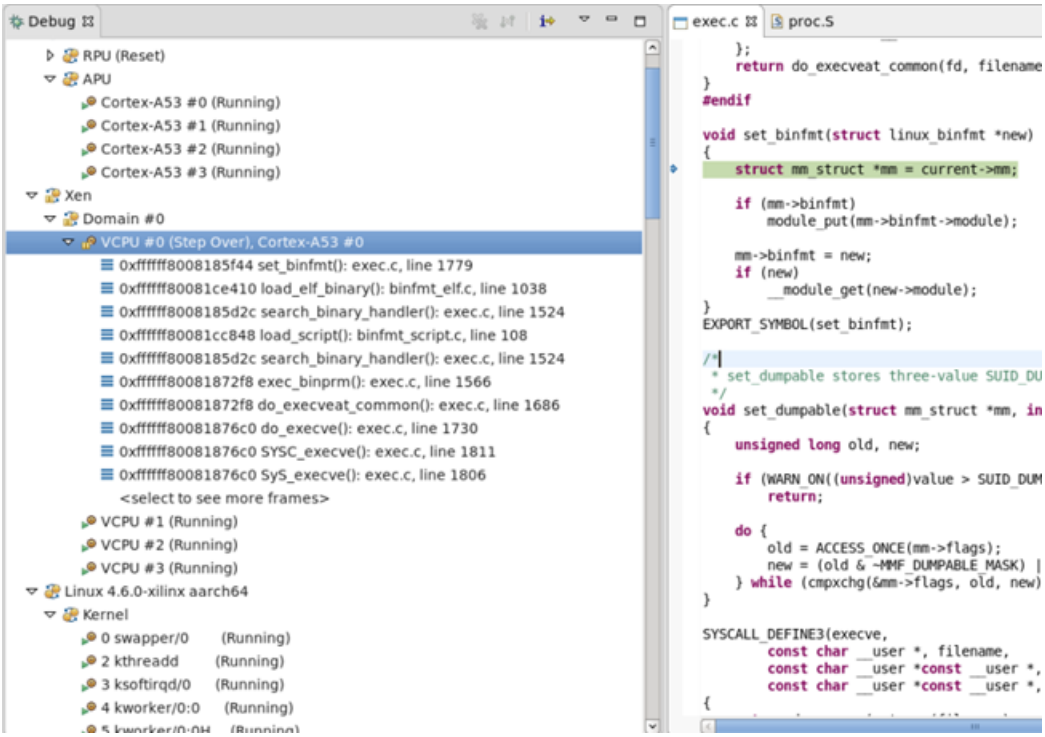
Note: Path Map is used for both symbols and source lookups.

5. Add a breakpoint or suspend the core. As soon as the breakpoint is hit, the debug view is updated with all the information.

6. You can now insert breakpoints, step in, step out, watch variables, stack trace or perform other source level debugging tasks.

Debugging a Dom-0/Dom-U Kernel

1. Boot Xen and Dom-0. For details on how to boot Xen and Dom-0, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).
2. Enable Xen awareness by enabling OS aware debug for Xen symbol file. Symbol files are added to a process context to enable source level debugging. For details on how to enable Xen awareness, refer to Enabling Xen Awareness.
3. Debug Dom-0 kernel.
   a. Enable OS awareness on the Linux symbol file in the Debug view for Dom-0 VCPU context. For details on OS aware debug, refer to OS Aware Debugging.
   b. Suspend the **Dom-0 VCPU#0** core. You can now insert breakpoints, step in, step out, watch variables, stack trace or perform other source level debugging tasks.



4. Debug the Dom-U Kernel:
   a. Copy the guest Linux images to Dom-0 file system.
   b. Create a Dom-U guest.
   c. Enable OS awareness on the Linux symbol file in the Debug view for Dom-U VCPU context. For details on OS aware debug, refer OS Aware Debugging.

d. Suspend the Dom-U VCPU#0 core. You can now insert breakpoints, step in, step out, watch variables, stack trace or perform other source level debugging tasks.

Debugging Dom-0/Dom-U User Space Processes

1. Boot Xen and Dom-0. For details on how to boot Xen and Dom-0, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).
2. Enable Xen awareness by enabling OS aware debug for Xen symbol file. Symbol files are added to a process context to enable source level debugging. For details on how to enable Xen awareness, refer to Enabling Xen Awareness.
3. Create a Linux application project. For details on how to create a Linux application project, refer Creating a Linux Application Project.
4. Configure the Dom-0 user space process by adding the symbol file of the application running on Linux for the debug context of the virtual CPU (VCPU#) of the host domain (Dom-0).
5. Configure Dom-U user space process.
   a. Copy the guest Linux images to Dom-0 file system.
   b. Create the Linux guests with para-virtual networking.

```
name = "guest 0"
kernel = "/boot/Image"
extra = "console=hvc0 rdinit=/sbin/init"
memory = 256
vcpus = 2
vif = [ 'bridge=xenbr0' ]
```

   c. Add the symbol file of the application running on Linux for the debug context of the virtual cpu (VCPU#) of the guest domain (Dom-U).
6. When the symbol files are set, you can insert breakpoints, step in, step out, watch variables, stack trace or perform other source level debugging tasks.

Debugging a Dom-U Standalone Application

1. Create a new standalone hypervisor guest application.
   a. Click **File → New → Application Project**. The New Application Project dialog box appears.

      Note: This is equivalent to clicking on **File → New → Project** to open the New Project wizard, selecting **Xilinx → Application Project**, and clicking **Next**.

   b. Type a project name into the Project Name field.
   c. Select the location for the project. You can use the default location as displayed in the Location field by leaving the **Use default location** check box selected. Otherwise, click the check box and type or browse to the directory location.
   d. The OS Platform allows you to select which operating system you will be writing code for. Select **standalone**.

Note: This selection alters what templates you view in the next screen and what supporting code is provided in your project.

e. In platform selection window, select **Create a New Platform from Hardware (XSA)**. Select **zcu102 design** and click **Next** to proceed.

f. Choose the preferred CPU and OS.

g. Click **Next** to advance to the Templates screen

h. From the Hypervisor Guest drop-down list, select **Yes** to create an application with a pre-defined linker script suitable to run the Xen.

i. Specify a board support package or domain. You can create a new customizable domain, or select an existing domain. The domain created by the wizard will have the `hypervisor_guest` parameter set to `true`. It will also ensure that the `stdin` and `stdout` are pointing to `psu_uart_1`.

j. Click **Next** to advance to the Templates screen.

k. The Vitis software platform provides useful sample applications listed in **Templates** dialog box that you can use to create your project. The **Description** box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.

l. Select the desired template. If you want to create a blank project, select the **Empty Application**. You can then add C files to the project, after the project is created.

m. Click **Finish** to create your application project and board support package (if it does not exist).

Note: Xilinx recommends that you use Managed Make flow rather than Standard Make C/C++ unless you are comfortable working with make files.

2. Build the newly created hypervisor guest standalone application to generate a `.bin` file. This file is needed to work with Xen.

3. Boot Xen and Dom-0. For details on how to boot Xen and Dom-0, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).

4. Enable Xen awareness by enabling OS aware debug for Xen symbol file. Symbol files are added to a process context to enable source level debugging. For details on how to enable Xen awareness, refer to Enabling Xen Awareness.

5. Copy the application to Dom-0 file system.

6. Create the guest domain `hello` using the Xen configuration file.

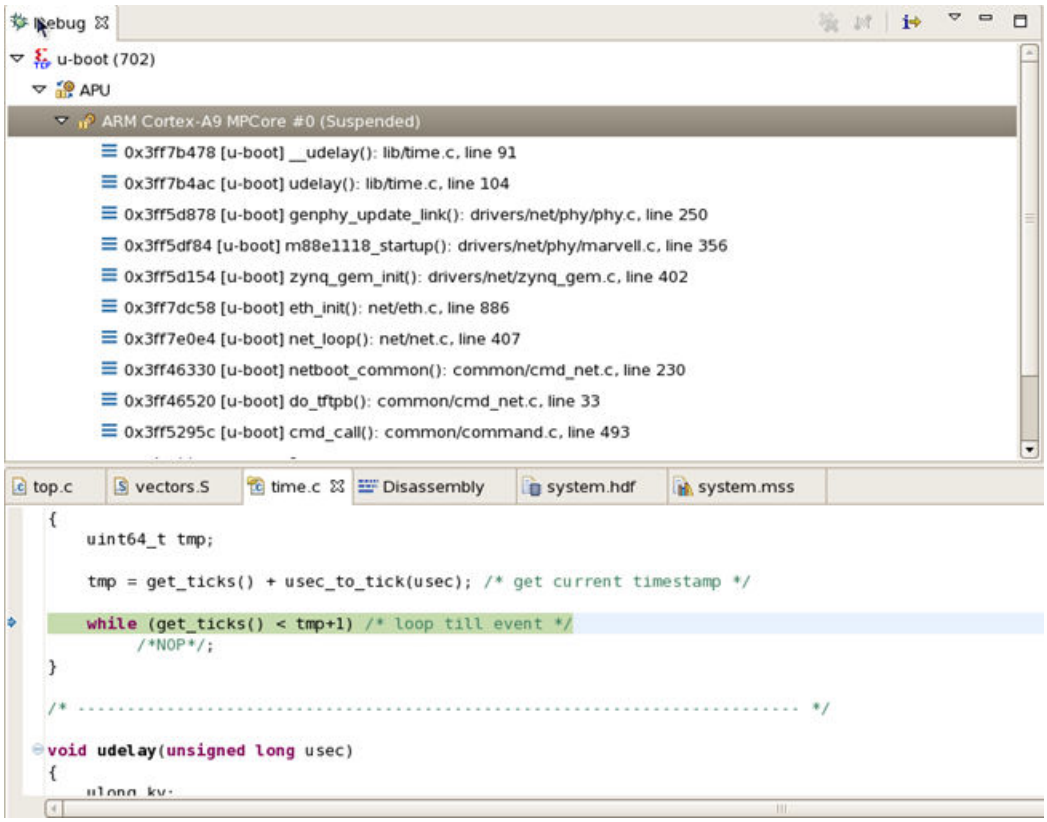
```
name = "hello"
kernel = "/boot/hello.bin"
memory = 8
vcpus = 1
cpus = [1]
irqs = [ 54 ]
iomem = [ "0xff010,1" ]
```

7. Suspend the **Dom-U VCPU#0** core. You can now insert breakpoints, step in, step out, watch variables, stack trace, or perform other source level debugging tasks.

Debugging Self-Relocating Programs

System debugger supports source level debugging of self-relocating programs such as U-boot. A self-relocating program is a program which relocates its own code and data sections during runtime. The debug information available in such files does not provide details about where the program sections have been relocated. For this reason, you must supply to the debugger the address where the program sections have been relocated. This can be done in two ways.

1. Update the system debugger launch configuration to provide the address to which program sections are relocated.
   a. Select **Debug As → Debug Configurations** to launch the system debugger launch configuration.
   b. Click the **Application** tab and select the application you wish to download.
   c. Select the **This is a self-relocating application** checkbox.
   d. Enter the address where all the program sections are to be relocated in the **Relative address to which the program sections are relocated** textbox.
   e. Launch the debug configuration. When the program sections are relocated during runtime, the debugger will have enough information to support source level debugging of the relocated sections.



Note: This method is supported only when the **Debug Type** is set to **Standalone**' in the **Target Setup** tab of the debug configuration.

2. Alternatively, you can also use the `memmap` command in XSDB to provide the address where the program sections are relocated. `memmap` command in XSDB can be used to add symbol files to the debugger. This is useful for debugging the applications which are already running on the target. For example, boot from flash. In case of relocatable ELF files, you can use the `-relocate-section-map` option, to provide the relocation address.

```
xsdb% targets 2
 1 APU
   2 ARM Cortex-A9 MPCore #0 (Suspended)
   3 ARM Cortex-A9 MPCore #1 (Suspended)
 4 xc7z020
xsdb% targets 2
xsdb% memmap -reloc 0x3bf37000 -file u-boot

xsdb% stop
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x3ff7b478
(Suspended)
xsdb% bt
 0 0x3ff7b478 __udelay()+1005809800: lib/time.c, line 91
 1 0x3ff7b4ac udelay()+1005809696: lib/time.c, line 104
 2 0x3ff5d878 genphy_update_link()+1005809860: drivers/net/phy/phy.c,
line 250
 3 0x3ff5df84 m88e1118_startup()+1005809712: drivers/net/phy/marvell.c,
line 356
 4 0x3ff5d154 zynq_gem_init()+1005810192: drivers/net/zynq_gem.c, line
402
 5 0x3ff7dc58 eth_init()+1005809720: net/eth.c, line 886
```

```
 6 0x3ff7e0e4 net_loop()+1005809728: net/net.c, line 407
 7 0x3ff46330 netboot_common()+1005809972: common/cmd_net.c, line 230
 8 0x3ff46520 do_tftpb()+1005809708: common/cmd_net.c, line 33
 9 0x3ff5295c cmd_process()+1005809824: common/command.c, line 493
 10 0x3ff5295c cmd_process()+1005809824: common/command.c, line 493
 11 0x3ff3b710 run_list_real()+1005811444: common/cli_hush.c, line 1656
 12 0x3ff3b710 run_list_real()+1005811444: common/cli_hush.c, line 1656
 13 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 14 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 15 0x3ff3b008 parse_string_outer()+1005809872: common/cli_hush.c, line
3254
 16 0x3ff3b6b8 run_list_real()+1005811356: common/cli_hush.c, line 1617
 17 0x3ff3b6b8 run_list_real()+1005811356: common/cli_hush.c, line 1617
 18 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 19 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 20 0x3ff3afd0 parse_string_outer()+1005809816: common/cli_hush.c, line
3248
 21 0x3ff5140c do_run()+1005809740: common/cli.c, line 131
 22 0x3ff5295c cmd_process()+1005809824: common/command.c, line 493
 23 0x3ff5295c cmd_process()+1005809824: common/command.c, line 493
 24 0x3ff3b710 run_list_real()+1005811444: common/cli_hush.c, line 1656
 25 0x3ff3b710 run_list_real()+1005811444: common/cli_hush.c, line 1656
 26 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 27 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 28 0x3ff3b008 parse_string_outer()+1005809872: common/cli_hush.c, line
3254
 29 0x3ff3b6b8 run_list_real()+1005811356: common/cli_hush.c, line 1617
 30 0x3ff3b6b8 run_list_real()+1005811356: common/cli_hush.c, line 1617
 31 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 32 0x3ff3be3c parse_stream_outer()+1005811244: common/cli_hush.c, line
2003
 33 0x3ff3afd0 parse_string_outer()+1005809816: common/cli_hush.c, line
3248
 34 0x3ff39ab4 main_loop()+1005809724: common/main.c, line 85
 35 0x3ff3c4f4 run_main_loop()+1005809672: common/board_r.c, line 675
 36 0x3ff73b54 initcall_run_list()+1005809716: lib/initcall.c, line 27
 37 0x3ff3c66c board_init_r()+1005809676: common/board_r.c, line 908
 38 0x3ff3837c clbss_l()+1005809688: arch/arm/lib/crt0.S, line 174
 39 unknown-pc
```

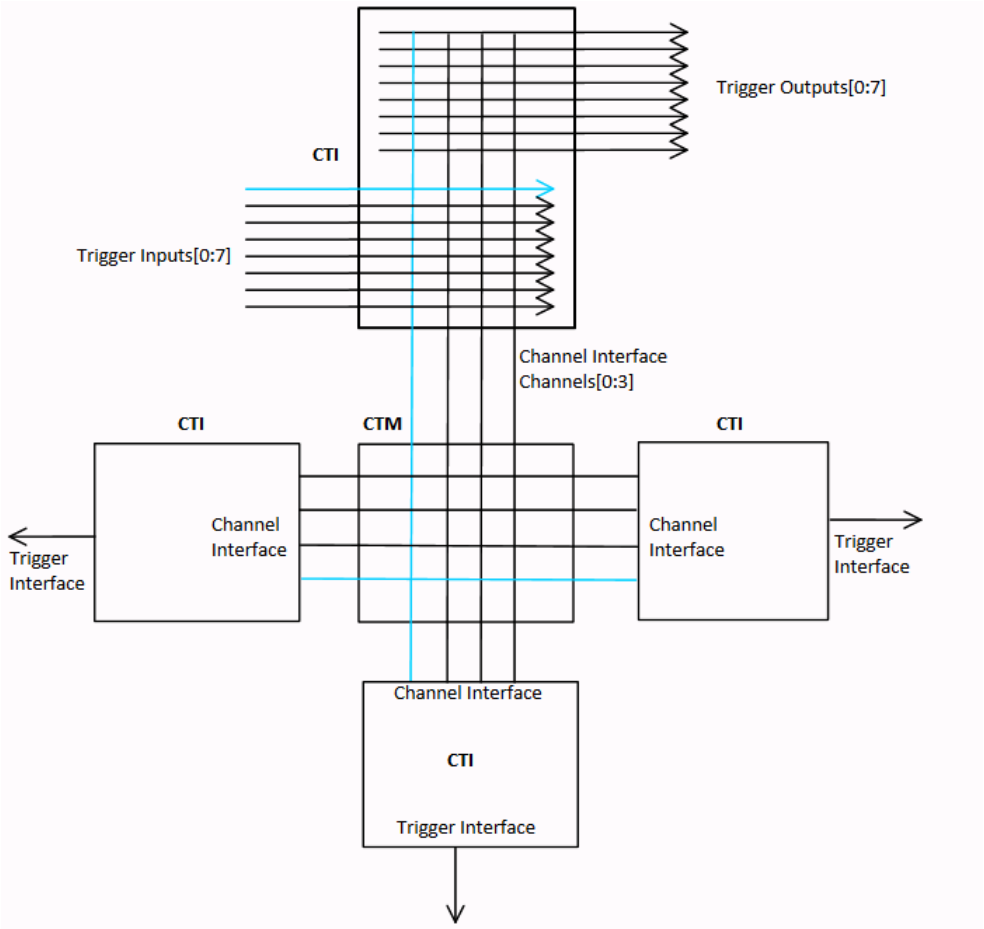
Cross-Triggering

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 3: **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 now create/edit/remove cross-trigger breakpoints and apply the breakpoints on the target using the **Debug Configurations** dialog box. To enable cross-triggering, do the following:

1. Launch the Vitis software platform.
2. Create a standalone application project. Alternatively, you can also select an existing project.
3. Right-click on the application and select **Debug As → Debug Configuration**.
4. Double-click **Launch on Hardware (Single Application Debug)** to create a new configuration.
5. On the **Target Setup** tab page, select **Enable Cross-Triggering**.
6. Click the button next to the **Enable Cross-Triggering** check box. The **Cross Trigger Breakpoints** dialog box appears.

   You can create new breakpoints and edit or remove existing breakpoints using the **Cross Trigger Breakpoints** dialog box. The options available on the dialog box are described below.

   - **Create:** Click to create a new cross trigger breakpoint. The New Cross Trigger Breakpoint dialog box appears. You need to select a cross trigger signal, which can be a source or destination of a cross-triggering breakpoint. The **OK** button enables only when you select at least one input and one output signal.
   - **Edit:** Click to edit an existing breakpoint. The Edit Cross Trigger Breakpoint dialog box appears that allows you to edit the selected input and output signals.
   - **Remove:** Click to remove the selected breakpoint.

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 8: **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 9: **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 9: **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 9: **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]

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 it's 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
```

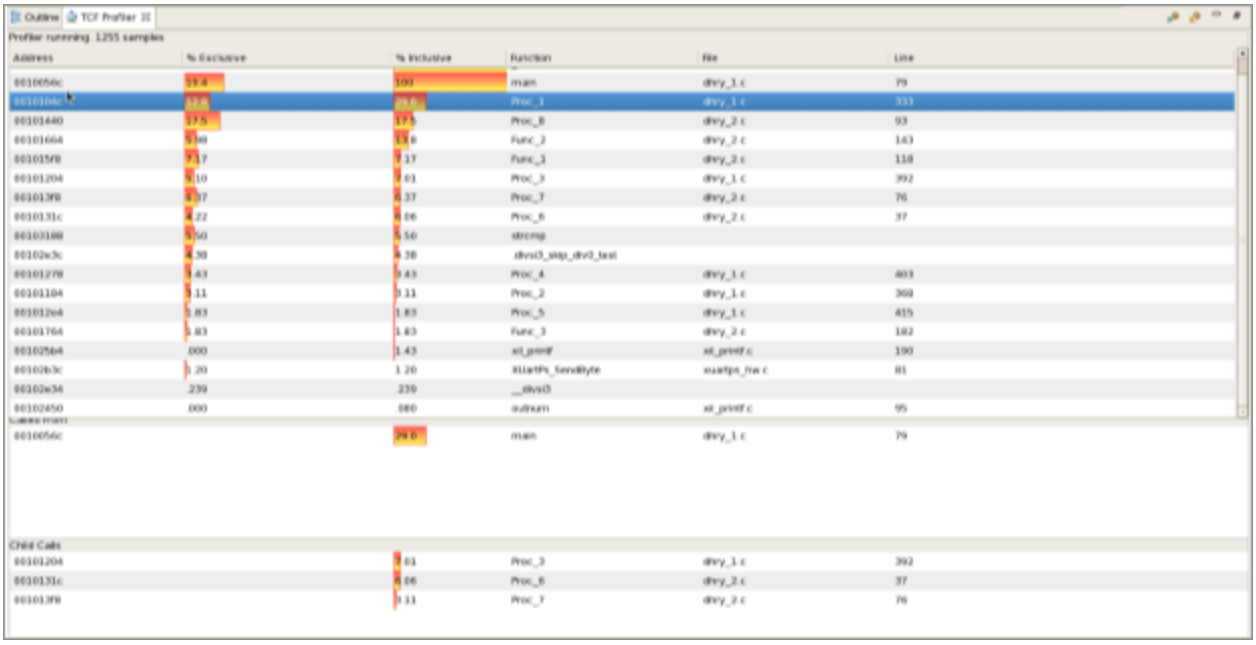
Profile/Analyze

TCF Profiling

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 doesn't 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. Select the application you want to profile.
2. Right-click the application and select **Run As → Launch on Hardware (Single Application Debug)**.
3. When the application stops at main, open the TCF profiler view by selecting **Window → Show View → Debug → TCF Profiler**.
4. Click the button to start profiling. The Profiler Configuration dialog box appears.
5. Select the **Aggregate Per Function** option, to group all the samples collected for different addresses in a single function together. When the option is disabled, the samples collected are shown as per the address.
6. Select the **Enable stack tracing** option, to show the stack trace for each address in the sample data. To view the stack trace for an address, click on that address entry in the profiler view.
7. Specify the **Max stack frames count** for the maximum number of frames that are shown in the stack trace view.
8. Specify the **View update interval** for the time interval (in milliseconds) the TCF profiler view is updated with the new results. Please note that this is different from the interval at which the profile samples are collected.

9. Resume your application. The profiler view will be updated with the data as shown the figure below.



gprof Profiling (Zynq-7000 SoC)

IMPORTANT! *This feature applies only to Zynq-7000 SoC devices.*

This feature applies only to Zynq-7000 SoC devices. GNU gprof provides two kinds of information that you can use to optimize the program:

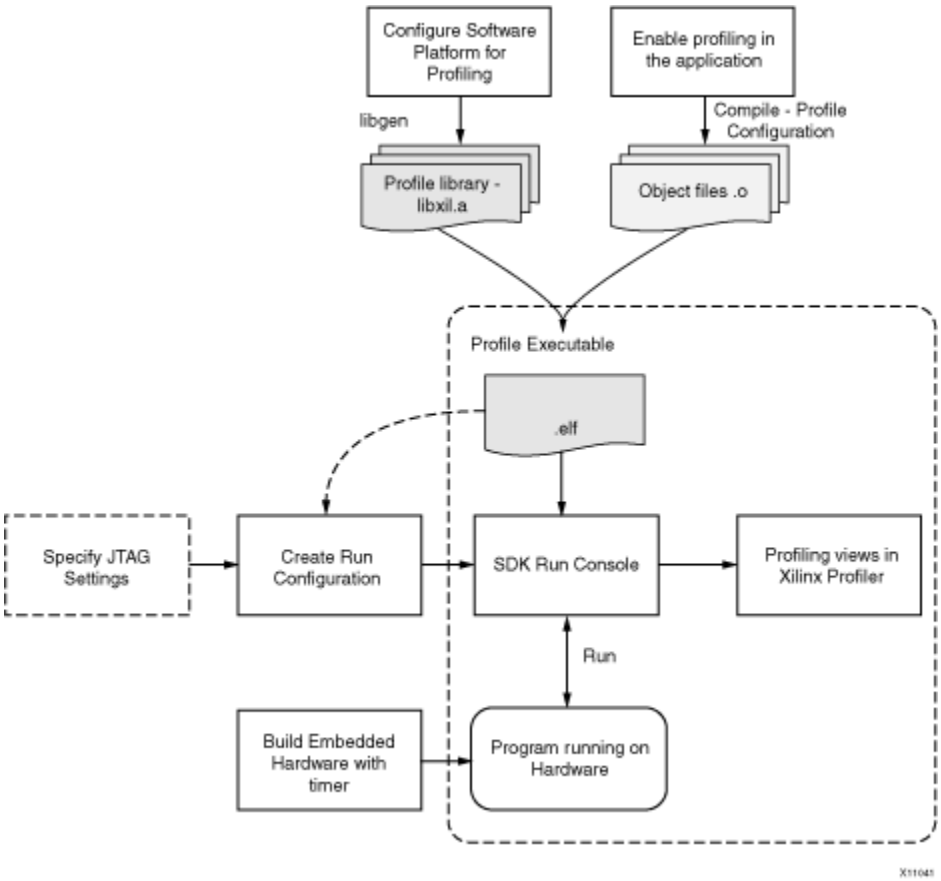
- A histogram with which you can identify the functions in the program that take up the most execution time
- A call graph that shows what functions called which other functions, and how many times

The execution flow of the program is altered so that gprof can obtain data. Consequently, this method of profiling is considered software-intrusive. The program flow is altered in two ways:

- To obtain histogram data, the program is periodically interrupted to obtain a sample of its program counter location. This user-defined interval is usually measured in milliseconds. The program counter location helps identify which function was being executed at that particular sample. Taking multiple samples over a long interval of a few seconds helps identify which functions execute for the longest time in the program.
- To obtain the call graph information, the compiler annotates every function call to store the caller and callee information in a data structure.

The profiling workflow is described in the following diagram:

Figure 4: **Profiling Workflow**



***Note*:** Xilinx recommends not to use garbage collector flags when you run profiling. Using garbage collector flags can cause errors.

For additional information about GNU gprof, refer to http://sourceware.org/binutils/docs-2.18/gprof/index.html.

Specifying Profiler Configuration

To configure options for the Profiler, do the following:

1. In the Project Explorer or C/C++ Projects view, select a project.
2. Select **Run → Run Configuration**.
3. In the Run Configurations dialog box, expand **Launch on Hardware (Single Application Debug)**.
4. Create a run configuration.
5. Click the **Application** tab.
6. Click the **Edit** button to view and configure the **Advanced Options**.

7. In the Profile Options area, select the **Enable Profiling** check box.
8. Specify the sampling frequency and the scratch memory to use for profiling, where:
   a. The sampling frequency is the interrupt interval that the profiling routine uses to periodically check which function is currently being executed. The routine performs the sampling by examining the program counter at each interrupt.
   b. The scratch memory address is the location in DDR3 memory that the domain profiling services use for data collection. The application program should never touch this space.
9. Click **Run** to profile the application.

Setting Up the Hardware for Profiling

To profile a software application, you must ensure that interrupts are raised periodically to sample the program counter (PC) value. To do this, you must program a timer and use the timer interrupt handler to collect and store the PC. The profile interrupt handler requires full access to the timer, so a separate timer that is not used by the application itself must be available in the system.

Xilinx profiling libraries that provide the profile interrupt handler support the AXI Timer core.

When profiling on Zynq-7000 SoC processors, the internal SCU timer should be used.

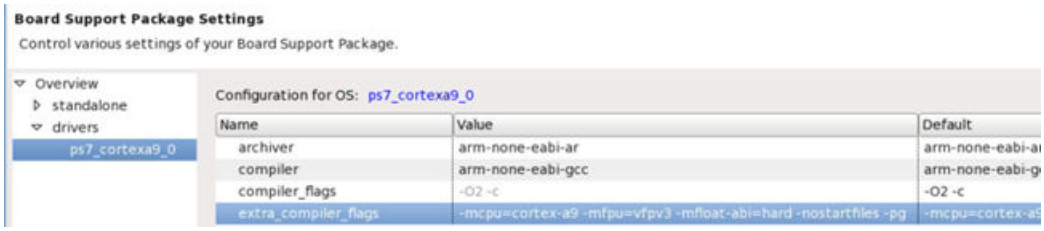
Setting Up the Software for Profiling

There are three important steps involved in setting up the software application for profiling:

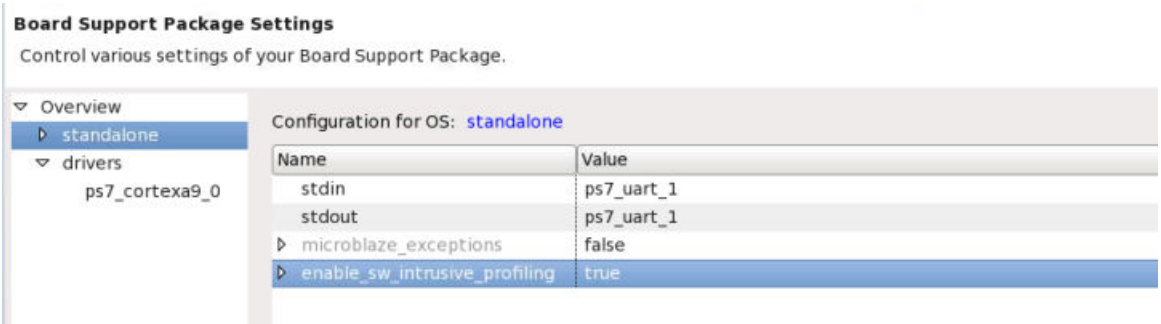
1. Enable profiling in the Software Platform to include profiling libraries.

   Note: Profiling is supported only for standalone software platforms.

   a. Add `-pg` to the extra compiler flags to build the domain with profiling.



   b. Set `enable_sw_intrusive_profiling` to true in the Board Support Package Settings window.



2. Enable profiling in application C/C++ build settings from **C/C++ Build → Settings → Profiling**

Setting up the Domain

1. Double-click **Application.prj**. This opens the Application Overview page. Click **Navigate to BSP Settings**. Click **Modify BSP Settings**.
2. Click on the OS name, such as **standalone**, to configure its parameters.
3. Set the **enable_sw_intrusive_profiling** field to **true** and select the timer for use by the profile libraries.
4. The domain should be compiled with the -pg compiler option. To perform this step, click on the drivers item and select the CPU driver. Add the **-pg** flag to the **extra_compiler_flags** option.
5. Click **OK**.

Setting Up the Software Application

1. Modify the software application code to enable interrupts. If there is an interrupt controller present in the system with multiple interrupt sources, you must enable interrupts in the processor and the interrupt controller to allow interrupts from the profile timer to reach the processor. Example code is shown below:

```
/* enable interrupt controller */
        XIntc_mMasterEnable(SYSINTC_BASEADDR);
        /* service all interrupts */
        XIntc_SetIntrSvcOption(SYSINTC_BASEADDR,
XIN_SVC_ALL_ISRS_OPTION);
        /* enable the profile timer interrupt */
        XIntc_mEnableIntr(SYSINTC_BASEADDR, PROFILE_TIMER_INTR_MASK);
        /* enable interrupts in the processor */
        microblaze_enable_interrupts();
```

2. If the profiling timer is the only entity that connects to the input of interrupt controller or directly to the processor, the tool sets up the interrupt for you automatically, and no change is required in the application code.
3. Right-click the software application and select **C/C++ Settings** (or **Properties → C/C++ Build → Settings**).

4. Select **gcc compiler → Profiling** and enable profiling by selecting **Enable Profiling (-pg)**.
5. Click **OK**.

Viewing the Profiling Results

When the program completes execution (reaches exit), or when you click the Stop button to stop the program, the Vitis software platform downloads the profile data and stores it in a file named `gmon.out`.

The Vitis software platform automatically opens the `gmon.out` file for viewing. The `gmon.out` file is generated in the `debug` folder of the application project.

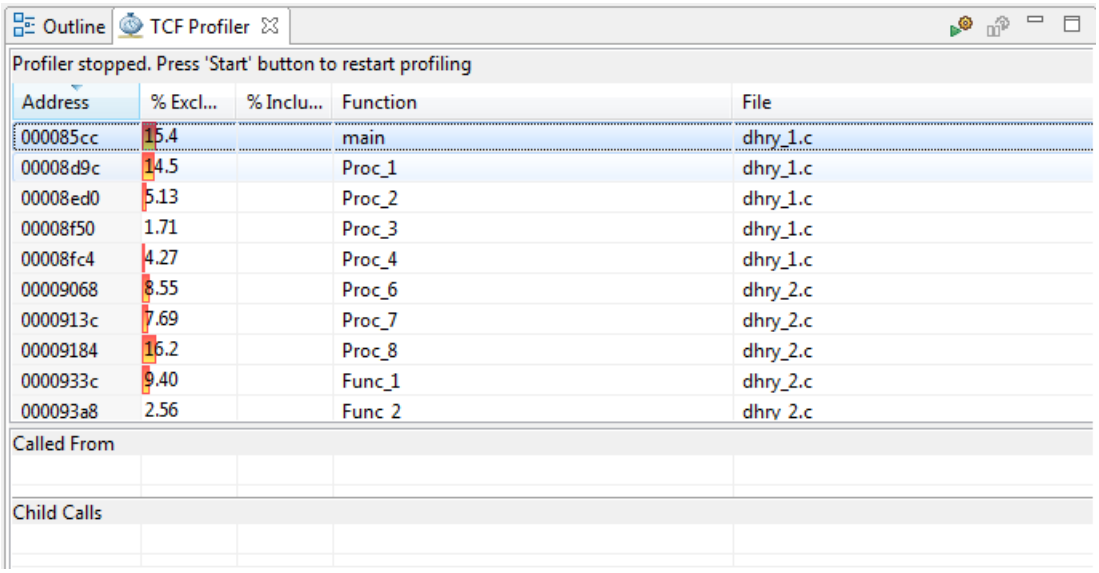
Profiling Linux Applications with System Debugger

To profile Linux applications using Xilinx System Debugger, perform the following:

1. Create an new Linux application for the target, using the Vitis IDE.

   Note: The instructions have been developed based on Cortex-A9 on ZC702 but should be valid for other targets as well.
2. Import your application sources in to the new project.
3. Build the application.
4. Boot Linux on ZC702 (for example, from the SD card) and start the TCF agent on the target.
5. Create a new target connection for the TCF agent, from the Target Connections icon.
6. Create a new **Xilinx System Debugger** debug configuration for the application, you wish to profile, and launch the debug configuration. Create a new **Launch on Hardware (Single Application Debug)**.
7. On the Main tab, select **Linux Application Debug** from the Debug Type list.
8. On the Application tab page, specify the local `.elf` file path and the remote `.elf` file path.
9. Click **Debug**.
10. When the process context stops at main(), launch the TCF Profiler view by selecting **Window → Show View → Debug → TCF Profiler**.
11. In the TCF Profiler view, click the **Start** toolbar icon to start profiling.

    Note: Set a breakpoint at the end of your application code, so that the process is not terminated. If not set, the data collected by the TCF Profiler is lost when the process terminates.
12. Resume the process context. TCF Profiler view will be updated with the profile date.



Non-Intrusive Profiling for MicroBlaze Processors

When extended debug is enabled in the hardware design, MicroBlaze supports non-intrusive profiling of the program instructions. You can configure whether the instruction count or the cycle count should be profiled. The profiling results are stored in a profiling buffer in the debug memory, which can be accessed by the debugger thru MDM debug registers. The size of the buffer can be configured from 4K to 128K, using the `C_DEBUG_PROFILE_SIZE` (a size of 0 indicates profiling is disabled) parameter.

The profile buffer is divided into number of portions known as bins. Each bin is 36 bit wide and can count the instructions or cycles of a program address range. The address range that is profiled by each bin is dependent on the total size of the program that is profiled. Bin size is calculated using the formula:

```
B = log2((H - L + S * 4) / S * 4)
```

Where B is the bin size, H, L are high and low address of the program address range being profiled, and S is the size of the profile buffer.

When profiling is enabled and program starts running, profile statistics for an address range are stored in its corresponding bin. Xilinx System Debugger can read these results, when needed.

Specifying Non-Intrusive Profiler Configuration

To configure options for the Profiler, do the following:

1. Launch the Vitis software platform.
2. Create a new standalone application project or select an existing one.

3. Select **Run → Run Configuration**.
4. In the Run Configurations dialog box, expand **Launch on Hardware (Single Application Debug)**.
5. Create a run configuration.
6. Click the **Application** tab.
7. Click the **Edit** button to view and configure the Advanced Options.
8. In the Profile Options area, select the **Enable Profiling** check box.
9. Select **Non-Intrusive**.
10. Specify the low address and the high address of the program range to be profiled. Alternatively, select the **Program Start** or the **Program End** check box to auto-calculate the low or high address from the program.
11. **Count Instructions** to count the number of instructions executed. Alternatively, select **Count Cycles** to count the number of cycles elapsed.
12. Select **Cumulative Profiling** to profile without clearing the profiling buffers from the last execution.
13. Click **OK** to save the configurations.
14. Click **Run** to profile the selected project.

Viewing the Non-Intrusive Profiling Results

When the application completes execution, or when you click the Stop button to stop the program, the Vitis software platform downloads the non-intrusive profile data and stores it in a file named `gmon.out`.

Note: The Vitis software platform automatically opens the `gmon.out` file for viewing. The `gmon.out` file is generated in the `debug` folder of the application project.

FreeRTOS Analysis using STM

The Vitis software platfrom supports collection and analysis of trace events generated by FreeRTOS based applications. Zynq UltraScale+ MPSoC processors support the Software Trace Microcell (STM) block which is a software application driven trace source to generate a software instrumentation trace (SWIT). To collect FreeRTOS events and analyze them, do the following:

1. Click **File → New → Application Project**. The New Application Project dialog box appears.
2. Type a project name into the Project Name field.
3. Select the location for the project. You can use the default location as displayed in the Location field by leaving the Use default location check box selected. Otherwise, click the check box and type or browse to the directory location.

4. In platform selection window, select **Create a new platform from hardware (XSA)** and choose **zcu102 design**. Click **Next** to proceed.
5. Choose CPU and OS as **freertos10_xilinx**.

   Note: The FreeRTOS version may vary in upcoming releases.
6. Click **Next** to advance to the Templates screen.

   The Vitis software platform provides useful sample applications listed in Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.
7. Select the desired template. If you want to create a blank project, select **Empty Application**. You can add C files to the project after the project is created.
8. Click the **Navigate to BSP** settings option in application project settings page. Select **Open BSP Settings → Overview → FreeRTOS** and change the value of **enable_stm_event_trace** to TRUE.
9. Right-click on the application and select **Debug As → Debug Configuration**.
10. In the Debug Configurations dialog box, double-click **Single Application Debug** to create a launch configuration for the selected project.
11. Click **Debug**. Debugging begins with the processors in the running state.
12. Debug the project using the system debugger on the required target.
13. Wait for project to be downloaded on to board and stop at main().
14. Click **Window → Show View → Xilinx**. The Show View dialog box appears.
15. Select **Trace Session Manager** from the Show View dialog box. The launch configuration related to the application being debugged can be seen in the Trace Session Manager view.
16. Click the start button in the Trace Session Manager view toolbar to start the FreeRTOS trace collection.
17. Switch to the Debug view and resume the project.
18. Allow the project to run.
19. Switch back to the Trace Session Manager view and stop the trace collection. All the trace data collected will be exported to suitable trace file and will be opened in Events editor and the FreeRTOS Analysis view.

Optimize

Performance Analysis

Performance analysis in the Vitis software platform provides functionality for viewing and analyzing different types of performance data. Its goal is to provide views, graphs, metrics, etc. to help extract useful information from the data, in a way that is more user-friendly and informative than huge text dumps.

Performance analysis provides the following features:

- Support for viewing Arm data.
- Support for viewing APM data with PS and MDM as master.
- Support for viewing MicroBlaze data.
- Support for viewing and analyzing live data.
- Support for offline viewing of data.
- Support for zooming out/in of the data.
- Event filtering and searching.
- Import and export of trace packages.

The Performance analysis feature in the Vitis software platform supports data collection from AXI Performance Monitor (APM) Event Counters, Arm Performance Monitor Unit (PMU) from a Zynq-7000 SoC processing system, and MicroBlaze performance monitoring counters. For an example usage of performance monitoring on a Zynq device, refer to System Performance Modeling. For a MicroBlaze design, APM can be used in a similar way as SPM.

To collect MicroBlaze performance data, the performance monitoring counters must be enabled in the Vivado hardware design. For more information, refer MicroBlaze Processor Reference Guide (UG984). The Vitis software platform monitors the following events for MicroBlaze processors:

- Number of clock cycles
- Any valid instruction executed
- Read or write data request from/to data cache
- Read or write data cache hit
- Pipeline stalled
- Instruction cache latency for memory read

The data is collected in the Vitis software platform in real time. The values from these counters are sampled every 10 ms. These values are used to calculate metrics shown in the Performance Counters view.

The Vitis software platform monitors the following PMU events for each Cortex-A9 CPU:

- Data cache refill
- Data cache access
- Data stall
- Write stall
- Instruction rename
- Branch miss

The following two Level-2 cache controller (L2C-PL330) counters are monitored:

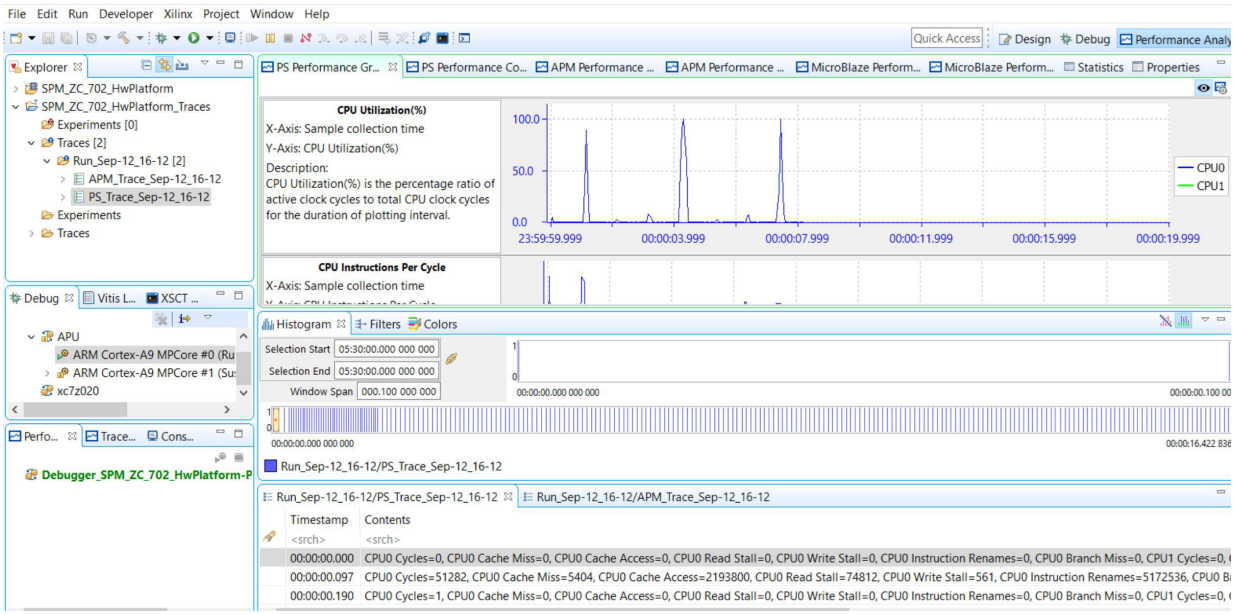
- Number of cache hits
- Number of cache accesses

The following APM counters for each HP and ACP port are monitored:

- Write Byte Count
- Read Byte Count
- Write Transaction Count
- Total Write Latency
- Read Transaction Count
- Total Read Latency

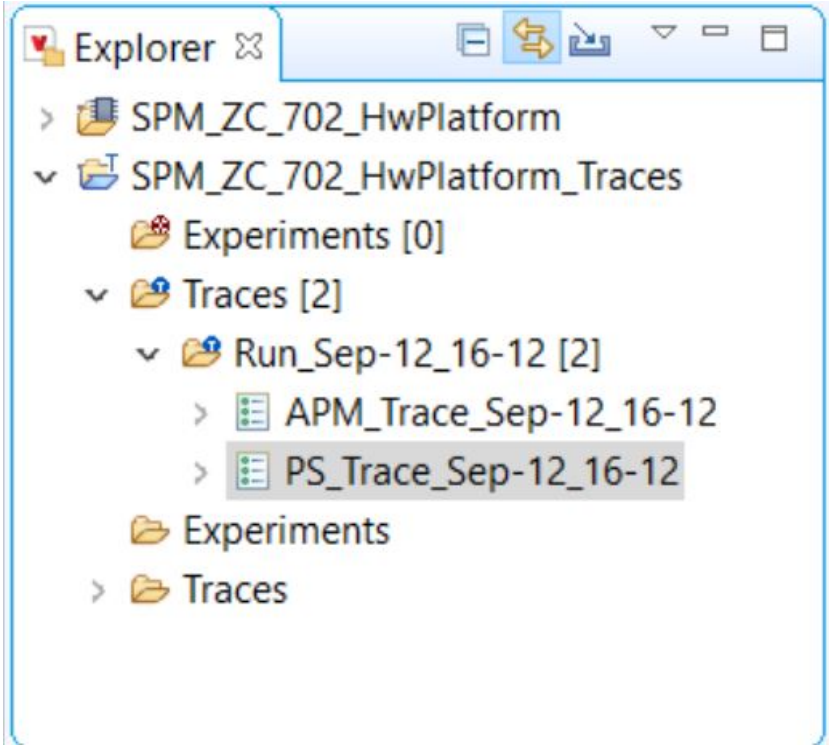
Working with the Performance Analysis Perspective

The Performance Analysis perspective is comprised of many views which provide the capability of collecting and analyzing the performance data referred as trace.

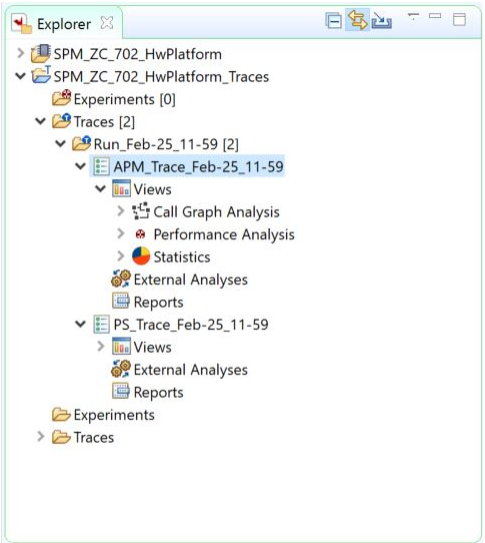


Explorer View

The Project Explorer view displays all the available projects in the workspace. When a performance analysis session is launched the data from the board is collected and stored as trace files in tracing project. Each of the hardware project contains a corresponding tracing project,*_Traces, where the data is stored. Performance counters data from single run is stored under designated Run_* folder. Data from different sections is stored in different files under the run folder.



To analyse the data double click the trace file to open it in an Events editor view. After the file is opened, the tree under the trace file can be expanded to view the list of available analysis views.



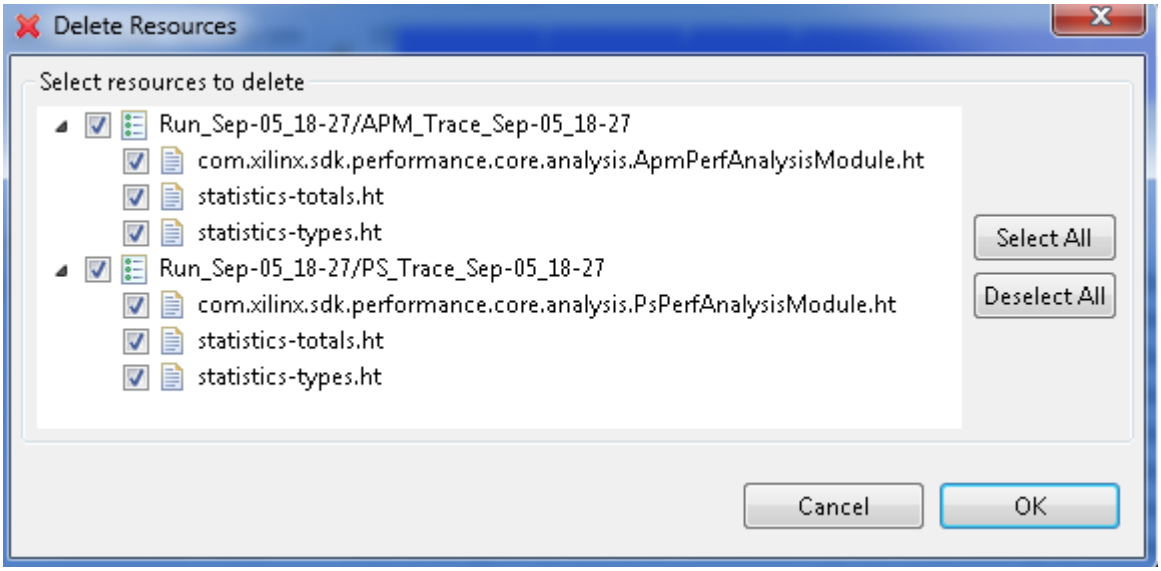
Deleting Supplementary Files

Supplementary files are by definition trace specific files that accompany a trace. These files could be temporary files, persistent indexes, or any other persistent data files created by the tool during parsing a trace.

All supplementary files are hidden from the user and are handled internally by the tool. However, there is a possibility to delete the supplementary files so that there are recreated when opening a trace.

To delete all supplementary files from one or many traces and experiments:

1. Select the relevant traces and experiments in the **Project Explorer** view.
2. Right-click and select **Delete Supplementary Files** from the context menu that appears. The Delete Resources dialog box, with a list of supplementary files, grouped under the trace or experiment they belong to, appears.



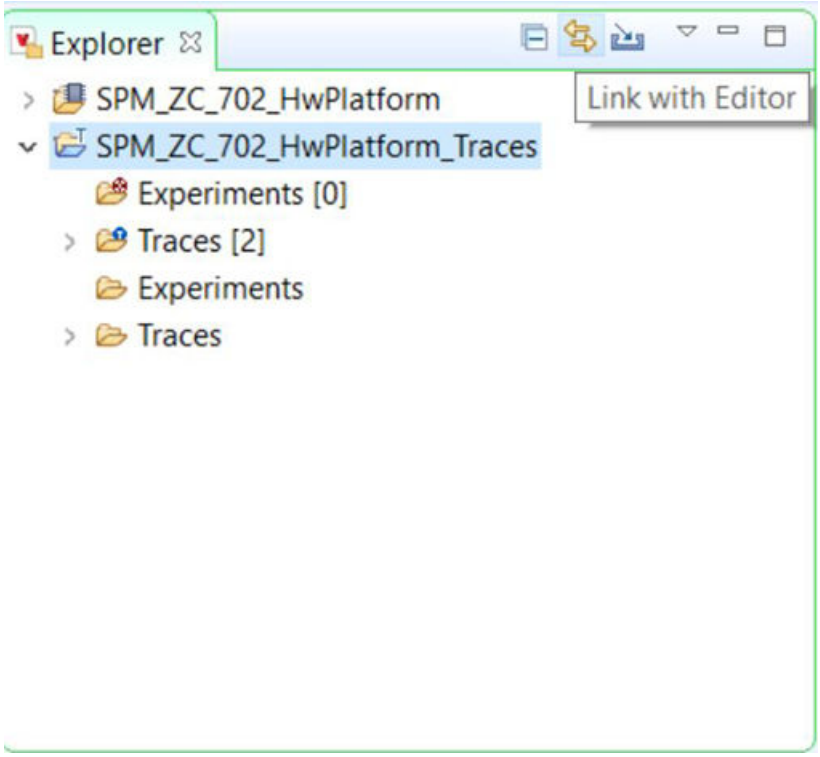
3. Select the file(s) to delete from the list.
4. Click **OK**.

Link with Editor

The tracing projects support the Link With Editor feature of the Project Explorer view. With this feature it is now possible to do the following:

- Select a trace element in the **Project Explorer** view and the corresponding **Events** editor will get focus, if the relevant trace is open.
- Select an **Events** editor and the corresponding trace element will be highlighted in the **Project Explorer** view.

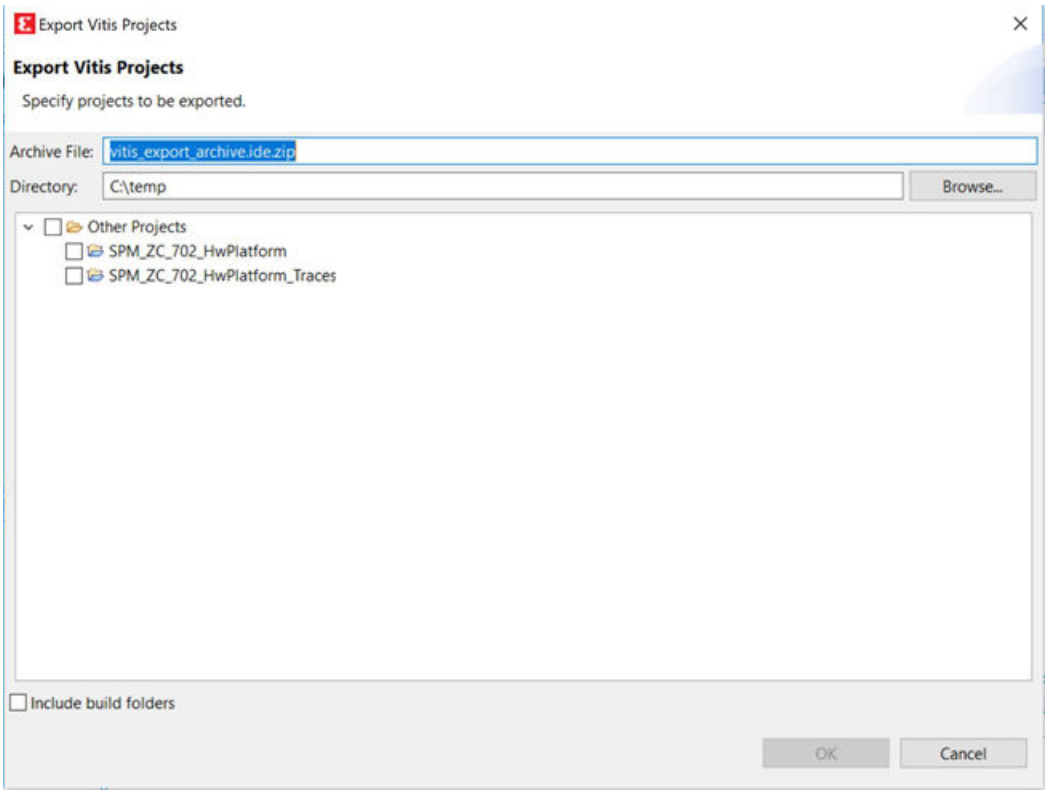
To enable or disable this feature toggle the **Link With Editor** button of the Project Explorer view as shown below.



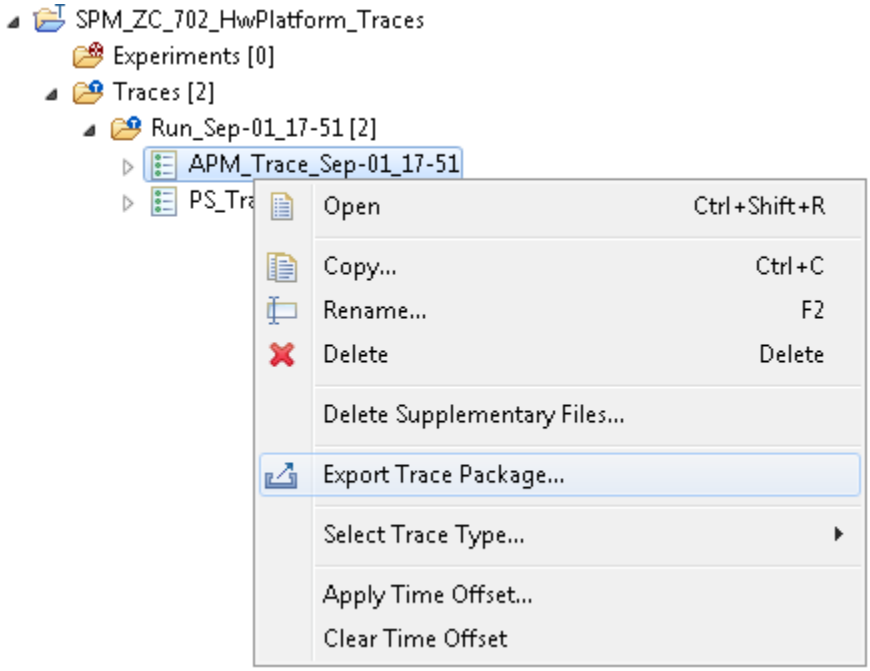
Exporting a Trace Package

The Export Trace Package wizard allows users to select a trace and export its files and bookmarks to an archive on a media. The `Traces` folder holds the set of traces available for a tracing project. To export traces contained in the `Traces` folder:

1. Select **File → Export**. **Export** dialog box appears.
2. Expand **Tracing** and select **Trace Package Export**.
3. Click **Next**. The Export trace package dialog box appears.



4. Select the project containing the traces and then the traces to be exported.
5. You can also open the Export trace package wizard by expanding the project in the Project Explorer view, selecting the traces under the Traces folder, and selecting the **Export Trace Package** from the context menu that appears.



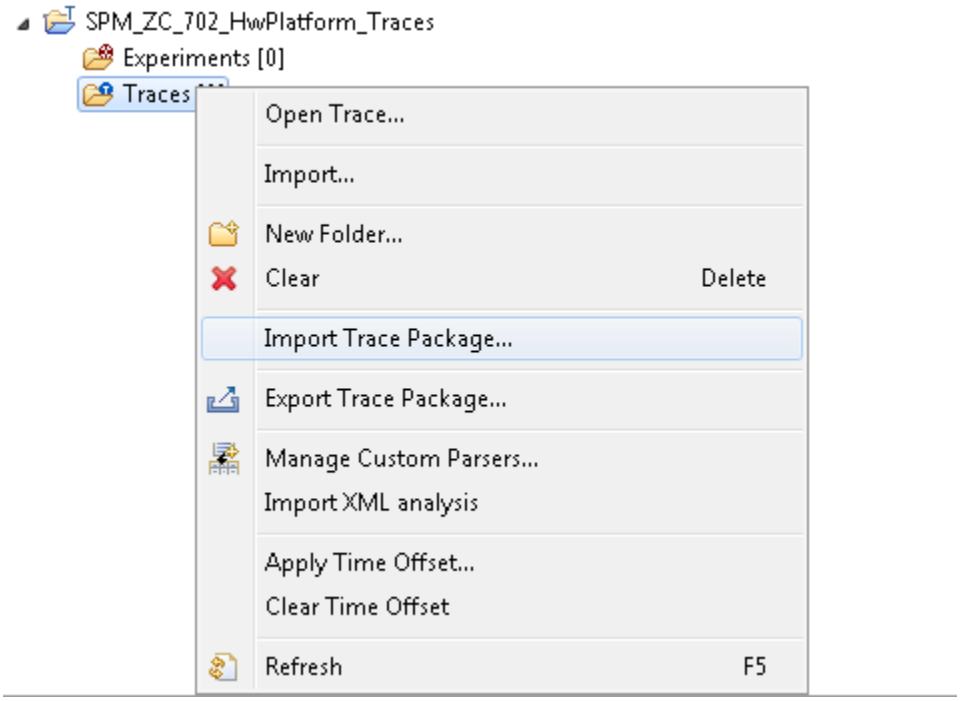
6. You can now select the content to export and various format options for the resulting file.
7. Click **Finish** to generate the package and save it to the media. The folder structure of the selected traces relative to the Traces folder is preserved in the trace package.

Importing a Trace Package

The Import Trace Package wizard allows users select a previously exported trace package from their media and import the content of the package in the workspace.

The `Traces` folder holds the set of traces available for a tracing project. To import a trace package to the `Traces` folder of a project:

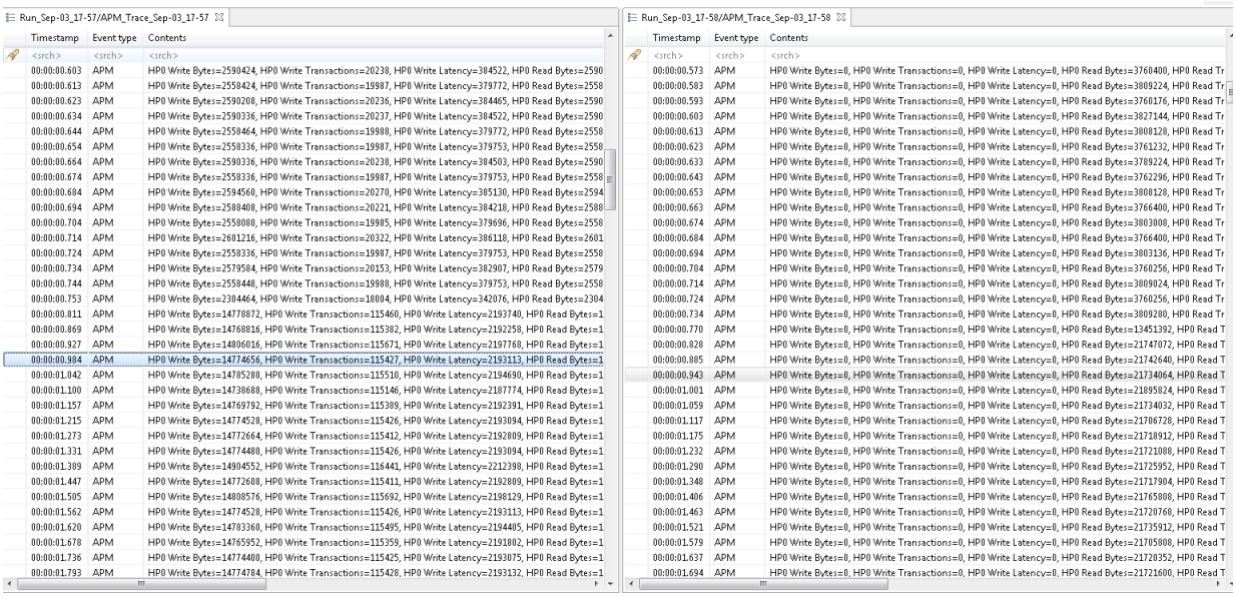
1. Select **File → Import** from the **File** main menu. The Import dialog box appears.
2. Expand **Tracing** and select **Trace Package Import**.
3. Click **Next**. The **Import trace package** dialog box appears.
4. Select the archive containing the traces and the destination project.
5. You can also open the Import Trace Package wizard by expanding the project in the Project Explorer view and selecting the **Import Trace Package** from the context menu that appears.



6. You can now select the content to import from the selected trace archive.
7. Click **Finish** to import the trace to the target folder. The folder structure from the trace package is restored in the `Traces` folder of the project.

Events Editor

The Events editor shows the basic trace data elements (events) in a tabular format. The editors can be dragged in the editor area so that several traces may be shown side by side, as shown in the following figure.



The header displays the current trace name. The page displays the following fields.

- **Timestamp:** The event timestamp.
- **Type:** The event type (PS/ APM/MicroBlaze).
- **Content:** The raw event content obtained from the hardware server.

The first row of the table is the header row. You can search and filter the information on the page, using this row.

The highlighted event is the current event, and is synchronized with the other views. If you select another event, the other views will be updated accordingly. The properties view will display a more detailed view of the selected event.

An event range can be selected by holding the **Shift** key while clicking another event or using any of the cursor keys (**Up**', **Down**, **PageUp**, **PageDown**, **Home**, and **End**). The first and last events in the selection will be used to determine the current selected time range for synchronization with the other views.

The Events editor can be closed, disposing a trace. When this is done, all the views displaying the information will be updated with the trace data of the next event editor tab. If all the editor tabs are closed, then the views will display their empty states.

Searching and Filtering Events

Searching and filtering of events in the table can be performed by entering matching conditions in one or multiple columns in the header row (the first row below the column header).

To toggle between searching and filtering, click on the **Search** or **Filter** icon in the left margin of the header row, or right-click on the header row and select **Show Filter Bar** or **Show Search Bar** in the context menu.

To apply a matching condition to a specific column, click on the column's header row cell, type in a regular expression and press the **Enter** key. You can also enter a simple text string and it will be automatically be replaced with a 'contains' regular expression.

When matching conditions are applied to two or more columns, all conditions must be met for the event to match (for example, 'and' behavior).

To clear all matching conditions in the header row, press the **Delete** key.

Searching an Event

When a searching condition is applied to the header row, the table selects the next matching event starting from the top currently displayed event. Wrapping occurs if there is no match until the end of the trace.

All matching events have a Search match icon in their left margin. Non-matching events are dimmed.

	Timestamp	Event type	Contents
	<srch>	<srch>	.*CPU0 Cycles=60.*
	00:00:04.049	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0
	00:00:04.107	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0
	00:00:04.165	PS	CPU0 Cycles=207866, CPU0 Cache Miss=88586, CPU0 Cache Access=1372742, CPU0 Read Stall=7644213, CPU0 Write Stall=57, CPU0 Instruction Renames=4813763, CPU0
	00:00:04.223	PS	CPU0 Cycles=601590, CPU0 Cache Miss=273693, CPU0 Cache Access=4236328, CPU0 Read Stall=21016365, CPU0 Write Stall=0, CPU0 Instruction Renames=14823508, CP
	00:00:04.280	PS	CPU0 Cycles=601554, CPU0 Cache Miss=267841, CPU0 Cache Access=4145496, CPU0 Read Stall=21392519, CPU0 Write Stall=0, CPU0 Instruction Renames=14512612, CP
	00:00:04.338	PS	CPU0 Cycles=601609, CPU0 Cache Miss=255995, CPU0 Cache Access=3960952, CPU0 Read Stall=22161892, CPU0 Write Stall=0, CPU0 Instruction Renames=13861652, CP
	00:00:04.396	PS	CPU0 Cycles=601556, CPU0 Cache Miss=271379, CPU0 Cache Access=4198920, CPU0 Read Stall=21171012, CPU0 Write Stall=0, CPU0 Instruction Renames=14696256, CP
	00:00:04.454	PS	CPU0 Cycles=207801, CPU0 Cache Miss=96117, CPU0 Cache Access=1486580, CPU0 Read Stall=7149054, CPU0 Write Stall=10, CPU0 Instruction Renames=5197283, CPU0
	00:00:04.511	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0
	00:00:04.569	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0
	00:00:04.627	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0
	00:00:04.685	PS	CPU0 Cycles=1, CPU0 Cache Miss=0, CPU0 Cache Access=0, CPU0 Read Stall=0, CPU0 Write Stall=0, CPU0 Instruction Renames=0, CPU0 Branch Miss=0, CPU1 Cycles=0

Press **Enter** to search for and selects the next matching event. Press **Shift+Enter** to search for and select the previous matching event. Wrapping occurs in both directions.

Press **Esc** to cancel an ongoing search.

Press **Del** to clear the header row and reset all events to normal.

Filtering an Event

When a filtering condition is entered in the head row, the table will clear all events and fill itself with matching events as they are found from the beginning of the trace.

A status row will be displayed before and after the matching events, dynamically showing how many matching events were found and how many events were processed so far. When the filtering is completed, the status row icon in the left margin will change from a stop to a filter icon.

<filter>	<filter>	.*CPU0 Cycles=60.*
16903/18015		
00:00:04.223	PS	CPU0 Cycles=601590, CPU0 Cache Miss=273693, CPU0 Cache Access=4236328, CPU0 Read Stall=21016365, CPU0 Write Stall=0, CPU0 Instruction Renames=148235
00:00:04.280	PS	CPU0 Cycles=601554, CPU0 Cache Miss=267841, CPU0 Cache Access=4145496, CPU0 Read Stall=21392519, CPU0 Write Stall=0, CPU0 Instruction Renames=145126
00:00:04.338	PS	CPU0 Cycles=601609, CPU0 Cache Miss=255995, CPU0 Cache Access=3960952, CPU0 Read Stall=22161892, CPU0 Write Stall=0, CPU0 Instruction Renames=138616
00:00:04.396	PS	CPU0 Cycles=601556, CPU0 Cache Miss=271379, CPU0 Cache Access=4198920, CPU0 Read Stall=21171012, CPU0 Write Stall=0, CPU0 Instruction Renames=146962
00:00:07.516	PS	CPU0 Cycles=601526, CPU0 Cache Miss=43083, CPU0 Cache Access=5868292, CPU0 Read Stall=831621, CPU0 Write Stall=0, CPU0 Instruction Renames=45081682,
00:00:10.751	PS	CPU0 Cycles=601391, CPU0 Cache Miss=788, CPU0 Cache Access=16925238, CPU0 Read Stall=357, CPU0 Write Stall=40, CPU0 Instruction Renames=49113377, CPI
00:00:10.809	PS	CPU0 Cycles=601555, CPU0 Cache Miss=799, CPU0 Cache Access=16927688, CPU0 Read Stall=454, CPU0 Write Stall=8, CPU0 Instruction Renames=49128637, CPU
00:00:10.867	PS	CPU0 Cycles=600333, CPU0 Cache Miss=888, CPU0 Cache Access=16891872, CPU0 Read Stall=756, CPU0 Write Stall=25, CPU0 Instruction Renames=49007316, CPI
00:00:10.925	PS	CPU0 Cycles=601474, CPU0 Cache Miss=898, CPU0 Cache Access=16924304, CPU0 Read Stall=345, CPU0 Write Stall=7, CPU0 Instruction Renames=49105458, CPU
00:00:10.982	PS	CPU0 Cycles=600257, CPU0 Cache Miss=880, CPU0 Cache Access=16886325, CPU0 Read Stall=511, CPU0 Write Stall=7, CPU0 Instruction Renames=49011838, CPU
00:00:11.040	PS	CPU0 Cycles=601554, CPU0 Cache Miss=907, CPU0 Cache Access=16929543, CPU0 Read Stall=516, CPU0 Write Stall=8, CPU0 Instruction Renames=49114263, CPU
00:00:11.098	PS	CPU0 Cycles=601724, CPU0 Cache Miss=899, CPU0 Cache Access=16932908, CPU0 Read Stall=408, CPU0 Write Stall=4, CPU0 Instruction Renames=49130137, CPU
00:00:11.155	PS	CPU0 Cycles=608775, CPU0 Cache Miss=923, CPU0 Cache Access=17127129, CPU0 Read Stall=502, CPU0 Write Stall=15, CPU0 Instruction Renames=49711639, CPI
00:00:11.214	PS	CPU0 Cycles=601982, CPU0 Cache Miss=894, CPU0 Cache Access=16943563, CPU0 Read Stall=612, CPU0 Write Stall=0, CPU0 Instruction Renames=49155914, CPU

Press **ESC** to stop an ongoing filtering. In this case the status row icon will remain as a 'stop' icon to indicate that not all events were processed.

Press **DEL** or right-click on the table and select **Clear Filters** from the context menu to clear the header row and remove the filtering. All trace events will be now shown in the table. Note that the currently selected event will remain selected even after the filter is removed.

You can also search on the subset of filtered events by toggling the header row to the Search Bar while a filter is applied. Searching and filtering conditions are independent of each other.

Bookmarking an Event

Any event of interest can be tagged with a bookmark.

To add a bookmark, double-click the left margin next to an event, or right-click the margin and select **Add bookmark**. Alternatively, use the **Edit→Add bookmark** menu. Edit the bookmark description as desired and click **OK**.

The bookmark will be displayed in the left margin, and hovering the mouse over the bookmark icon will display the description in a tooltip.

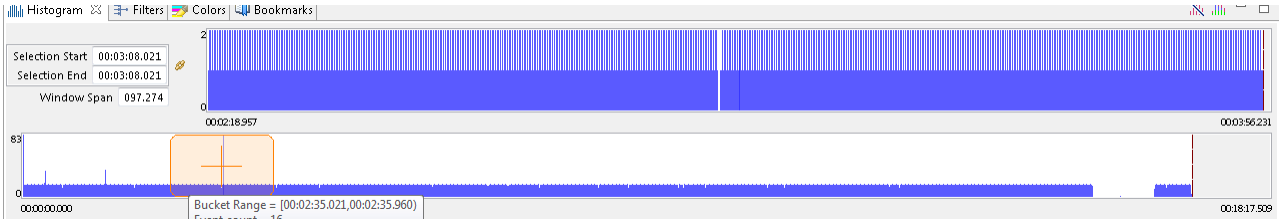
The bookmark will be added to the Bookmarks view. In this view, the bookmark description can be edited, and the bookmark can be deleted. Double-clicking the bookmark or selecting **Go to** from its context menu will open the trace or experiment and go directly to the event that was bookmarked.

<filter>	<filter>	.*Max Read Latency=610.*
14670/180...		
00:03:07.963	APM	HP0 Write Bytes=21880944, HP0 Write Transactions=170945, HP0 Write Latency=3247955, HP0 Read Bytes=23056640, HP0 Read Transactions=180130, HP0 Read Latency=
00:03:08.021	APM	HP0 Write Bytes=25287936, HP0 Write Transactions=197562, HP0 Write Latency=3753678, HP0 Read Bytes=25013872, HP0 Read Transactions=195420, HP0 Read Latency=
00:03:08.079	APM	HP0 Write Bytes=25432320, HP0 Write Transactions=198690, HP0 Write Latency=3775129, HP0 Read Bytes=24998696, HP0 Read Transactions=195303, HP0 Read Latency=
00:03:08.136	APM	HP0 Write Bytes=25088080, HP0 Write Transactions=196001, HP0 Write Latency=3724000, HP0 Read Bytes=24963680, HP0 Read Transactions=195029, HP0 Read Latency=
00:03:08.194	APM	HP0 Write Bytes=25001312, HP0 Write Transactions=195323, HP0 Write Latency=3711137, HP0 Read Bytes=24998176, HP0 Read Transactions=195298, HP0 Read Latency=
00:03:08.252	APM	HP0 Write Bytes=25426496, HP0 Write Transactions=198644, HP0 Write Latency=3774255, HP0 Read Bytes=25099752, HP0 Read Transactions=196092, HP0 Read Latency=
00:03:08.309	APM	HP0 Write Bytes=25300352, HP0 Write Transactions=197659, HP0 Write Latency=3755502, HP0 Read Bytes=25110736, HP0 Read Transactions=196177, HP0 Read Latency=

To remove a bookmark, double-click its icon, select **Remove Bookmark** from the left margin context menu, or select **Delete** from the Bookmarks view.

Histogram View

The Histogram View displays the trace events (counters data) distribution with respect to time. When performance analysis is running, this view is dynamically updated as the events are received.



The controls on the view are described below.

- **Selection Start:** Displays the start time of the current selection.
- **Selection End:** Displays the end time of the current selection.
- **Window Span:** Displays the current zoom window size in seconds.

The controls can be used to modify their respective value. After validation, the other controls and views will be synchronized and updated accordingly. To modify both selection times simultaneously, press the **link** icon which disables the Selection End control input.

The large (full) histogram, at the bottom, shows the event distribution over the trace. It also has a smaller semi-transparent orange window, with a cross-hair, that shows the current zoom window.

The smaller (zoom) histogram, on top right, corresponds to the current zoom window, a sub-range of the event set.

The x-axis of each histogram corresponds to the event timestamps. The start time and end time of the histogram range is displayed. The y-axis shows the maximum number of events in the corresponding histogram bars.

The vertical blue line(s) show the current selection time (or range). If applicable, the region in the selection range will be shaded.

The mouse actions that can be used to control the histogram are listed below.

- **Left-click:** Sets a selection time
- **Left-drag:** Sets a selection range
- **Shift+left-click or drag:** Extend or shrink the selection range
- **Middle-click or CTRL+Left-click:** Centers the zoom window

- **Middle-drag or CTRL+left-drag:** Moves the zoom window
- **Right-drag:** Sets the zoom window
- **SHIFT+Right-click or drag:** Extend or shrink the zoom window
- **Mouse wheel up:** Zoom in
- **Mouse wheel down:** Zoom out

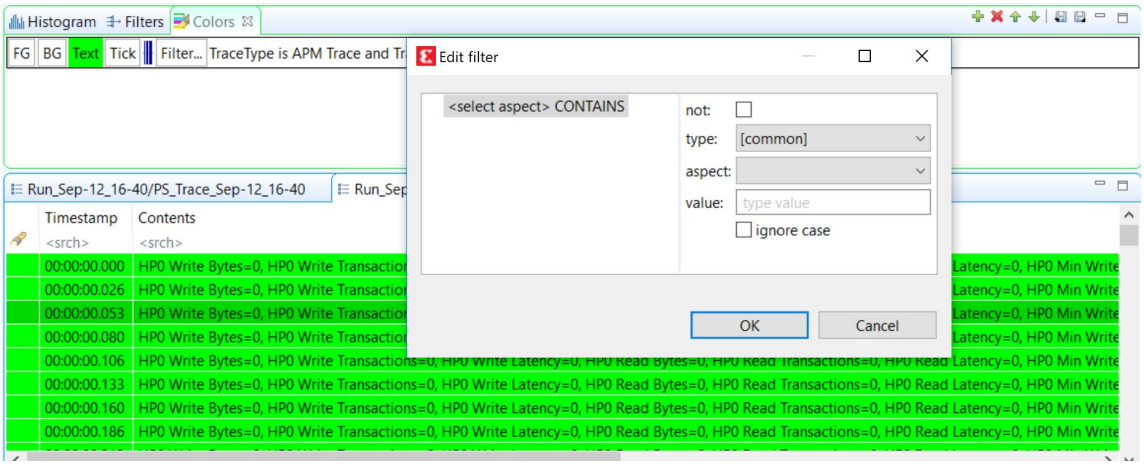
Hovering the mouse over an histogram bar pops up an information window that displays the start/end time of the corresponding bar, as well as the number of events it represents. If the mouse is over the selection range, the selection span in seconds is displayed.

The actions performed by various keystrokes when they are used in the Histogram view are listed below.

- **Left Arrow:** Moves the current event to the previous non-empty bar.
- **Right Arrow:** Moves the current event to the next non-empty bar.
- **Home:** Sets the current time to the first non-empty bar.
- **End:** Sets the current time to the last non-empty histogram bar.
- **Plus (+):** Zoom in
- **Minus (-):** Zoom out

Colors View

The Colors view allows you to define a prioritized list of color settings.



A color setting associates a foreground and background color (used in any events table), and a tick color (used in the Time Chart view), with an event filter.

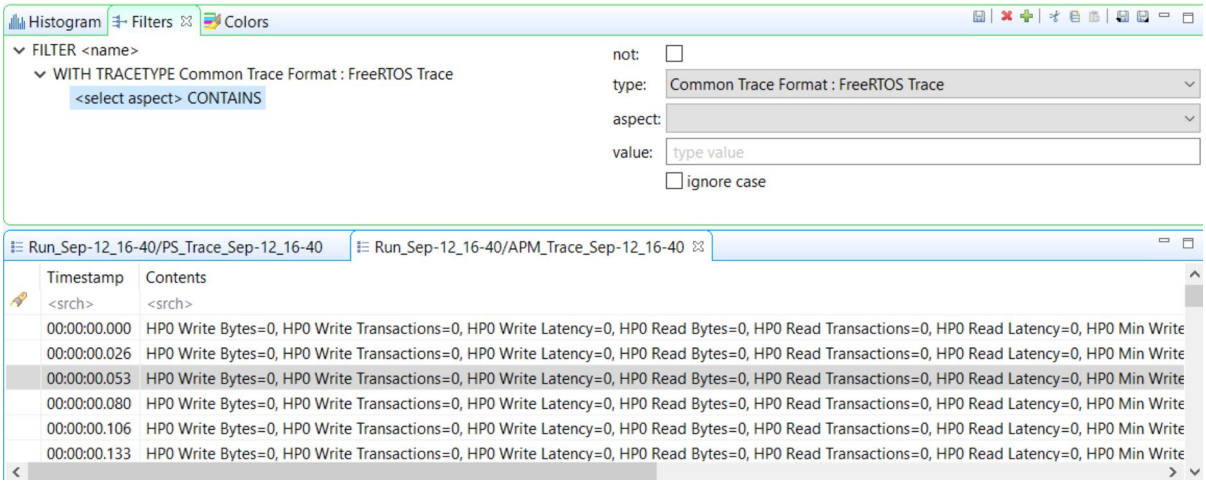
In an events table, any event row that matches the event filter of a color setting will be displayed with the specified foreground and background colors. If the event matches multiple filters, the color setting with the highest priority will be used.

The same principle applies to the event tick colors in the Time Chart view. If a tick represents many events, the tick color of the highest priority matching event will be used.

Color settings can be inserted, deleted, reordered, imported and exported using the buttons in the Colors view toolbar. Changes to the color settings are applied immediately, and are persisted to disk.

Filters View

The Filters view allows you to define preset filters that can be applied to any events table.



The filters can be more complex than what can be achieved with the filter header row in the events table. The filter is defined in a tree node structure, where the node types can be any of TRACETYPE, AND, OR, CONTAINS, EQUALS, MATCHES. or COMPARE. Some nodes types have restrictions on their possible children in the tree.

The TRACETYPE node filters against the trace type of the trace as defined in a plug-in extension or in a custom parser. When used, any child node will have its aspect combo box restricted to the possible aspects of that trace type.

The AND node applies the logical `and` condition on all of its children. All children conditions must be true for the filter to match. A `not` operator can be applied to invert the condition.

The OR node applies the logical `or` condition on all of its children. At least one children condition must be true for the filter to match. A `not` operator can be applied to invert the condition.

The CONTAINS node matches when the specified event `aspect` value contains the specified `value` string. A `not` operator can be applied to invert the condition. The condition can be case sensitive or insensitive.

The EQUALS node matches when the specified event `aspect` value equals exactly the specified `value` string. A `not` operator can be applied to invert the condition. The condition can be case sensitive or insensitive.

The MATCHES node matches when the specified event `aspect` value matches against the specified `regular expression`. A `not` operator can be applied to invert the condition.

The COMPARE node matches when the specified event `aspect` value compared with the specified `value` gives the specified `result`. The result can be set to `smaller than`, `equal` or `greater than`. The type of comparison can be numerical, alphanumerical or based on time stamp. A `not` operator can be applied to invert the condition.

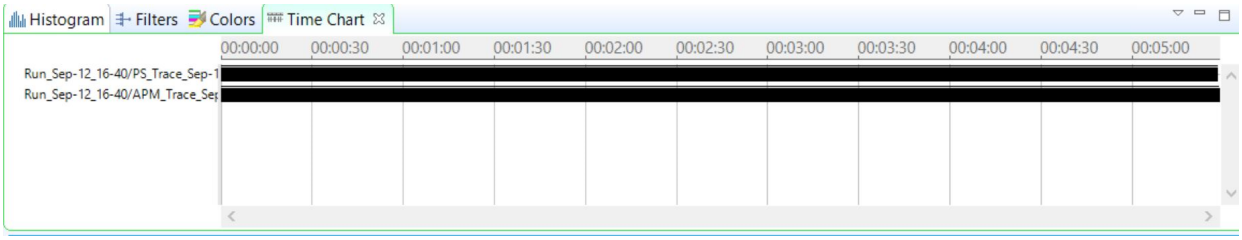
For numerical comparisons, strings prefixed by "0x", "0X" or "#" are treated as hexadecimal numbers and strings prefixed by "0" are treated as octal numbers.

For time stamp comparisons, strings are treated as seconds with or without fraction of seconds. This corresponds to the TTT format in the Time Format preferences. The value for a selected event can be found in the Properties view under the `Timestamp` property. The common 'Timestamp' aspect can always be used for time stamp comparisons regardless of its time format.

Filters can be added, deleted, imported and exported using the buttons in the Filters view toolbar. The nodes in the view can be Cut (Ctrl-X), Copied (Ctrl-C) and Pasted (Ctrl-V) by using the buttons in the toolbar or by using the key bindings. This makes it easier to quickly build new filters from existing ones. Changes to the preset filters are only applied and persisted to disk when the Save filters button is pressed.

Time Chart View

The Time Chart view allows you to visualize every open trace in a common time chart. Each trace is displayed in its own row, and ticks are displayed for every punctual event. As you zoom using the mouse wheel, or by right-clicking and dragging in the time scale, more detailed event data is computed from the traces.



Time synchronization is enabled between the time chart view and other trace viewers such as the events table.

Color settings defined in the Colors view can be used to change the tick color of events displayed in the Time Chart view.

When a search is applied in the events table, the ticks corresponding to matching events in the Time Chart view are decorated with a marker below the tick.

When a bookmark is applied in the events table, the ticks corresponding to the bookmarked event in the Time Chart view is decorated with a bookmark above the tick.

When a filter is applied in the events table, the non-matching ticks are removed from the Time Chart view.

The Time Chart view only supports traces that are opened in an editor. The use of an editor is specified in the plug-in extension for that trace type, or is enabled by default for custom traces.

Analysis Views

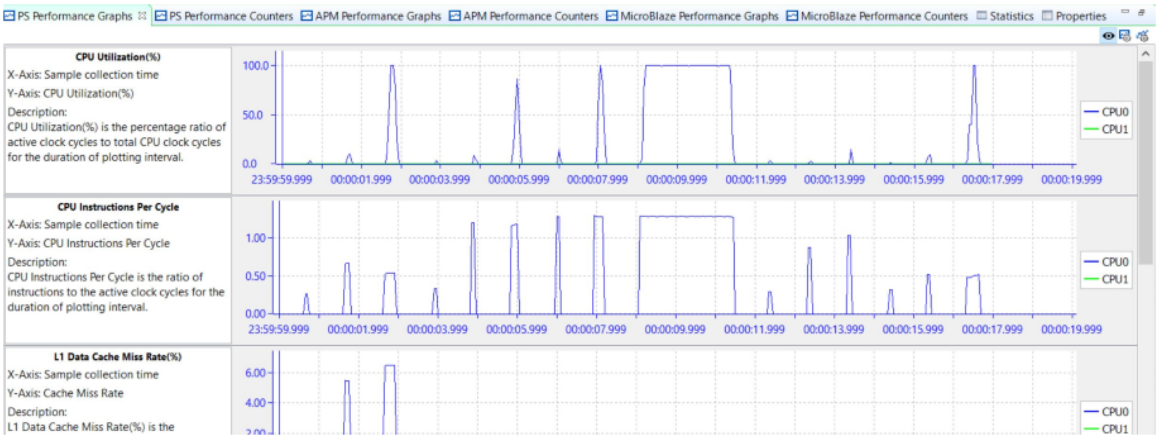
For each of the different types of trace (PS, APM, and so on) collected, there is a set of views to help in analyzing it. There are two types of views; tabular and graphical.

You can view the analysis of trace data both in live mode, when the data collection is running, and in offline mode. In live mode, tabular view displays analysis for the entire trace duration, whereas the graphical view displays analysis for the recent 20 seconds. In the offline mode, graphical view displays the zoomed region whereas the tabular view displays the selection region or zoomed region depending on whichever is the last user action. In live mode, to pause the views and view the past data, us the icon present in the analysis views. When the views are paused, the Histogram View can be used to zoom and analyze any portion of the data.

These analysis views display the data only when corresponding trace file is opened in the Events Editor; otherwise they will be empty.

PS Performance Graphs

All the PS (Arm) metrics will be displayed using these graphs.



PS Performance Counters

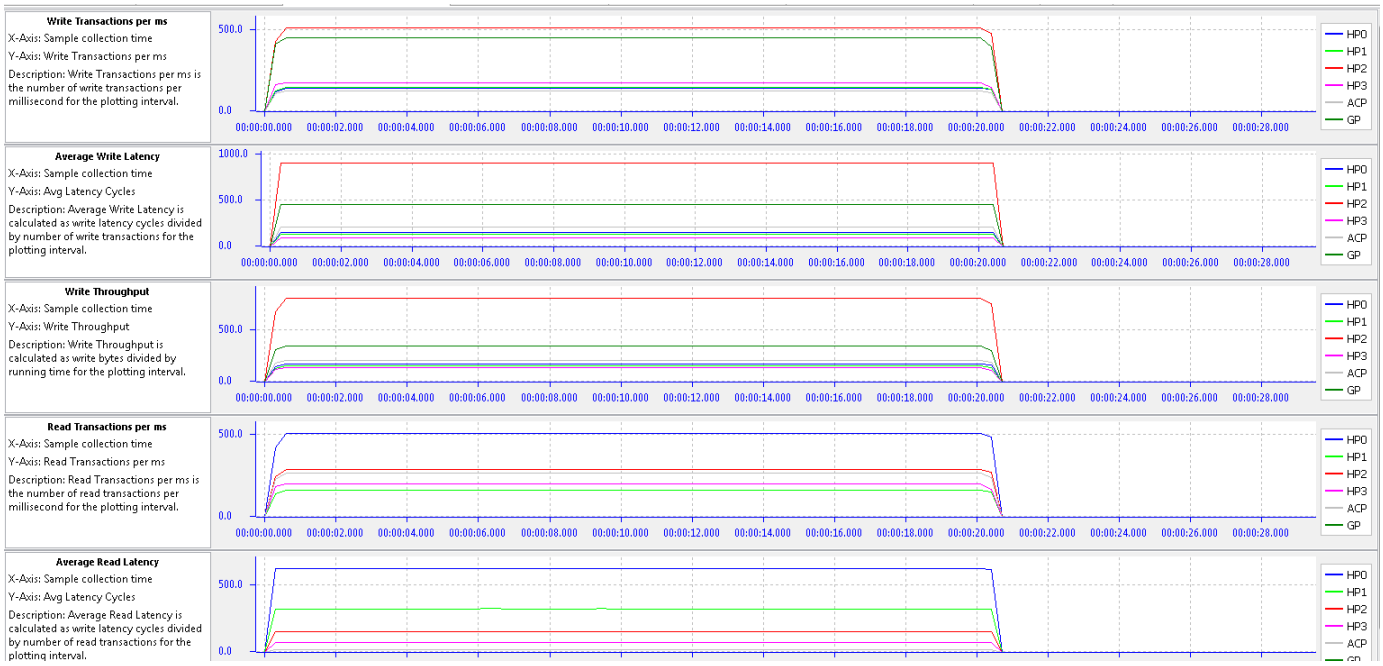
Tabular representation of the PS (Arm) metrics.

PS Performance Graphs | PS Performance Counters | APM Performance Graphs

00:00:00.000-00:00:53.092	CPU0	CPU1
CPU Utilization(%)	57.1	0.01
CPU Instructions Per Cycle	0.31	0.00
L1 Data Cache Miss Rate(%)	18.1	0.00
L1 Data Cache Access per ms	30.0k	0.00
CPU Write Stall Cycles Per Instructi...	0.00	0.00
CPU Read Stall Cycles Per Instruction	1.66	0.00

APM Performance Graphs

APM metrics are displayed using the graphs.



APM Performance Counters

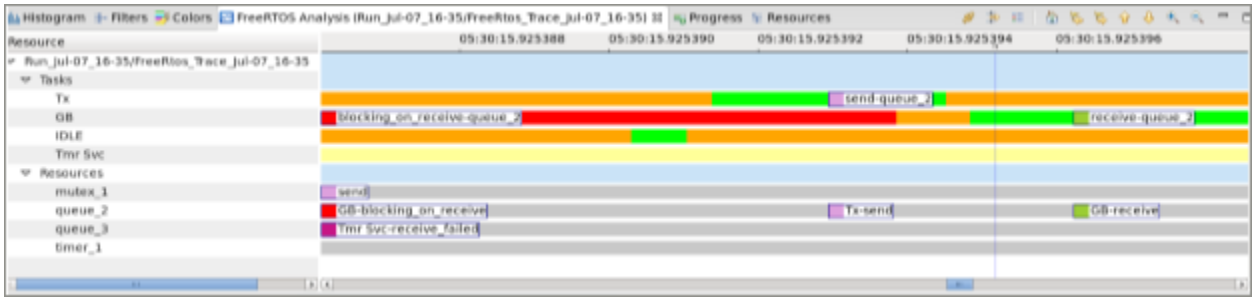
APM metrics displayed in tabular format.

PS Performance Graphs | PS Performance Counters | APM Performance Graphs | APM Performance Counters | MicroBlaze Performance Graphs | MicroBlaze Performance Counters | Statistics | Properties

00:00:00.000-00:00:29.571	HP0	HP1	HP2	HP3	ACP	GP
Write Transactions per ms	139.9	146.7	509.4	175.5	123.0	449.9
Minimum Write Latency	0.00	0.00	0.00	0.00	0.00	0.00
Maximum Write Latency	191.0	132.0	1194.0	96.0	213.0	460.0
Average Write Latency	154.0	132.0	903.3	96.0	213.0	460.0
Write Throughput (MB/sec)	169.0	151.4	798.8	130.6	206.6	340.1
Read Transactions per ms	504.7	162.7	289.5	199.7	263.5	0.00
Minimum Read Latency	0.00	0.00	0.00	0.00	0.00	0.00
Maximum Read Latency	1173.0	718.0	485.0	409.0	21.0	0.00
Average Read Latency	621.5	323.6	153.1	72.6	16.4	0.00
Read Throughput (MB/sec)	730.7	322.7	264.0	38.3	12.6	0.00

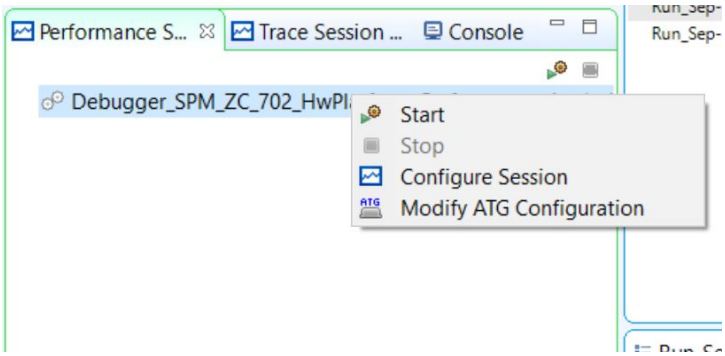
FreeRTOS Analysis

FreeRTOS event trace displayed in different states.



Performance Session Manager

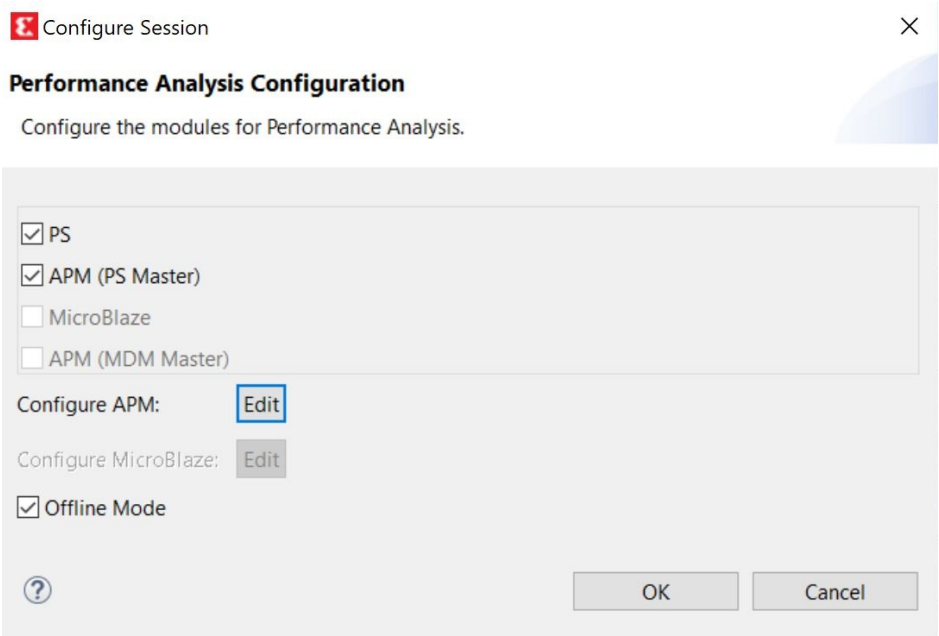
The Performance Session Manager view provides you with the capability to control the sessions. You can start and stop a performance session from this view. Each time a session is started, a set of trace files is created based on your configuration.



Whenever an application is debugged or performance analysis is launched, the view automatically populates the entry for the active configuration.

Configure Session

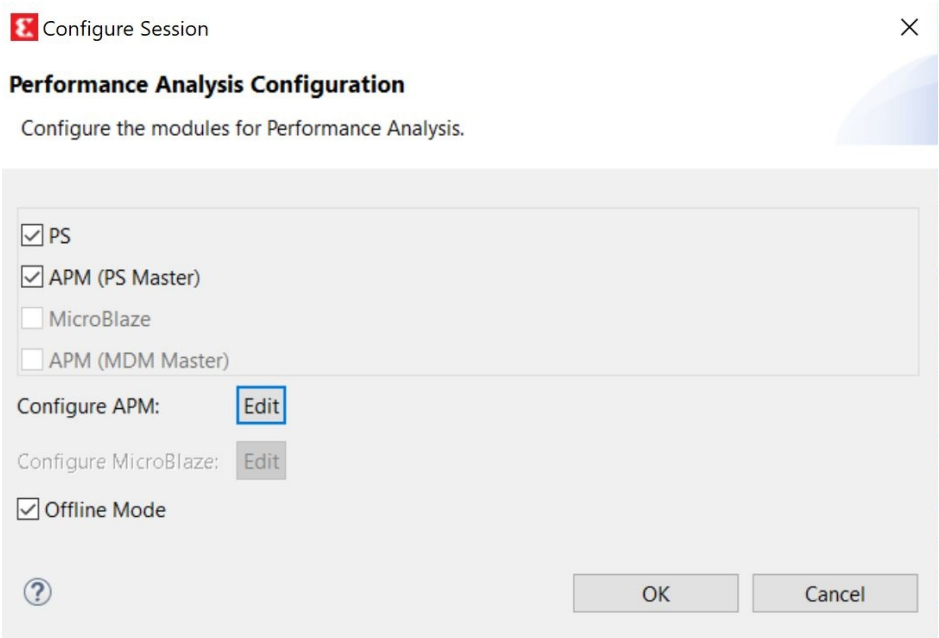
You can configure a session by choosing the list of modules for which the data has to be collected. Each of the modules will be enabled based on the design information.



If you wish to configure the modules prior to starting performance analysis, use the **Configure Performance Analysis** option on the hardware Project.

Configure APM

You can choose which APM slots to be monitored by selecting the **Configure APM** option on the Configure Session dialog box.



Configure MicroBlaze

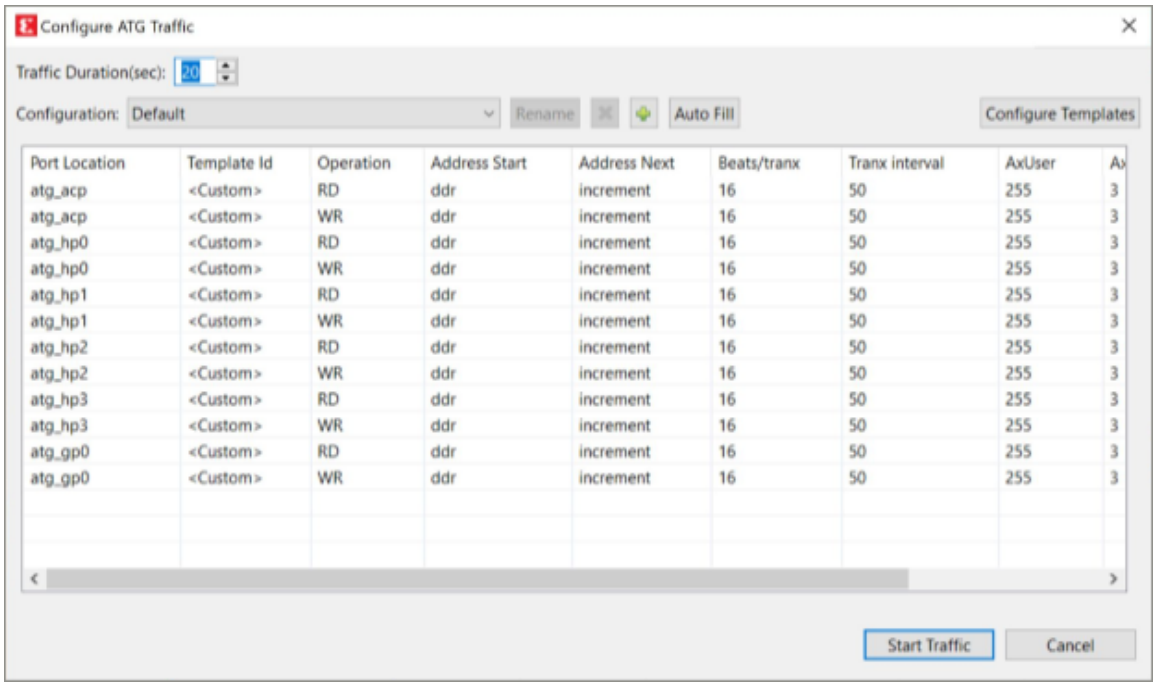
You can choose the MicroBlaze instances for performance analysis use the option **Configure MicroBlaze** in the Configure Session dialog box. By default, only instances from the first MDM module will be selected.

Offline Mode

Viewing the live performance analysis is supported only for duration of 10 mins and stops automatically after the elapsed time. When Offline Mode is selected, the performance analysis runs indefinitely until you stop it manually from the view.

Modify ATG Configuration

You can modify the ATG traffic configuration using the Modify ATG Configuration option available in the Performance Session Manager.



Port Location	Template Id	Operation	Address Start	Address Next	Beats/tranx	Tranx interval	AxUser	Ax
atg_acp	<Custom>	RD	ddr	increment	16	50	255	3
atg_acp	<Custom>	WR	ddr	increment	16	50	255	3
atg_hp0	<Custom>	RD	ddr	increment	16	50	255	3
atg_hp0	<Custom>	WR	ddr	increment	16	50	255	3
atg_hp1	<Custom>	RD	ddr	increment	16	50	255	3
atg_hp1	<Custom>	WR	ddr	increment	16	50	255	3
atg_hp2	<Custom>	RD	ddr	increment	16	50	255	3
atg_hp2	<Custom>	WR	ddr	increment	16	50	255	3
atg_hp3	<Custom>	RD	ddr	increment	16	50	255	3
atg_hp3	<Custom>	WR	ddr	increment	16	50	255	3
atg_gp0	<Custom>	RD	ddr	increment	16	50	255	3
atg_gp0	<Custom>	WR	ddr	increment	16	50	255	3

System Performance Modeling

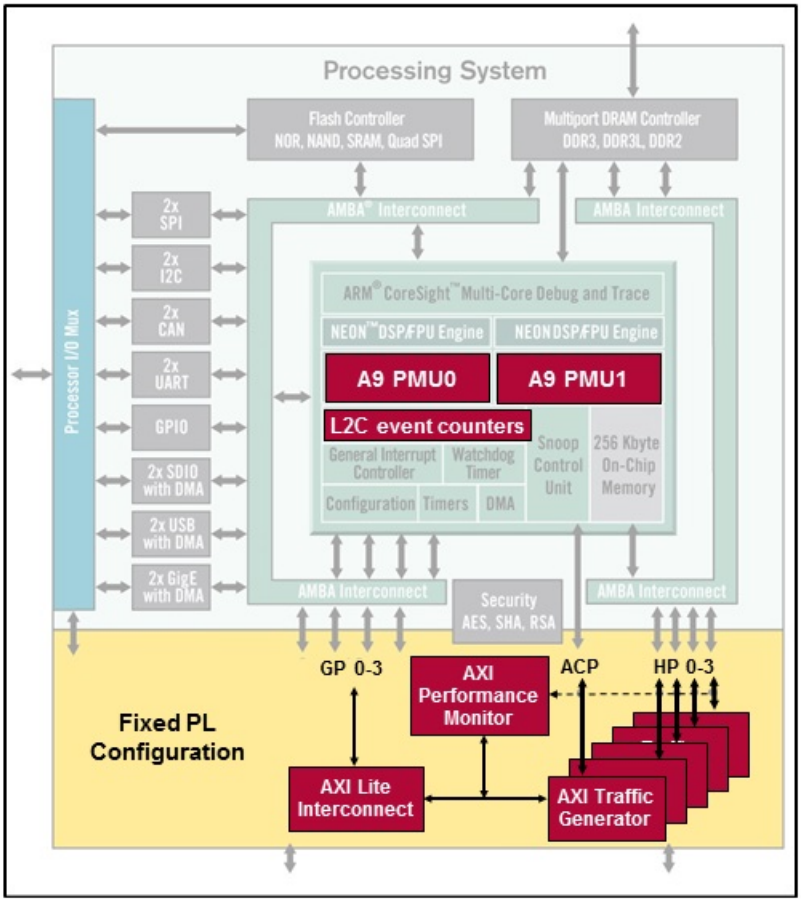
System Performance Modeling (SPM) offers system-level performance analysis for characterizing and evaluating the performance of hardware and software systems. In particular, it enables analysis of the critical partitioning trade-offs between the Arm® Cortex A9 processors and the programmable fabric for a variety of different traffic scenarios. It provides graphical visualizations of AXI transaction traces and system-level performance metrics such as throughput, latency, utilization, and congestion.

SPM can be used in two ways:

- Using a predefined design provided with the Vitis software platform
- With the user design

In the current release, SPM is supported only for baremetal/standalone applications.

The following diagram shows the system performance modeling flow.



Predefined Design Flow

The predefined flow provided with the Vitis software platform uses the fixed design and comes with a fixed bitstream. In this design, there are five AXI Traffic Generators (ATGs), with one connected to each of the four High Performance ports (HP0-3) and one connected to the Accelerator Coherency Port (ACP). The ATGs are set up and controlled using one of the General Purpose (GP) ports. In addition, an AXI Performance Monitor (APM) is included in order to monitor the AXI traffic on the HP0-3 and ACP ports.

System Performance Modeling Using the Predefined Design

Creating the System Performance Modeling Project

1. Select **File → New → Other → Xilinx → SPM Project...** to start the System Performance Modeling application.
2. Click **Finish**.
3. The SPM Launcher opens.
4. The following figure shows the Edit Configuration dialog box for a local connection. To start the SPM with the default traffic configuration, click **Debug**.

5. It first programs the FPGA and then starts the SPM.

Selecting an ATG Traffic Configuration

To select a traffic configuration:

1. In the Project Explorer, right-click the hardware platform and select **Run As → Run Configurations**.
2. Under Performance Analysis, select **Performance Analysis on <filename>.elf**.
3. You can use the ATG Configuration tab to define multiple traffic configurations and select the traffic to be used for the current run. The following figure shows the traffic that is defined in the Default configuration.
4. The port location is taken from the Hardware handoff file. If no ATG was configured in the design, the ATG Configuration tab is empty.
5. You can use the ATG Configuration dialog box to add and edit configurations.
6. To add a configuration to the list of configurations, click the **+** button.
7. To edit a configuration, select the **Configuration:** drop-down list to choose the configuration that you want to edit.
8. For ease of defining an ATG configuration, you can create Configuration Templates. These templates are saved for the user workspace and can be used across the Projects for ATG traffic definitions. To create a template, do the following:
9. Click **Configure Templates**.
10. Click the **+** button to add a new user-defined configuration template.
11. The newly created template is assigned a Template ID with the pattern of "UserDef_*" by default. You can change the ID and also define the rest of the fields.
12. You can use these defined templates to define an ATG configuration. To delete a Configuration Template, select it and click the **X** button.

TIP: *In an ATG configuration, to set a port so that it does not have any traffic, set the Template ID for that port to* ***None****.*

Configure FSBL Parameters

Changing the first stage bootloader (FSBL) configuration is only available for the fixed design flow of the System Performance Modeling application.

To invoke the FSBL Configuration Change dialog box, right-click the configuration name and select **Configure FSBL Parameters**.

Below are the details about the first stage bootloader (FSBL) parameters.

Table 10: **FSBL Parameters**

Parameter	Description	Default Value
PS Clock Frequency (MHz)	The clock frequency of the Zynq-7000 SoC PS (specified in MHz).	666.7 MHz
PL Clock Frequency (MHz)	The clock frequency of the Zynq-7000 SoC PL (specified in MHz).	100.0 MHz
DDR Clock Frequency (MHz)	The clock frequency of the DDR memory (specified in MHz).	533.3 MHz
DDR Data Path Width	The bit width used in the DDR memory data path. Possible values are 16 and 32 bits.	32 bits
DDR Port 0 - Enable HPR	This enables the usage of high priority reads on DDR port 0. This port is used by the CPUs and the ACP via the L2 Cache.	Unchecked
DDR Port 1 - Enable HPR	This enable the usage of high priority reads on DDR port 1. This port is used by other masters via the central interconnect.	Unchecked
DDR Port 2 - Enable HPR	This enables the usage of high priority reads on DDR port 2. This port is used by HP2 and HP3.	Unchecked
DDR Port 3 - Enable HPR	This enable the usage of high priority reads on DDR port 3. This port is used by HP0 and HP1.	Unchecked
HPR/LPR Queue Partitioning	Indicates the desired partitioning for high and low priority reads in the queue. Note that the queue has a depth of 32 read requests. There are four values provided in a drop-down menu.	HPR(0)/LPR(32)
LPR to Critical Priority Level	The number of clocks that the LPR queue can be starved before it goes critical. Unit: 32 DDR clock cycles. This value sets the DDR LPR_reg register [1]. Valid values are between 0 and 2047.	2
HPR to Critical Priority Level	The number of clocks that the HPR queue can be starved before it goes critical. Unit: 32 DDR clock cycles. This value sets the DDR HPR_reg register [1]. Valid values are between 0 and 2047.	15
Write to Critical Priority Level	The number of clocks that the write queue can be starved before it goes critical. Unit: 32 DDR clock cycles. This value sets the DDR WR_reg register [1]. Valid values are between 0 and 2047.	2

For more information about the FSBL, refer to *Zynq-7000 SoC Software Developers Guide* (UG821).

User-Defined Flow

Performance analysis can be done on any user-defined applications.

System Performance Modeling Using a User-Defined Flow

The Vitis software platform provides the capability to monitor a running target regardless of the target operating system.

***Note*:** If no ATG is configured in the hardware, the ATG Configuration window will be empty. Make sure to remove the Breakpoints by selecting **Window → Show View → Breakpoints**.

1. If your design is defined in the Vivado Design Suite, then it is recommended to create a platform specification based on the design. To do performance analysis based on the specification:

a. Build and export your bitstream using **File → Export → Export Hardware** in the Vivado Design Suite.

b. In the Vitis™ software platform, select **File → New → Platform Project** and import the generated file `<your design>.xsa` into the Vitis software platform.

c. Right-click on the application and select **Run As → Run Configuration**.

d. Select **SPM Analysis** and click the **New** button to create a performance analysis configuration.

e. Select **Standalone Application Debug** from the **Debug Type** dropdown list.

f. Select the imported hardware platform specification from the Hardware platform dropdown list.

g. Select the **Reset entire system** and **Program FPGA** check boxes.

h. Click **Run** to launch the **Performance Analysis** perspective.

2. For any reason, if you cannot create a hardware platform specification, or do not have one, you can still do performance analysis in the Vitis software platform. To do performance analysis in absence of the specification:

   a. Select **Run → Run Configurations**. Click **OK** to save the details and close the Configure APM dialog box.

   b. Select **Single Application Debug** and click the **New** button to create a new configuration.

   c. Select **Attach** to running target from the Debug Type dropdown list.

   d. Click **Performance Analysis Configuration** to edit the PS and APM settings.

   e. Click **ATG configuration** to see the ATG configuration.

   f. Click **Apply** and run to see the data.

Limitations

The Vitis software platform supports SPM only for baremetal/standalone applications.

Packaging the System/Utilities

Bootgen Utility

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

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

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

Program Flash

Program Flash is a Vitis software platform used to program the flash memories in the design. Various types of flash types 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 used in different configurations such as QSPI SINGLE, QSPI DUAL PARALLEL, and QSPI DUAL STACKED.

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

- **Hardware Platform:** Select the hardware platform you plan to use.
- **Connection:** Select the connection to hardware server.
- **Device:** Select a device. Auto Detect selects the first device on the chain, by default.
- **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 are only 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.
- **FSBL File:** The FSBL `.elf` file is mandatory for the NOR flash types in Zynq devices.

  ***Note*:** Not required for non Zynq devices.
- **Flash Type:** Select a flash type.
  - Zynq devices:
    - qspi_single
    - qspi_dual_parallel

  - qspi_dual_stacked
  - nand_8
  - nand_16
  - nor
  - emmc

    Note: emmc flash type is applicable for Zynq UltraScale+ MPSoC devices only.

- Non Zynq devices:
  - The flash type drop down list is populated based on the FPGA detected in the connection. If the connection to hardware server does not exist, an error message stating `"Could not retrieve Flash Part information. Please check hardware server connection"` is displayed on the dialog box. Based on the device detected, the dialog populates all the flash parts supported for the device.

Note: Appropriate part can be selected based on design. For Xilinx boards, the part name can found from the respective boards' user guide.

- **Convert ELF to Bootable SREC format and program:** The ELF file provided as the image file is converted into SREC format and programmed. This is a typical use case in non zynq devices. The SREC bootloader can be built and used to read the SREC converted ELF from flash, load it into RAM and boot.
- **Blank check after erase:** The blank check is performed to verify if the erase operation was properly done. The contents are read back and check if the region erased is blank.
- **Verify after Flash:** The verify operation is cross check the flash programming operation. The flash contents are read back and cross checked against the programmed data.

Creating a Bootable Image and Program the Flash

An example XSCT session that demonstrates creating two applications (FSBL and Hello World) is shown in the following snippet. It also allows you to create a bootable image using the applications along with bitstream and program the image on to the flash.

Note: Assuming the board to be ZCU702, `-flash_type qspi_single` is used as an option in `program_flash`.

```
setws /tmp/wrk/workspace
app create -name a9_hello -hw /tmp/wrk/system.xsa -os standalone proc
ps7_cortexa9_0 -template {Zynq FSBL}
app create -name a9_fsbl -hw /tmp/wrk/system.xsa -os standalone proc
ps7_cortexa9_0 -template {Hello World}
app build -name a9_hello
app build -name a9_fsbl
exec bootgen -arch zynq -image output.bif -w -o /tmp/wrk/BOOT.bin
exec program_flash -f /tmp/wrk/BOOT.bin -flash_type qspi_single -
blank_check -verify -cable type xilinx_tcf url tcp:localhost:3121
```

Chapter 9

Vitis Utilities

Xilinx Software Command-Line Tool

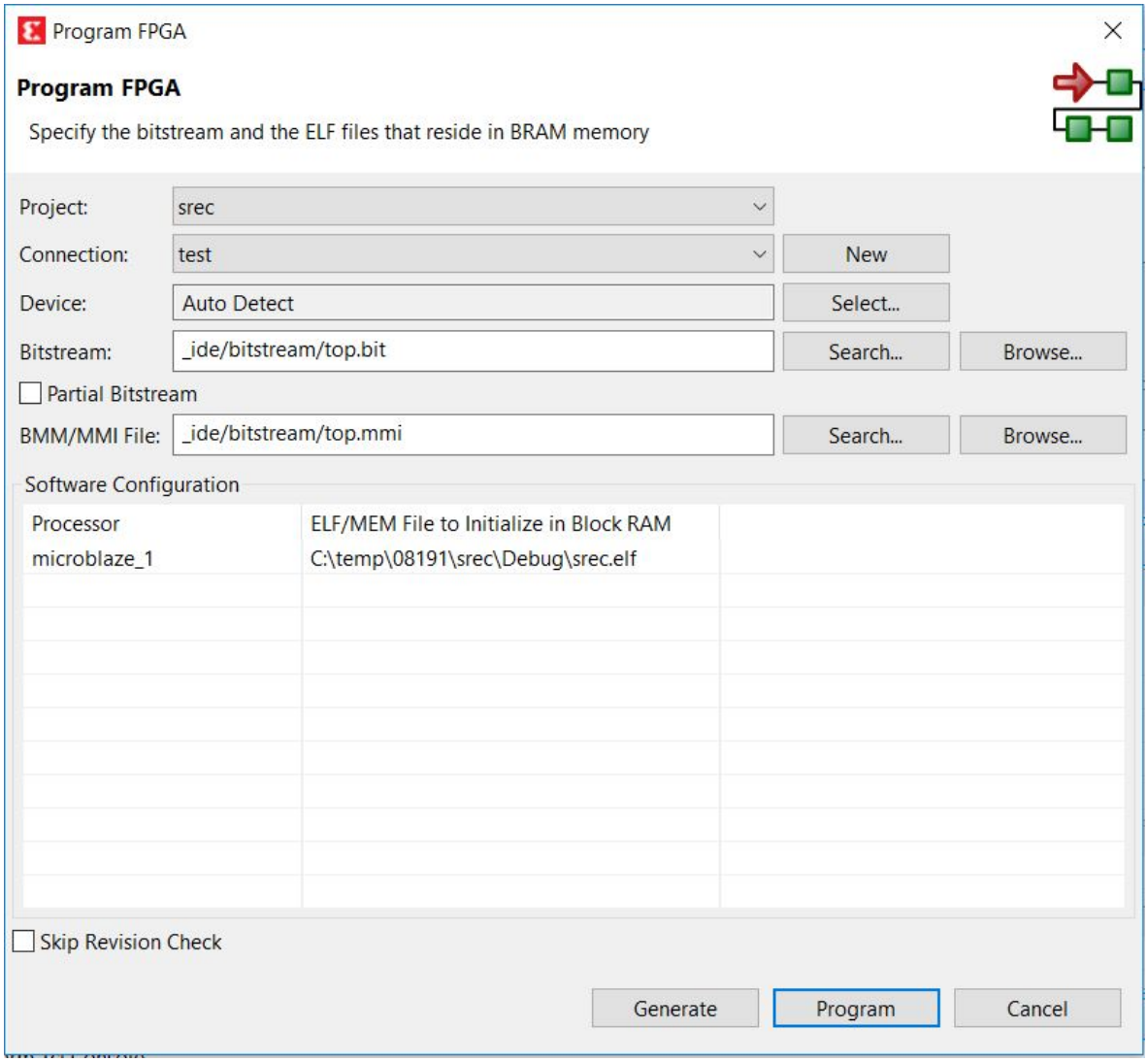
Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis IDE. As with other Xilinx tools, the scripting language for XSCT is based on Tools Command Language (Tcl). You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions:

- Create hardware, domains/board support packages (BSPs), and application projects
- Manage repositories
- Set toolchain preferences
- Configure and build domains/BSPs and applications
- Download and run applications on hardware targets
- Create and flash boot images by running Bootgen and program_flash tools.

For more information about XSCT, refer to the *Xilinx Software Command-Line Tool (XSCT) Reference Guide* (UG1208).

Program FPGA

Program the FPGA with the bitstream.



The following table lists the options available on the **Program FPGA** dialog box:

- **Hardware Configuration:** Specify the Bitstream and BMM files. These are provided by Vivado® Design Suite when you export your hardware design to the Vitis software platform.
- **Bitstream File:** Specify the FPGA Bitstream.
- **BMM File:** Specify the BMM file.
- **Software Configuration:** Specify the program that is initialized at the reset start address for each processor in the Block RAM.
- **Processor:** Name of the processor in the system.
- **ELF file:** Specify the ELF file to initialize.
- **Program:** Click this button to program the FPGA.

Dump/Restore Data File

The Vitis software platform allows you to copy the contents of a binary file to the target memory, or copy binary data from target memory to a file, through JTAG.

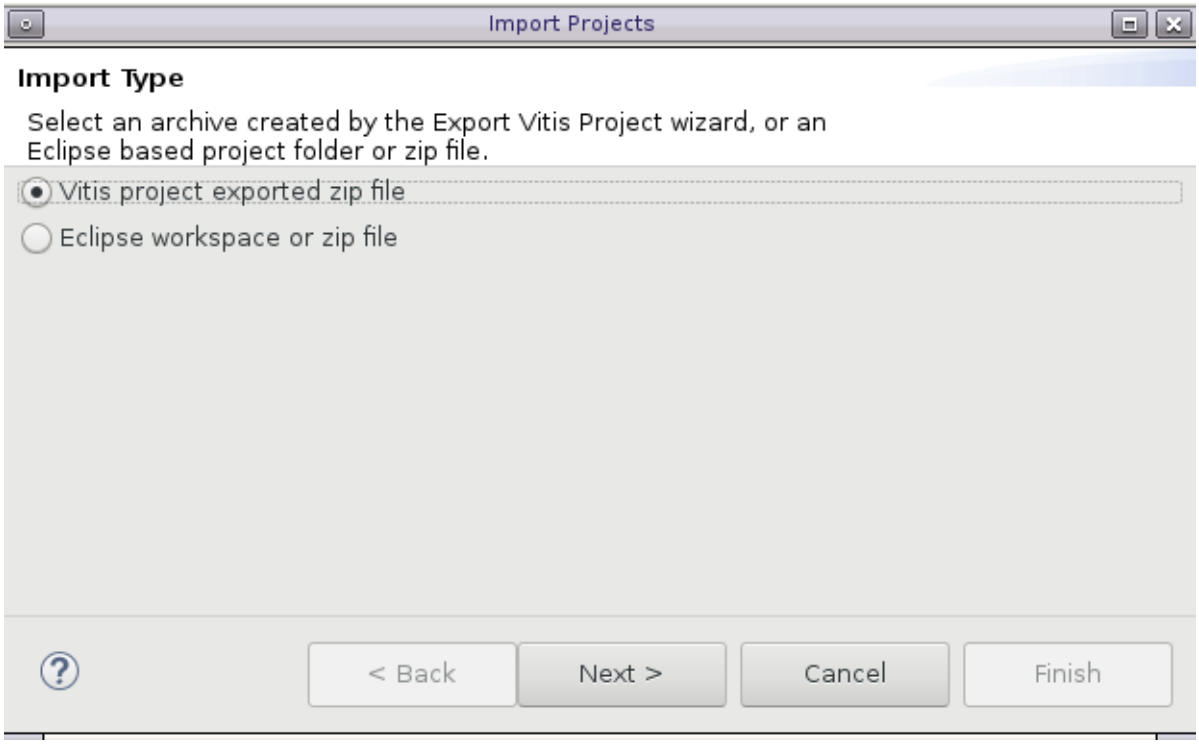
Launch Shell

Launch a command console window with Xilinx settings. This shell can be used for running XSDB, XSCT commands.

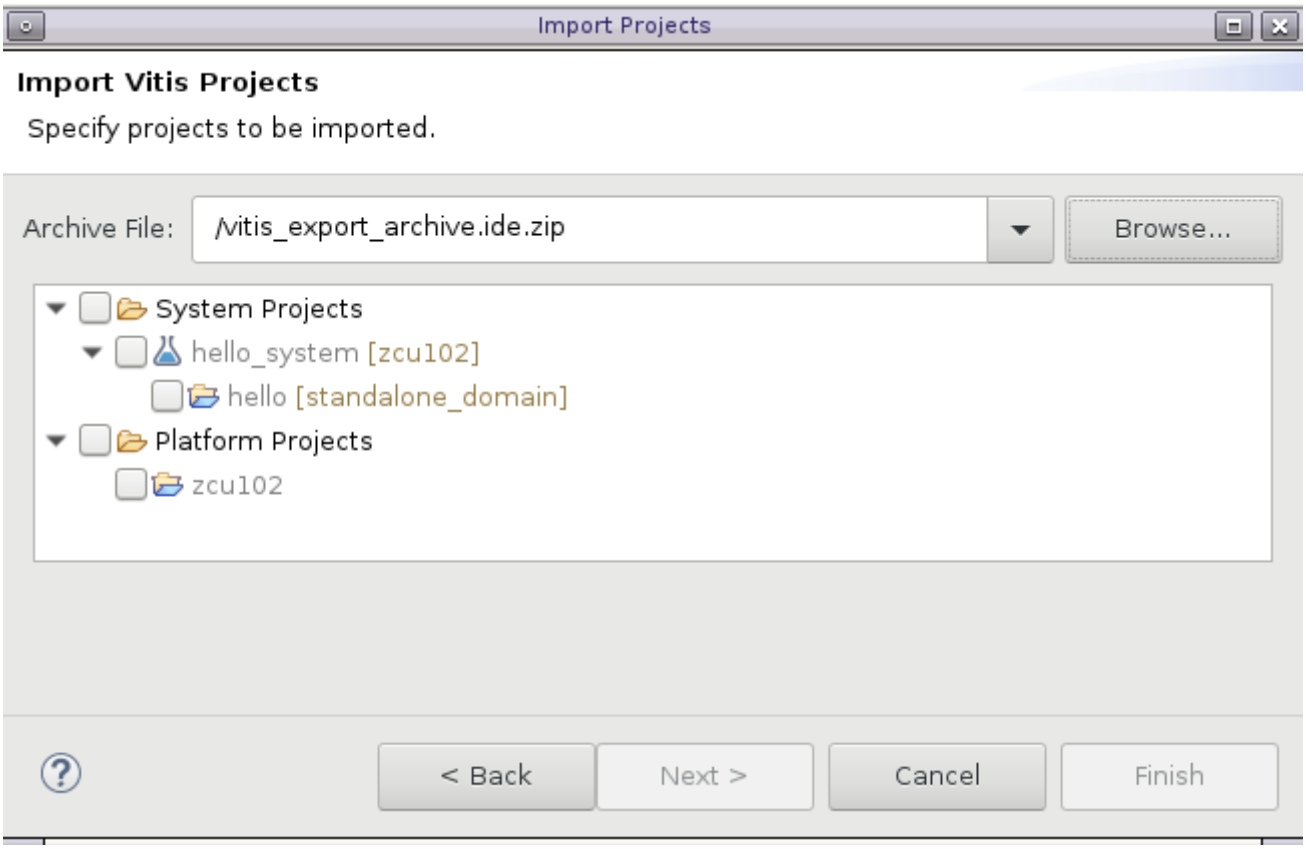
Import

In the Vitis IDE, you can also import projects that have previously been exported from the Vitis software platform.

1. Go to **File → Import → Vitis project exported zip file**.



2. Select the zip file exported from the Vitis software platform.

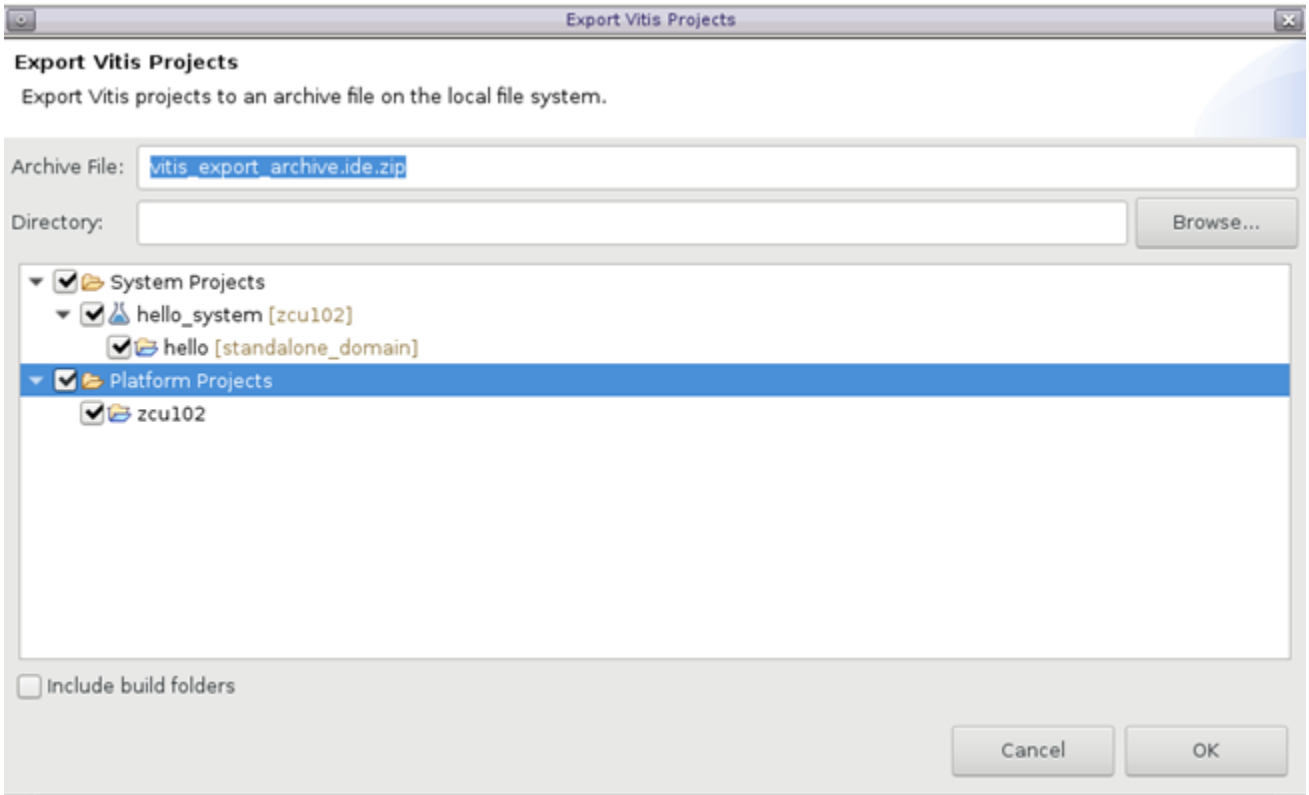


Note: If projects with the same name exist in the current workspace, the project in the exported zip cannot be imported.

Export

Projects managed in the Vitis IDE can be exported to that you can move them around easily.

1. Go to **File → Export** to open the Export Vitis Projects window.



2. Select the system projects or platform projects you want to export.
3. Set the export archive file name and destination directory. Selecting the **Include build folders** option includes build folders in the export zip file. This is generally not required because these files can be generated at the destination.

   Note: If any files are added to project by links, the referenced file will be added to the exported .zip file so that the project can be imported without referencing.

Generating Device Tree

The Vitis™ IDE can generate device trees. To generate a device tree, follow these steps:

1. Select **Xilinx → Repositories**.
2. Click **New**.
3. Provide the local path for the device tree generato, which can be downloaded from GitHub.
4. Select **Xilinx → Generate Device Tree** to open the Device Tree Generator.
5. Provide the hardware specification file and the output directory (the output will be created here).

You can change the settings for device tree blob (DTB) using the Modify Device Tree settings. The device tree path displays after successful generation.

Chapter 10

Embedded Software Development Use Cases in the Vitis Software Platform

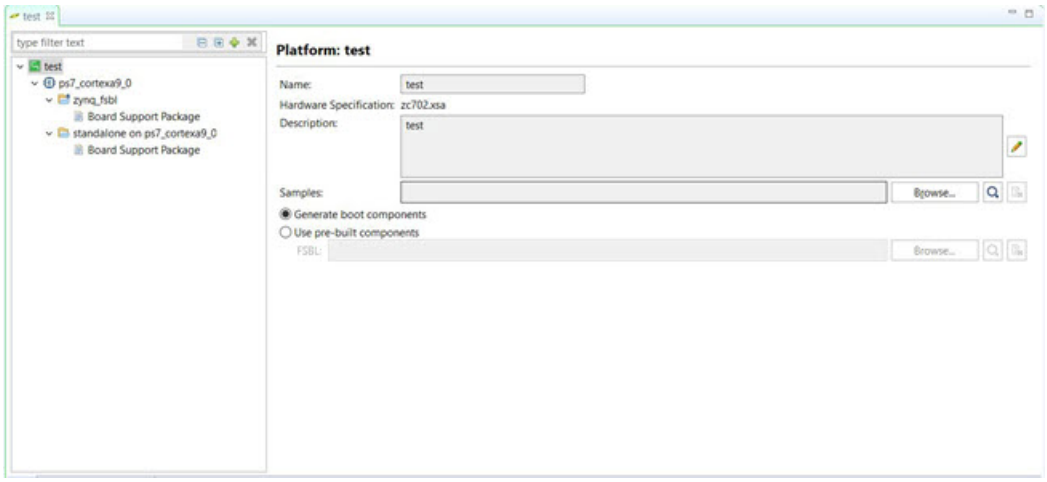
Creating a Platform Project

A platform project is a customizable platform in the Vitis™ software platform. Platform projects can be created by importing hardware configuration XSA files, which are exported from the Vivado® Design Suite.

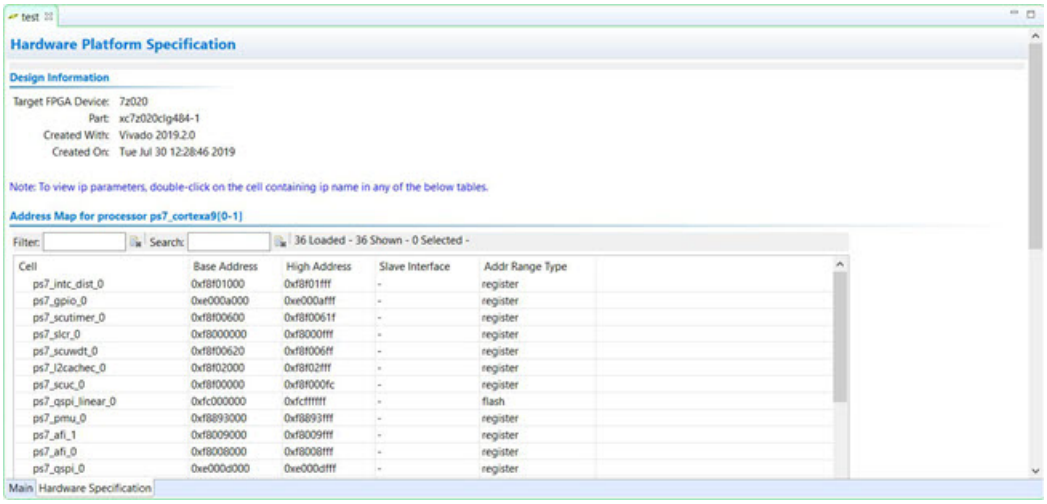
You can create a platform project by using the New Platform Project wizard. To create a platform project, follow these steps:

1. Launch the New Platform Project dialog box using any one of the following methods:
   a. Go to **File → New → Platform Project**.
   b. Click **File → New → Other** to open the New Project wizard. Then select **Xilinx → Platform Project**, and click **Next**. The New Platform Project dialog box appears.
2. Provide a project name in the Project name field.
3. Click **Next**.
4. In the Platform Project dialog box, choose **Create from hardware specification (XSA)** if you have the XSA exported from Vivado®. If you have not built the hardware yet, select **Create from existing platform** and choose one of the pre-defined platforms from the list.
   a. If you choose **Create from hardware specification (XSA)**, use the following steps:
      i. Provide the XSA.
      ii. Select the appropriate operating system and processor to create the platform based on your selection.
      iii. Click **Finish** to create your platform project.
   b. If you choose **Create from existing platform**, use the following steps:
      i. Select a platform from the list of available platforms in the Load Platform Definition dialog box and click **Finish**.

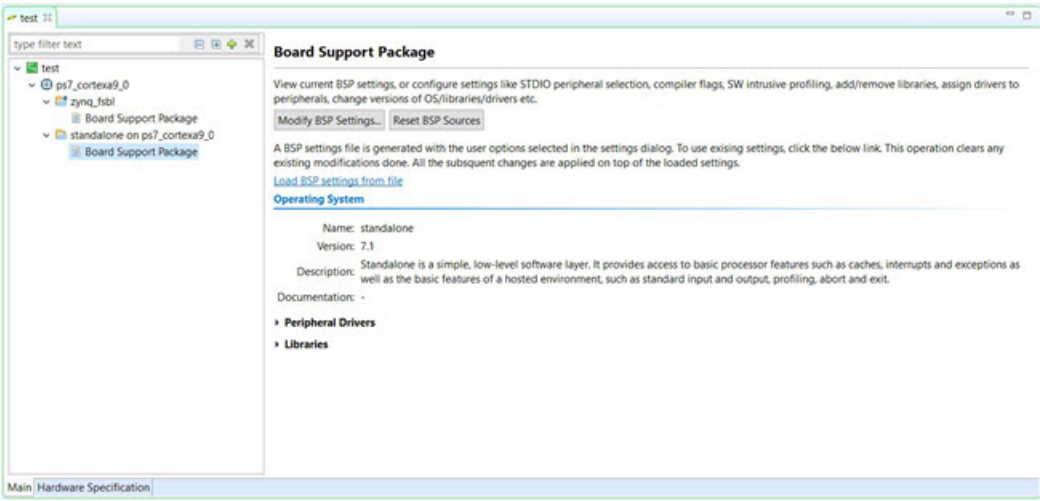
The platform project editor opens.



5. Click to generate the platform. The Console tab shows the status of the platform generation.
6. Optional: To see the hardware specification, switch from the Main tab to the Hardware Specification tab on the bottom left of the platform details window.



7. Optional: You can change the sources and settings by clicking **Board Support Package → Modify BSP Settings** in the platform details window.



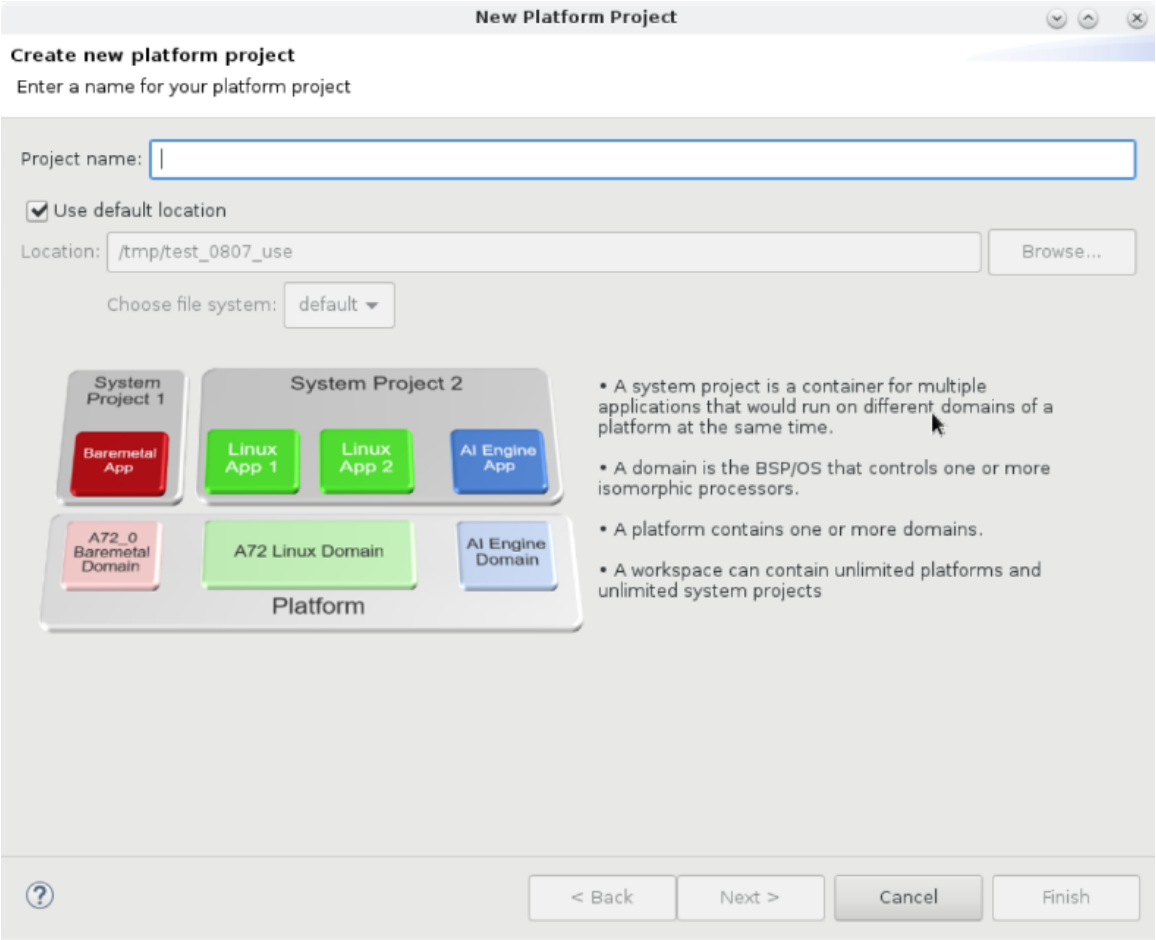
8. Click the **platform.spr** file to reopen the platform project.

Customizing a Pre-Built Platform

A pre-built platform is not editable when it is not in the workspace. To customize a pre-built platform, use the following flow.

1. Launch the New Platform Project dialog box using any one of the following methods:
   a. Go to **File → New → Platform Project**.
   b. Click **File → New → Other** to open the New Project wizard. Then select **Xilinx → Platform Project**, and click **Next**.

   The New Platform Project wizard appears.



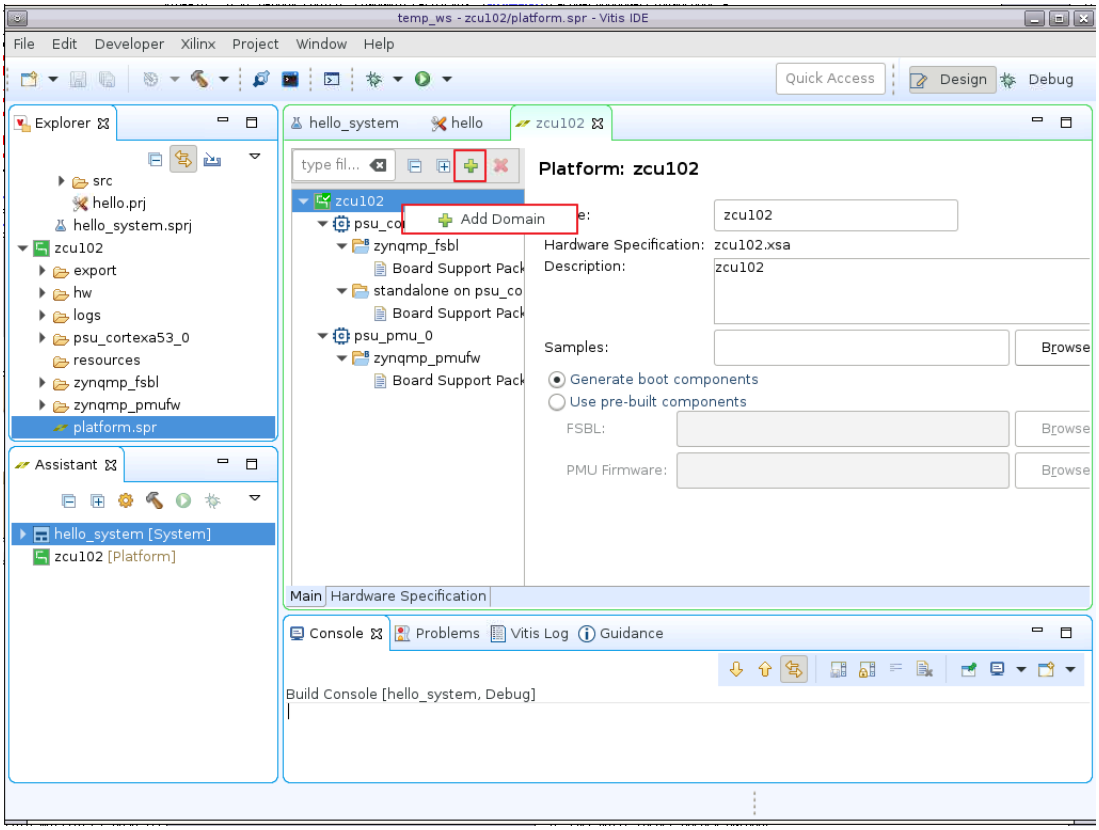
2. Provide a project name in the Project name field.
3. Click **Next**.
4. In the Platform Project page, select **Create from existing platform**. You can add domains to this platform, as described in Adding Domains to a Platform Project.

Adding Domains to a Platform Project

A platform can contain multiple domains. Each domain contains the settings and source files for a processor. The following steps detail how to add a domain the a platform with at least one domain.

1. In the Explorer view, double-click **platform.spr** to open platform settings. This platform might already have one domain.

2. Add a domain by clicking the green plus icon in the Platform Settings window, or right-click the platform name and select **Add Domain**.



3. Fill in the displayed fields and select the desired processor to target.
4. To make the changes take effect, build the domain by clicking the hammer icon on the toolbar, or right-clicking **Platform Project** in the Explorer view and select **Build Project**.

Creating Applications from Domains in a Platform

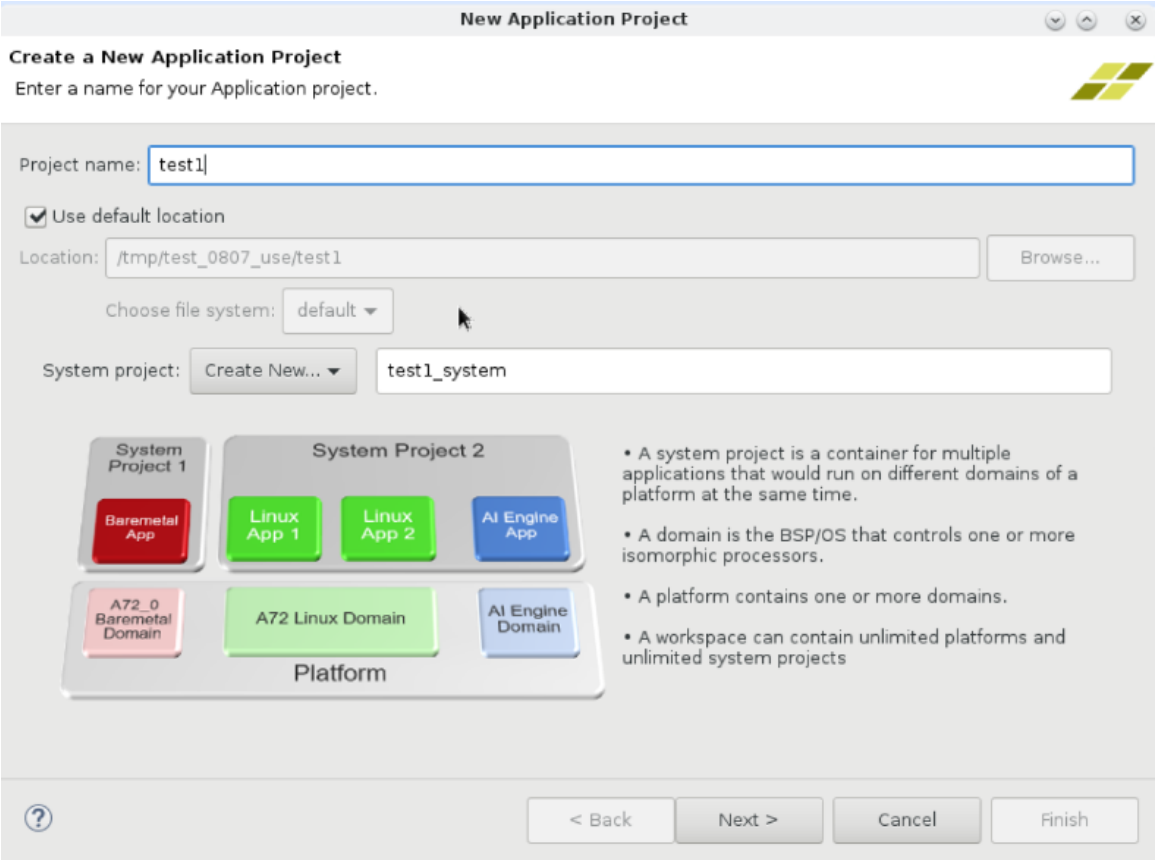
Application projects are the final containers for applications. The project contains or links to C/C++ source files, executable output files, and associated utility files such as the Makefiles used to build the project. Each application project produces one executable file called `<project name>.elf`. You can configure the following items on an application project:

- C/C++ build settings
- Run and debug configurations
- Build configurations

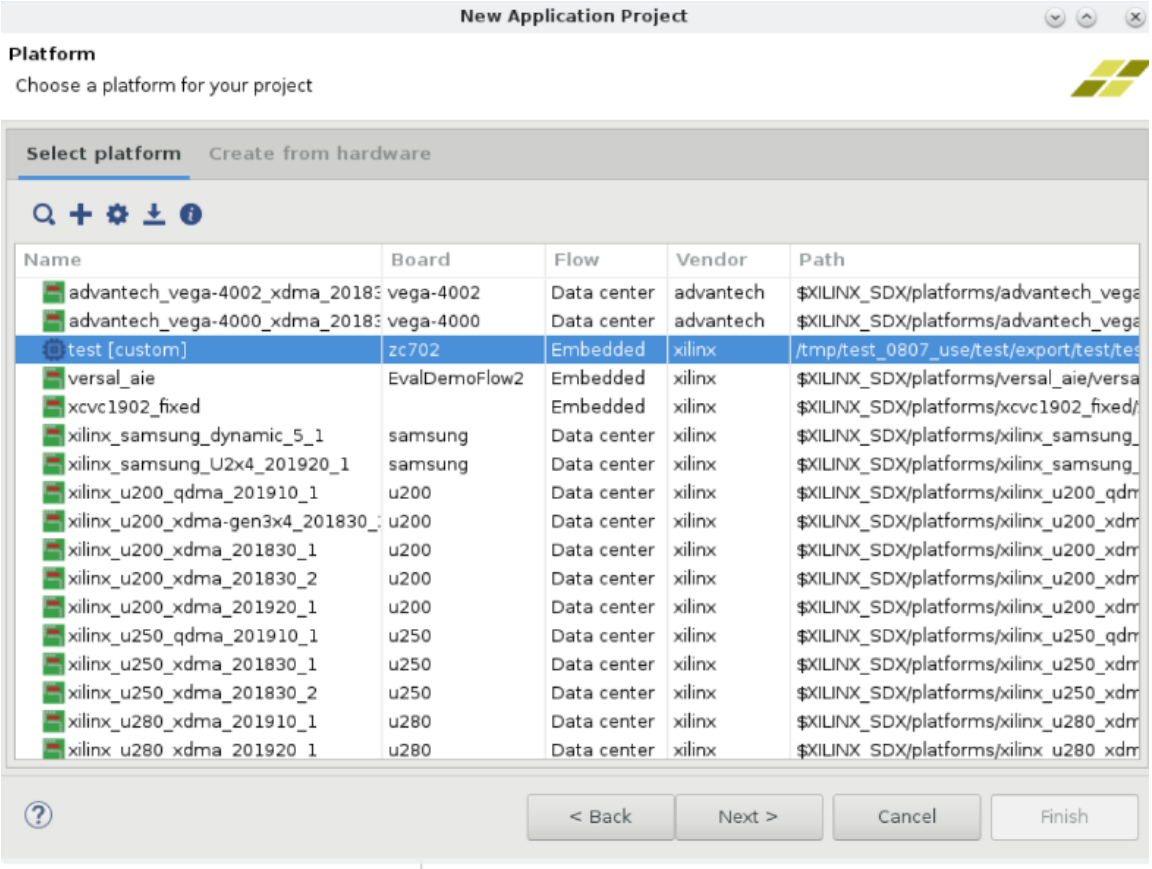
You can create many different applications for a given platform. This allows you to develop software for a given hardware within the same workspace.

To create a C or C++ standalone application project using the New Application Project wizard, use the following steps:

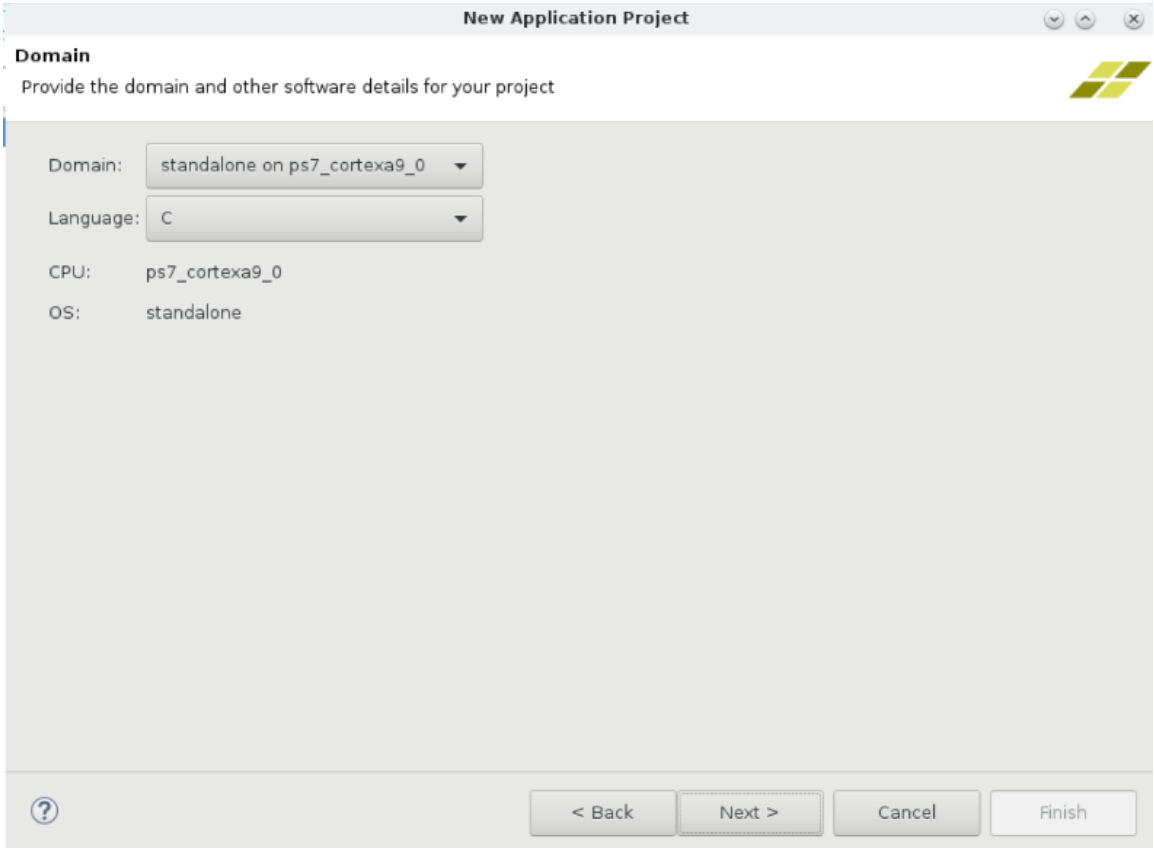
1. Select **File → New → Application Project** to launch the New Application Project wizard.
2. Provide the project name.
3. Click **Next** to create the application from a custom platform.



4. Optional: Add another custom platform by clicking **Add Custom Platform**.



5. Select the platform to see the available domains.
6. Click **Next** to continue. Select the domain in this platform for the application. If the domain you expect is not available in this platform, customize the platform project as described in Customizing a Pre-Built Platform.



7. Click **Next** to see all the available templates for the processor and OS combination.

   The Vitis software platform provides sample applications listed in the Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source, header files, and linker scripts.

8. Select the desired template.

   If you want to create a blank project, select **Empty Application**. You can add C files to the project after the project is created.

9. Click **Finish** to create the application project and platform.

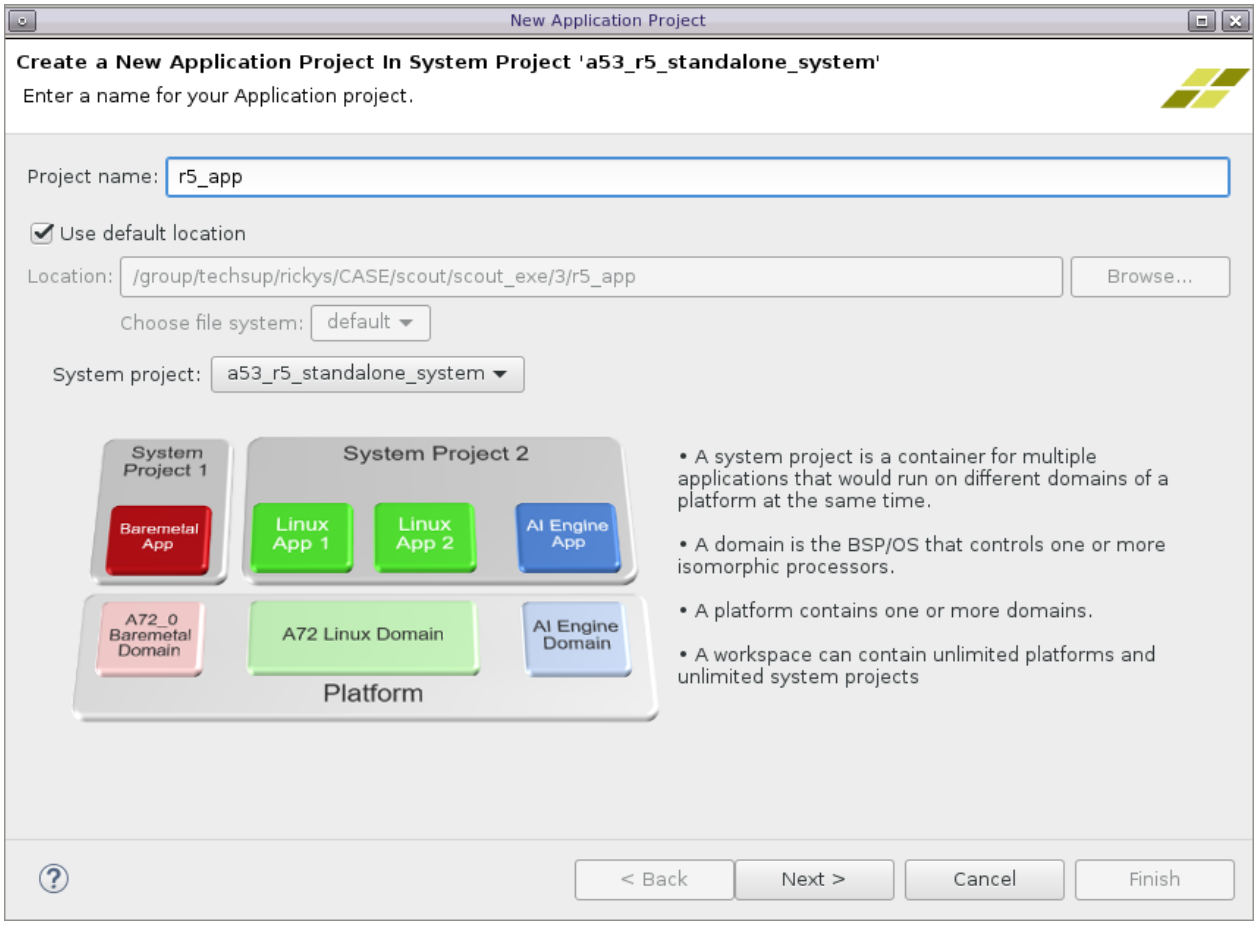
Managing Multiple Applications in a System Project

A system project can contain multiple applications that can run on the device simultaneously. Two applications for the same processor cannot sit together in a system project.

For example, on a Zynq® UltraScale+™ MPSoC device, a Hello World standalone application on A53_0 and a Hello World application on R5_0 can be held in one system project if they are expected to run at the same time. A Hello World standalone application on A53 and a Hello World application in Linux *cannot* be combined in one system project, because these applications use the same A53 processors and cannot run simultaneously on them.

The following steps detail the flow to add two applications to one system project.

1. Create an application with one domain in the platform (see Creating Applications from Domains in a Platform).
2. Create a new application: **File → New → Application Project**.
3. Give a name to this application in the New Application Project wizard.

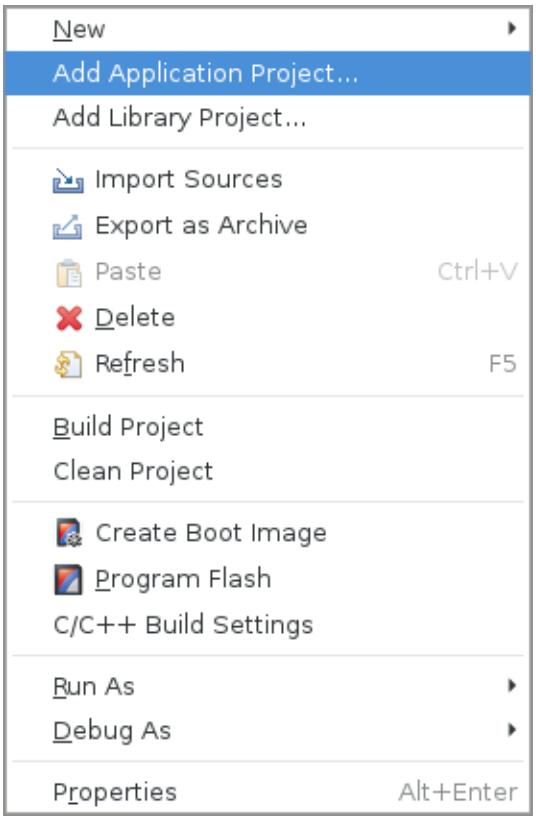


4. From the System project dropdown menu, select an existing system project. It can be the one created in step 1. Click **Next**.
5. Complete the flow detailed in Creating Applications from Domains in a Platform.

The following steps detail the flow to add an application to one system project.

1. Create an application with one domain in the platform (see Creating Applications from Domains in a Platform).

2. Right-click the system project in the Explorer view, and select **Add Application Project**.

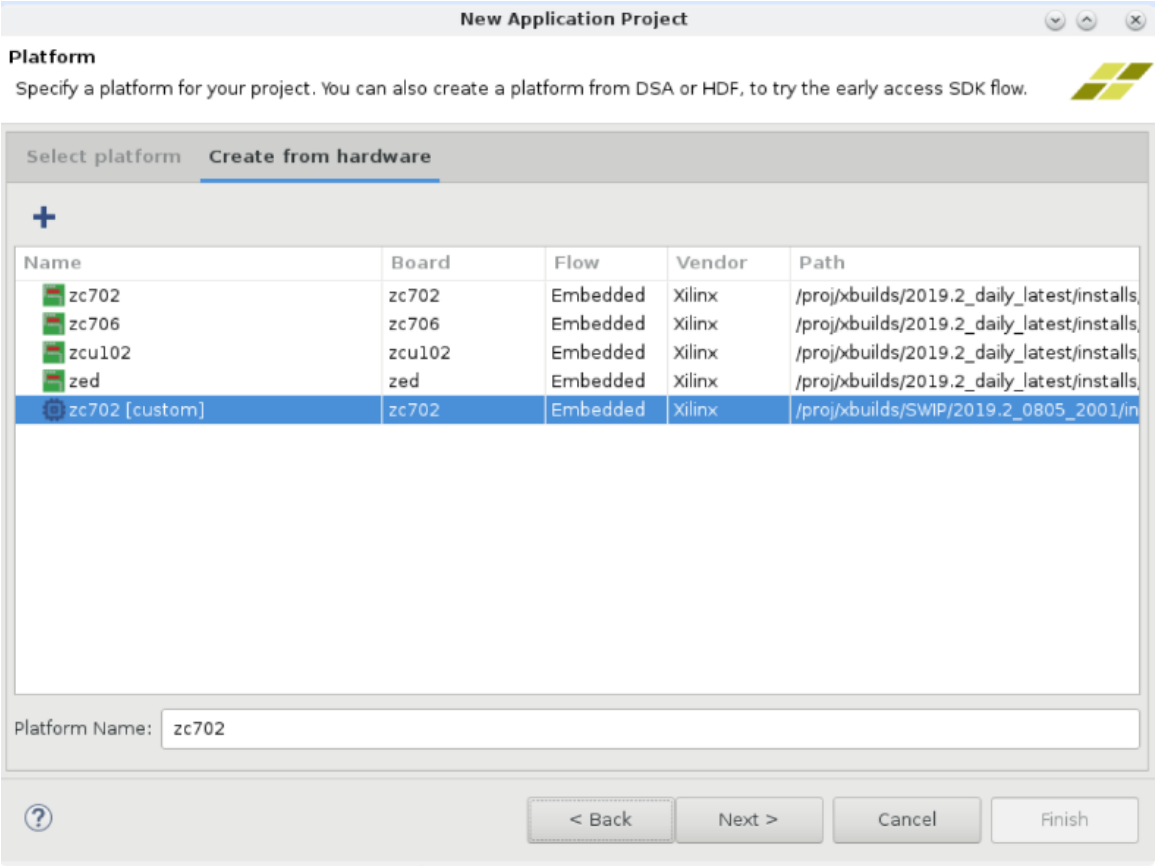


3. Give a name to this application in the New Application Project view.
4. The system project name is automatically updated (no manual change required). Click **Next**.
5. Complete the flow detailed in Creating Applications from Domains in a Platform.

Creating and Building Applications for XSA Exported from the Vivado Design Suite

You can create a C or C++ standalone application project by using the New Application Project wizard.

1. Select **File → New → Application Project** to launch the New Application Project dialog box.
2. Provide a project name.
3. Click **Next**.
4. Click **Create a new platform from hardware (XSA)** to select the XSA.
5. Click the **+** icon, and then click **Next** to add the XSA to the list.



6. Select the processor from the CPU drop-down list.

   Note: This is an important step when there are multiple processors in your design such as any Zynq® devices.

7. Select **standalone** from the OS drop-down list.

   Note: This selection alters what templates you view in the next screen and what supporting code is provided in your project.

8. Select **C** or **C++** as your preferred language.

9. Select **Next** to view all the templates available for the processor and OS combination.

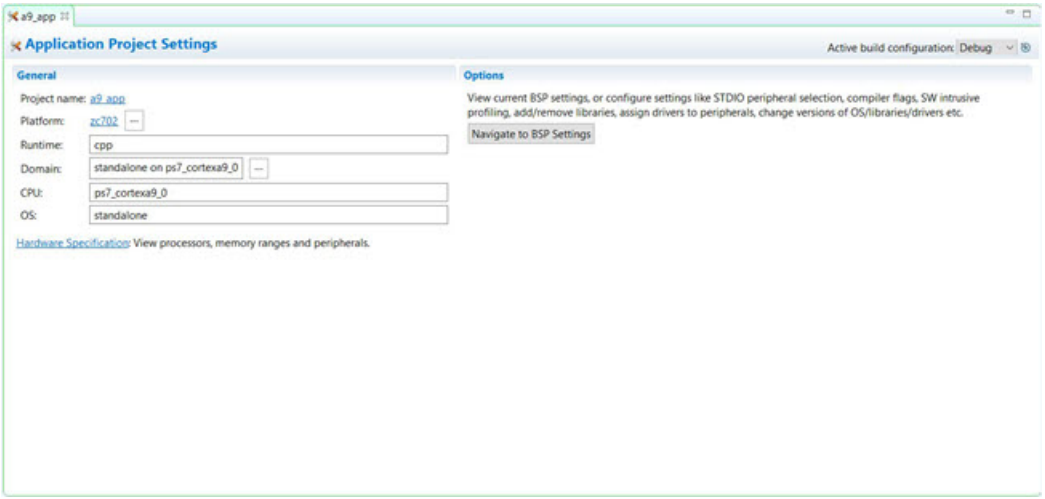
   The Vitis software platform provides useful sample applications listed in the Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source, header files, and linker scripts.

10. Select the desired template.

    If you want to create a blank project, select **Empty Application**. You can add C files to the project after the project is created.

11. Click **Finish** to create your application project and platform.

The application project is created and the appropriate platform is generated in the background. The project editor is as shown in the following figure.



Click **Hardware Specification** to see the hardware peripheral view. Click **Navigate to BSP Settings** to view and change the domain settings.

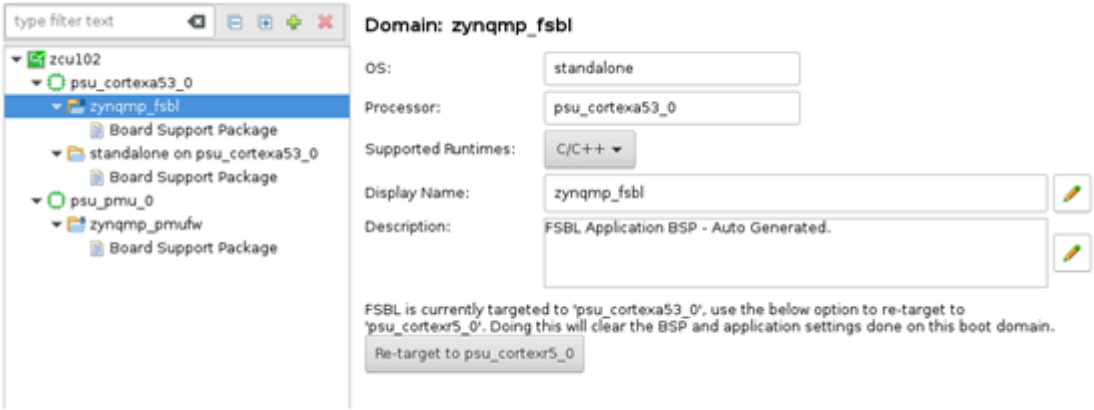
Exporting the XSA files from the Vivado Design Suite

You can export an XSA from the Vivado IDE by clicking **File → Export → Hardware**. Bitstream is optional.

Switching FSBL Targeting Processor

The FSBL in platform is created for Cortex-A53, by default, on a Zynq UltraScale+ MPSoC device. You can re-target it to Cortex-R5F when necessary.

1. Double click **platform.spr**.
2. Select **psu_cortexa53_0 → zynqmp_fsbl**.



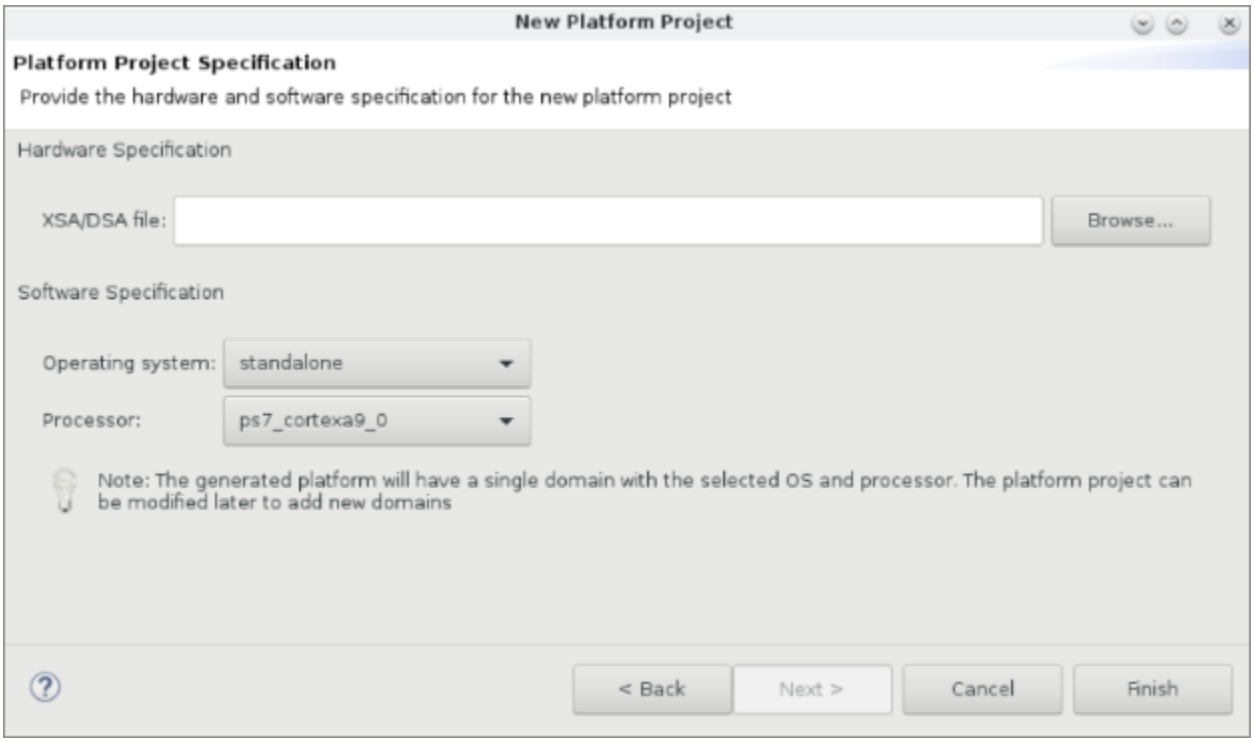
3. Click **Re-target to psu_cortexr5_0**.
4. Build the platform.

Creating Multiple Domains for a Single Hardware

In the Vitis™ software platform, the hardware (XSA) and domains are referred to as the *platform*. A platform is a combination of hardware (XSA) and software (BSPs, boot components like FSBL, etc.) components. BSP or OS are referred to as *domains* in the platform. Each domain can have settings of one processor or a cluster of isomorphism processors, for example, Linux on 4x Cortex™-A53. A platform can contain unlimited domains.

You can create an application using the New Platform Project wizard. To create a platform project, follow these steps:

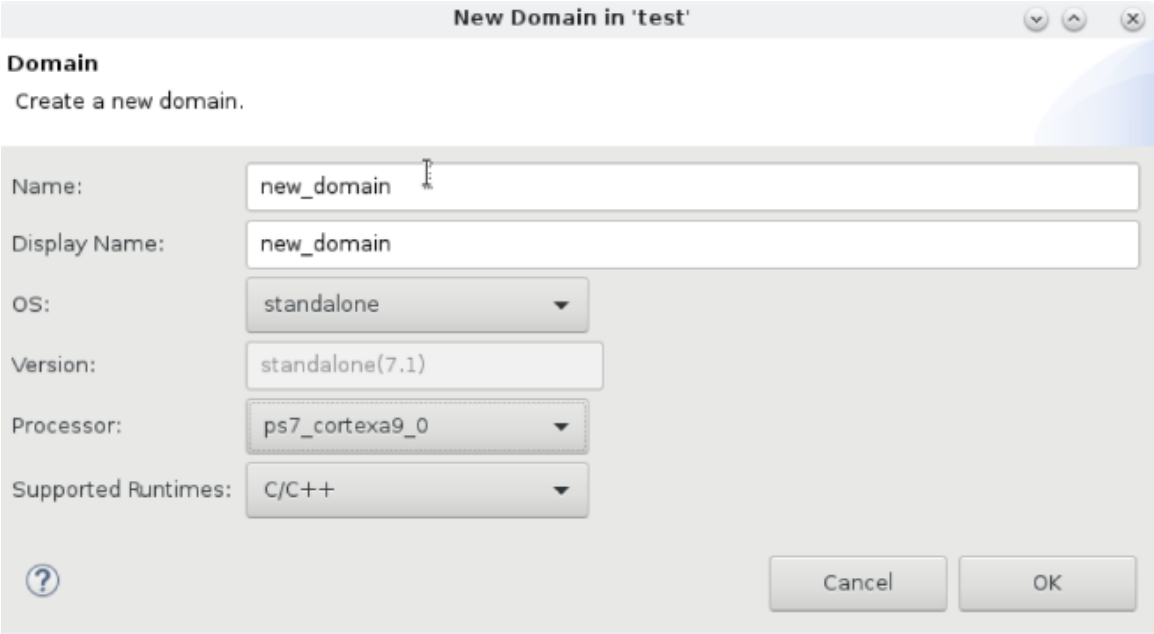
1. Launch the New Platform Project dialog box using any one of the following methods:
   a. Select **File → New → Platform Project**.
   b. Click **File → New → Other** to open the New Project wizard. Then select **Xilinx → Vitis Platform Project**, and click **Next**.
2. Provide a name for the platform you want to create.
3. Click **Next**.
4. Select **Create from hardware specification (XSA)** from the New Platform Project window if you have the XSA exported from Vivado®.
5. Click **Next**.
6. Specify the XSA file to create the platform project.
7. Select the appropriate OS and processor.



8. Click **Finish** to create your platform project.

   The Vitis software platform creates the platform based on the selection. The platform project editor opens. The domain contains boot components.

9. Click to add a new domain.
10. Click **System Configuration** to create multiple system configurations in the existing platform.
11. Fill the Name, Display Name, and Description fields.
12. Click **OK**.
13. Click to add a new domain to the existing system configuration.
14. Fill the Name and Display Name fields in the Domain dialog box.
15. Select an OS from the drop-down list.
16. Select a processor from the Processor drop-down list. This is an important step when there are multiple processors in your design such as Zynq® devices.



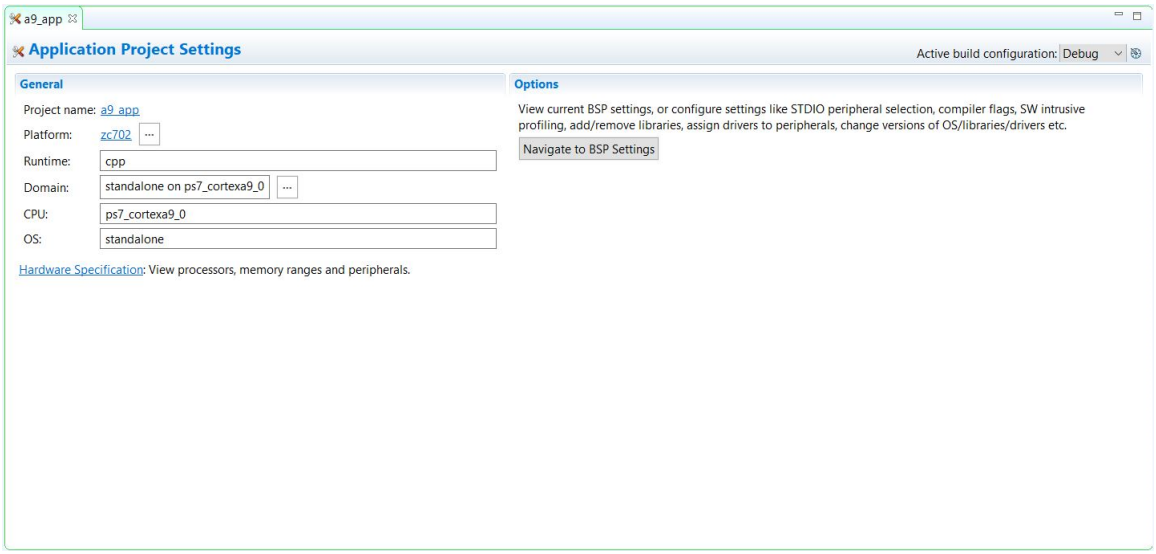
17. Click **OK**.
18. Click to generate the platform.

Changing a Referenced Domain

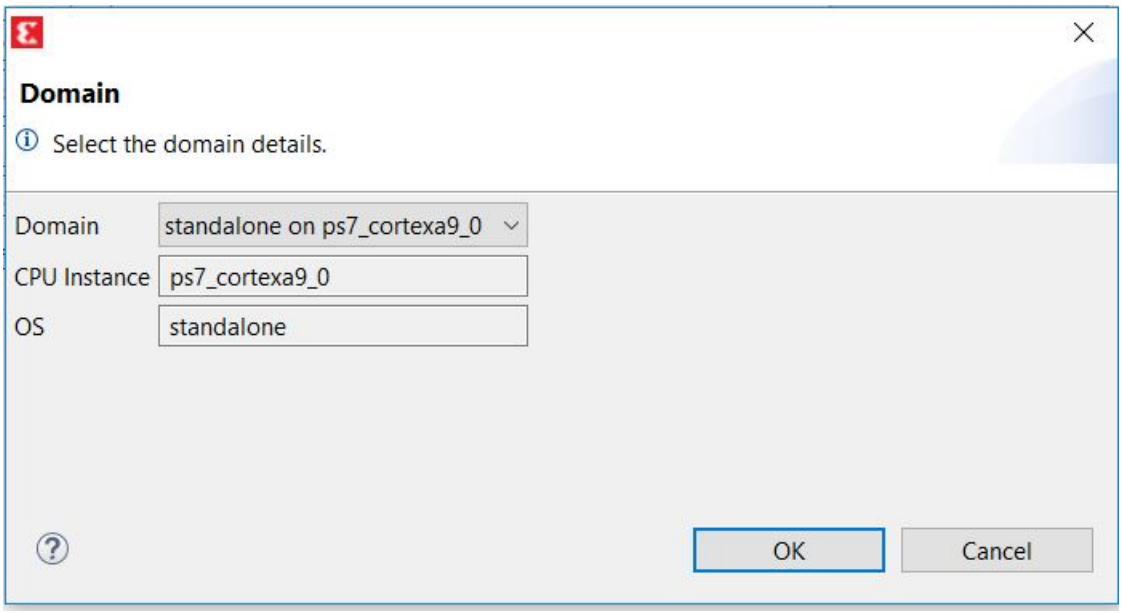
You can re-target an application project to a different platform. The Vitis™ software platform lists all the applicable system configurations available in the re-targeted platform. You must select the right domain from the available domains of a selected system configuration. To change the referenced domain, follow these steps:

IMPORTANT! *The new platform should have domain(s) matching the current domain.*

1. Click the ellipses (**...**) beside the Domain field in the Application Project Settings to see the available configurations in the platform.



2. Select the domain to re-target.

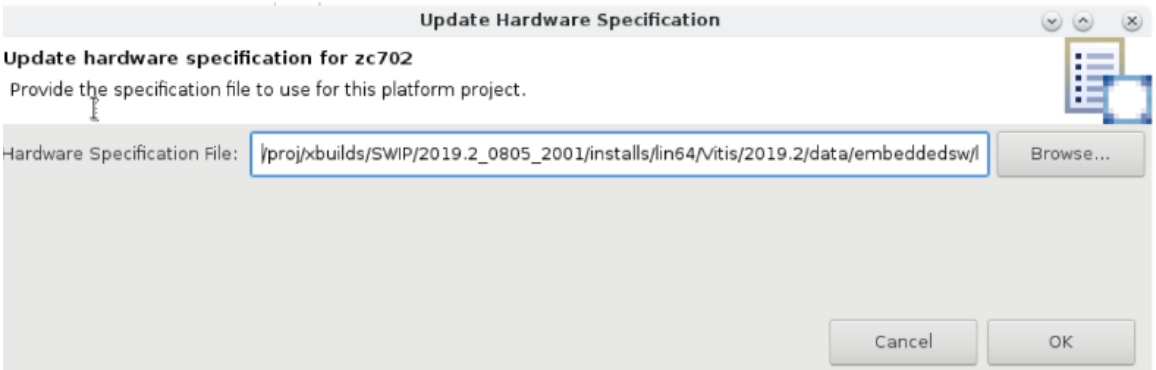


Changing and Updating the Hardware Specification

The Vitis™ software platform allows you to update a platform project with a new hardware by updating the software components under the hood. If your Vivado® project and its exported XSA have been 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 project in the Explorer view, and select **Update Hardware Specification**.
2. Specify the source hardware specification file in the Update hardware specification for test dialog box.



3. Click **OK** to see the hardware specification status.

Debugging the Application on Hardware

The Vitis software platform debugger enables you to examine your code line by line. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system.

The customized Xilinx® system debugger is derived from open-source tools and is integrated with the Vitis software platform.

To debug the application on hardware, follow these steps:

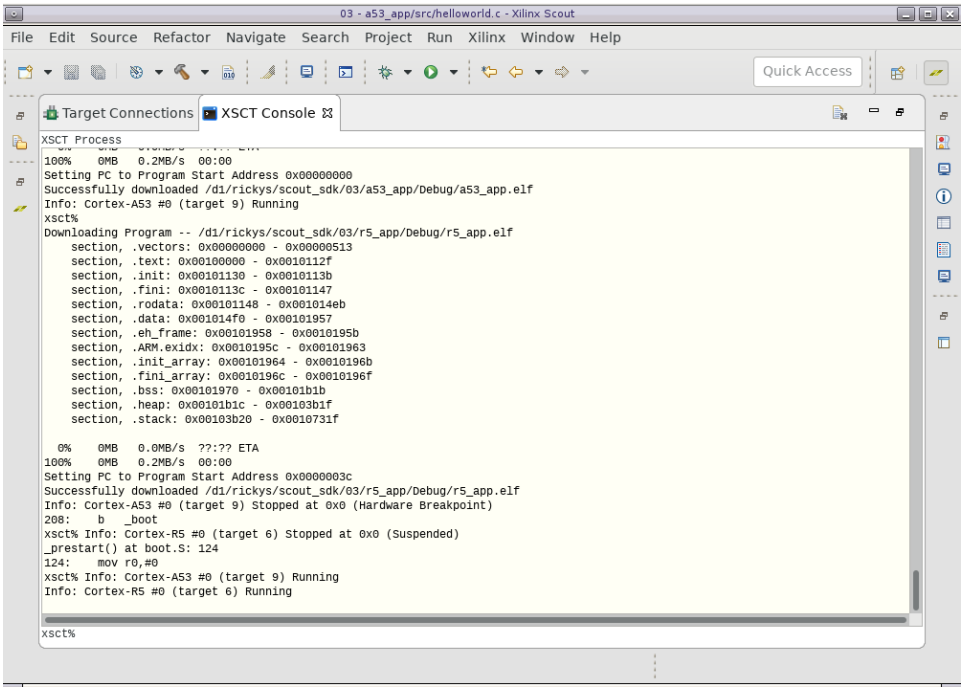
1. Right-click the application project and select **Build Project** to build an application.
2. Right-click on the application project and select **Debug As → Launch on Hardware (Single Application Debug)**.
3. When prompted to switch perspectives, click **Yes** to move to the Debug perspective.

   Operations like step-in, step-into, and more can be done in the Debug perspective. You can check the Breakpoints view, Registers view, Variable view, Memory view, and more in this perspective.

Running and Debugging Applications under a System Project Together

Each application of a system project can run standalone. Applications in a system project can be launched together as well. The Vitis software platform can download them one by one and launch them one after another. In debug mode, all applications stop at main(). The following steps detail how to run applications under a system project together.

1. Right-click the system project in the Explorer view, select **Run as** or **Debug as**, then select **Launch on Hardware (System Project Debug)**.
2. Double-click the **XSCT Console** tab at the bottom right of the IDE to see the detailed commands and logs.



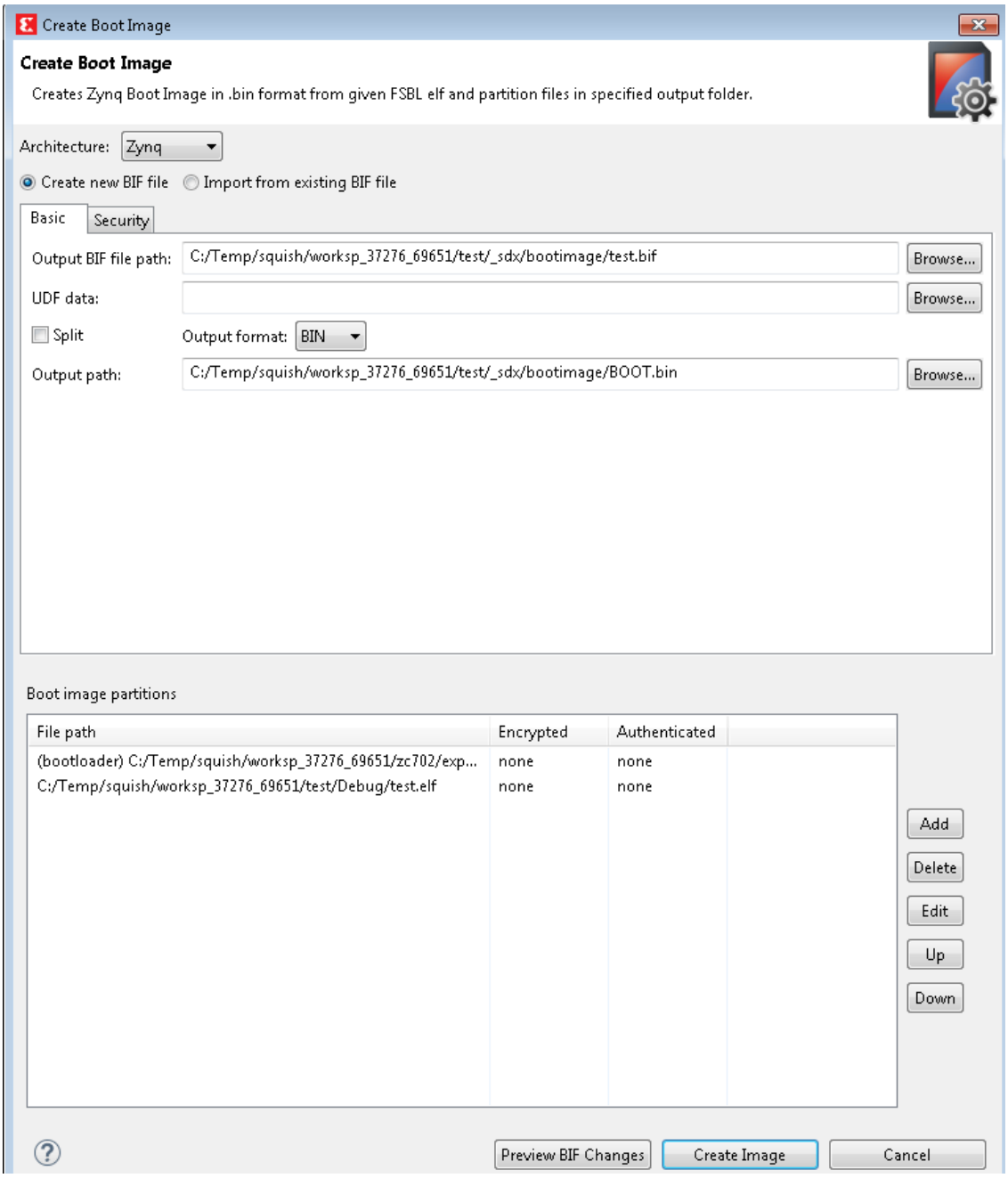
Creating a Bootable Image

When a system project is built, the Vitis software platform builds all applications in the system project and creates a bootable image according to a pre-defined BIF or an auto-generated BIF.

You can create bootable images using Bootgen. In the Vitis IDE, the Create Boot Image menu option is used to create the boot image.

To create a bootable image, follow these steps:

1. Select the application project in Explorer view.
2. Right-click the application and select **Create Boot Image** to open the Create Boot Image dialog box.
3. Specify the boot loader and the partitions.



4. Click **Create Image** to create the image and generate the `BOOT.bin` in the `<Application_project_name>/_ide/bootimage` folder.

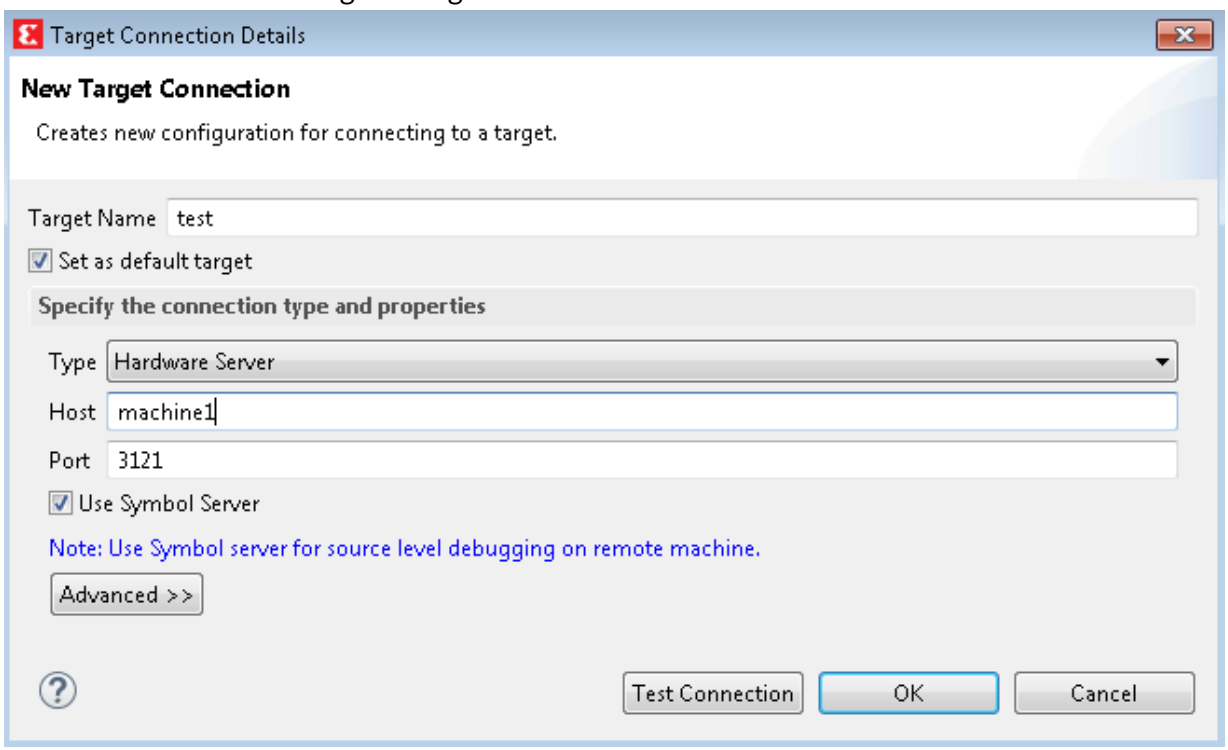
Flash Programming

Program Flash is a Vitis™ software platform tool used to program the flash memories in the design. The types of flash supported by the Vitis software platform for programming are:

- For non-Zynq® family devices: Parallel Flash (BPI) and Serial Flash (SPI) from Micron and Spansion.
- For Zynq family devices: Quad SPI, NAND, and NOR. QSPI can be used in different configurations such as QSPI single, QSPI dual parallel, QSPI dual stacked.

To program the flash memories, follow these steps:

1. Connect to the board using the target connections icon



2. Select the application in which you created the boot image.
3. Select **Xilinx → Program Flash**.
4. Fill the required information.
5. Select the appropriate target connection.
6. Select the flash type.
7. Click **Flash** to start the program flash operation. After the operation is complete and you can see the status of the flash programming, check it in the Vitis software platform log.

Generating Device Tree

The Vitis™ IDE can generate device trees. To generate a device tree, follow these steps:

1. Select **Xilinx → Repositories**.
2. Click **New**.
3. Provide the local path for the device tree generato, which can be downloaded from GitHub.
4. Select **Xilinx → Generate Device Tree** to open the Device Tree Generator.
5. Provide the hardware specification file and the output directory (the output will be created here).

   You can change the settings for device tree blob (DTB) using the Modify Device Tree settings. The device tree path displays after successful generation.

Debugging an Application using the User-Modified/Custom FSBL

Creating a Hello World Application

1. Select **File → New → Application Project**.
2. Provide a name for your project in the project name field.
3. Select the platform that you created and generate the project.
4. Click **Next**.
5. Provide the system configuration and software details and click **Next**.
6. Select a template to create your project (Example: Hello World).
7. Click **Finish** to build the application project.

Modifying the Source Code of the FSBL in Platform

The source code of FSBL in platform can be modified in place. Building platform again compiles the FSBL in platform. For Zynq UltraScale+ MPSoC, the FSBL source code is located in `<Platform>/zynqmp_fsbl` and for Zynq-7000, the FSBL source code is located in `<Platform>/zynq_fsbl`. After you modify the source code, build the platform again to compile the FSBL in platform.

Modifying the BSP Settings of the FSBL in Platform

To modify the BSP settings of the FSBL, perform the following steps.

1. Double-click **platform.spr**.
2. Select **Board Support Package** on the platform page that opens.
3. Click **Modify BSP Settings**. In the dialog box that opens, you can modify the options and click **OK** to update the settings.
4. Select the platform in the Explorer view and build the platform using the icon.

Debugging the "Hello World" Application using the Modified FSBL

1. Right-click the application project and select **Debug As → Debug Configurations**.
2. Double-click **Single Application Debug** to create a new debug configuration.
3. Click the **Target Setup** tab.
4. Check the **Use FSBL flow for initialization** check-box.
5. Click **Debug** to switch the perspective.
6. Select **Yes** to open the debug perspective.
7. Browse to the modified FSBL .elf file path for FSBL File.
8. Click **Debug** to switch the perspective.
9. When prompted, select **Yes** to open the Debug perspective.

Modifying the Domain Sources (Driver and Library Code)

To add/modify the domain sources (driver and library code) using the 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 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 Vitis™ software platform automatically infers all the components contained within the repository and makes them available for use in its environment.

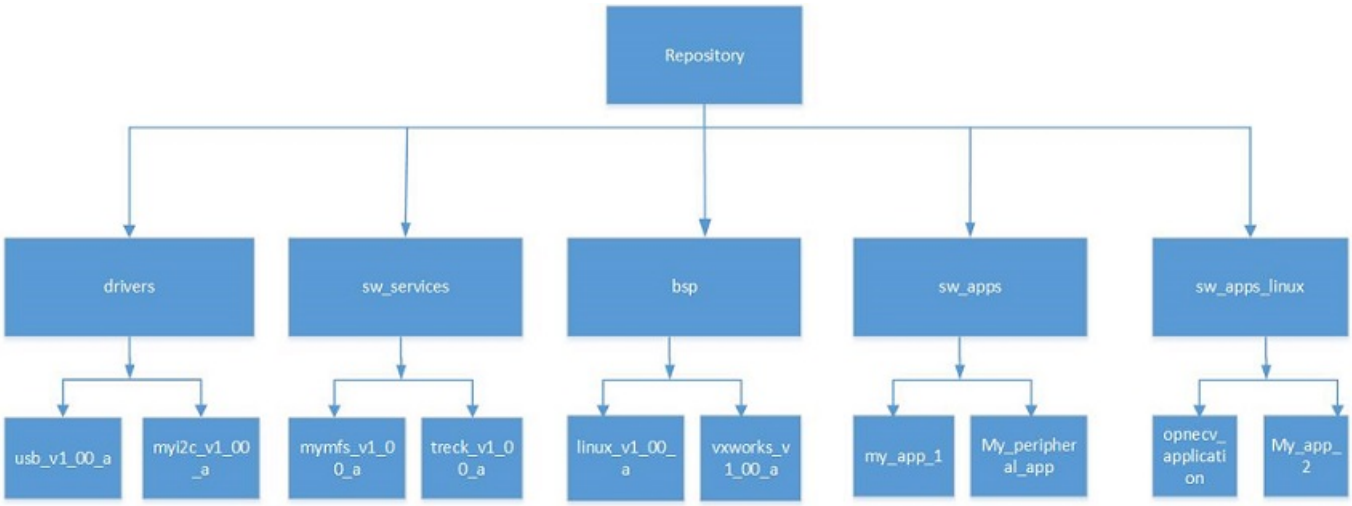
Your Vitis software platform workspace can point to multiple software repositories. The scope of the software repository can be global (available across all workspaces) or local (available only to the current workspace). Components found in any local software repositories added to a Vitis software platform 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 5: **Repository Structure**

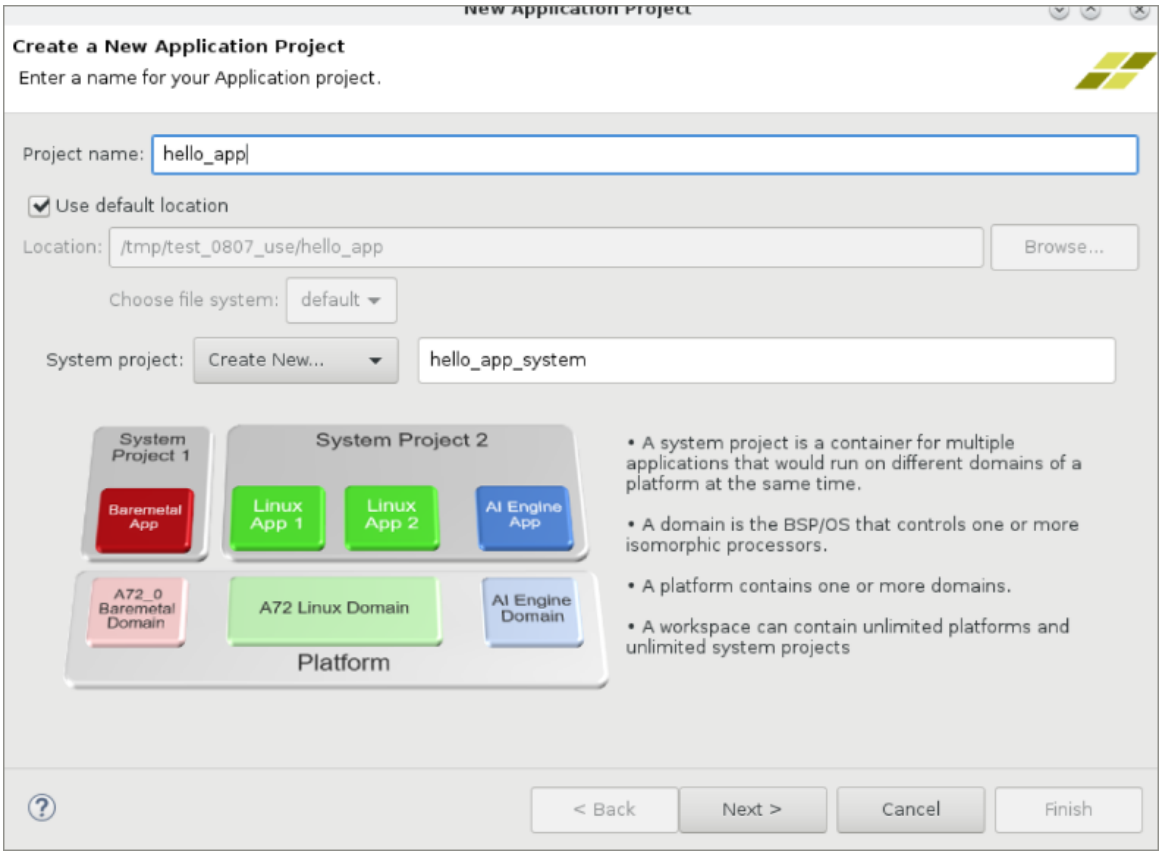


Adding the Repository

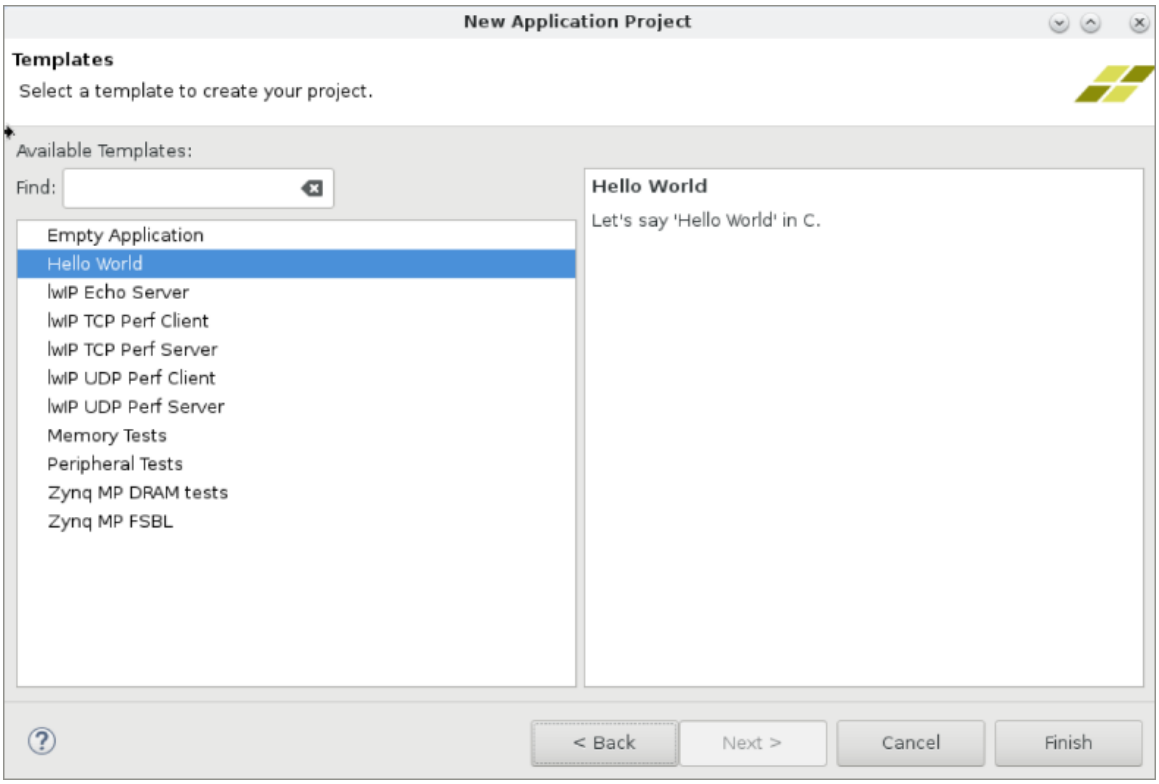
1. Select **Xilinx → Repositories**.
2. To add the repository you created in Creating a Repository, follow one of these two steps:
   - To ensure that your repository driver/library repository is limited to the current workspace, click **New** to add it under Local Repositories.
   - To ensure that your repository driver/library repository is available across all workspaces, click **New** to add it under Global Repositories.
3. Select **Apply and Close** to add the custom drivers/libraries from the repositories.

Creating the Application Project

1. Select **File → New → Application Project**.
2. Provide a project name in the Project name field. Click **Next** to open the New Application Project wizard.



3. Select an existing platform or select **Create from XSA** to see the list of all the available hardware designs in installation.
4. Select a hardware design file and click **Next**.
5. A list of supported CPUs and operating systems is displayed. Select **cortexa53_0** from the CPU drop-down list if you want to debug the A53 application.
6. Select **standalone** from the OS drop-down list. Click **Next**.
7. A list of supported application projects for this operating system and processor combination is displayed. Select any of the application templates from the list and click **Finish**.



8. Select **Overview → Drivers** to see a list of available drivers/libraries for the IP. You can modify/edit the driver options by clicking the drop-down icons.
9. Build the application project.
10. Debug the application by selecting **Debug As → Launch on Hardware (Single Application Debug)**.

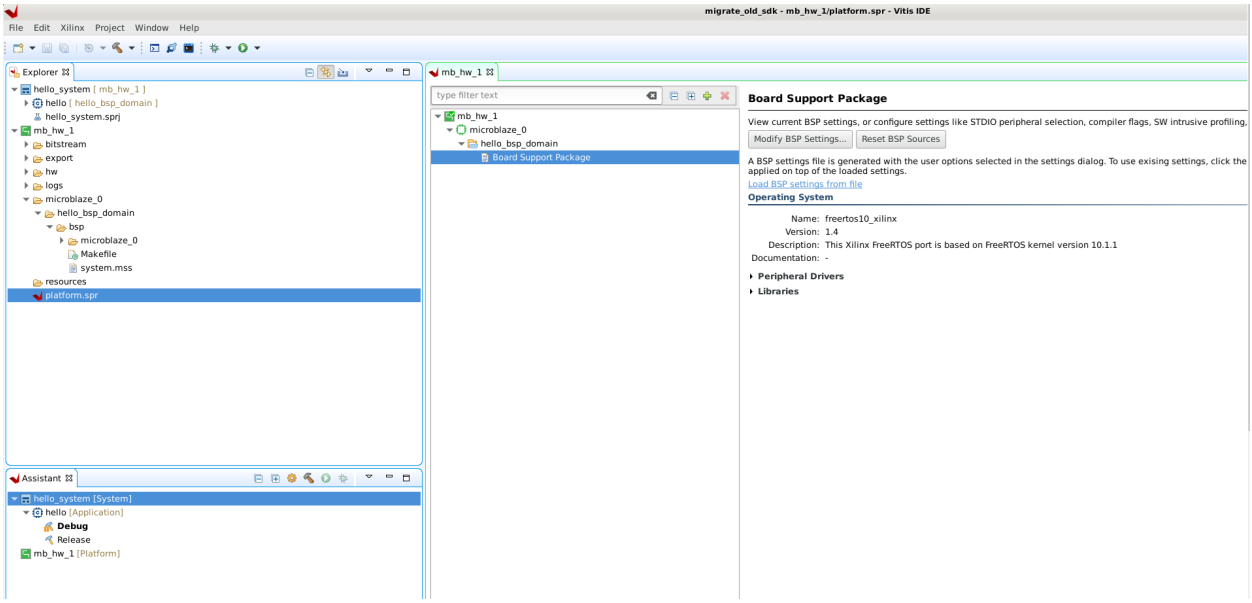
Migrating an SDK Project to the Vitis Software Platform

To migrate an SDK project to the Vitis software platform, perform the following steps.

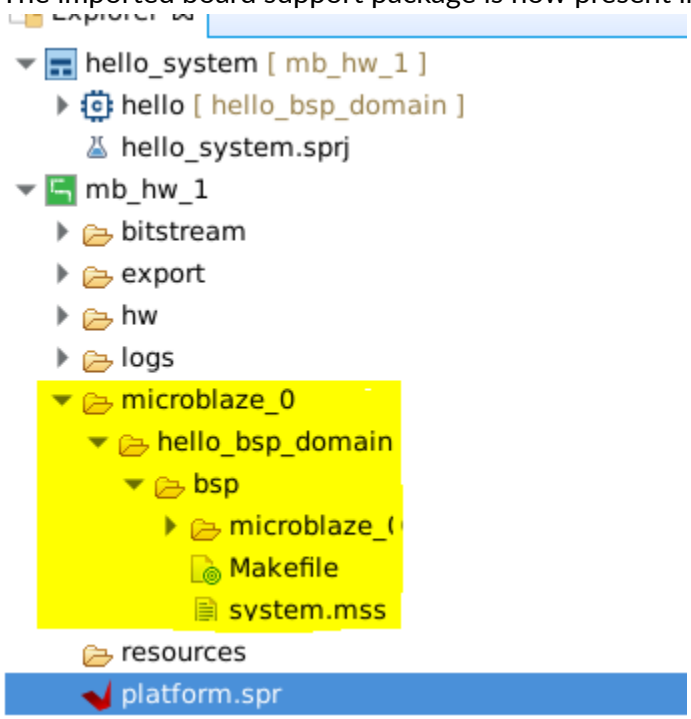
IMPORTANT! *Remember to back up your project before migration.*

1. Launch the Vitis IDE.
2. Import the SDK project by navigating to **File → Import**.
3. In the Import Projects window, select **Eclipse workspace or zip file** at the import type, and click **Next**.
4. In the next window, select the root directory and projects to be imported.

5. After a successful import, you can see the imported application project(s) or platform project(s) as shown below.



The imported board support package is now present inside the platform project.



6. You now need to update the hardware specification to the latest version. Right-click on the platform project, and click **Update Hardware Specification**.
7. Select the XSA file that was exported from Vivado and click **OK**.
8. After the update, an Out-of-date tag appears next to the platform project. Right-click on the platform project and then select **Build Project**. Build the application project again.

Section III

Bootgen Tool

Chapter 11

Introduction

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

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

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

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

Installing Bootgen

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

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

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

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

Boot Time Security

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

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

- *Zynq-7000 SoC Technical Reference Manual* (UG585)
- *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)

See Using Encryption and Using Authentication for more information about encrypting and authenticating content when using Bootgen.

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

Chapter 12

Boot Image Layout

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

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

Building a boot image involves the following steps:

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

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

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

Zynq-7000 SoC Boot and Configuration

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

BootROM on Zynq-7000 SoC

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

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

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

Zynq-7000 SoC Boot Image Layout

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

Figure 6: **Boot Header**

<table>
<tr><td colspan="4">Boot Header</td></tr>
<tr><td colspan="4">Register Initialization Table</td></tr>
<tr><td colspan="4">Image Header Table</td></tr>
<tr><td>Image Header 1</td><td>Image Header 2</td><td>...</td><td>Image Header n</td></tr>
<tr><td>Partition Header 1</td><td>Partition Header 2</td><td>...</td><td>Partition Header n</td></tr>
<tr><td colspan="4">Header Authentication Certificate (Optional)</td></tr>
<tr><td colspan="3">Partition 1 (FSBL)</td><td>AC (Optional)</td></tr>
<tr><td colspan="3">Partition 2</td><td>AC (Optional)</td></tr>
<tr><td colspan="4">.
.</td></tr>
<tr><td colspan="3">Partition n</td><td>AC (Optional)</td></tr>
</table>

Zynq-7000 SoC Boot Header

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

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

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

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

Table 11: **Zynq-7000 SoC Boot Header**

Address Offset	Parameter	Description
0x00-0x1F	Arm® Vector table	Filled with dummy vector table by Bootgen (Arm Op code `0xEAFFFFFE`, 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`.

Table 11: **Zynq-7000 SoC Boot Header** *(cont'd)*

Address Offset	Parameter	Description
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 Intialization Pairs and INT File Attributes.

Table 12: **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 which is common across all these images, and information like; the number of images, partitions present in the boot image, and the pointer to the other header tables. The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC device.

Table 13: **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)

Table 13: **Zynq-7000 SoC Image Header Table** *(cont'd)*

Address Offset	Parameter	Description
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 will also contain the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC device.

Table 14: **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 Zynq®-7000 SoC Partition Header.

Note: An ELF file with three (3) loadable sections has one image header and three (3) partition header tables.

Table 15: **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
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 a Zynq®-7000 SoC device.

Table 16: **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

Table 16: **Zynq-7000 SoC Partition Attribute Bits** *(cont'd)*

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

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

Table 17: **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)

Table 17: **Zynq-7000 SoC Authentication Certificate** *(cont'd)*

Authentication Certificate Bits		Description
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, SHA-256 (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 Zynq®-7000 SoC Authentication Certificate Header.

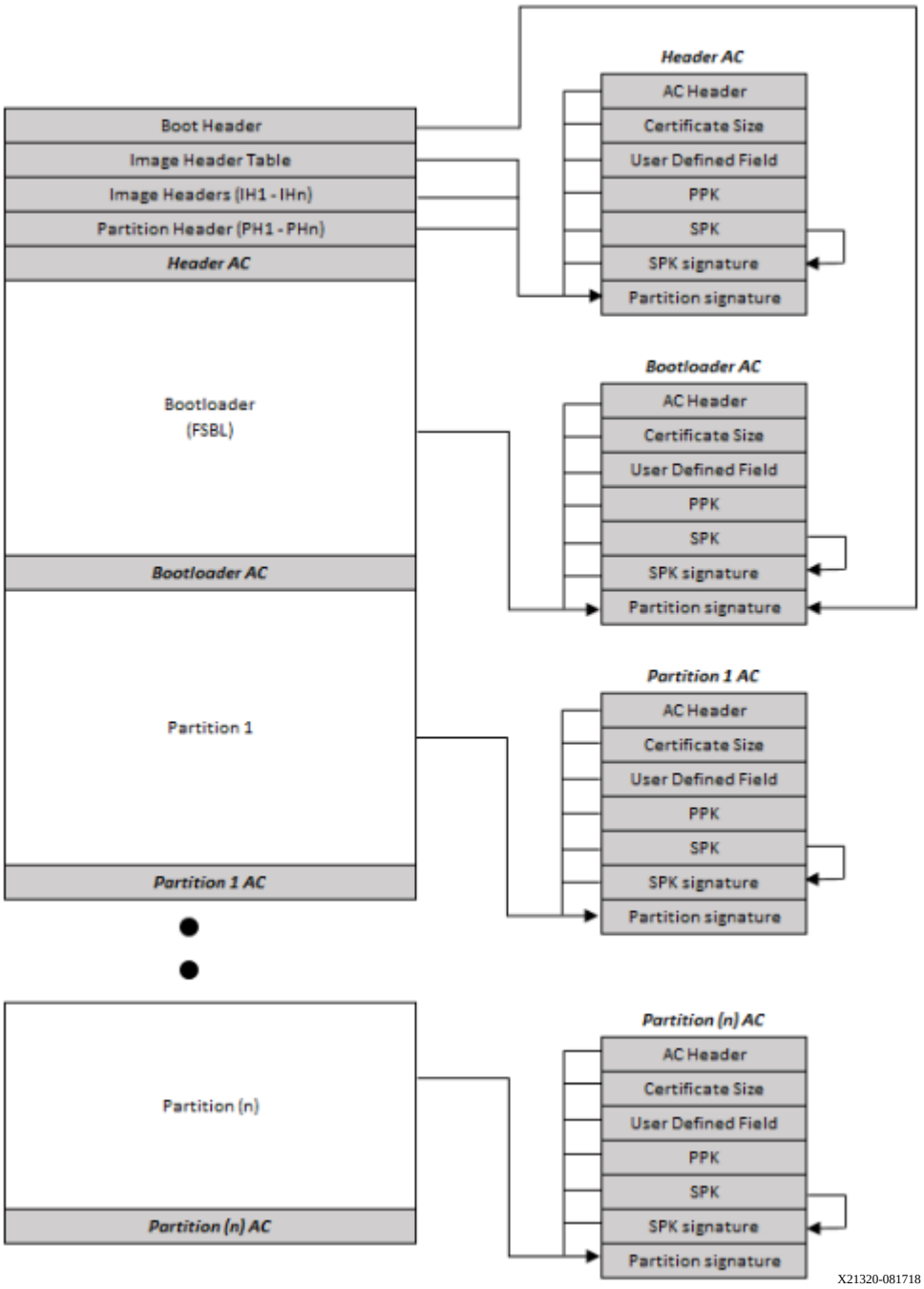
Table 18: **Zynq-7000 SoC Authentication Certificate Header**

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

Zynq-7000 SoC Boot Image Block Diagram

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

Figure 7: **Zynq-7000 SoC Boot Image Block Diagram**



Zynq UltraScale+ MPSoC Boot and Configuration

Introduction

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

During initial boot, the following steps occur:

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

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

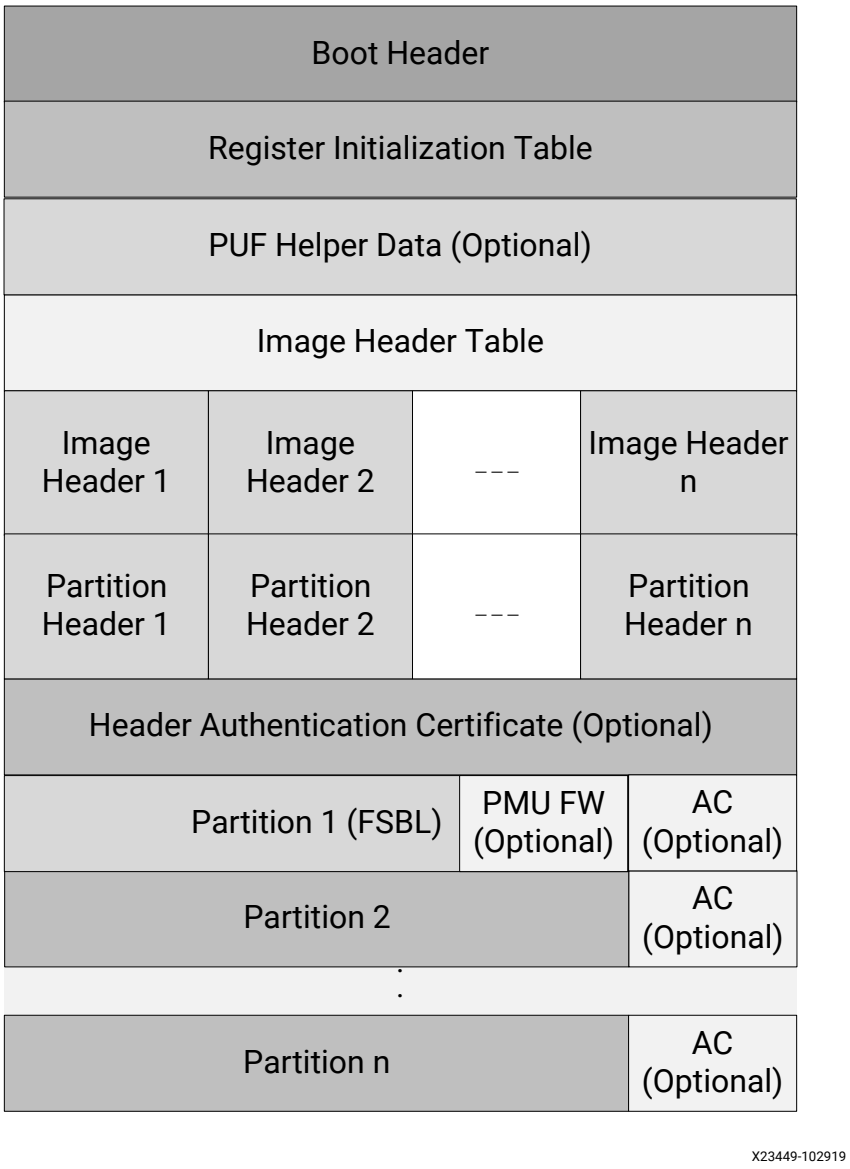
- Checks to determine if authentication is required by the FSBL or the user application.
- Performs an authentication check and proceeds only if the authentication check passes. Then checks the image for any encrypted partitions.
- If the CSU detects partitions that are encrypted, the CSU performs decryption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

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

Zynq UltraScale+ MPSoC Boot Image

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

Figure 8: **Zynq UltraScale+ MPSoC Boot Image**



Zynq UltraScale+ MPSoC Boot Header

About the Boot Header

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

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

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

Table 19: **Zynq UltraScale+ MPSoC Device Boot Header**

Address Offset	Parameter	Description
0x00-0x1F	Arm® vector table	XIP ELF vector table: 0xEAFFFFFE: 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 FW 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 19: **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 20: **Zynq UltraScale+ MPSoC Boot Header Attribute Bits**

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

Table 20: **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 About Register Intialization Pairs and INT File Attributes.

Table 21: **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 this link to the section on "PUF Helper Data" in *Zynq UltraScale+ Device Technical Reference Manual* (UG1085).

Table 22: **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 23: **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	1st Partition Header Offset	Pointer to first partition header. (word offset)
0x0C	1st 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 will also contain the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the Zynq® UltraScale+™ MPSoC.

Table 24: **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 `0xFFFFFFF` 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 Zynq® UltraScale+™ MPSoC.

Table 25: **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 25: **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 Zynq® UltraScale+™ MPSoC.

Table 26: **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 26: **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 26: **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 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 27: **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 || UDF || PPK || SPK || SPK Signature))	

Note: FSBL Signature is calculated as follows:

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

Zynq UltraScale+ MPSoC Authentication Certification Header

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

Table 28: **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 9: **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	Key0	IV0+0x01	Key1	IV1	Key0	IV0+0x02	Key1	IV1
Block #0	Key0	IV1	-	-	Key1	IV1	-	-	Key1	IV1	-	-

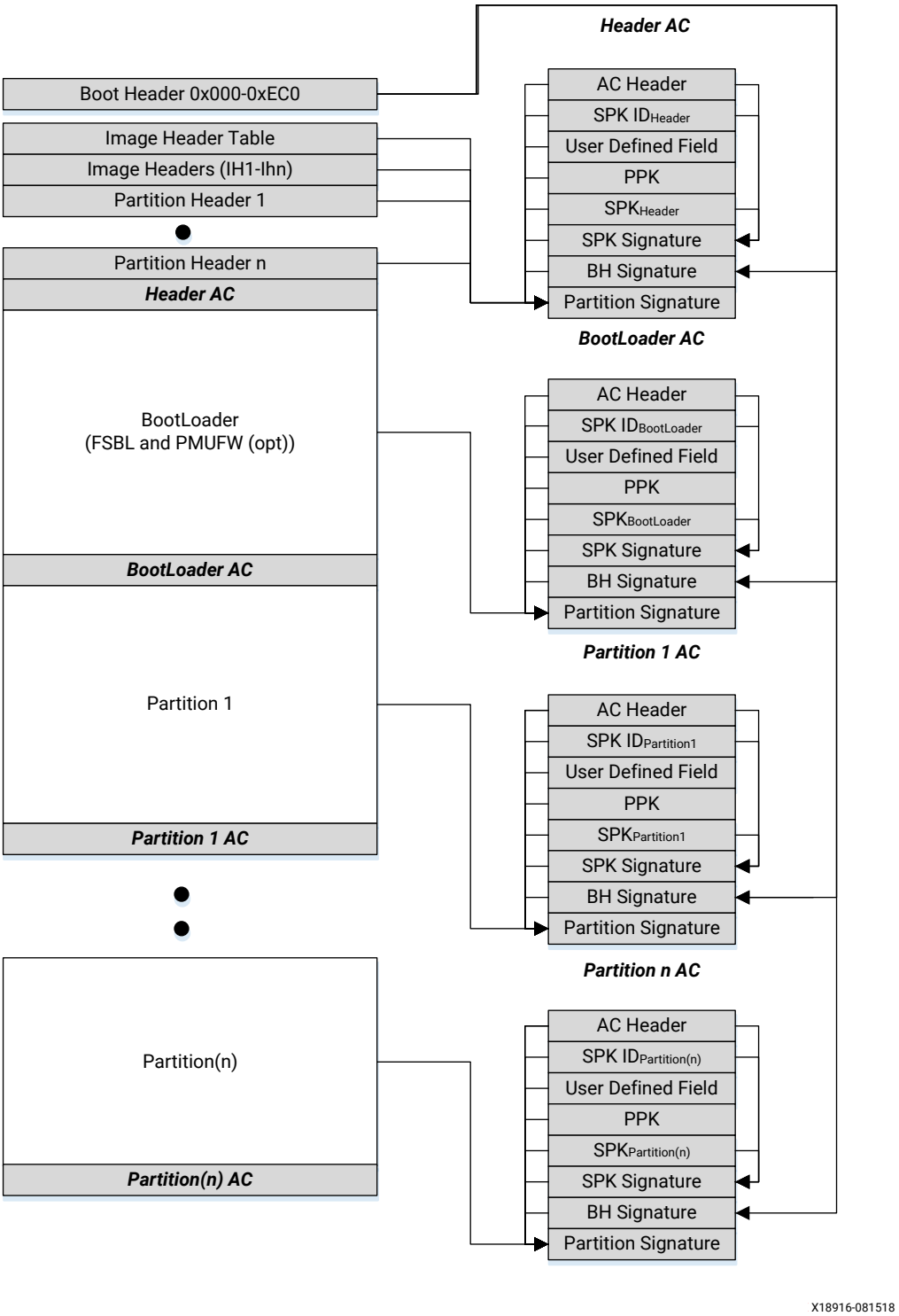
AES with Key rolling

	Partition#0 (FSBL)				Partition#1				Partition#2			
	Encrypted Using		Contents		Encrypted Using		Contents		Encrypted Using		Contents	
Secure Header	Key0	IV0	-	IV1	Key0	IV0+0x01	Key1	IV1	Key0	IV0+0x02	Key1	IV1
Block #0	Key0	IV1	Key 2	IV2	Key1	IV1	Key2	IV2	Key1	IV1	Key2	IV2
Block #1	Key2	IV2	Key 3	IV3	Key2	IV2	Key 3	IV3	Key2	IV2	Key 3	IV3
Block #2	Key3	IV3	Key 4	IV4	Key3	IV3	Key 4	IV4	Key3	IV3	Key 4	IV4
...	...	...	...	...	...	...	...	...	...	...	...	...

Zynq UltraScale+ MPSoC Boot Image Block Diagram

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

Figure 10: **Zynq UltraScale+ MPSoC Device Boot Image Block Diagram**



Chapter 13

Creating Boot Images

Boot Image Format (BIF)

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

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

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

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

- First Stage Boot Loader (FSBL)
- Secondary Stage Boot Loader (SSBL) like U-Boot
- Bitstream
- Linux

- Software applications to run on processors
- User Data
- Bootgen generated boot image

BIF Syntax and Supported File Types

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

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

    // partition attributes
    [attribute2, attribute3=<argument>] <elf>
    [attribute2, attribute3=<argument>, attibute4=<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>,
        attibute=<argument>
    ] <bit>

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

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

Bootgen Supported Files

The following table lists the Bootgen supported files.

Table 29: **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	
	`.pk1/.pub/.pem`	RSA key	
	`.sig`	Signature files	Signature files generated by bootgen or HSM.
Zynq/Zynq UltraScale+ MPSoC/FPGA	`.bit/.rbt`	Bitstream	Strips the BIT file header.

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 Xilinx® devices uses that attribute:

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

Table 30: **Bootgen Attributes and Description**

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

Table 30: **Bootgen Attributes and Description** *(cont'd)*

Option/Attribute	Description	Used By
bh_keyfile <filename>	256-bit obfuscated key or black key to be stored in the Boot Header. This is only valid when `[keysrc_encryption]=bh_gry_key` or `[keysrc_encryption]=bh_blk_key`.	zynqmp
bhsignature <filename>	Imports Boot Header signature into authentication certificate. This can be used if you do not want to share the secret key PSK. You can create a signature and provide it to Bootgen. The file format is `bootheader.sha384.sig`	zynqmp
big_endian	To specify the binary file is in big endian format	zynqmp
blocks <block sizes>	Specify block sizes for key-rolling feature in Encrytion. 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
boot_device <options>	Specifies the secondary boot device. Indicates the device on which the partition is present. Options are: qspi32 qspi24 nand sd0 sd1 sd-ls sdls mmc usb ethernet pcie sata ***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 TRM.	zynqmp
bootimage <filename.bin>	Specifies that the listed input file is a boot image that was created by Bootgen.	zynq zynqmp
bootloader <partition>	Specifies the partition is a bootloader (FSBL). This attribute is specified along with other partition BIF attributes.	zynq zynqmp
bootvector <vector_values>	Specifies the vector table for execute in place (XIP).	zynqmp

Table 30: **Bootgen Attributes and Description** *(cont'd)*

Option/Attribute	Description	Used By
checksum <options>	Specifies that the partition needs to be checksummed. This option is not supported along with more secure features like authentication and encryption. Checksum algorithms are: `md5`: for Zynq®-7000 SoC devices. For Zynq® UltraScale+™ MPSoC, options are `none`:No checksum operation. `sha3`: sha3 checksum. ***Note***: Zynq devices do not support checksum for boot loaders. Zynq UltraScale+ MPSoC devices do support checksum operation for bootloaders.	zynq zynqmp
destination_device <device_type>	This specifies if the partition is targeted for PS or PL. The options are: ps: the partition is targeted for PS (default). pl: the partition is targeted for PL, for bitstreams.	zynqmp
destination_cpu <device_core>	Specifies the core on which the partition should be executed. a53-0 a53-1 a53-2 a53-3 r5-0 (default) r5-1 pmu r5-lockstep	zynqmp
early_handoff	This flag ensures that the handoff to applications that are critical immediately after the partition is loaded; otherwise, all the partitions are loaded sequentially first, and then the handoff also happens in a sequential fashion.	zynqmp
encryption <option>	Specifies the partition to be encrypted. Encryption algorithms are: Zynq uses AES-CBC, and Zynqmp uses AES-GCM. The partition options are: none: Partition not encrypted. aes: Partition encrypted using AES algorithm.	All
exception_level <options>	Exception level for which the core should be configured. Options are: el-0 el-1 el-2 el-3	zynqmp

Table 30: **Bootgen Attributes and Description** *(cont'd)*

Option/Attribute	Description	Used By
familykey	Specifies the family key.	zynqmp fpga
fsbl_config	Specifies the sub-attributes used to configure the bootimage. Those sub-attributes are: `bh_auth_enable`: RSA authentication of the boot image is done excluding the verification of PPK hash and SPK ID. `auth_only`: boot image is only RSA signed. FSBL should not be decrypted. `opt_key`: Operational key is used for block-0 decryption. Secure Header has the opt key. `pufhd_bh`: PUF helper data is stored in Boot Header. (Default is `efuse`). PUF helper data file is passed to Bootgen using the `[puf_file]` option. `puf4kmode`: PUF is tuned to use in 4k bit configuration. (Default is 12k bit). `shutter = <value>`32 bit `PUF_SHUT` register value to configure PUF for shutter offset time and shutter open time.	zynqmp
headersignature <signature_file>	Imports the header signature into an Authentication Certificate. This can be used in case the user does not want to share the secret key, The user can create a signature and provide it to Bootgen.	zynq zynqmp
hivec	Specifies the location of exception vector table as hivec (Hi-Vector). The default value is lovec (Low-Vector). This is applicable with A53 (32 bit) and R5 cores only. `hivec`: exception vector table at `0xFFFF0000`. `lovec`: exception vector table at `0x00000000`.	zynqmp
init <filename>	Register initialization block at the end of the Bootloader, built by parsing the init (.int) file specification. A maximum of 256 address-value init pairs are allowed. The init files have a specific format.	zynq zynqmp
keysrc_encryption	Specifies the Key source for encryption. The keys are: `efuse_gry_key`: Grey (Obfuscated) Key stored in eFUSE. See Gray/Obfuscated Keys `bh_gry_key`: Grey (Obfuscated) Key stored in boot header. `bh_blk_key`: Black Key stored in boot header. See Black/PUF Keys `efuse_blk_key` : Black Key stored in eFUSE. `kup_key`: User Key. `efuse_red_key`: Red key stored in eFUSE. See Rolling Keys `bbram_red_key`: Red key stored in BBRAM.	zynq zynqmp

Table 30: **Bootgen Attributes and Description** *(cont'd)*

Option/Attribute	Description	Used By
load <partition_address>	Sets the load address for the partition in memory.	zynq zynqmp
offset <offset_address>	Sets the absolute offset of the partition in the boot image.	zynq zynqmp
partition_owner <option>	Owner of the partition which is responsible to load the partition. Options are: fsbl: Partition is loaded by FSBL uboot: Partition is loaded by U-Boot.	zynq zynqmp
pid <ID>	Specifies the Partition ID. PID can be a 32-bit value (0 to 0xFFFFFFFF).	zynqmp
pmufw_image <image_name>	PMU firmware image to be loaded by BootROM, before loading the FSBL.	zynqmp
ppkfile <key filename>	Primary Public Key (PPK). Used to authenticate partitions in the boot image. See Using Authentication for more information.	zynq zynqmp
presign <sig_filename>	Partition signature (.sig) file.	All
pskfile <key filename>	Primary Secret Key (PSK). Used to authenticate partitions in the boot image. See the Using Authentication for more information.	zynq zynqmp
puf_file <filename>	PUF helper data file. PUF is used with black key as encryption key source. PUF helper data is of 1544 bytes.1536 bytes of PUF HD + 4 bytes of HASH + 3 bytes of AUX + 1 byte alignment.	zynqmp
reserve	Reserves the memory, which is padded after the partition.	zynq zynqmp
spk_select <SPK_ID>	Specify an SPK ID in user eFUSE.	zynqmp
spkfile <filename>	Keys used to authenticate partitions in the boot image. See Using Authenticationfor more information. SPK - Secondary Public Key	All
spksignature <signature_file>	Imports the SPK signature into an Authentication Certificate. See Using AuthenticationThis can be used in case the user does not want to share the secret key PSK, The user can create a signature and provide it to Bootgen.	zynq zynqmp

Table 30: **Bootgen Attributes and Description** *(cont'd)*

Option/Attribute	Description	Used By
split <options>	Splits the image into parts, based on the mode. Split options are: slaveboot: Supported for zynqmp only. Splits as follows: Boot Header + Bootloader Image and Partition Headers Rest of the partitions normal: Supported for both zynq and zynqmp. Splits as follows: Bootheader + Image Headers + Partition Headers + Bootloader Partiton1 Partition2 and so on Along with the split mode, output format can also be specified as `bin` or `mcs`. ***Note*:** The option split mode normal is same as the command line option split. This command line option is deprecated.	zynq zynqmp
sskfile <key filename>	Secondary Secret Key (SSK) key authenticates partitions in the Boot Image. The primary keys authenticate the secondary keys; the secondary keys authenticate the partitions.	All
startup=<address>	Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.	zynq zynqmp
trustzone= <option>	The trustzone options are: secure nonsecure	zynqmp
udf_bh <data_file>	Imports a file of data to be copied to the user defined field (UDF) of the Boot Header. The UDF is provided through a text file in the form of a hex string. Total number of bytes in UDF are: zynq = 76 bytes; zynqmp= 40 bytes.	zynq zynqmp
udf_data <data_file>	Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate.	zynq zynqmp
xip_mode	Indicates eXecute In Place (XIP) for FSBL to be executed directly from QSPI flash.	zynq

Chapter 14

Using Bootgen Interfaces

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

Bootgen GUI Options

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

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

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

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

   Note the following:

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

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

2. Populate the Create boot image dialog box with the following information:

   a. From the **Architecture** drop-down, select the required architecture.

   b. Select either **Create a BIF file** or **Import an existing BIF file**.

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

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

Using Bootgen on the Command Line

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

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

Commands and Descriptions

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

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

Table 31: **Bootgen Command and Descriptions**

Commands	Description and Options	Used by
arch <type>	Xilinx® device architecture: Options: zynq (default) zynqmp fpga	All
bif_help	Prints out the BIF help summary.	All
dual_qspi_mode <configuration>	Generates two output files for dual QSPI configurations: parallel stacked <size>	zynq zynqmp
efuseppkbits <PPK_filename>	Generates a PPK hash for eFUSE.	zynq zynqmp
encrypt <options>	AES Key storage in device. Options are: bbram (default) efuse	zynq fpga
encryption_dump	Generates encryption log file, aes_log.txt.	zynqmp
fill <hex_byte>	Specifies the fill byte to use for padding.	zynq zynqmp
generate_hashes	Output SHA2/SHA3 hash files with padding in PKCS#1v1.5 format.	zynq zynqmp

Table 31: **Bootgen Command and Descriptions** *(cont'd)*

Commands	Description and Options	Used by
generate_keys <key_type>	Generate the authentication keys. Options are: pem rsa obfuscatedkey	zynq zynqmp
h, help	Prints out help summary.	All
image <filename(.bif)>	Provides a boot image format (.bif) file name.	All
log<level_type>	Generates a log file at the current working directory with following message types: error warning (default) info debug trace	All
nonbooting	Create an intermediate boot image.	zynq zynqmp
o <filename>	Specifies the output file. The format of the file is determined by the file name extension. Valid extensions are: .bin (default) .mcs .pdi	All
p <partname>	Specify the part name used in generating the encryption key.	All
padimageheader <option>	Pads the image headers to force alignment of following partitions. Options are: 0 1 (default)	zynq zynqmp
process_bitstream <option>	Specifies that the bitstream is processed and outputs as .bin or .mcs. For example, if encryption is selected for bitstream in BIF file, the output is an encrypted bitstream.	zynq zynqmp

Table 31: **Bootgen Command and Descriptions** *(cont'd)*

Commands	Description and Options	Used by
read <options>	Used to read boot headers, image headers, and partition headers based on the options. bh: To read boot header from bootimage in human readable form iht: To read image header table from bootimage ih: To read image headers from bootimage. pht: To read partition headers from bootimage ac: To read authentication certificates from bootimage	zynq zynqmp
split <options>	Splits the boot image into partitions and outputs the files as `.bin` or `.mcs`. • Bootheader + Image Headers + Partition Headers + `Fsbl.elf` • `Partition1.bit` • `Partition2.elf`	zynq zynqmp
spksignature <filename>	Generates an SPK signature file.	zynq zynqmp
verify	This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image will be verified against the available partitions.	zynq zynqmp
verify_kdf	This option is used to validate the Counter Mode KDF used in bootgen for generation AES keys.	zynqmp
w <option>	Specifies whether to overwrite the output files: on(default) off ***Note***: The `-w` without an option is interpreted as `-w on`.	All
zynqmpes1	Generates a boot image for ES1 (1.0). The default padding scheme is ES2 (2.0).	zynqmp

Chapter 15

Boot Time Security

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

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

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

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

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

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

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

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

Zynq UltraScale+ MPSoC Device Security

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

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

Using Encryption

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

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

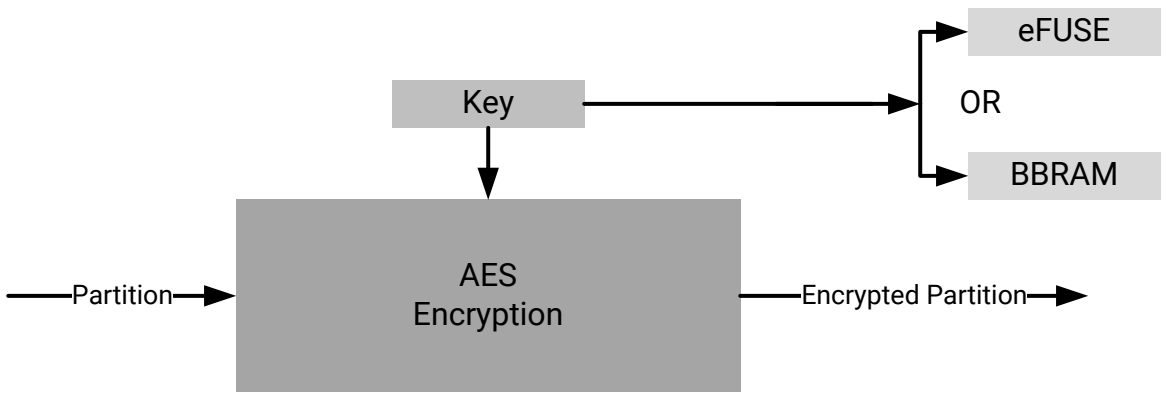
AES is an iterated symmetric block cipher, which means that it does the following:

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

Encryption Process

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

Figure 11: **Encryption Process Diagram**

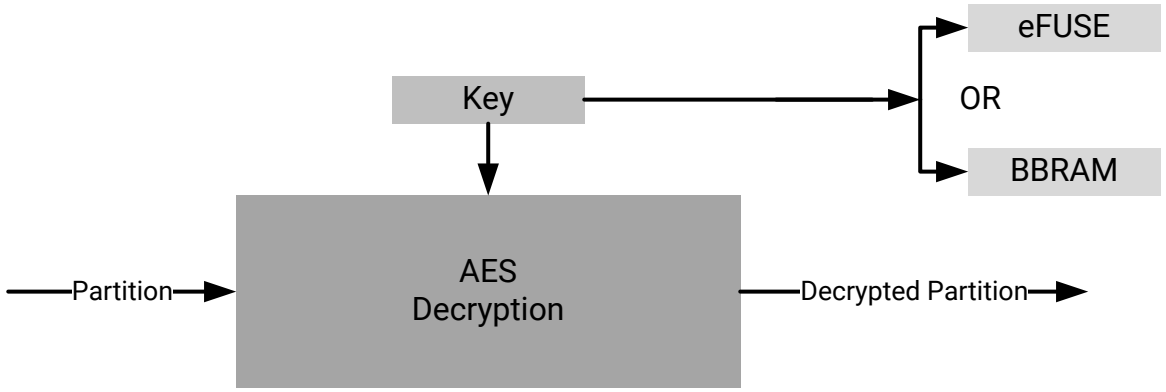


X21274-102919

Decryption Process

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

Figure 12: **Decryption Process Diagram**



X21274-102919

Encrypting Zynq-7000 Device Partitions

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

Example BIF File

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

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

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

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

Key Generation

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

Figure 13: **Sample Key File**

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

Encrypting Zynq MPSoC Device Partitions

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

Operational Key

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

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

```
image:
{
    [fsbl_config] opt_key
    [keysrc_encryption] bbram_red_key

    [bootloader,
     destination_cpu = a53-0,
     encryption      = aes,
     aeskeyfile      = aes_p1.nky]fsbl.elf

    [destination_cpu = a53-3,
     encryption      = aes,
     aeskeyfile      = aes_p2.nky]hello.elf

}
```

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

Figure 14: **Operational Key**

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

Bootgen generates the encryption key file. The operational key `opt_key` is then generated in the `.nky` file, if `opt_key` has been enabled in the BIF file, as shown in the previous example.

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

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

Rolling Keys

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

```
image:
{
    [keysrc_encryption] bbram_red_key

    [
        bootloader,
        destination_cpu = a53-0,
        encryption      = aes,
        aeskeyfile      = aes_p1.nky,
        blocks          = 1024(2);2048;4096(2);8192(2);4096;2048;1024
    ]    fsbl.elf

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

***Note*:**

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

Gray/Obfuscated Keys

The user key is encrypted with the family key, which is embedded in the metal layers of the device. This family key is the same for all devices in the Zynq® UltraScale+™ MPSoC. The result is referred to as the *obfuscated key*.

The obfuscated key can reside in either the Authenticated Boot Header or 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 first partition.
- If an encryption file is generated for the first partition and other encryption file with Key0/IV0 is specified for a later partition, then Bootgen exits and returns the error that an incorrect Key0/IV0 pair was used.

Key Generation

A sample key file is shown below.

Figure 15: **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 Xilinx® 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), IV0, and Operational Key. Bootgen does not allow creating boot images if these are different in each encryption key file. You must specify multiple encryption key files, one for each of partition in the image. The partitions are encrypted using the key that is specified for the partition.

Note: You can have unique key files for each of the partition created due to multiple loadable sections by having key file names appended with `".1", ".2"..."."n"` 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`.

Using Authentication

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

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

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

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

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

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

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

Bootgen can create 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:

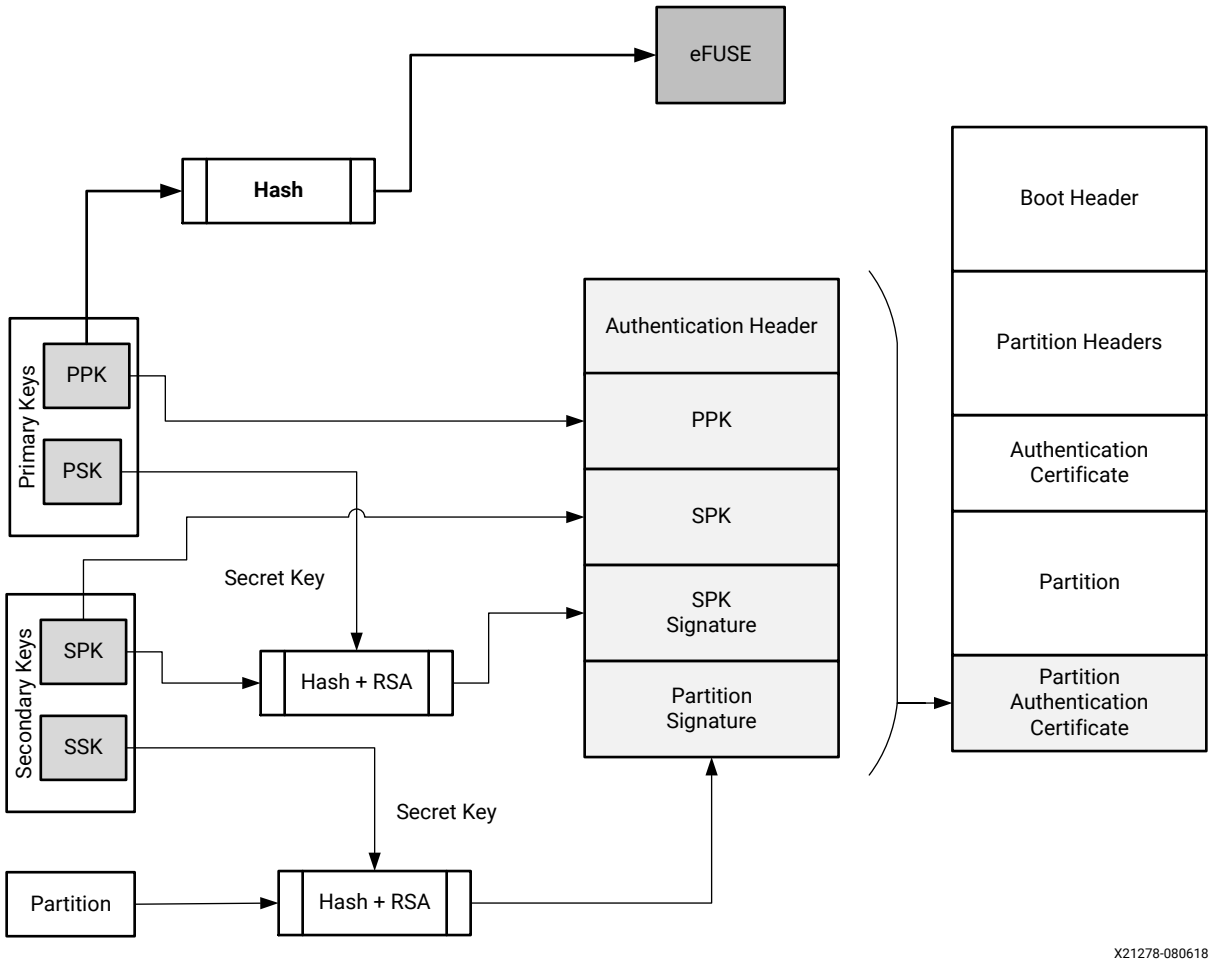
- Zynq-7000 Authentication Certificates
- Zynq UltraScale+ MPSoC Authentication Certificates

Signing

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

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

Figure 16: **RSA Partition Signature**



The following table shows the options for Authentication.

Table 32: **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.	*.txt *.pem *.pub *.pk1
PSK	Primary Secret Key	This key is used to authenticate a partition. It should always be specified when authenticating a partition.	*.txt *.pem *.pk1
SPK	Secondary Public Key	This key, when specified, is used to authenticate a partition.	*.txt *.pem *.pub *.pk1

Table 32: **Supported File Formats for Authentication Keys** *(cont'd)*

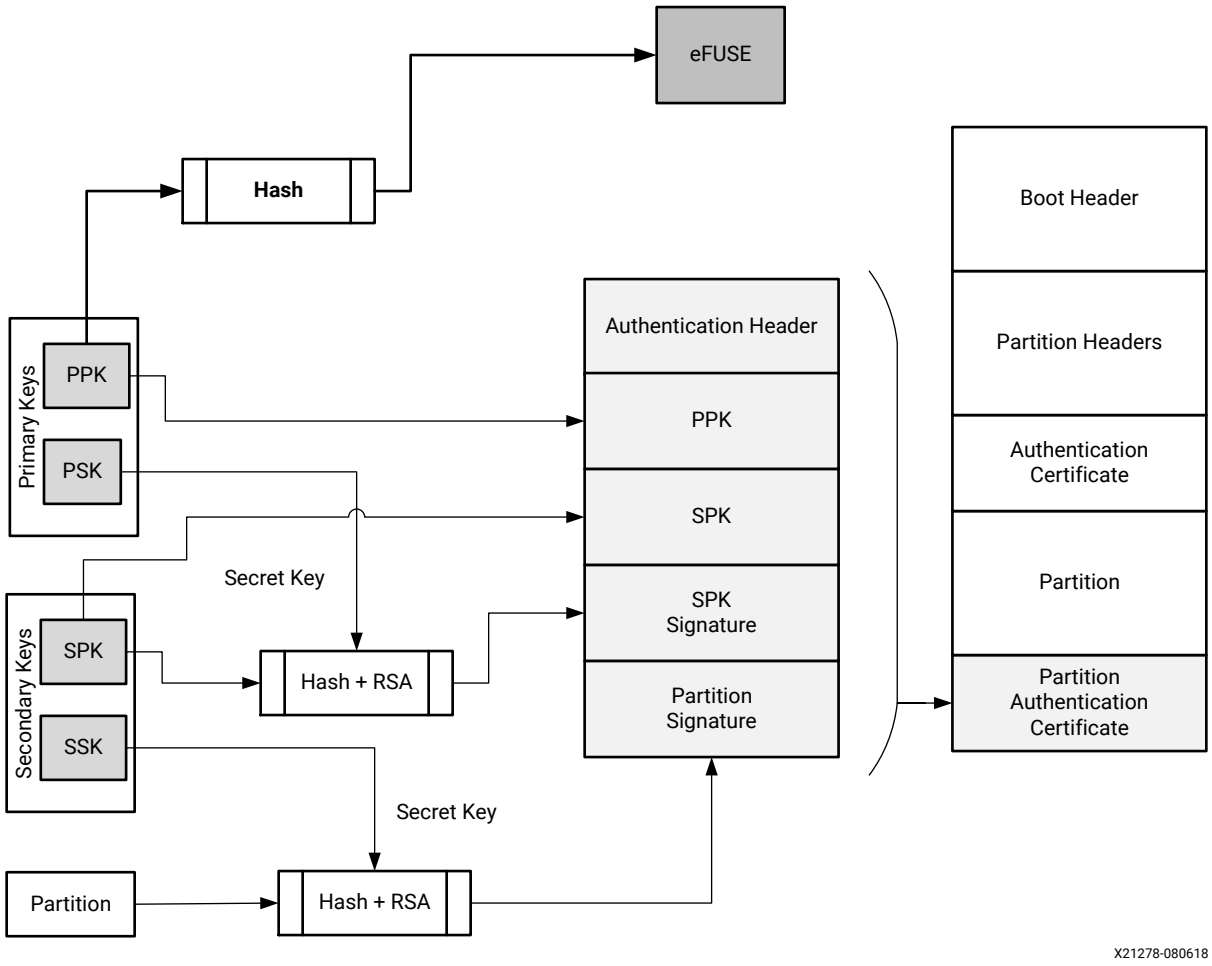
Key	Name	Description	Supported File Format
SSK	Secondary Secret Key	This key, when specified, is used to authenticate a partition.	*.txt *.pem *.pk1

Verifying

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

1. Verify PPK - This step establishes the authenticity of primary key, which is used to authenticate secondary key.
   a. PPK is read from AC in boot image
   b. Generate PPK hash
   c. Hashed PPK is compared with the PPK hash retrieved from eFUSE
   d. If same, then primary key is trusted, else secure boot fail
2. Verify secondary keys: This step establishes the authenticity of secondary key, which is used to authenticate the partitions.
   a. SPK is read from AC in boot image
   b. Generate SPK hashed
   c. Get the SPK hash, by verifying the SPK signature stored in AC, using PSK
   d. Compare hashes from step (b) and step (c)
   e. If same, then secondary key is trusted, else secure boot fail.
3. Verify partitions - This step establishes the authenticity of partition which is being booted.
   a. Partition is read from the boot image.
   b. Generate hash of the partition.
   c. Get the partition hash, by verifying the Partition signature stored in AC, using SSK.
   d. Compare the hashes from step (b) and step (c)
   e. If same, then partition is trusted, else secure boot fail

Figure 17: **Verification Flow Diagram**



Bootgen can create a authentication certificate in two ways:

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

Zynq UltraScale+ MPSoC Authentication Support

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

NIST SHA-3 Support

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

The generated signature uses the Keccak-SHA3 or NIST-SHA3 based on following table:

Table 33: **Authentication Signatures**

Which Authentication Certificate (AC)?	Signature	SHA Algorithm and SPK eFUSE	Secret Key used for Signature Generation
Header AC (loader by FSBL/FW)	SPK Signature	If SPKID eFUSEs, then Keecak; If User eFUSE, then NIST	PSK
	BH Signature	Always Keecak	SSK_{header}
	Header Signature	Always Nist	SSK_{header}
BootLoader AC (loaded by ROM)	SPK Signature	Always Keecak; Always SPKID eFUSE for SPK	PSK
	BH Signature	Always Keecak	$SSK_{Bootloader}$
	Header Signature	Always Keecak	$SSK_{Bootloader}$
Partition AC (loaded by FSBL FW)	SPK Signature	If SPKID eFUSEs then Keecak; If User eFUSE then NIST	PSK
	BH Signature	Always Keecak	$SSK_{Partition}$
	Header Signature	Always NIST	$SSK_{Partition}$

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
    [authenication=rsa, destination_cpu=r5-0] hello.elf
}
```

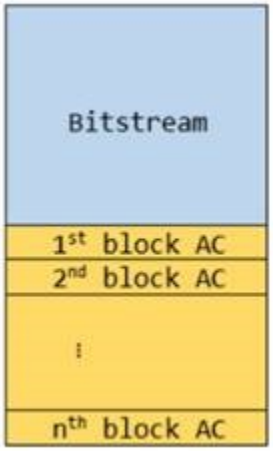
Example 2: BIF file for authenticating the partitions with separate secondary key for each partition:

```
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 = 0x01,
      sskfile = secondary_p2.pem
    ] bl31.elf

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

Bitstream Authentication Using External Memory

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

Figure 18: **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 = 0x200,
      sskfile = ssk3.pem
```

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

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

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

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

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

Key Generation

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

The figure shows the sample RSA private key file.

Figure 19: **Sample RSA Private Key File**


```
-----BEGIN RSA PRIVATE KEY-----
MIIJKAIBAAKCAgEA4ppimme6TvPT5+JB2CgXQLU9AyStbnEr2lEJu+ZpR9HZ5Plq
6KbOcFuV6q3EKvI5PJsMS0yHpVr/ll/uTPxyUT6Im5goMyaskz0PS3xTWuYoSDba
YD50ZlPi5xBrswWvys6YcIbLTbk2+o86o0Rr/sdQtLR0pbsLfuBFoKMEsKl9N12k
Ell6DMlTjh9KSpZOzmj7yew2Rm857QqQp8su1Vi4qdtIr58+MoQxeETeHcN+zuq4
drlUsUqX3msVb9z0rRwYrBVtSksWr5d+xj+cAUpiPjeMGRXg00L6gEGGPTjnqQtG
YFCoCFcBL4JknHF/yMyV7f6wh2xtkKbme+Kuovcz/pQVKEGELkQ9kjweBf5c8Vmk
b13NvkrAUOXYLM+py0uY/PGjtz6B5W964LOcrT+TRROi4FGotYzk2XmJtODO5dYH
Lw58IOT3zAYwaC/98bUDGYP6kJ9+YqprerLmZU55Ew30PPodjHYihLmBj1pvmu4g
oZ9tXJPch/uRk/tv3e53P2JhWKwdb72FUi8hEgSkCWWAFfJwCVFwATettzGlhtz+
Ww3eBAQi9fFbgr6YERwxOOLopaRQiZPaC/8XG8u0bTE3MdvsJK/IIOAqVnTl7Dfs
QKzTZap8+Iwx/vuaWAiLd0qYCDKKKmlGGz5bQhEgRnk0I/1pOKIlPRL8wH0CAwEA
AQKCAgA3qhscuOxgZq8gYEkycy67G4pgUks0PSK7n3qXqNMl7FvtToO/oPJHUYgz
PPpaXmRHCgNsH+GWchM08gDU8pKWeJkQN8FwR0jPZolyTpkfVDiC/M6KI+luEZ9E
iZkbQgNb+4Ig6kvYzO2/gR2za6Rn0shli3q4F4mMYkVYX5NQXmI/Doa2phlANdQX
roIObnvvYoSvppHynXIKU7UTMutPRlsdhpuFYMXjnOuWErzJbPOimrAzofU3FA7Y
eU+ryghk2ekJpL3TKTzqZ3mh85A8FOyrQfPtWZl/6A0nInF6apclxHpGQKn2WoEV
DZ/vekYcqn0OGKl+qtkDVqx5tEaXlXGlc0PBWg5aofkpNZ0K0wOG6iCueNvATcJ9
RoMq7c7zZOYh4SzWgSjP3a8neGcnhG0T6BGYCGjPXRW2Y6ri/7lrDcOBVSc3zS8p
IVKABpl3PIgZlhMnxdC60RPh8dhXRR0TUa3+lSyGx37Ad9260UeHHJIpz28DkzTg
CY7RU5SDSh6wDuDbhelu4nzZDGWeKq9zeAzXGZhIn0zcxpWvG54uHTHNnqBEFJ2S
ZSJ8sq4aYiZCiW/PrqKgg8wBygKcEtr3/LcAm4r3pl9mHk1555QQNdpk+ba+3GLp
bEy0889KwCyPKfWY5pl6VNgLycxe/TofMDCHQARAZwLRnN1sQQKCAQEA80nn83su
OYN9Oc22owfm/MHGJ6mFi5LpRtGylWbcAbDsZs7rjQ4IZ46JQlMpiQl0IpNbVub7
sW0FUX7sVo0XZMSl+EpsZDq021+7hY6+MGALtPpg9n2Jz9lfCyVXfNqv5SiMv6Te
6/jur69KiwhztYFi7JK4GGUdcCWyAwMTdgm3pQDDH99Vp436klvk4lMyjeQaIpO/
FzkiklfYN84j9jvtagoMk0fzaickiOGSs4ciOdS3DEgGC9x1hDkIs9UFPk1Pfw+7
qYnsT7XIwoTCBrvQlllKp5fLZUhSRsIQV82u44IPfcU3xWgeyInSGx0RfSV5RWOv
v9sJPVsFlXE5EQKCAQEA7nFNK5gbPKA0nxKTeM1ZMHp99/YqRxpj5irmXmrF54cn
sZPpG/dvbJBXILAd9hsSYjW8FNY5ehJhL9IQzEVavFr8SAvu2FyI9MN0d9wUvpJG
55JxX9K090uSzaXZVimV/5xumbnynwx2ZwgxslSAYoNy+8soviZlXQxZzeUaohaM
VVuLlHdRzE0afrcFsnfugIDl72MbI4t2cKTfTek/iYAvF9bkO76upkPmWu4V7yFT
of9QFkq8qBRthEpvaKNTObpU5TrzsxUH3rYXVnAZgpEXEJdeVVFYzSLf4SC45mxe
GPp37pYetPBKVrUesuEvQ70IeoiCGRXFgC9TPmYwrQKCAQEA5D1CoPbAD+7ejVsD
a4FfX2K+7rin86A4vlq9hlzAK8n6jhyzeRpgIh7jgFiaj9hRnppVx3pdSD+6DJGh
UTV+a+fcuMnBVGqK/3+ZYhvfK2z/rqJyUuzFXDxWYROANz7GY5seKDCZfhGEg0dV
DIg6XV5sGvsuQJyj+HE0xoSdPlCxe9fyNrWEGvQkzxgX64gX1mvXZPbs//F3EIne
68O1kyz3d1LEJ2wJ3V2pdc0BnvE4175K1/f9zCTgDtKe6m7/Q0quOMreyJf/HWyy
UmLP0BdlAogfdIkApR0rKvym7milGQUMWXaq8sTSlFpPxWYI4TpFwiZaXAg2a9w3
qdKVsQKCAQBH8noloFT/mxulsBY9ikDSRvPBoU6qe8UPC3zNmowyy25nv8jD/opp
iLgxjdLMkuieJ7aj1uwq8GbQ5iLZcEfrs8yR9L/SG0HcESoQjKDZzAuHDoIVNuAS
CoS2dse4nv26zjn1Os2BvmHvvuI3/BVtJFrKrUeS8MT/KZ3jabD6nbEkhGX+m25c
JhvLhnA6pMOblMlMzWu/8vH/FVCoEqxwUfRjzhy6BlRuqOhWIacOq9CvffltcImy
cc+F7mvld/rB3X6GWJ52N+9S/UDXfSXFZwA9ql7lgYE5DL/fD1+bb7GI+fK8VCHZ
2POlbCtiMF5xoxVu28fdx9r7TcxhdLZVAoIBADmGYfxvgEqhALqdWZQmtRRNisWQ
y0/RfED7dNtN8o5vjBCbrOV/tQ3Ddbb7a0kwo1NFr1xR7KIki98SkKN0EiCrpRfc
+ccs6kASTZcPH/nGG9lbrOAm9FOG2q5cX6kDK1hqHe+lUYm/34a+2wN0/CwAh7MH
gECABtqx9QCD/DJI+n5ocrYk5RsQJrtnwoP4L8X24dRiMiRMIsS4V9uyyRLQTWV/
k3TOjRgL5eRKbcVwV7c8kmaGDWfM/eVLlQW+wEa0wY+TdSUhlYvgsG5yijkhCAEe
/+Az0w5Zu1vnLbj5eXKiULWISlOsDCBfJepuINHoUpBwsGzFb7ZXtpK2XlM=
-----END RSA PRIVATE KEY-----
```


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

BIF Example

A sample BIF file, `generate_pem.bif`:

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

Command

The command to generate keys is, as follows:

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

PPK Hash for eFUSE

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

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

BIF File Example

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

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

Command

The command to generate PPK hash for eFUSE programming is:

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

Using HSM Mode

In current cryptography, all the algorithms are public, so it becomes critical to protect the private/secret key. The hardware security module (HSM) is a dedicated crypto-processing device that is specifically designed for the protection of the crypto key lifecycle, and increases key handling security, because only public keys are passed to the Bootgen and not the private/secure keys. A *Standard* mode is also available; this mode does not require passing keys.

In some organizations, an Infosec staff is responsible for the production release of a secure embedded product. The Infosec staff might use a HSM for digital signatures and a separate secure server for encryption. The HSM and secure server typically reside in a secure area. The HSM is a secure key/signature generation device which generates private keys, signs the partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys reside in the HSM only.

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

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

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

Using Advanced Key Management Options

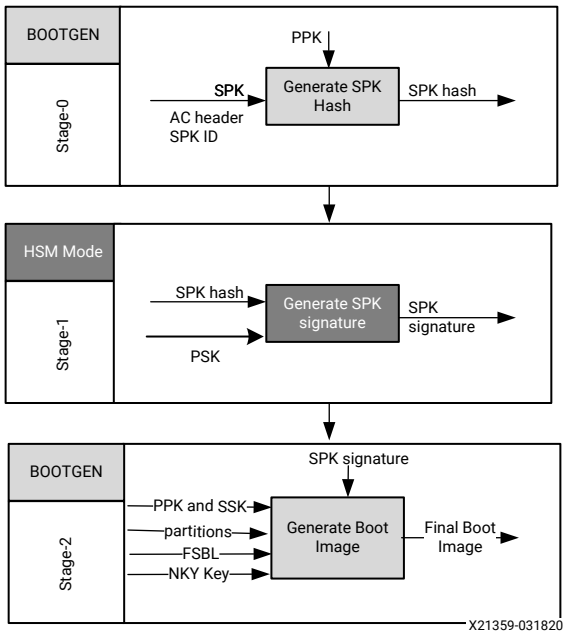
The public keys associated with the private keys are `ppk.pub` and `spk.pub`. The HSM accepts hash values of partitions generated by Bootgen and returns a signature block, based on the hash and the secret key.

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

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

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

Figure 20: **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=0x12345678
    [keysrc_encryption]bbram_red_key
    [pskfile]primary.pem
    [sskfile]secondary.pem
    [
     bootloader,
     encryption=aes,
     aeskeyfile=aes.nky,
     authentication=rsa
    ]fsbl.elf
    [destination_cpu=a53-0,authentication=rsa]hello_a53_0_64.elf
}
```

Stage 0: Create a boot image using HSM Mode

A trusted individual creates the SPK signature using the Primary Secret Key. The SPK Signature is on the Authentication Certificate Header, SPK, and SPK ID. To generate a hash for the above, use the following BIF file snippet.

```
stage 0:
{
    [auth_params] ppk_select=1;spk_id=0x12345678
    [spkfile]keys/secondary.pub
}
```

The following is the Bootgen command:

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

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

Stage 1: Distribute the SPK Signature

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

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

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

Stage 2: Encrypt using AES in FSBL

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

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

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

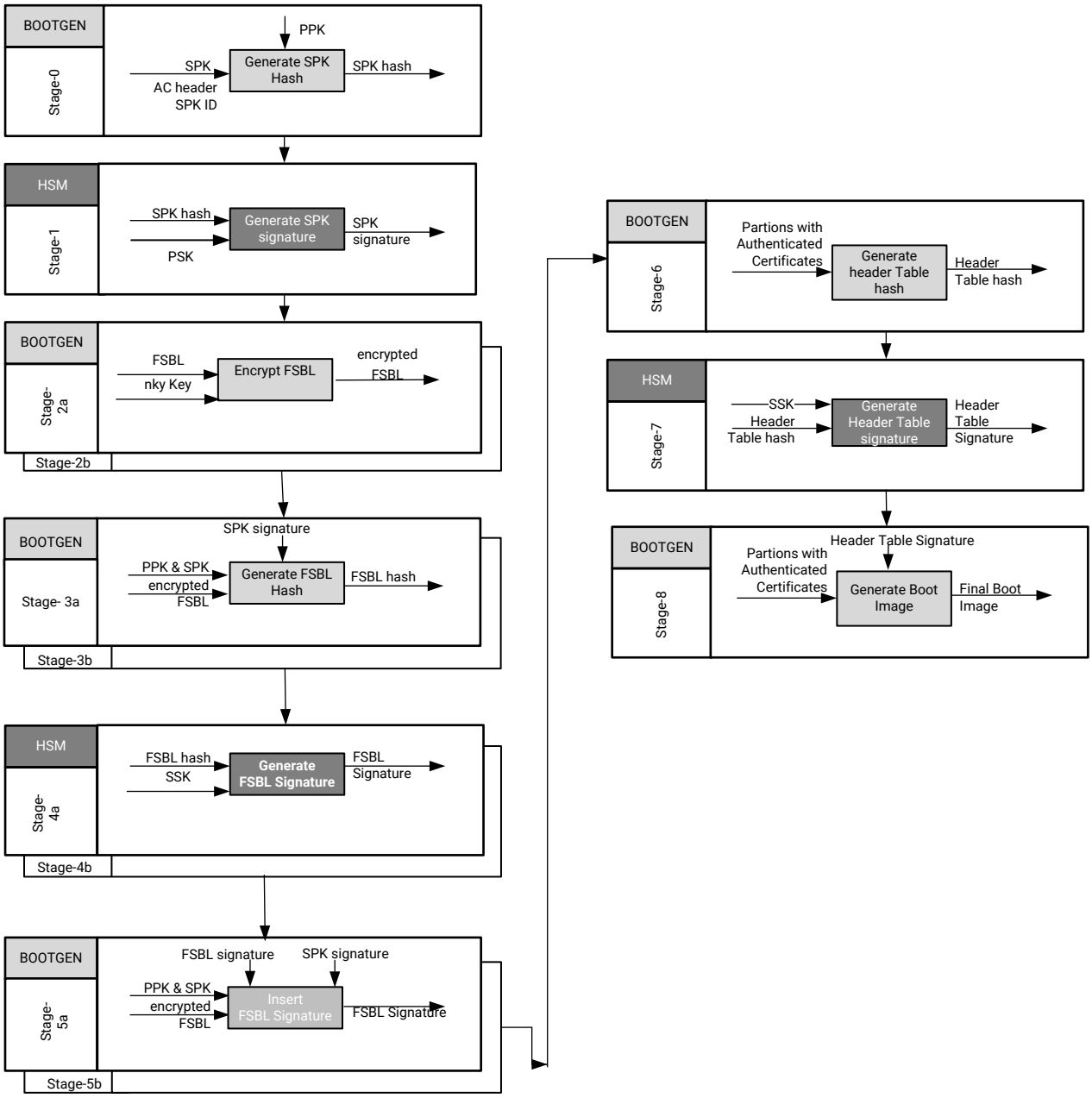
The Bootgen command is:

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

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

The following figure provides a diagram of an HSM mode boot image for a Zynq®-7000 SoC device. The steps to create this boot image are immediately after the diagram.

Figure 21: **Stage 0 to 8 Boot Process**



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

```
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.

Stage3a: Generate the FSBL Hash

The BIF file is as follows:

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

The Bootgen command is as follows.

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

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

Stage 3b: Generate the bitstream hash

The stage3b BIF file is as follows:

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

The Bootgen command is as follows.

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

The output is the hash file `system.bit.0.sha256`.

Stage 4: Sign the Hashes

This stage creates signatures from the partition hash files created.

Stage 4a: Sign the FSBL partition hash

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

Or by using the following OpenSSL program.

```
#Swap the bytes in FSBL hash
objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256

#Generate FSBL signature using OpenSSL
openssl rsautl -raw -sign -inkey secondary.pem -in zynq_fsbl_0.elf.0.sha256
> zynq_fsbl_0.elf.0.sha256.sig

#Swap the bytes in FSBL signature
objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256.sig
```

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

Stage 4b: Sign the bitstream hash

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

Or by using the following OpenSSL program.

```
#Swap the bytes in bitstream hash
objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256

#Generate bitstream signature using OpenSSL
openssl rsautl -raw -sign -inkey secondary.pem -in system.bit.0.sha256 >
system.bit.0.sha256.sig

#Swap the bytes in bitstream signature
objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256.sig
```

The output is the signature file `system.bit.0.sha256.sig`.

Stage 5: Insert Partition Signatures

Insert partition signatures created above are changed into authentication certificates.

Stage 5a: Insert the FSBL signature

The stage5a.bif is as follows.

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

The Bootgen command is as follows.

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

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

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

The Bootgen command is as follows.

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

The authenticated output file is `system_e_ac.bin`.

Stage 6: Generate Header Table Hash

This stage generates the hash for the header tables.

The stage6.bif is as follows.

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

The Bootgen command is as follows.

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

The output hash file is `ImageHeaderTable.sha256`.

Stage 7: Generate Header Table Signature

This stage generates the header table signature.

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

Or by using the following OpenSSL program:

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

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

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

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

Stage 8: Combine Partitions, Insert Header Table Signature

The stage8.bif is as follows:

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

The Bootgen command is as follows:

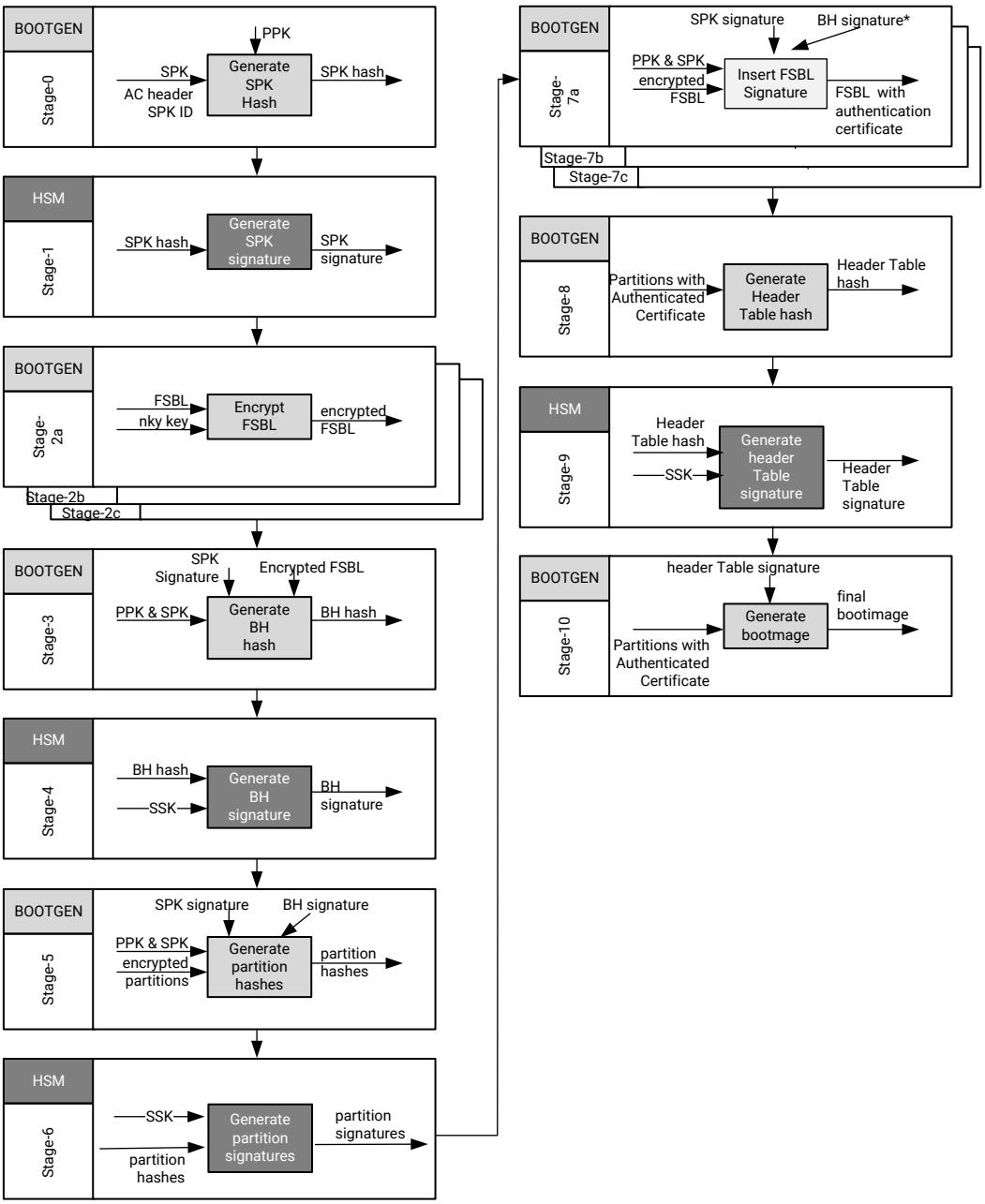
```
bootgen -image stage8.bif -w -o final.bin
```

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

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

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

Figure 22: **0 to 10 Stage Boot Process**



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

```
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
}
```

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 will be generated for a bitstream partition. For more details, see Bitstream Authentication Using External Memory.

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

Stage 6: Sign Partition Hashes

Create the following files using OpenSSL:

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

Stage 7: Insert Partition Signatures into Authentication Certificate

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

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

The Bootgen command is as follows:

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

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

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

The Bootgen command is:

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

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

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

The Bootgen command is:

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

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

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

The Bootgen command is:

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

Stage 8: Combine Partitions, Get Header Table Hash

Enter the following in a BIF file:

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

The Bootgen command is:

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

Stage 9: Sign Header Table Hash

Generate the following files using OpenSSL:

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

Stage 10: Combine Partitions, Insert Header Table Signature

Enter the following in a BIF file:

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

The Bootgen command is:

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

Chapter 16

FPGA Support

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

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

Encryption and Authentication

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

Encryption Example

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

```
bootgen -arch fpga -image secure.bif -w -o 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 Xilinx support and request the family key file for the target device family. The path to where this file is stored must be passed as a `bif` option before attempting obfuscated encryption. Contact secure.solutions@xilinx.com to obtain the Family Key.

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

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

Figure 23: **AES Key Sample**

```
Device xcku115;
EncryptKeySelect BBRAM;
KeyObfuscate 94da9014cb2203f502f81d14fa2471f4a8902b16d9d408c9c66db214c1640db7, 0;
StartIvObfuscate c485144e397a92081ad20c867a005272, 0;
Key0 dcd2e72ad1b281ecca5e0790b65b94090ec1c8fc010eb01e56717345df4c7010, 0;
StartIV0 3fe826e5495db1bdaf0c2ca2e8640911, 0;
KeyObfuscate 967a6d1ecccefddd1990241007de18f41d69ca7231852c0061fb6c78e204c5f3, 1;
StartIvObfuscate 7ab9a7ca88474d7f95ed1b548523451b, 1;
Key0 af84947a9cc256c090d5ae1c53ed3fd33bb553d7039e445829ba4cffbe56ffe3, 1;
StartIV0 a50026e212363e1d71fa6f4fb540ce42, 1;
```

HSM Mode

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

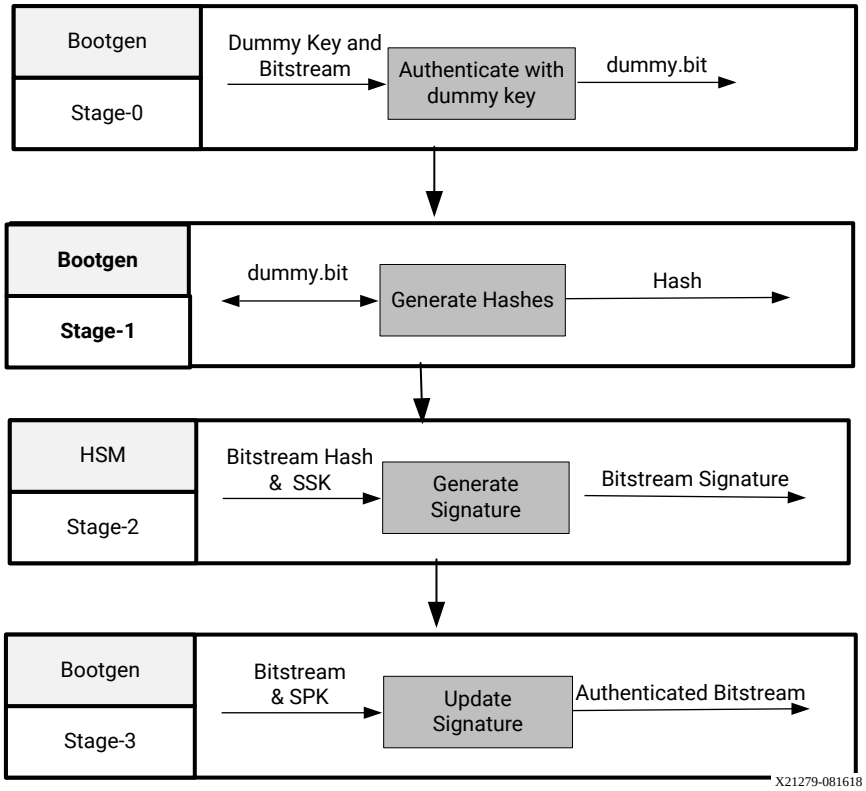
Standard Mode

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

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

The following steps listed below describe how to generate an authenticated bitstream in HSM mode, where the secret keys are maintained by secure team and not available with the user. The following figure shows the HSM mode flow:

Figure 24: **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.

```
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.

Chapter 17

Use Cases and Examples

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

Zynq MPSoC Use Cases

Simple Application Boot on Different Cores

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

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

PMUFW Load by BootROM

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

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

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

PMUFW 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 a Zynq® UltraScale+™ MPSoC device (`arch=zynqmp`).

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

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

Encryption Flow: BBRAM Red Key

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

```
the_ROM_image:
{
    [keysrc_encryption] bbram_red_key
    [
      bootloader,
      encryption=aes,
      aeskeyfile=aes0.nky,
```

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

Encryption Flow: Red Key Stored in eFUSE

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

```
the_ROM_image:
{
    [keysrc_encryption] efuse_red_key

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

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

Encryption Flow: Black Key Stored in eFUSE

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

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

Note: Boot image authentication is compulsory for using black key encryption.

Encryption Flow: Black Key Stored in Boot Header

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

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

```
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-A manages the secret red key. Team-B builds encrypted images for development and test. However, it does not have access to the secret red key.

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

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

The following procedures describe the steps to build an image:

Procedure-1

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

1. Encrypt Bootloader with device key:

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

   Example stage1.bif:

   ```
   stage1:
   {
       [fsbl_config] opt_key
       [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. 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 xc7z020clg484;
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
IV 0 F7F8FDE08674A28DC6ED8E37;
```

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

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

Example of `stage3.bif`:

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

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

Single Partition Image

This features 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=0x12345678
       [authentication = rsa]Data.bin
   }
   ```

2. When the image is generated, download the authenticated image to the DDR.
3. Execute the U-Boot command to authenticate the secure image as shown in the following example.

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

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

Only Encryption Use Case

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

Authentication Flow

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

```
the_ROM_image:
{
    [fsbl_config] bh_auth_enable
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] PSK.pem
    [sskfile] SSK.pem

    [
```

```
      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 boot header authentication enabled to bypass the PPK hash verification:

```
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 (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.

Chapter 18

BIF Attribute Reference

aarch32_mode

Syntax

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

Description

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

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

Arguments

Specified partition.

Example

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

aeskeyfile

Syntax

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

Description

The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file must be written to eFUSE or BBRAM. If the key file is not present in the path specified, a new key is generated by Bootgen, which is used for encryption.

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

Arguments

Specified file name.

Return Value

None

Zynq-7000 SoC Example

The partitions `fsbl.elf` and `hello.elf` are encrypted using keys in `test.nky`.

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

Sample key (.nky) file - `test.nky`

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

Zynq UltraScale+ MPSoC Example

Example 1:

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

```
all:
{
     [keysrc_encryption] bbram_red_key
     [bootloader,encryption=aes,aeskeyfile=test.nky] fsbl.elf
     [encryption=aes,aeskeyfile=test1.nky] hello.elf
     [encryption=aes,aeskeyfile=test2.nky] app.elf
}
```

Example 2:

Consider Bootgen creates three partitions for `hello.elf`, called `hello.elf.0`, `hello.elf.1`, and `hello.elf.2`. Sample BIF - `test_mulitple.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.

alignment

Syntax

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

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

Arguments

Number of bytes to be aligned.

Example

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

auth_params

- **Syntax:**

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

Description

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

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

***Note*:**

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

Example

Sample BIF 1 - test.bif

```
all:
{
    [auth_params]ppk_select=0;spk_id=0x12345678
    [pskfile] primary.pem
    [sskfile]secondary.pem
    [bootloader, authentication=rsa]fsbl.elf
}
```

Sample BIF 2 - test.bif

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

Sample BIF 3 - test.bif

```
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             = 0x12345678,
    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

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

Description

This specifies the partition to be authenticated.

Arguments

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

Example

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

big_endian

Syntax

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

Description

To specify the binary file is in big endian format.

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

Arguments

Specified partition.

Example

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

bh_keyfile

Syntax

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

Description

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

Arguments

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

Example

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

bh_key_iv

Syntax

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

Description

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

Arguments

Path to file.

Example

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

bhsignature

Syntax

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

Description

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

Example

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

blocks

Syntax

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

Description

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

Arguments

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

Example

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

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

boot_device

Syntax

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

Description

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

Arguments

Options are:

- qspi32
- qspi24
- nand
- sd0
- sd1

- sd-ls
- mmc
- usb
- ethernet
- pcie
- sata

Example

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

bootimage

Syntax

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

Description

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

Arguments

Specified file name.

Example

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

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

Example fsbl.bin generation

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

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

Example system.bin generation

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

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

bootloader

Syntax

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

Description

Identifies an ELF file as the FSBL.

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

Arguments

Specified file name.

Example

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

bootvectors

Syntax

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

Description

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

Example

```
all:
{

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

checksum

Syntax

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

Description

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

Arguments

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

- SHA3: Checksum operation for Zynq® UltraScale+™ MPSoC devices.

core

Syntax

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

Description

This attributes specifies which core executes the partition.

Arguments

- *a72-0
- a72-1
- r5-0
- r5-1
- psm

Example

```
test:
{
    id = 0x2
    image
    {
        {type=bootloader, file=plm.elf}
        {type=pmcdata, load=0xf2000000, file=pmc_cdo.bin}
        {type=cdo, file=ps_cdo.bin}
        {core=a72-0, file=a72_app.elf}
        {core=r5-0, file=r5_app.elf}
    }
}
```

destination_cpu

Syntax

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

Description

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

Note:

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

Arguments

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

Example

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

destination_device

Syntax

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

Description

Specifies whether the partition is targeted for PS or PL.

Arguments

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

Example

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

early_handoff

Syntax

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

Description

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

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

Example

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

encryption

Syntax

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

Description

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

Arguments

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

Example

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

exception_level

Syntax

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

Description

Exception level for which the core should be configured.

Arguments

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

Example

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

familykey

Syntax

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

Description

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

Arguments

Path to file.

Example

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

fsbl_config

Syntax

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

Description

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

Arguments

- bh_auth_enable: Boot Header Authentication Enable: RSA authentication of the bootimage will be done excluding the verification of PPK hash and SPK ID.
- auth_only: Boot image is only RSA signed. FSBL should not be decrypted.
- opt_key: Optional key is used for block-0 decryption. Secure Header has the opt key.

- pufhd_bh: PUF helper data is stored in Boot Header. (Default is `efuse`)/ PUF helper data file is passed to bootgen using the [puf_file] option.
- puf4kmode: PUF is tuned to use in 4k bit configuration. (Default is 12k bit). `shutter = <value>` 32 bit `PUF_SHUT` register value to configure PUF for shutter offset time and shutter open time.

Example

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

headersignature

Syntax

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

Description

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

Arguments

`<signature_file>`

Example

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

hivec

Syntax

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

Description

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

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

Arguments

None

Example

A sample BIF file is shown below :

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

init

Syntax

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

Description

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

Example

A sample BIF file is shown below:

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

keysrc_encryption

Syntax

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

Description

This specifies the Key source for encryption.

Arguments

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

Example

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

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

load

Syntax

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

Description

Sets the load address for the partition in memory.

Example

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

offset

Syntax

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

Description

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

Arguments

Specified value and partition.

Example

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

partition_owner

Syntax

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

Description

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

Arguments

- fsbl (default)
- u-boot

Example

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

pid

Syntax

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

Description

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

Example

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

pmufw_image

Syntax

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

Description

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

Arguments

Filename

Example

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

ppkfile

Syntax

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

Description

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

See Using Authentication.

Arguments

Specified file name.

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

Example

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

presign

Syntax

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

Description

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

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

Example

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

pskfile

Syntax

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

Description

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

Arguments

Specified file name.

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

Example

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

puf_file

Syntax

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

Description

PUF helper data file.

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

See Black/PUF Keys for more information.

Example

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

reserve

Syntax

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

Description

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

Arguments

Specified partition

Example

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

split

Syntax

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

Description

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

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

Normal mode splits as follows:

- Bootheader + Image Headers + Partition Headers + Bootloader
- Partiton1
- Partition2 and so on

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

Options

The available options for argument mode are:

- slaveboot
- normal
- bin
- mcs

Example

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

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

spkfile

Syntax

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

Description

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

Arguments

Specified file name.

Example

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

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

spksignature

Code Example

```
[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

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

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 will always use spk_efuse.

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

The user eFUSE specifies which key ID is *not* valid (has been revoked). Hence, the firmware (non-ROM) checks to see if a given user eFUSE that represents the SPK ID has been programmed. `spk_select = user-efuse` indicates that user eFUSE will be used for that partition.

Example

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

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

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

    [
      destination_cpu =a53-0,
      authentication = rsa,
```

```
      spk_select = spk-efuse,
      spk_id =0x12345678,
      sskfile = ssk4.pem
    ] application2.elf
}
```

sskfile

Syntax

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

Description

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

Arguments

Specified file name.

Example

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

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

startup

Syntax

```
[startup=<address_value>] <pattiion>
```

Description

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

Example

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

trustzone

Syntax

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

Description

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

- secure
- nonsecure (default)

Example

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

udf_bh

Syntax

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

Description

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

- zynq: 76 bytes
- zynqmp: 40 bytes

Arguments

Specified file name.

Example

```
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
2664315301020304050607080 90a0b0c0d0e0f101112131415161718191a1b
1c1d1
```

udf_data

Syntax

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

Description

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

Arguments

Specified file name.

Example

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

xip_mode

Syntax

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

Description

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

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

Arguments

Specified partition.

Example

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

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

Chapter 19

Command Reference

arch

Syntax

```
-arch [options]
```

Description

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

Arguments

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

Return Value

None

Example

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

bif_help

Syntax

```
bootgen -bif_help
```

```
bootgen -bif_help aeskeyfile
```

Description

Lists the supported BIF file attributes. For a more detailed explanation of each bif attribute, specify the attribute name as argument to `-bif_help` on the command line.

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 or 32 or 64 or 128).

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>

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:

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

encrypt

Syntax

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

Description

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

Arguments

Key source arguments:

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

bbram: The AES key is stored in BBRAM.

encryption_dump

Syntax

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

Description

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

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

Example

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

fill

Syntax

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

Description

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

Outputs

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

Example

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

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

generate_hashes

Syntax

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

Description

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

Table 34: **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 Zynq® UltraScale+™ MPSoC format:

Table 35: **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

image

Syntax

```
-image <BIF_filename>
```

Description

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

Arguments

bif_filename

Example

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

The Sample BIF file is shown in the following example:

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

log

Syntax

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

Description

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

Arguments

- error: Only the error information is captured.
- warning: The warnings and error information is captured. This is the default value.
- info: The general information and all the above info 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 Xilinx® device. This is needed for generating a 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 partitions. This feature is enabled by default. Specifying a `0` disables this feature. The `boot.bin` has the image header tables and partition header tables in actual and no extra tables are padded. If nothing is specified or if `-padimageheader=1`, the total image header tables and partition header tables are padded to max partitions.

Arguments

- 1: Pad the header tables to max partitions. This is the default value.

- 0: Do not pad the header tables.

Image or Partition Header Lengths

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

process_bitstream

Syntax

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

Description

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

Arguments

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

Returns

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

read

Syntax

```
-read [options] <filename>
```

Description

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

Arguments

- bh: To read boot header from bootimage in human readable form

- iht: To read image header table from bootimage
- ih: To read image headers from bootimage.
- pht: To read partition headers from bootimage
- ac: To read authentication certificates from bootimage
- bootgen -arch zynqmp -read BOOT.bin

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 will be verified against the available partitions. Verification is performed in the following steps:

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

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

verify_kdf

Syntax

```
bootgen -arch zynqmp -verify_kdf testVec.txt
```

Description

The format of the `testVec.txt` file is as below.

```
L = 256
KI = d54b6fd94f7cf98fd955517f937e9927f9536caebe148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4a0c69526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acbdbbcd4f6142d7b8515bad09ec72f7af
```

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

w

Syntax

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

Description

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

Arguments

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

zynqmpes1

Syntax

```
bootgen -arch zynqmp -image test.bif -o boot.bin -zynqmpes1
```

Description

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

Initialization Pairs and INT File Attribute

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

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

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

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

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

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

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

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

```
* = multiply/
 = divide
% = mod
an address value
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.

Chapter 20

Bootgen Utility

CAUTION! *This utility has been deprecated. Instead, use the -read option.*

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

The utility generates the following files as output:

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

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

Usage:

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

Example:

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

Sample output file looks like the following:

Figure 25: **Example Output**

```
Xilinx Bootgen Debug Utility
Version: 2018.2    Date: Jan 03, 2018

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::   B O O T   H E A D E R   ::::::::::::::::::::::
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<Flash Address>     <Offset>          <Description>                   <Interpretation>

[0x00000000]         (0x00)        ARM Vector Table                 - 0xeafffffe
[0x00000004]         (0x04)        ARM Vector Table                 - 0xeafffffe
[0x00000008]         (0x08)        ARM Vector Table                 - 0xeafffffe
[0x0000000c]         (0x0C)        ARM Vector Table                 - 0xeafffffe
[0x00000010]         (0x10)        ARM Vector Table                 - 0xeafffffe
[0x00000014]         (0x14)        ARM Vector Table                 - 0xeafffffe
[0x00000018]         (0x18)        ARM Vector Table                 - 0xeafffffe
[0x0000001c]         (0x1C)        ARM Vector Table                 - 0xeafffffe
[0x00000020]         (0x20)        Width Detection Word             - 0xaa995566
[0x00000024]         (0x24)        Header Signature                 - 0x584c4e58
[0x00000028]         (0x24)        Encryption Key Source            - 0x00      (Not Encrypted)
[0x0000002c]         (0x24)        Header Version                   - 0x1010000
[0x00000030]         (0x30)        Load Image Byte Offset           - 0x202000
[0x00000034]         (0x34)        Load Image Byte Length           - 0x00
[0x00000038]         (0x38)        Image Load Byte Address          - 0x00
[0x0000003c]         (0x3C)        Image Execution Byte Address     - 0xfc202000
[0x00000040]         (0x40)        Total Image Byte Length          - 0x00
[0x00000044]         (0x44)        QSPI Config Word                 - 0x01
[0x00000048]         (0x48)        Header Checksum                  - 0xffd51c40
[0x00000098]         (0x98)        Image Header Table Offset        - 0x8c0
[0x0000009c]         (0x9C)        Partition Header Table Offset    - 0xc80

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
:::::::::::::::::::::::::::::::::::::::::::   R E G I S T E R   I N I T I A L I Z A T I O N   T A B L E   :::
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[0x000000a0]
        Refer file "info_boot_new_register_init_table.txt"
[0x000008a0]

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
:::::::::::::::::::::::::::::::::::::::::::::::::::::   I M A G E   H E A D E R   T A B L E   :::::::::::::
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<Flash Address>     <Offset>          <Description>                   <Interpretation>

[0x000008c0]         (0x00)        Version                          - 0x1020000
[0x000008c4]         (0x04)        Count of Image Headers           - 0x01
[0x000008c8]         (0x08)        Offset to 1st Partition Header   - 0x320
[0x000008cc]         (0x0C)        Offset to 1st Image Header       - 0x240
[0x000008d0]         (0x10)        Offset to Header Auth. Cert.     - 0x00


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::   I M A G E   H E A D E R   ::::::::::::::::::::::
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   Image 1   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<Flash Address>     <Offset>          <Description>                   <Interpretation>

[0x00000900]         (0x00)        Next Image Header Pointer        - 0x00
[0x00000904]         (0x04)        Next 1st Partition Header Pointer - 0x320
[0x00000908]         (0x08)        Partition Count (Wrong Info)     - 0x00
[0x0000090c]         (0x0C)        Image Length (Wrong Info)        - 0x01
[0x00000910]         (0x10)        Image Name                       - fsbl_xip.elf
```

Section IV

Xilinx Software Command-Line Tool

Chapter 21

Xilinx Software Command-Line Tool

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 the 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 will be run nightly or running a set of commands that are used often by the developer.

Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis IDE. As with other Xilinx tools, the scripting language for XSCT is based on Tools Command Language (Tcl). You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions:

- Create hardware, domains, platform projects, system projects, and application projects
- Manage repositories
- Set toolchain preferences
- Configure and build domains/BSPs and applications
- Download and run applications on hardware targets
- Create and flash boot images by running Bootgen and program_flash tools.

This reference content is intended to provide the information you need to develop scripts for software development and debug targeting Xilinx processors.

As you read the document you will notice usage of some abbreviations for various products produced by Xilinx. For example:

- Use of `ps7` in the source code implies that these files are targeting the 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 a 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 Xilinx 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 like optimization and debug information level.

System Requirements

If you plan to use capabilities that are offered through the Vitis IDE or the Xilinx Software Command-Line Tool (XSCT), then you also need to meet the hardware and software requirements that are specific to that capability.

Hardware Requirements

The table below lists the hardware requirements.

Table 36: **Hardware Requirements**

Requirement	Description
CPU Speed	2.2 GHz minimum or higher; Hyper-threading (HHT) or multicore recommended.
Processor	Intel Pentium 4, Intel Core Duo, or Xeon Processors; SSE2 minimum
Memory/RAM	2 GB or higher
Display Resolution	1024×768 or higher at normal size (96 dpi)
Disk Space	Based on the components selected during the installation

Software Requirements

The table below lists the supported operating systems.

***Note*:** 32-bit machine support is now only available through Lab Edition and Hardware Server standalone product installers.

Table 37: **Software Requirements**

Operating System	Supported Version
Windows	• Windows 7 SP1 (64-bit) • Windows 8.1 (64-bit) • Windows 10 Pro (64-bit)

Table 37: **Software Requirements** *(cont'd)*

Operating System	Supported Version
Linux	• Red Hat Enterprise Linux: ◦ 6.6-6.9 (64-bit) ◦ 7.0-7.1 (64-bit) • CentOS: ◦ 6.7-6.8 (64-bit) ◦ 7.2-7.3 (64-bit) • SUSE Linux Enterprise: ◦ 11.4 (64-bit) ◦ 12.2 (64-bit) • Ubuntu Linux 16.04.2 LTS (64-bit) ***Note*:** Additional library installation required.

Chapter 22

XSCT Commands

The Xilinx® Software Command-Line tool allows you to create complete Vitis workspaces, investigate the hardware and software, debug and run the project, all from the command line.

XSCT commands are broadly classified into the following categories. The commands in each category are described subsequently.

- Target Connection Management
- Target Registers
- Program Execution
- Target Memory
- Target Download FPGA/BINARY
- Target Reset
- Target Breakpoints/Watchpoints
- JTAG UART
- Miscellaneous
- JTAG Access
- Target File System
- SVF Operations
- Device Configuration System
- Vitis Projects

TIP:

- Help for each of the commands can be viewed by running `help <command>` or `<command> -help` in the XSCT console. All the available XSCT commands can be listed by running `help commands`.
- You can use **Ctrl+C** to terminate long running commands like `fpga` or `elf download` or `for/while` loops.
- You can terminate XSCT by pressing **Ctrl+C** twice in succession.
- Windows style paths are supported when the path is enclosed within curly brackets `{}`.

Target Connection Management

The following is a list of connections commands:

- connect
- disconnect
- targets
- gdbremote connect
- gdbremote disconnect

connect

Connect to the hw_server/TCF agent.

Syntax

```
connect [options]
```

Allows users to connect to a server, list connections or switch between connections.

Options

Option	Description
`-host <host name/ip>`	Name/IP address of the host machine
`-port <port num>`	TCP port number
`-url <url>`	URL description of the hw_server/TCF agent
`-list`	List open connections
`-set <channel-id>`	Set active connection
`-new`	Create a new connection, even one exist to the same url
`-xvc-url <url>`	Open Xilinx Virtual Cable connection
`-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

Example(s)

```
connect -host localhost -port 3121
```

Connect to the hw_server/TCF agent on host localhost and port 3121.

```
connect -url tcp:localhost:3121
```

Identical to previous example.

disconnect

Disconnect from the 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 comibination with -filter option. An error will be generate if list is empty or contains more than one entry.

Option	Description
`-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 references by name, while Tcl variables are accessed using the $ syntax, string must be quoted. Operators ==, !=, <=, >=, <, >, && and || are supported as well as (). There 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 dict's containing target properties.
`-index <index>`	Include targets based on 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. The timeout value is in seconds. Default timeout is 3 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.

Example(s)

```
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 1st Jtag Device.

gdbremote connect

Connect to GDB remote server.

Syntax

```
gdbremote connect [options] server
```

Connect to a GDB remote server, for example qemu. A special client named tcfgdbclient is used to connect to remote GDB server.

Options

Option	Description
`-architecture <name>`	Specify default architecture is remote server does not provide it.

Returns

Nothing, if the connection is successful. Error string, if the connection failed.

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, 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
`-defs`	Read register definitions instead of values
`-no-bits`	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.

Example(s)

```
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, processor core register can be written to. For a target representing a group of processor cores, system registers or IOU registers can be written.

Returns

Nothing, if successful. Error string, if an invalid register is specified or the register cannot be written.

Example(s)

```
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
- stpout
- dis
- print
- locals
- backtrace
- 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.

Example(s)

```
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 5 sec 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 instruction is function call, stop at 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 instruction is 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.

stpout

Step out from current function.

Syntax

```
stpout [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 current PC. Default value for `<num>` is 1.

Returns

Disassembled instructions if successful. Error string, if the target instructions cannot be read.

Example(s)

```
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 a complex data types like a structure can be accessed through '.' operator. For example, var1.int_type refers to int_type element in 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
`-add <expression>`	Add the `<expression>` to auto expression list. The values or definitions of the expressions in auto expression list are displayed when expression name is not specified. Frequently used expressions should be added to the auto expression list.
`-defs [expression]`	Return the expression definitions like address, type, size and RW flags. Not all definitions are available for all the expressions. For example, address is available only for variables and not when the expression includes an operator.
`-dict [expression]`	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.
`-remove [expression]`	Remove the expression from auto expression list. Only expressions previously added to the list through -add option can be removed. When the expression name is not specified, all the expressions in the auto expression list are removed.
`-set <expression>`	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.

`<none>` or -add: Expression value(s)

`-defs`: Expression definition(s)

`-remove or -set`: Nothing

Error string, if expression value cannot be read or set.

Example(s)

```
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 variable name and value are not specified, values of all the local variables are returned. Elements of a complex data types like a structure can be accessed through '.' operator. For example, var1.int_type refers to int_type element in 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.
`-dict [expression]`	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.

Example(s)

```
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 int_type element in var1 struct, where var1 is a member of 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 var1 struct, which is in turn a member of tmp_var struct.

backtrace

Stack back trace.

Syntax

```
backtrace
```

Return stack trace for current target. Target must be stopped. Use debug information for best result.

Returns

Stack Trace, if successful. Error string, if Stack Trace cannot be read from the target.

profile

Configure and run the GNU profiler.

Syntax

```
profile [options]
```

Configure and run the GNU profiler. The profiling needs to enabled while building bsp and application to be profiled.

Options

Option	Description
`-freq <sampling-freq>`	Sampling frequency.
`-scratchaddr <addr>`	Scratch memory for storing the profiling related data. It needs to be assigned carefully, as it should not overlap with the program sections.
`-out <file-name>`	Name of the output file for writing the profiling data. This option also runs the profiler and collects the data. If 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.

Example(s)

```
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 MB. The output file is generated in gmon.out format. The results can be viewed using gprof editor. In case of cycle count, an annotated disassembly file is also generated clearly marking 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 100MHz.
`-count-instr`	Count no. of executed instructions. By default no. 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 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.

Example(s)

```
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 no. 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`	Execute the command and output trace. ***Note:*** The -con option is only available with embedded trace.
`-stp`	
`-nxt`	

Option	Description
`-out <filename>`	Output trace data to a file. `<filename>` Name of the output file for writing the trace data. If not specified, data is output to standard output.
`-level <level>`	Set the trace level to "full", "flow", "event", or "cycles". If not specified, "flow" is used.
`-halt`	Set to halt program execution when the trace buffer is full. If not specified, trace is stopped but program execution continues.
`-save`	Set to enable capture of load and get instruction new data value.
`-low <addr>`	Set low and high address of the external trace buffer address range. The address range must indicate an unused accessible memory space. Only used with external trace.
`-high <addr>`	
`-format <format>`	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.

Example(s)

```
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
`-force`	Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.
`-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 will be aligned to access-size before reading memory, if '-unaligned-access' option is not used. For targets which do not support double-word access, debugger uses 2 word accesses. If number of data values to be read is more than 1, then debugger selects appropriate access size. For example, 1. mrd -size b 0x0 4 Debugger accesses one word from the memory, displays 4 bytes. 2. mrd -size b 0x0 3 Debugger accesses one half-word and one byte from the memory, displays 3 bytes. 3. mrd 0x0 3 Debugger accesses 3 words from the memory and displays 3 words.
`-value`	Return a Tcl list of values, instead of displaying the result on console.
`-bin`	Return data read from the target in binary format.
`-file <file-name>`	Write binary data read from the target to `<file-name>`.

Option	Description
`-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 -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 - 0xfffc, where the higher 8 bits select an AP and lower 8 bits are the register address for that AP.
`-unaligned-access`	Memory address is not aligned to 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 a 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.

Example(s)

```
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 3 bytes at address 0x1.

```
mrd -size h 0x2 2
```

Read 2 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 8 bits (0x1) select the APB-AP and lower 8 bits (0x0) is the address of CSW.

```
mrd -address-space APR 0x04
```

Read AHB-AP TAR on Zynq. The higher 8 bits (0x0) select the AHB-AP and lower 8 bits (0x4) is the address of TAR.

```
mrd -address-space AP1 0x80090088
```

Read address 0x80090088 on DAP APB-AP. 0x80090088 corresponds to DBGDSCR register of Cortex-A9#0, on Zynq AP 1 selects the APB-AP.

```
mrd -address-space AP0 0xe000d000
```

Read address 0xe000d000 on DAP AHB-AP. 0xe000d000 corresponds to QSPI device on Zynq AP 0 selects the AHB-AP.

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 specifed 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 specifed 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, debugger flushes/invalidates caches to make sure caches are in sync.

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 will be aligned to accesss-size before writing to memory, if '-unaligned-access' option is not used. If target does not support double-word access, the debugger uses 2 word accesses. If number of data values to be written is more than 1, then debugger selects appropriate access size. For example, 1. mwr -size b 0x0 {0x0 0x13 0x45 0x56} Debugger writes one word to the memory, combining 4 bytes. 2. mwr -size b 0x0 {0x0 0x13 0x45} Debugger writes one half-word and one byte to the memory, combining the 3 bytes. 3. mwr 0x0 {0x0 0x13 0x45} Debugger writes 3 words to the memory.
-bin	Read binary data from a file and write it to target address space.
-file <file-name>	File from which binary data is read to write to 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 -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 - 0xfffc, where the higher 8 bits select an AP and lower 8 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 a APU target to access ARM DAP and MEM-AP address space.

Returns

Nothing, if successful. Error string, if the target memory cannot be written.

Example(s)

```
mwr 0x0 0x1234
```

Write 0x1234 to address 0x0.

```
mwr 0x0 {0x12 0x23 0x34 0x45}
```

Write 4 words from the list of values to address 0x0.

```
mwr 0x0 {0x12 0x23 0x34 0x45} 10
```

Write 4 words from the list of values to address 0x0 and fill the last word from the list at remaining 6 address locations.

```
mwr -size b 0x1 {0x1 0x2 0x3} 3
```

write 3 bytes from the list at address 0x1.

```
mwr -size h 0x2 {0x1234 0x5678} 2
```

write 2 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 8 bits (0x1) select the APB-AP and lower 8 bits (0x0) is the address of CSW.

```
mwr -arm-dap 0x04 0xf8000120
```

Write 0xf8000120 to AHB-AP TAR on Zynq The higher 8 bits (0x0) select the AHB-AP and lower 8 bits (0x4) is the address of TAR.

```
mwr -arm-ap 1 0x80090088 0x03186003
```

Write 0x03186003 to address 0x80090088 on DAP APB-AP 0x80090088 corresponds to DBGDSCR register of Cortex-A9#0, on Zynq AP 1 selects the APB-AP.

```
mwr -arm-ap 0 0xe000d000 0x80020001
```

Write 0x80020001 to address 0xe000d000 on DAP AHB-AP 0xe000d000 corresponds to QSPI device on Zynq AP 0 selects the AHB-AP.

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 specifed and only one symbol file exists in target's memory map, then that symbol file is used. If no symbol file is specifed and multiple symbol files exist in target's memory map, then 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 will not be tracked for debug and are not visible in the debug targets view.
`-fast-step`	Enable fast stepping. Only the current process will be re-synced after stepping. All other processes will not be re-synced when this flag is turned on.

Note(s)

- fast-exec and fast-step options are not valid with disable option.

Returns

Nothing, if OSA is configured successfully. Error, if ambiguous options are specified.

Example(s)

```
osa -file <symbol-file> -fast-step -fast-exec
```

Enable OSA for `<symbole-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.
`-size <memory-size>`	Size of the memory region.

Option	Description
`-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 memmap command can be removed.

Returns

Nothing, while setting the memory map, or list of memory maps when -list option is used.

Example(s)

```
memmap -addr 0xfc000000 -size 0x1000 -flags 3
```

Add the memory region 0xfc000000 - 0xfc000fff to 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 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
`-clear`	Clear uninitialized data (bss).
`-keepsym`	Keep previously downloaded elfs in the list of symbol files. Default behavior is to clear the old symbol files while downloading an elf.
`-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 elf download. Without this option, debugger flushes/invalidates caches to make sure caches are in sync.
`-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 debug symbol info for the code. `<addr>` is the relative address, to which all the program sections are relocated.
`-vaddr`	Use vaddr 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 `<file>` is downloaded correctly to active target.

```
verify -data <file> <addr>
```

Verify if the binary file `<file>` is downloaded correctly to active target address specified by `<addr>`.

Options

Option	Description
`-force`	Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.
`-vaddr`	Use vaddr 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
`-file <bitstream-file>`	Specify file containing bitstream.
`-partial`	Configure FPGA without first clearing current configuration. This options should be used while configuring partial bitstreams created before 2014.3 or any partial bitstreams in binary format.
`-no-revision-check`	Disable bitstream vs silicon revision revision compatibility check.
`-skip-compatibility-check`	Disable bitstream vs FPGA device compatibility check.
`-state`	Return whether the FPGA is configured.
`-config-status`	Return configuration status.
`-ir-status`	Return IR capture status.
`-boot-status`	Return boot history status.

Option	Description
`-timer-status`	Return watchdog timer status.
`-cor0-status`	Return configuration option 0 status.
`-cor1-status`	Return configuration option 1 status.
`-wbstar-status`	Return warm boot start address status.

Note(s)

- If no target is selected or if the current target is not a supported FPGA device, and only one supported FPGA device is found in the targets list, then this device will be 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 Configutation 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
`-processor`	Reset the active processor target.
`-cores`	Reset the active processor group. This reset type is supported only on Zynq. A processor group is defined as a set of processors and on-chip peripherals like OCM.
`-system`	Reset the active System.

Option	Description
`-srst`	Generate system reset for active target. With JTAG this is done by generating a pulse on the SRST pin on the JTAG cable assocated with the active target.
`-por`	Generate power on reset for active target. With JTAG this is done by generating a pulse on the POR pin on the JTAG cable assocated with the active target.
`-ps`	Generate PS only reset on Zynq MP. This is supported only through MicroBlaze PMU target.

Returns

Nothing, if reset if successful. Error string, if reset is unsupported.

Target Breakpoints/Watchpoints

The following is a list of breakpoints commands:

- bpadd
- bpremove
- bpenable
- 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.

Option	Description
`-line <line-number>`	Specify the `<line-number>` within the file, where 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 will trigger the breakpoint. `<breakpoint-mode>` can be a bitwise OR of the values below: 0x1 = Triggered by a read from the breakpoint location 0x2 = Triggered by a write to the breakpoint location 0x4 = Triggered by an instruction execution at the breakpoint location This is the default for Line and Address breakpoints 0x8 = 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 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 = supress trigger on stepping over a code breakpoint. 2 = supress trigger on any kind of stepping.
`-properties <dict>`	Specify advanced breakpoint properties.
`-meta-data <dict>`	Specify meta-data of advanced breakpoint properties.
`-target-id <id>`	Specify a target id for which the breakpoint should be set. A breakpoint can be set for all the targets by specifying the `<id>` as "all". If this option is not used, then the breakpoint is set for the active target selected through targets command. If there is no active target, then the breakpoint is set for all targets.

Note(s)

- Breakpoints can be set in XSDB before connecting to the hw_server/TCF agent. If there is an active target when a Breakpoint is set, the Breakpoint will be enabled only for that active target. If there is no active target, the Breakpoint will be enabled for all the targets. 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.

Example(s)

```
bpadd -addr 0x100000
```

Set a Breakpoint at address 0x100000. Breakpoint type is chosen by the hw_server/TCF agent.

```
bpadd -addr &main
```

Set a function Breakpoint at main. Breakpoint type is chosen by the 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 \"-all\" option is used.

Options

Option	Description
`-all`	Remove all breakpoints.

Returns

Nothing, if the breakpoint is removed successfully. Error string, if the breakpoint specified by `<id>` is not set.

Example(s)

```
bpremove 0
```

Remove Breakpoint 0.

```
bpremove 1 2
```

Remove Breakpoints 1 and 2.

```
bpremove -all
```

Remove all Breakpoints.

bpenable

Enable Breakpoints/Watchpoints.

Syntax

```
bpenable <id-list> | -all
```

Enable the Breakpoints/Watchpoints specified by `<id-list>` or enable all the breakpoints when \"-all\" option is used.

Options

Option	Description
`-all`	Enable all breakpoints.

Returns

Nothing, if the breakpoint is enabled successfully. Error string, if the breakpoint specified by `<id>` is not set.

Example(s)

```
bpenable 0
```

Enable Breakpoint 0.

```
bpenable 1 2
```

Enable Breakpoints 1 and 2.

```
bpenable -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 \"-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.

Example(s)

```
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 Beakpoints/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 Breakpoint hitcount 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 hyper-terminal.

Syntax

```
jtagterminal [options]
```

Start/Stop a Jtag based hyper-terminal to communicate with ARM DCC or MDM UART interface.

Options

Option	Description
`-start`	Start the Jtag Uart terminal. This is the default option.
`-stop`	Stop the Jtag Uart terminal.
`-socket`	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 runnning 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. 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 a MDM or ARM/MicroBlaze processor target before runnning this command.

- While running a script in non-interactive mode, output from Jtag uart may not 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.

Example(s)

```
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
- unloadhw
- mdm_drwr
- mb_drwr
- mdm_drrd
- mb_drrd
- configparams
- version
- xsdbserver start
- xsdbserver stop
- xsdbserver disconnect
- xsdbserver version

loadhw

Load a Vivado HW design.

Syntax

```
loadhw [options]
```

Load a Vivado HW design, and set the memory map for the current target. If the current target is a parent for a group of processors, memory map is set for all its child processors. If current target is a processor, memory map is set for all the child processors of it's parent. This command returns the HW design object.

Options

Option	Description
`-hw`	HW design file.
`-list`	Return a list of open designs for the targets.
`-mem-ranges [list {start1 end1} {start2 end2}]`	List of memory ranges from which the memory map should be set. Memory map is not set for the addresses outside these ranges. If this option is not specified, then memory map is set for all the addresses in the hardware design.

Returns

Design object, if the HW design is loaded and memory map is set successfully. Error string, if the HW design cannot be opened.

Example(s)

targets -filter {name =~ "APU"}; loadhw design.xsa Load the HW design named design.xsa and set memory map for all the child processors of APU target. targets -filter {name =~ "xc7z045"}; loadhw design.xsa Load the HW design named design.xsa and set memory map for all the child processors for which xc7z045 is the parent.

unloadhw

Unload a Vivado HW design.

Syntax

```
unloadhw
```

Close the Vivado HW design which was opened during loadhw command, and clear the memory map for the current target. If the current target is a parent for a group of processors, memory map is cleared for all its child processors. If the current target is a processor, memory map is cleared for all the child processors of it's parent. This command does not clear memory map explicitly set by users.

Returns

Nothing.

mdm_drwr

Write to MDM Debug Register.

Syntax

```
mdm_drwr [options] <cmd> <data> <bitlen>
```

Write to MDM Debug Register. cmd is 8-bit MDM command to access a Debug Register. data is the register value and bitlen is the register width.

Options

Option	Description
`-target-id <id>`	Specify a target id representing MicroBlaze Debug Module or MicroBlaze instance to access. If this option is not used and `-user` is not specified, then the current target is used.
`-user <bscan number>`	Specify user bscan port number.

Returns

Nothing, if successful.

Example(s)

```
mdm_drwr 8 0x40 8
```

Write to MDM Break/Reset Control Reg.

mb_drwr

Write to MicroBlaze Debug Register.

Syntax

```
mb_drwr [options] <cmd> <data> <bitlen>
```

Write to MicroBlaze Debug Register available on MDM. cmd is 8-bit MDM command to access a Debug Register. data is the register value and bitlen is the register width.

Options

Option	Description
`-target-id <id>`	Specify a target id representing MicroBlaze instance to access. If this option is not used and -user is not specified, then the current target is used.
`-user <bscan number>`	Specify user bscan port number.
`-which <instance>`	Specify MicroBlaze instance number.

Returns

Nothing, if successful.

Example(s)

```
mb_drwr 1 0x282 10
```

Write to MB Control Reg.

mdm_drrd

Read from MDM Debug Register.

Syntax

```
mdm_drrd [options] <cmd> <bitlen>
```

Read a MDM Debug Register. cmd is 8-bit MDM command to access a Debug Register and bitlen is the register width. Returns hex register value.

Options

Option	Description
`-target-id <id>`	Specify a target id representing MicroBlaze Debug Module or MicroBlaze instance to access. If this option is not used and
`-user is not specified, then the current target is used.`	
`-user <bscan number>`	Specify user bscan port number.

Returns

Register value, if successful.

Example(s)

```
mdm_drrd 0 32
```

Read XMDC ID Reg.

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
`-target-id <id>`	Specify a target id representing MicroBlaze instance to access. If this option is not used and -user is not specified, then the current target is used.
`-user <bscan number>`	Specify user bscan port number.
`-which <instance>`	Specify MicroBlaze instance number.

Returns

Register value, if successful.

Example(s)

```
mb_drrd 3 28
```

Read MB Status Reg.

configparams

List, get or set configuration parameters.

Syntax

```
configparams <options>
```

List 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 may change depending on current connection.

```
configparams <options> <name>
```

Get configuration parameter value(s).

```
configparams <options> <name> <value>
```

Set configuration parameter value.

Options

Option	Description
`-all`	Include values for all contexts in result.
`-context [context]`	Specify context of value to get or set. The default context is "" which represet the global default. Not all options support context specific values.
`-target-id <id>`	Specify target id or value to get or set. This is an alternative to the -context 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 unsupported parameter is specified.

Example(s)

```
configparams force-mem-accesses 1
```

Disable access protection for 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 TCF server version.

Syntax

```
version [options]
```

Get Vitis or TCF server version. When no option is specified, Vitis build version is returned.

Options

Option	Description
`-server`	Get the TCF server build version, for the active connection.

Returns

Vitis or TCF 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. 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 the 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 incomming connections.
`-port <port>`	Specifies port to listen on. If this option is not specified or if the port is zero then a dynamically allocated port number is used.

Returns

Server details are disaplayed on the console if server is started. successfully, or error string, if a server has been already started.

Example(s)

```
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 incomming 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 comibination with -filter option. An error will be generate if list is empty or contains more than one entry.
`-regexp`	Use regexp for filter matching.
`-nocase`	Use case insensitive filter matching.

Option	Description
`-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 references by name, while Tcl variables are accessed using the $ syntax, string must be quoted. Operators ==, !=, <=, >=, <, >, && and || are supported as well as (). There 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 3 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.

Example(s)

```
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) then state machine will move 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 will be 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 Cature 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 delay between sequence commands. No JTAG clocks will be 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 is cable specific.

```
sequence set_pin pin value
```

Set value of <pin> to <value>. Supported pins is 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 will return the result from shift commands using -capture option and from get_pin commands. Available options: -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.

Example(s)

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.

Example(s)

```
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 will ensure 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 since 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 will attept 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 will return 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 use to implement support for different types of JTAG cables. An open JTAG server will enumberate or connect to available JTAG ports.

Options

Option	Description
`-list`	List opened servers. This is the default if no other option is given.
`-format`	List format of supported server strings.
`-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.

Example(s)

```
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

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

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

tfile rmdir

Remove directory

Syntax

```
tfile rmdir <path>
```

Remove directory `<path>`

Returns

tfile mkdir

Create directory

Syntax

```
tfile mkdir <path>
```

Make directory `<path>`

Returns

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

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

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 content

Returns

Directory content

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 config

Configure options for SVF file

Syntax

```
svf config [options]
```

Configure and generate SVF file.

Options

Options	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 used to specify the chunk size in bytes for each transaction while downloading. Supported only for Microblaze processors.

Returns

Nothing

Example(s)

```
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.

Example(s)

```
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.

Example(s)

```
svf mwr 0xffff0000 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.

Example(s)

```
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

Example(s)

```
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

Example(s)

```
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

Example(s)

```
svf delay 1000
```

Delay of 1000 tcks is added to the SVF file.

Device Configuration System

The following is a list of device commands:

- device program
- device status

device program

Program PDI/BIT

Syntax

```
device program <file>
```

Program PDI or BIT file into device device.

***Note*:** 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.

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.
`-hex`	Format the return data in hexadecimal.

Returns

Status report

Vitis Projects

The following is a list of projects commands:

- getaddrmap
- getperipherals
- repo
- platform
- domain
- bsp
- library
- setws
- getws
- app
- sysproj
- importprojects
- importsources
- toolchain

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.

Example(s)

```
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.

Example(s)

```
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.

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
`-set <path-list>`	Set the repository path and load all the software cores available. Multiple repository paths can be specified as Tcl list.
`-get`	Get the repository path(s).
`-scan`	Scan the repositories. Used this option to scan the repositories, when some changes are done.
`-os`	Return a list of all the OS from the repositories.
`-libs`	Return a list of all the libs from the repositories.
`-drivers`	Return a list of all the drivers from the repositories.
`-apps`	Return a list of all the applications from the repositories.
`-add-platforms <platform-name>`	Add the platform specified by `<platform-name>` to the 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.

Example(s)

```
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.

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 `<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. fsbl - Specify extra compiler/linker flags for fsbl. generate - Build the platform. list - List all the platforms in workspace. pmufw - Specify extra compiler/linker flags for pmufw. 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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command. Refer to sub-command help for details.

Example(s)

Refer to sub-command help for details.

platform active

Set/Get active platform

Syntax

```
platform active [platform-name]
```

Set or get the active platform. If platform-name is specified, it is made as active platform, otherwise the name of active platform is returned. If no active platform exists, this command returns an empty string.

Options

None

Returns

Empty string, if a platform is set as active or no active platform exists. Platform name, when active platform is read.

Example(s)

```
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 like fsbl, pmufw etc.

Options

None

Returns

Nothing. Build log will be printed on the console.

Example(s)

platform active zcu102

```
platform clean
```

Set zcu102 as 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 application. "repo -apps `<platform-name>`" can be used to list the application templates available for the given platform-name.
`-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.

Returns

Empty string, if the platform is configured successfully. Error string, if no platform is active or if the platform cannot be configured.

Example(s)

platform active zc702

```
platform config -desc "ZC702 with memory test application"
```

-samples /home/user/newDir Make zc702 as active platform, configure the description of the platform and make samples in /home/user/newDir part of the platform.

```
platform config -updatehw /home/user/newdesign.xsa
```

Updates the platform project with the new xsa.

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 hw 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 is not needed as the platform will be created in workspace. If the workspace is not set and this option is not specified, then platform will be generated in current working directory.
`-prebuilt`	Mark the platform to be built from already built sw artifacts. This option should be used only if you have existig software platform artifacts.
`-proc <processor>`	The processor to be used; the tool will create default domain.
`-samples <samples-directory>`	Make the samples in `<samples-directory>`, part of the platform.
`-os <os>`	The os to be used; the tool will create 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.

Returns

Empty string, if the platform is created successfully. Error string, if the platform cannot be created.

Example(s)

```
platform create -name "zcu102_test" -hw zcu102
```

Defines a software platform for a pre-defined hardware desciption file.

```
platform create -name "zcu102_test" -hw zcu102 -proc psu_cortexa53_0 -os
standalone
```

Defines a software platform for a pre-defined hardware desciption file. Create a default domain with standalone os running on psu_cortexa53_0.

```
platform create -xpfm /path/zc702.xpfm
```

This will create 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 desciption file.

platform fsbl

Configure fsbl

Syntax

```
platform fsbl
```

Configure extra compiler and linker flags for fsbl.

Options

Option	Description
`-extra-compiler-flags <flags>`	Set extra compiler flags for fsbl to the flags specified by `<flags>`.
`-extra-linker-flags <flags>`	Set extra linker flags for fsbl to the flags specified by `<flags>`.
`-report`	Return a table of extra compiler and linker flags set for fsbl.

Returns

Empty string, if the flag is set successfully. Error string, if the flag cannot be set.

Example(s)

```
platform fsbl -extra-compiler-flags "-DFSBL_DEBUG_INFO"
```

Add -DFSBL_DEBUG_INFO to the compiler options, while building the fsbl application.

```
platform fsbl -report
```

Return table of extra compiler and extra linker flags that are set.

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 plafform are built.

Returns

Empty string, if the platform is generated successfully. Error string, if the platform cannot be built.

Example(s)

```
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

None

Returns

List of platforms, or "No active platform present" string if no platforms exist.

Example(s)

```
platform list
```

Return a list of all the platforms in the workspace and repository.

platform pmufw

Configure pmufw

Syntax

```
platform pmufw
```

Configure pmufw to build with extra compiler and linker flags.

Options

Option	Description
`-extra-compiler-flags <value>`	Set extra compiler flag for pmufw with the provided value.
`-extra-linker-flags <value>`	Set extra linker flag for pmufw with the provided value.
`-report`	Return the list of the flags set to pmufw.

Returns

Empty string, if the flag is set successfully. Error string, if the flag cannot be set.

Example(s)

```
platform pmufw -extra-compiler-flags "-DDEBUG_INFO"
```

Add -DDEBUG_INFO to the compiler options, while building the pmufw application.

platform read

Read from the platform file

Syntax

```
platform read [platform-file]
```

Read platform settings from the platform file and makes it available for edit. Platform file gets created during the creation of platform itself and it contains all details of platform like hw specification file, processor information etc

Options

None

Returns

Empty string, if the platform is read successfully. Error string, if the platform file cannot be read.

Example(s)

```
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 platform-name is not specified, active platform is deleted.

Options

None

Returns

Empty string, if the platform is deleted successfully. Error string, if the platform cannot be deleted.

Example(s)

```
platform remove zc702
```

Removes zc702 platform from the disk.

platform report

Report the details of a platform

Syntax

```
platform report [platform-name]
```

Return details like domains, processors, etc. created in a platform. If platform-name is not specified, details of the active platform are returned.

Options

None

Returns

Table with details of platform, or error string if no platforms exist.

Example(s)

```
platform report
```

Return a table with details of the active platform.

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 "platform read" command.

Options

None

Returns

Empty string, if the platform settings are written successfully. Error string, if the platform settings cannot be written.

Example(s)

```
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 `<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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command. Refer to sub-command help for details.

Example(s)

Refer to 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 as active domain, otherwise the name of 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.

Example(s)

```
domain active
```

Return the name of the active domain .

```
domain active test_domain
```

Set test_domain as active domain.

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 as active domain, otherwise the name of 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.

Example(s)

```
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
`-display-name <display name>`	Display name of the domain.

Option	Description
`-desc <description>`	Brief description about the domain.
`-image <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.
`-sw-repo <repositories-list>`	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.
`-mss <mss-file>`	Use mss from specified by `<mss-file>`, instead of generating mss file for the domain.
`-prebuilt-data <directory-name>`	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.
`-readme <file-name>`	Add README file for the domain, with boot instructions, etc.
`-inc-path <include-path>`	Additional include path which should be added while building the application created for this domain.
`-lib-path <library-path>`	Additional library search path which should be added to the linker settings of the application created for this domain.
`-sysroot <sysroot-dir>`	The Linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build.
`-boot <boot-dir>`	Directory to generate components after Linux image build.
`-bif <file-name>`	Bif file used to create boot image for Linux boot.
`-qemu-args <file-name>`	File with all PS QEMU args listed. This is used to start PS QEMU.
`-pmuqemu-args <file-name>`	File with all PMC QEMU args listed. This is used to start PMU QEMU.
`-pmcqemu-args <file-name>`	File with all pmcqemu args listed. This is used to start pmcqemu.
`-qemu-data <data-dir>`	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.

Example(s)

```
domain config -display-name zc702_MemoryTest
```

-desc "Memory test application for Zynq" -prebuilt-data /home/user/build_dir/ Configure display name, description, and set prebuilt-data directory 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 about 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.
`-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. "repo -apps" command can be used to list the available application.
`-auto-generate-linux`	Generate the Linux artifacts automatically.
`-image <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.

Example(s)

```
domain create -name "ZUdomain" -os standalone -proc psu_cortexa53_0
```

-support-app {Hello World} Create a standalone domain and configure settings needed for "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 list

List domains

Syntax

```
domain list
```

List domains in the active platform.

Options

None

Returns

List of domains in the active platform, or empty string if no domains exist.

Example(s)

platform active platform1

```
domain list
```

Display all the domain created in platform1.

domain remove

Delete a domain

Syntax

```
domain remove [domain-name]
```

Delete a domain from active platform. If domain-name is not specified, active domain is deleted.

Options

None

Returns

Empty string, if the domain is deleted successfully. Error string, if the domain deletion fails.

Example(s)

```
domain remove test_domain
```

Removes test_domin from the active platform.

domain report

Report the details of a domain

Syntax

```
domain report [domain-name]
```

Return details like platform, processor core, OS, etc. 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.

Example(s)

```
domain report
```

Return a table with details of the active domain.

bsp

Configure bsp settings of baremetal domain

Syntax

```
bsp <sub-command> [options]
```

Configure the bsp settings which includes library, driver and OS version of a active domain, based on `<sub-command>` specified. Following sub-commands are supported. config - Modify the configurable parameters of bsp settings. getdrivers - List IP instance and it's 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. 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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command. Refer to sub-command help for details.

Example(s)

Refer to 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>` and `<value>` are not specified, returns the details of all configurable parameters of processor, os, or all libraries in BSP. If `<param>` is specified and `<value>` 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 paramter if `<value>` is not specified. Error string, if the parameter cannot be set/Appended.

Example(s)

```
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

None

Returns

Table with IP, it's corresponding driver and driver version. Empty string, if there are no IP's.

Example(s)

```
bsp getdrivers
```

Return the list of IP's and it's driver.

bsp getlibs

list libraries added in the bsp settings

Syntax

```
bsp getlibs
```

Display list of libraries added in the bsp settings.

Options

None

Returns

List of library/(ies). Empty string, if there are no library added.

Example(s)

```
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 it's version.

Options

None

Returns

OS name and it's version.

Example(s)

```
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 Library in BSP.
`-os`	Return the configurable parameters of OS in BSP.
`-proc`	Return the configurable parameters of processor in BSP.

Returns

parameter names, empty string, if no parameter exist.

Example(s)

```
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 BSP.

Options

None

Returns

Nothing, if the bsp is generated successfully. Error string, if the bsp cannot be generated.

Example(s)

```
bsp regenerate
```

Regenerate the BSP sources with the changes done in the BSP settings.

bsp removelib

Remove library from bsp settings

Syntax

```
bsp removelib -name <lib-name>
```

Remove the library from bsp settings of the active domain.

Options

Option	Description
`-name <lib-name>`	Library to be removed from bsp settings.

Returns

Nothing, if the library is removed successfully. Error string, if the library cannot be removed.

Example(s)

```
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.

Example(s)

```
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]
```

Add the library to the bsp settings of active domain.

Options

Option	Description
`-name <lib-name>`	Library to be added to the bsp settings.
`-ver <version>`	Library version.

Returns

Nothing, if the library is set successfully. Error string, if the library cannot be set.

Example(s)

```
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 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.

Example(s)

```
bsp setosversion -ver 6.6
```

Set the OS version 6.6 in 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 `<sub-command>` specified. Following sub-commands are supported. build - Build the library project. clean - Clean 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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command.

Example(s)

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.

Example(s)

```
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.

Example(s)

```
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, then active platform and domain are used for Creating library project. For creating library project and adding them to existing system project, refer to 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, then active domain is used.

Options

Option	Description
-name <project-name>	Project name that should be created.
-type <library-type>	<library-type> can be 'static' or 'shared'
-platform <platform-name>	Name of the platform. Use "repo -platforms" to list available pre-defined platforms.
-domain <domain-name>	Name of the domain. Use "platform report <platform-name>" to list the available domains in a platform.
-sysproj <system-project>	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.

Example(s)

```
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.

Example(s)

```
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.

Example(s)

```
library remove lib1
```

Removes lib1 from workspace.

library report

Report details of the library project

Syntax

```
library report <project-name>
```

Return details like platform, domain etc. of the library project.

Options

None

Returns

Details of the library project, or error string, if library project does not exist.

Example(s)

app report lib1 Return all the details of library lib1.

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.

Example(s)

```
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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command. Refer to sub-command help for details.

Example(s)

Please refer to 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.

Returns

Nothing. Build log will be printed on the console.

Example(s)

```
app build -name helloworld
```

Build helloworld application.

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.

Example(s)

```
app clean -name helloworld
```

Clean helloworld 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.
`-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> <paramater value>`: Nothing if the value is set successfully, or error, if unsupported parameter is specified.

Example(s)

```
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
```

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.

Options

Option	Description
`-name <application-name>`	Name of the application to be created.
`-platform <platform-name>`	Name of the platform. Use "repo -platforms" to list available pre-defined platforms.
`-domain <domain-name>`	Name of the domain. Use "platform report `<platform-name>`" to list the available system configurations in a platform.
`-hw <hw-spec>`	HW specification file exported from Vivado (XSA).
`-sysproj <system-project>`	Name of the system project. Use "sysproj list" to know available system projects in the workspace.
`-proc <processor>`	Processor core for which the application should be created.
`-template <application template>`	Name of the template application. Default is "Hello World". Use "repo -apps" to list available template applications.
`-os <os-name>`	OS type. Default type is standalone.
`-lang <programming language>`	Programming language can be c or c++.

Returns

Nothing, if the application is created successfully. Error string, if the application creation fails.

Example(s)

```
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

None

Returns

List of applications in the workspace. If no applications exist, "No application exist" string is returned.

Example(s)

```
app list
```

Lists all the applications in the workspace.

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.

Example(s)

```
app remove zynqapp
```

Removes zynqapp from workspace.

app report

Report details of the application

Syntax

```
app report <app-name>
```

Return details like platform, domain, processor core, OS, etc. of an application.

Options

None

Returns

Details of the application, or error string, if application does not exist.

Example(s)

```
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.

Option	Description
`-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 avaliable 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.

Example(s)

```
app switch -name helloworld -platform zcu102
```

Switch the helloworld 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

Depends on the sub-command. Refer to sub-command help for details.

Returns

Depends on the sub-command.

Example(s)

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.

Example(s)

```
sysproj build -name helloworld_system
```

Build the system project specified.

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 suceessfully. Error string, if the application build clean fails.

Example(s)

```
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.

Example(s)

```
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.

Example(s)

```
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 like platform, domain, etc. of a system project.

Options

None

Returns

Details of the system project, or error string, if system project does not exist.

Example(s)

```
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.

Example(s)

```
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.
`-path <source-path>`	Path from which the source files should be imported. If `<source-path>` is a file, it is imported to application project. If `<source-path>` is a directory, all the files/sub-directories from the `<source-path>` 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 -linker-script option is used.
`-linker-script`	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.

Example(s)

```
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.

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

Chapter 23

XSCT Use Cases

As with Vitis IDE, the first step to use Xilinx Software Command-line Tool (XSCT) involves selecting a workspace. For creating and managing projects, XSCT launches Vitis IDE in the background. XSCT workspaces can be seamlessly used with Vitis IDE and vice-versa.

Note: At any given point of time, a workspace can either be used only from Vitis IDE or XSCT.

The following is a list of use cases describing how you can use the tool to perform common tasks:

- Running Tcl Scripts
- Creating an Application Project Using an Application Template (Zynq UltraScale+ MPSoC FSBL)
- Modifying BSP Settings
- Changing Compiler Options of an Application Project
- Working with Libraries
- Creating a Bootable Image and Program the Flash
- Switching Between XSCT and Vitis Integrated Development Environment
- Performing Standalone Application Debug
- Running an Application in Non-Interactive Mode
- Debugging a Program Already Running on the Target
- Using JTAG UART
- Debugging Applications on Zynq UltraScale+ MPSoC
- Editing FSBL/PMUFW Source File
- Editing FSBL/PMUFW Settings

Changing Compiler Options of an Application Project

Below is an example XSCT session that demonstrates creating an empty application for Cortex™-A53 processor, by adding the compiler option `-std=c99`.

```
setws /tmp/wrk/workspace
app create -name test_a53 -hw /tmp/wrk/system.xsa -os standalone -proc
psu_cortexa53_0 -template {Empty Application}
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)

Below 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. `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 a Bootable Image and Program the Flash

Below is an example XSCT session that demonstrates creating two applications (FSBL and Hello World). Further, create a bootable image using the applications along with bitstream and program the image on to the flash.

Note: Assuming the board to be zc702. Hence `-flash_type qspi_single` is used as an option in `program_flash`.

```
setws /tmp/wrk/workspace
app create -name a9_hello -hw /tmp/wrk/system.xsa -os standalone proc
ps7_cortexa9_0 -template {Zynq FSBL}
app create -name a9_fsbl -hw /tmp/wrk/system.xsa -os standalone proc
ps7_cortexa9_0 -template {Hello World}
app build -name a9_hello
app build -name a9_fsbl
exec bootgen -arch zynq -image output.bif -w -o /tmp/wrk/BOOT.bin
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). Users will need to 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 debug info.

Below 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 dhrystone/Debug/dhrystone.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 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 A53 and R5 cores are held in reset. Users must clear resets on each core, before debugging on these cores. '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 A53 #0 is the current target, rst -cores clears resets on all the A53 cores in APU.

Below is an example XSCT session that demonstrates standalone application debug on A53 #0 core on Zynq UltraScale+ MPSoC.

***Note*:** Similar steps can be used for debugging applications on R5 cores and also on A53 cores in 32 bit mode. However, the A53 cores must be put in 32 bit mode, before debugging the applications. This should be done after POR and before the 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 dhrystone/Debug/dhrystone.elf
Downloading Program -- dhrystone/Debug/dhrystone.elf
                section, .text: 0xfffc0000 - 0xfffd52c3
                section, .init: 0xfffd5300 - 0xfffd5333
                section, .fini: 0xfffd5340 - 0xfffd5373
                section, .note.gnu.build-id: 0xfffd5374 - 0xfffd5397
                section, .rodata: 0xfffd5398 - 0xfffd6007
                section, .rodata1: 0xfffd6008 - 0xfffd603f
                section, .data: 0xfffd6040 - 0xfffd71ff
                section, .eh_frame: 0xfffd7200 - 0xfffd7203
                section, .mmu_tbl0: 0xfffd8000 - 0xfffd800f
                section, .mmu_tbl1: 0xfffd9000 - 0xfffdafff
                section, .mmu_tbl2: 0xfffdb000 - 0xfffdefff
                section, .init_array: 0xfffdf000 - 0xfffdf007
                section, .fini_array: 0xfffdf008 - 0xfffdf047
                section, .sdata: 0xfffdf048 - 0xfffdf07f
                section, .bss: 0xfffdf080 - 0xfffe197f
                section, .heap: 0xfffe1980 - 0xfffe397f
                section, .stack: 0xfffe3980 - 0xfffe697f
100%    0MB   0.4MB/s  00:00
Setting PC to Program Start Address 0xfffc0000
Successfully downloaded dhrystone/Debug/dhrystone.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 0xfffc0d5c (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: 000000000000001c    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: 00000000fffc1f4c    sp: 00000000fffe5980    pc: 00000000fffc0d5c
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 0xfffc0d64 (Step)

# View stack trace
xsdb% bt
    0  0xfffc0d64 main()+8: ../src/dhry_1.c, line 79
    1  0xfffc1f4c _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.

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
platform generate
app build -name mb_app
```

Performing Standalone Application Debug

Xilinx® 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.

Below is an example XSCT session that demonstrates standalone application debug on Zynq®-7000 SoC. 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 0xfffffe1c (Suspended)
Info: ARM Cortex-A9 MPCore #1 (target 3) Stopped at 0xfffffe18 (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 dhrystone/Debug/dhrystone.elf
Downloading Program -- dhrystone/Debug/dhrystone.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 dhrystone/Debug/dhrystone.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: 0010fca0       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 and 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`, users can download another elf onto `A9 #1` and debug it, using similar steps. Note that, it's not necessary to re-connect to the hw_server, initialize the PS or configure the FPGA in such cases. Users can just 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. Below is a example XSCT script to generate an SVF file:

```
# 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 0xffff0000 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 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>
```

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 dhrystone/Debug/dhrystone.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 Development Environment

Below is an example XSCT session that demonstrates creating two applications using XSCT and modifying the BSP settings. After the 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

Xilinx® System Debugger Command-line Interface (XSDB) supports virtual UART through JTAG, which is useful when the physical UART doesn't exist or is non-functional. To use JTAG UART, the SW 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.

Below is an example XSCT session that demonstrates how to use a JTAG terminal for STDIO.

```
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
```

Below is an example XSCT session that demonstrates how to use the XSCT console as STDOUT for JTAG UART.

```
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
```

Below is an example XSCT session that demonstrates how to redirect the STDOUT from JTAG UART to a file.

```
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

Below is an example XSCT session that demonstrates creating a default domain and adding XILFFS and XILRSA libraries to the BSP. 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
```

Chapter 24

Packaging Drivers for Vitis

XSCT Interface Examples

Tcl Examples

This chapter demonstrates how to load a `.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]
```

Property	Type	Readonly	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

Property	Type	Readonly	Visible	Value
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]
```

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

Property	Type	Readonly	Visible	Value
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_pl310_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_slcr_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]
```

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
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

Property	Type	Readonly	Visible	Value
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	
VLNV	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 38:

Property	Type	Read-only	Visible	Value
BASE_NAME	string	true	true	C_BASEADDR
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

```
hsi::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

```
hsi::create_sw_design swdesign -proc ps7_cortexa9_0 -os standalone
swdesign
```

To switch to active software design

```
hsi::current_sw_design
swdesign
```

Properties of the current software design

```
common::report_property [hsi::current_sw_design ]
```

Table 39: **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	
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_pl310_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
hsi% get_osstandalone
```

Properties of the OS object

```
common::report_property[hsi::get_os]
```

Table 40: **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_compiler_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 dhrystone 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
```

#Genereate 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

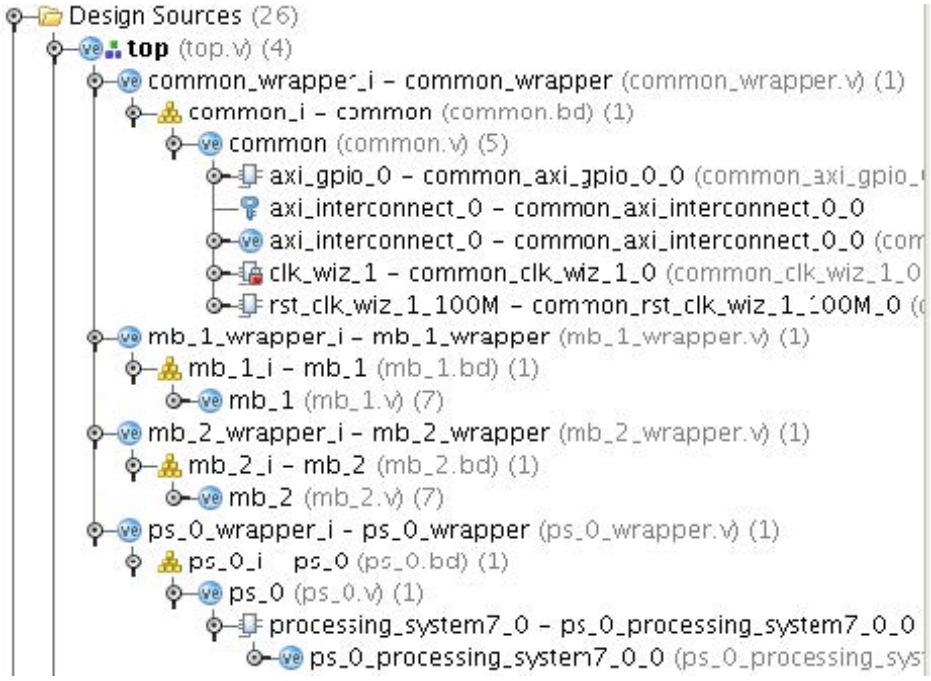
```
hsi::generate_bsp -dir advanced_bsp -compile
```

#Delete the library added to software design

```
hsi::delete_objs $lib
```

Generating and Compiling BSP for a Multi-Block Design

Figure 26: **Example Design with Multiple Block Design Instances in the Active Top Design**



#Open hardware design with multiple block design instances

```
hsi% 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 andtheir names are prefixed with their hierarchy

```
hsi% 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

```
hsi::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
```

Input and Output Files

Input Files

XSA

Xilinx® Support Archive (`.xsa`) is a Xilinx proprietary file format and only Xilinx software tools understand it. Third-party software tools can communicate to the XSCT Tcl interface to extract data from the `.xsa` file.

Note: Xilinx does not recommend manually editing the XSA file or altering its contents.

XSA is a container and contains:

- One or more `.hwh` files
  - Vivado® tool version, part, and board tag information
  - IP - instance, name, VLNV, 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 *OS 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. Refer to Drivers, OS Block, and Libraries for more information.

- **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` or `/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 SDK 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/libsrc` 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/libsrc` 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 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 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 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:

- **DRC Section:** This section contains Tcl routines that validate your OS and library parameters for consistency.
- **Generation Section:** This section 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 dialog box.

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 meaning is self-explanatory: `NAME, DESC, TYPE, DEFAULT, RANGE, 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 GUI 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 GUI 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 GUI 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 GUI 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 SDK 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 GUI 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 SDK GUI does not display this value by default. This is displayed only for the advanced option in the GUI.
  - **ALL_USERS:** The value can be modified by all. The SDK GUI 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 PARAMs, and only PARAMs. 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 dialog box.

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 has been called on all drivers and libraries, and `execs_generate` called after the libraries and drivers have been 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_l.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_l.h.

```
BEGIN INTERFACE stdout
PROPERTY header = xuartlite_l.h;
FUNCTION name = outbyte, value = XUartLite_SendByte;
END INTERFACE
BEGIN INTERFACE stdio
PROPERTY header = xuartlite_l.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 GUI 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_uartlte_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 SDK GUI does not display this value by default. It is displayed only as an advanced option in the GUI.

- **ALL_USERS:** The value can be modified by all. The SDK GUI 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 dialog box.
- For warnings, DRC procedures return a string value that can be printed on the console.
- On success, DRC procedures just 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 27: **Example Design with an IP with custom driver**

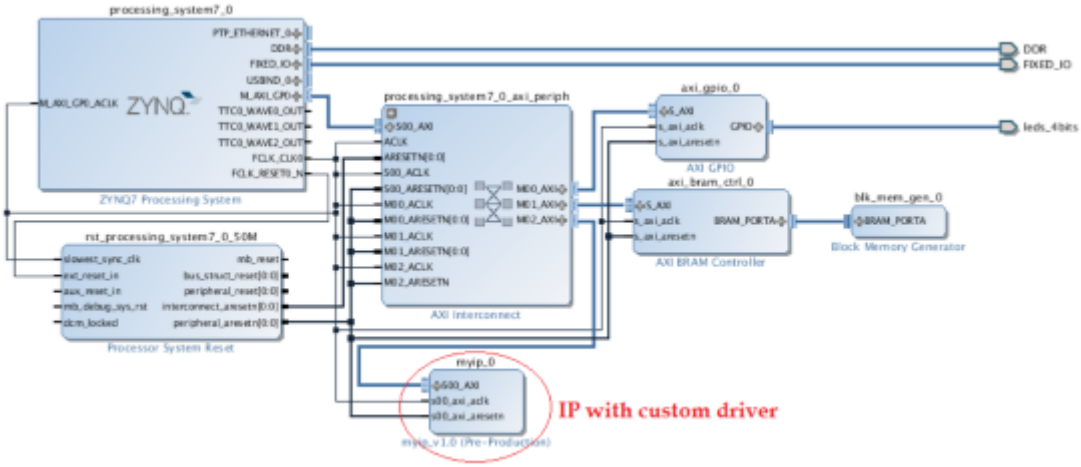


Figure 28: **Custom driver specified in IPXACT specification of an IP**

```
<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. Refer to the Hardware Hand-off section for more information. Custom driver for each IP is packaged in 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*]
```

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.

Tcl Commands Listed Alphabetically

This section contains all hardware and software interface Tcl commands, arranged alphabetically.

common::create_property

Description

Create property for class of objects(s).

Syntax

```
create_property [-description <arg>] [-type <arg>] [-enum_values <args>] [-
default_value <arg>] [-file_types <args>] [-display_text <arg>] [-quiet] [-
verbose] <name> <class>
```

Returns

The property that was created if success, "" if failure.

Usage

Name	Description
[-description]	Description of property
[-type]	Type of property to create; valid values are: string, int, long, double, bool, enum, file Default: string
[-enum_values]	Enumeration values
[-default_value]	Default value of type string
[-file_types]	File type extensions (without the dot)
[-display_text]	Text to display when selecting the file in file browser
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Name of property to create
<class>	Object type to create property for; valid values are: design, net, cell, pin, port, pblock, interface, fileset

Categories

PropertyAndParameter

Description

Creates a new property of the type specified with the user-defined namefor the specified class of objects. The property that is created can be assigned to an object of the specified class with the set_property command, but is not automatically associated with all objects of that class.

The `report_property -all` command will not report the newly created property for an object of the specified class until the property has been assigned to that object

Arguments

`-description` - (optional) Provide a description of the property being created. The description will be returned by the HSM help system when the property is queried.

`-type` - (optional) The type of property to create. Allowed property types include:

- string - Allows the new property to be defined with string values. This is the default value when -type is not specified.
- int - Allows the new property to be defined with short four-byte signed integer values with a range of -2,147,483,648 to 2,147,483,647. If a floating point value is specified for an int property type, the HSM tool will return an error.
- long - Specifies signed 64-bit integers with value range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. If a floating point value is specified for an long property type, the tool will return an error.
- double - Allows the new property value to be defined with a floating point number.
- bool - Allows the new property to be defined as a boolean with a true (1, or yes) or false (0, or no) value.
- enum - An enumerated data type, with the valid enumerated values specified by the -enum_values option.
- string_list - A Tcl list of string values.
- int_list - A Tcl list of integer values.
- double_list - A Tcl list of floating point values.

`-enum_values` <args> - (optional) A list of enumerated values that the property can have. The list must be enclosed in braces, {}, with values separated by spaces. This option can only be used with -type enum.

`-default_value` <args> - (optional) The default value to assign to the property. This option can be used for string, int, bool, and enum type properties.

<name> - (required) The name of the property to be defined. The name is case sensitive.

<class> - (required) The class of object to assign the new property to. All objects of the specified class will be assigned the newly defined property. Valid classes are: design, net, cell, pin, port, Pblock, interface, and fileset.

`-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 property called PURPOSE for cell objects:

```
common::create_property PURPOSE cell
```

Note: Because the type was not specified, the value will default to "strings".

Create a pin property called COUNT which holds an Integer value:

```
common::create_property -type int COUNT pin
```

common::get_msg_config

Description

Returns the current message count, limit, or the message configuration rules previously defined by the `set_msg_config` command.

Syntax

```
get_msg_config [-id <arg>] [-severity <arg>] [-rules] [-limit] [-count] [-
quiet] [-verbose]
```

Returns

Nothing.

Usage

Name	Description
[-id]	The message id to match. Should be used in conjunction with -limit or -count Default: empty
[-severity]	The message severity to match. Should be used in conjunction with -limit or -count Default: empty
[-rules]	Show a table displaying all message control rules for the current project
[-limit]	Show the limit for the number of messages matching either -id or -severity that will be displayed
[-count]	Show the number of messages matching either -id or -severity that have been displayed
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Report

common::get_param

Description

Get a parameter value.

Syntax

```
get_param [-quiet] [-verbose] <name>
```

Returns

Parameter value.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Parameter name

Categories

PropertyAndParameter

Description

Gets the currently defined value for a specified tool parameter. These parameters are user-definable configuration settings that control various behaviors within the tool. Refer to `report_param` for a description of what each parameter configures or controls.

Arguments

`name` - (required) The name of the parameter to get the value of. The list of user-definable parameters can be obtained with `list_param`. This command requires the full name of the desired parameter. It does not perform any pattern matching, and accepts only one parameter at a time.

`-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

Get the current value of the messaging parameter used for enabling the description:

```
common::get_param messaging.enableDescription
```

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::help

Description

Display help for one or more topics.

Syntax

```
help [-category <arg>] [-args] [-syntax] [-long] [-prop <arg>] [-class
<arg>] [-message <arg>] [-quiet] [-verbose] [<pattern_or_object>]
```

Returns

Nothing.

Usage

Name	Description
[`-category`]	Search for topics in the specified category
[`-args`]	Display arguments description
[`-syntax`]	Display syntax description
[`-long`]	Display long help description
[`-prop`]	Display property help for matching property names Default: *
[`-class`]	Display object type help
[`-message`]	Display information about the message with the given message. Every message delivered by the tool has a unique global message ID that consists of an application sub-system code and a message identifier. Example: -message {Common 17-8}.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<pattern_or_object>]	Display help for topics that match the specified pattern Default: *

Categories

Project

common::list_param

Description

Get all parameter names.

Syntax

```
list_param [-quiet] [-verbose]
```

Returns

List.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution

Categories

PropertyAndParameter

Description

Gets a list of user-definable configuration parameters. These parameters configure a variety of settings and behaviors of the tool. For more information on a specific parameter use the `report_param` command, which returns a description of the parameter as well as its current value.

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.

Examples

Get a list of all user-definable parameters:

```
common::list_param
```

common::list_property

Description

List properties of object.

Syntax

```
list_property [-class <arg>] [-regexp] [-quiet] [-verbose] [<object>]
[<pattern>]
```

Returns

List of property names.

Usage

Name	Description
[`-class`]	Object type to query for properties. Ignored if object is specified.
[`-regexp`]	Pattern is treated as a regular expression
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<object>]	Object to query for properties
[<pattern>]	Pattern to match properties against Default: *

Categories

Object, PropertyAndParameter

Description

Gets a list of all properties on a specified object or class.

Note: `report_property` also returns a list of properties on an object or class of objects, but also reports the property type and property value.

Arguments

`-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>.

`-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 CELL object:

```
common::list_property [get_cells microblaze_0]
```

common::list_property_value

Description

List legal property values of object.

Syntax

```
list_property_value [-default] [-class <arg>] [-quiet] [-verbose] <name>
[<object>]
```

Returns

List of property values.

Usage

Name	Description
[`-default`]	Show only the default value.
[`-class`]	Object type to query for legal property values. Ignored if object is specified.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<name>	Name of property whose legal values is to be retrieved
[<object>]	Object to query for legal properties values

Categories

Object, PropertyAndParameter

common::load_features

Description

Load Tcl commands for a specified feature.

Syntax

```
load_features [-quiet] [-verbose] [<features>...]
```

Returns

Nothing.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<features>]	Feature(s) to load, use `list_features` for a list of available features.

Categories

Tools

common::register_proc

Description

Register a Tcl proc with Vivado.

Syntax

```
register_proc [-quiet] [-verbose] <proc> [<tasknm>]
```

Returns

Nothing.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<proc>	Name of proc to register. Proc must be known to Tcl
[<tasknm>]	Name of Tcl task that wraps the proc. Default: Register the proc using the root name proc (no namespaces).

Categories

Tools

common::report_environment

Description

Report system information.

Syntax

```
report_environment [-file <arg>] [-format <arg>] [-append] [-return_string]
[-quiet] [-verbose]
```

Returns

Nothing.

Usage

Name	Description
[-file]	Write system information to specified file.
[-format]	Specifies how to format the report. Default is 'text', another option is 'xml'. Only applies if -file is used. If xml output is used, -append is not allowed. Default: text
[-append]	Append report to existing file
[-return_string]	Return report content as a string value
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Report

common::report_param

Description

Get information about all parameters.

Syntax

```
report_param [-file <arg>] [-append] [-non_default] [-return_string] [-
quiet] [-verbose] [<pattern>]
```

Returns

Param report.

Usage

Name	Description
[-file]	Filename to output results to. (send output to console if -file is not used)
[-append]	Append the results to file; do not overwrite the results file
[-non_default]	Report only params that are set to a non default value
[-return_string]	Return report as string
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<pattern>]	Display params matching pattern Default: *

Categories

PropertyAndParameter, Report

Description

Gets a list of all user-definable parameters, the current value, and a description of what the parameter configures or controls.

Arguments

`-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.

`-return_string` - (optional) Directs the output to a Tcl string rather than to the standard output. The Tcl string can be captured by a variable definition and parsed or otherwise processed.

Note: This argument cannot be used with the `-file` option.

`-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.

<pattern> (optional) Match parameters against the specified pattern. The default pattern is the wildcard `*` which gets all user-definable parameters.

Examples

The following example returns the name, value, and description of all user-definable parameters:

```
common::report_param
```

The following example returns the name, value, and description of user-definable parameters that match the specified search pattern:

```
common::report_param *coll*
```

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
[-all]	Report all properties of object even if not set
[-class]	Object type to query for properties. Not valid with <object>
[-return_string]	Set the result of running report_property in the Tcl interpreter's result variable
[-file]	Filename to output result to. Send output to console if -file is not used
[-append]	Append the results to file; do not overwrite the results file
[-regexp]	Pattern is treated as a regular expression
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<object>]	Object to query for properties
[<pattern>]	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]
```

common::reset_msg_config

Description

Resets or removes a message control rule previously defined by the set_msg_config command.

Syntax

```
reset_msg_config [-string <args>] [-id <arg>] [-severity <arg>] [-limit] [-
suppress] [-count] [-default_severity] [-regexp] [-quiet] [-verbose]
```

Returns

Nothing.

Usage

Name	Description
[`-string`]	A qualifier, apply the selected operation only to messages that contain the given strings Default: empty
[`-id`]	A qualifier, the message id to match. If not specified, all message ids will be matched

Name	Description
[-severity]	A qualifier, apply the selected operation only to messages at the given severity level
[-limit]	Resets the limit values for message controls that match the given qualifiers for the current project
[-suppress]	Stops suppressing messages that match the given qualifiers for the current project
[-count]	Resets the count of messages for all message controls that match the given qualifiers for the current project. This will prevent messages from being suppressed by a -limit control until the message count once again exceeds the specified limit.
[-default_severity]	Resets the message severity of all messages controls for the current project that match the given qualifiers to their default value
[-regexp]	The values used for -string are full regular expressions
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Report

common::reset_msg_count

Description

Reset message count.

Syntax

```
reset_msg_count [-quiet] [-verbose] <id>
```

Returns

New message count.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<id>	Unique message ID to be reset, e.g. "Common 17-99". "reset_msg_count -id *" reset all counters

Categories

Report

common::reset_param

Description

Reset a parameter.

Syntax

```
reset_param [-quiet] [-verbose] <name>
```

Returns

Original value.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Parameter name

Categories

PropertyAndParameter

Description

Restores a user-definable configuration parameter that has been changed with the `set_param` command to its default value.

You can use the `report_param` command to see which parameters are currently defined.

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> - (required) The name of a parameter to reset. You can only reset one parameter at a time.

Examples

The following example restores the tcl.statsThreshold parameter to its default value:

```
common::reset_param tcl.statsThreshold
```

common::reset_property

Description

Reset property on object(s).

Syntax

```
reset_property [-quiet] [-verbose] <property_name> <objects>...
```

Returns

The value that was set if success, "" if failure.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<property_name>	Name of property to reset
<objects>	Objects to set properties

Categories

Object, PropertyAndParameter

Description

Restores the specified property to its default value on the specified object or objects. If no default is defined for the property, the property is unassigned on the specified object.

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.

<property_name> - (required) The name of the property to be reset.

<objects> - (required) One or more objects on which the property will be restored to its default value.

Examples

The following example sets the archiver property on the specified processor, and then resets the property:

```
common::set_property CONFIG.archiver armar [get_sw_processor]
```

```
common::reset_property CONFIG.archiver armar [get_sw_processor]
```

common::set_msg_config

Description

Configure how the Vivado tool will display and manage specific messages, based on message ID, string, or severity.

Syntax

```
set_msg_config [-id <arg>] [-string <args>] [-severity <arg>] [-limit
<arg>] [-new_severity <arg>] [-suppress] [-regexp] [-quiet] [-verbose]
```

Returns

Nothing.

Usage

Name	Description
[`-id`]	A qualifier, apply the selected operation only to messages that match given message id. Example: '-id {Common 17-35}'. Default: match any id
[`-string`]	A qualifier, apply the selected operation only to messages that contain the given list of strings. Default: none
[`-severity`]	A qualifier, apply the selected operation only to messages at the given severity level. Example: '-severity INFO' Default: match any severity

Name	Description
[-limit]	for the messages that match the qualifiers, limit the number of messages displayed to the given integer value. Can only be used in conjunction with one of -id or -severity.
[-new_severity]	for the messages that match the qualifiers, change the severity to the given value for the current project
[-suppress]	for the messages that match the qualifiers, suppress (do not display) any messages for the current project
[-regexp]	The values used for -string are full regular expressions
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Report

common::set_param

Description

Set a parameter value.

Syntax

```
set_param [-quiet] [-verbose] <name> <value>
```

Returns

Newly set parameter value.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Parameter name
<value>	Parameter value

Categories

PropertyAndParameter

Description

Sets the value of a user-definable configuration parameter. These parameters configure and control various behaviors of the tool. Refer to `report_param` for a description of currently defined parameters.

You can use the `reset_param` command to restore any parameter that has been modified back to its default setting.

Note: Setting a specified parameter value to -1 will disable the feature.

Arguments

`<name>` - (required) The name of the parameter to set the value of. You can only set the value of one parameter at a time.

`<value>` - (required) The value to set the specified parameter to.

`-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

```
common::set_param messaging.defaultLimit 1000
```

common::set_property

Description

Set property on object(s).

Syntax

```
set_property [-dict <args>] [-quiet] [-verbose] <name> <value> <objects>...
```

Returns

Nothing.

Usage

Name	Description
[-dict]	list of name/value pairs of properties to set
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Name of property to set. Not valid with -dict option
<value>	Value of property to set. Not valid with -dict option
<objects>	Objects to set properties on

Categories

Object, PropertyAndParameter

Description

Assigns the defined property <name> and <value> to the specified <objects>.

This command can be used to define any property on an object in the design. Each object has a set of predefined properties that have expected values, or a range of values. The `set_property` command can be used to define the values for these properties. To determine the defined set of properties on an object, use `report_property`, `list_property`, or `list_property_values`.

You can also define custom properties for an object, by specifying a unique <name> and <value> pair for the object. If an object has custom properties, these will also be reported by the `report_property` and `list_property` commands.

This command returns nothing if successful, and an error if it fails.

Note: You can use the `get_property` command to validate any properties that have been set on an object.

Arguments

`-dict` - (optional) Use this option to specify multiple properties (<name> <value> pairs) on an object with a single `set_property` command. Multiple <name> <value> pairs must be enclosed in braces, {}, or quotes, "".

```
-dict "name1 value1 name2 value2 ... nameN valueN"
```

`-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) Specifies the name of the property to be assigned to the object or objects. The <name> argument is case sensitive and should be specified appropriately.

`<value>` - (required) Specifies the value to assign to the <name> on the specified object or objects. The value is checked against the property type to ensure that the value is valid. If the value is not appropriate for the property an error will be returned.

Note: In some cases the value of a property may include special characters, such as the dash character (`-`), which can cause the tool to interpret the value as a new argument to the command. In this case, you must use the explicit arguments (-name, -value, -objects) instead of the implied positional arguments (name, value, objects) as described here. This is shown in the Examples section below.

`<objects>` - (required) One or more objects to assign the property to.

Examples

Create a user-defined boolean property, TRUTH, for cell objects, and set the property on a cell:

```
common::create_property -type bool truth cell
common::set_property truth false [lindex [get_cells] 1]
```

The following example sets the compiler and archiver property value for the specified software processor:

```
common::set_property CONFIG.archiver armar [get_sw_processor]
common::set_property CONFIG.compiler armcc [get_sw_processor]
```

common::unregister_proc

Description

Unregister a previously registered Tcl proc.

Syntax

```
unregister_proc [-quiet] [-verbose] <tasknm>
```

Returns

Nothing.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<tasknm>	Name of Tcl task to unregister. The task must be wrapping a proc.

Categories

Tools

common::version

Description

Returns the build for hsi and the build date.

Syntax

```
version [-short] [-quiet] [-verbose]
```

Returns

Hsi version.

Usage

Name	Description
[-short]	Return only the numeric version number
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Report

hsi::add_library

Description

Add software library to software design.

Syntax

```
add_library [-sw <arg>] [-quiet] [-verbose] <name> [<version>]
```

Returns

The Software Library object. Returns nothing if the command fails.

Usage

Name	Description
[`-sw`]	Software design name
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<name>	Software library name
[<version>]	Version of software Library

Categories

Software

Description

Adds library to the active software design. The software design must previously have been created using the `create_sw_design` command. This command returns a message with the name of the library, or returns an error if the command fails.

Arguments

<-sw> - (optional) Name of the software design to which library to the added.

`-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> - Name of the library.

<version> - (optional) Version of the library name. Version less library will be picked if version is not specified.

Examples

The following adds the specified library to the current software design:

```
hsi::add_library xilffs
```

The following adds version of the library:

```
hsi::add_library xilrsa 1.0
```

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
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<name>	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::close_sw_design

Description

Close a software design.

Syntax

```
close_sw_design [-quiet] [-verbose] <name>...
```

Returns

Returns nothing, error message if failed.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<name>	Name of design to close

Categories

Software

Description

Closes the specified software design in the current Hardware Software Interface session.

IMPORTANT! *If the design has been modified, you will not be prompted to save the design prior to closing. In the current release the persistence option is not available.*

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 software design object to close.

Examples

Close the current software design in the current session:

```
hsi::close_sw_design [current_sw_design]
```

```
hsi::close_sw_design
```

Close the specified software design in the current session:

```
hsi::close_sw_design [current_sw_design]
```

hsi::create_comp_param

Description

Add parameter.

Syntax

```
create_comp_param [-quiet] [-verbose] <name> <value> <objects>
```

Returns

Parameter object. Returns nothing if the command fails.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Parameter name
<value>	Parameter value
<objects>	List of nodes

Categories

Software

Description

Create a new param to list of nodes (driver/os/proc/node).

If successful, this command returns the name of the param created. 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.

<name> - The name of the param to create.

<value> - The value of the param to create. Default type of param is string.

<objects> - List of nodes to which new param is created.

Examples

The following example creates a new param called p1 to specified driver:

```
hsi::create_comp_param p1 [get_drivers ps7_uart_1]
```

The following example creates a new param called p2 to all drivers.

```
hsi::create_comp_param p2 [get_drivers]
```

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
`-name`	Child DT node name
[`-unit_addr`]	Unit address of node
[`-label`]	Label of node
[`-objects`]	List of nodes
[`-quiet`]	Ignore command errors
[`-verbose`]	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:

```
hsi::create_dt_tree -dts_file pl.dtsi -dts_version /dts-v1/
```

```
hsi::create_dt_tree -dts_file system.dts -dts_version /dts-v3/ -
header "include pl.dtsi, include ps.dtsi"
```

```
hsi::create_dt_tree -dts_file ps.dtsi -dts_version /dts-v3/ -
header "PS system info"
```

hsi::create_node

Description

Add node.

Syntax

```
create_node [-quiet] [-verbose] <name> <objects>
```

Returns

Node object. Returns nothing if the command fails.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Child node name
<objects>	List of nodes

Categories

Software

Description

Create a new node to list of existing nodes (driver/os/prco/node).

If successful, this command returns the name of the node created. 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.

<name> - The name of the node to create.

<objects> - List of nodes to which new node is created.

Examples

The following example creates a new node called n1 to the specified driver:

```
hsi::create_node n1 [get_drivers ps7_uart_1]
```

The following example creates a new node called n2 to all drivers:

```
hsi::create_node n2 [get_drivers]
```

hsi::create_sw_design

Description

Create a software design.

Syntax

```
create_sw_design -proc <arg> [-app <arg>] [-os <arg>] [-os_ver <arg>] [-
quiet] [-verbose] <name>...
```

Returns

Software design object. Returns nothing if the command fails.

Usage

Name	Description
`-proc`	Processor name
[`-app`]	Application name
[`-os`]	OS name Default: standalone
[`-os_ver`]	OS version
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<name>	Software design name

Categories

Software

Description

Create a new software design module to add to the current session. If successful, this command returns the name of the software design created. Otherwise it returns an error.

Arguments

`-proc` - The processor instance name targeted for the software design.

`-app` - The template application name.

`-os` - (optional) The OS name targeted for the software design. Default value is standalone.

`-os_ver` - (optional) The OS version targeted for the software design. Default value is the latest OS version.

`-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 software design module to create.

Examples

Create a new software design module called sw_design_1 and add the module to the current session:

```
hsi::create_sw_design sw_design_1 -proc microblaze_0 -os xilkernel
```

```
hsi::create_sw_design sw_design_1 -proc microblaze_0 -os xilkernel -
os_ver 6.0
```

```
hsi::create_sw_design sw_design_1 -proc ps7_cortexa9_0
```

hsi::current_dt_tree

Description

Set or get current tree.

Syntax

```
current_dt_tree [-quiet] [-verbose] [<tree>]
```

Returns

Tree object, "" if failed.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<tree>]	Tree to be set

Categories

DeviceTree

Description

Defines the current DT tree for use with the Hardware Software Interface, or returns the name of the current DT tree in the active session. The current DT tree is the target of the Hardware Software Interface - DeviceTree Tcl commands. You can use the `get_dt_trees` command to get a list of created DT trees in the active session.

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.

<dt_tree> - (optional) The name of a DT tree to set as the current DT tree in the session. If a <dt_tree> is not specified, the command returns the current DT tree of the active session.

Examples

Get the current DT tree object:

```
hsi::current_dt_tree
```

OR

Set the specified dt_tree as the current session:

```
hsi::current_dt_tree pl.dtsi
```

hsi::current_hw_design

Description

Set or get current hardware design.

Syntax

```
current_hw_design [-quiet] [-verbose] [<design>]
```

Returns

Current hardware design object, "" if failed.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<design>]	design to be set

Categories

Hardware

Description

Defines the current hardware design for use with the Hardware Software Interface, or returns the name of the current design in the active project. The current hardware design is the target of the Hardware Software Interface hardware Tcl commands. You can use the get_hw_designs command to get a list of open hardware designs in the active project.

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.

<design> - (optional) The name of a hardware design to set as the current design in the session. If a design is not specified, the command returns the current hardware design of the active session.

Examples

Get the current hardware design object:

```
hsi::current_hw_design
```

OR

Set the specified hardware design object as the current design:

```
hsi::current_hw_design hw_design_1
```

hsi::current_hw_instance

Description

Set or reset current hardware instance.

Syntax

```
current_hw_instance [-quiet] [-verbose] [<instance>]
```

Returns

Current cell instance object, "" if failed.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<instance>]	hardware cell instance to be set

Categories

Hardware

Description

Sets or resets the current hardware instance to the given hierarchial instance, for scoping the hardware data access commands in a hierarchial design. Default is top design.

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.

<instance> - (optional) Hardware cell instance (hierarchial instance) to be set.

Examples

The following example sets the current hardware instance to axi_ethernet_0:

```
hsi::current_hw_instance [get_cells axi_ethernet_0]
```

```
hsi::get_cells
```

Returns the hardware cell instances inside current hierarchial instance `axi_ethernet_0`.

hsi::current_sw_design

Description

Set or get current software design.

Syntax

```
current_sw_design [-quiet] [-verbose] [<design>]
```

Returns

Software design object, "" if failed.

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<design>]	design to be set

Categories

Software

Description

Sets the current software design for use with the Hardware Software Interface, or returns the name of the current design in the active project.

The current software design is the target of the Hardware Software Interface software Tcl commands.

You can use the `get_sw_designs` command to get a list of open software designs in the active project.

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.

<design> - (optional) The name of a software design to set as the current design in the Hardware Software Interface. If a design is not specified, the command returns the current software design of the active project.

Examples

Get the current software design object:

```
hsi::current_sw_design
```

OR

Set the specified software design object as the current design:

```
hsi::current_sw_design sw_design_1
```

hsi::delete_objs

Description

Delete specified objects.

Syntax

```
delete_objs [-quiet] [-verbose] <objects>...
```

Returns

Pass if successful in deleting objects.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<objects>	The objects to be deleted

Categories

Software

Description

Delete specified objects from the current software design.

Objects must be passed directly to the delete_objs command, and not simply referenced by the object name.

This command returns nothing if it is successful, and returns an error if it fails.

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.

<objects> - A list of objects to delete from the current software design.

Examples

The following example deletes the specified objects from the current software design:

```
hsi::delete_objs [get_libs xilffs] [get_drivers gpio]
```

hsi::generate_app

Description

Generates template application.

Syntax

```
generate_app [-dir <arg>] [-hw <arg>] [-hw_xml <arg>] [-sw <arg>] [-proc
<arg>] [-os <arg>] [-os_ver <arg>] [-app <arg>] [-lapp] [-sapp] [-compile]
[-quiet] [-verbose]
```

Returns

Returns nothing.

Usage

Name	Description
[-dir]	Output directory where App needs to be generated
[-hw]	Hardware Design Name
[-hw_xml]	Hardware Design XML File Path
[-sw]	Software Design Name
[-proc]	Instance Name of the Processor to which App needs to be generated
[-os]	Name of Operating System for App
[-os_ver]	Version of Operating System for App
[-app]	Name of the Application
[-lapp]	List all the Applications in Repositories
[-sapp]	List all the Supported Applications for Hardware and Software Designs
[-compile]	Compile the generated source files
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Software

Description

Generates a template application for a processor on an operating system.

If successful, this command returns nothing. Otherwise it returns an error.

Arguments

`-dir` - (optional) The output where the application needs to be generated.

`-hw` - (optional) Hardware design name.

`-hw_xml` - (optional) XML file path of the hardware design.

`-sw` - (optional) Software design name.

`-proc` - Instance Name of the Processor to which App needs to be generated.

`-os` - (optional) Name of Operating System for App.

`-os_ver` - (optional) Version of Operating System for App.

`-app` - Name of the application.

`-lapp` - (optional) List all the Applications in Repositories.

`-sapp` - (optional) List all the Supported Applications for Hardware and Software Designs.

`-compile` - (optional) Compile the generated source files.

`-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

List all applications in the repositories:

```
hsi::generate_app -lapp
```

The following example gets a list of all supported applications for processor ps7_cortexa9_0 and standalone operating system:

```
hsi::generate_app -sapp -proc ps7_cortexa9_0
```

The following example generates a hello_world application for processor microblaze_0 for xilkernel OS:

```
hsi::generate_app -app hello_world -proc microblaze_0 -os xilkernal
```

hsi::generate_bsp

Description

Generates Board Support Package.

Syntax

```
generate_bsp [-dir <arg>] [-hw <arg>] [-hw_xml <arg>] [-sw <arg>] [-sw_mss
<arg>] [-proc <arg>] [-os <arg>] [-os_ver <arg>] [-compile] [-quiet] [-
verbose]
```

Returns

Returns nothing.

Usage

Name	Description
[-dir]	Output directory where BSP needs to be generated
[-hw]	Hardware Design Name
[-hw_xml]	Hardware Design XML File Path
[-sw]	Software Design Name
[-sw_mss]	Software Design MSS File Path
[-proc]	Instance Name of the Processor to which BSP needs to be generated
[-os]	Name of Operating System for BSP
[-os_ver]	Version of Operating System for BSP
[-compile]	Compile the generated source files
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

Software

Description

Generates a template application for a processor on an operating system

If successful, this command returns nothing. Otherwise it returns an error.

Arguments

`-dir` - (optional) Output directory where BSP needs to be generated.

`-hw` - (optional) Hardware design name .

`-hw_xml` - (optional) XML file path of the hw design .

`-sw` - (optional) Software design name .

`-sw_mss` - (optional) Software Design MSS File Path .

`-proc` - Instance Name of the Processor to which BSP needs to be generated .

`-os` - (optional) Name of Operating System for BSP .

`-os_ver` - (optional) Version of Operating System for BSP .

`-compile` - (optional) Compile the generated source files .

`-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

Generates a BSP for the processor ps7_cortexa9_0:

```
hsi::generate_bsp -proc ps7_cortexa9_0
```

The following example generates BSP for the processor ps7_cortexa9_0 for a MSS file `sw_app.mss`:

```
hsi::generate_bsp -sw_mss sw_app.mss
```

hsi::generate_target

Description

Generates target.

Syntax

```
generate_target [-dir <arg>] [-compile] [-quiet] [-verbose] [<name>]
[<objects>]
```

Returns

Returns nothing.

Usage

Name	Description
[-dir]	Output directory where target needs to be generated
[-compile]	Compile the generated source files
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<name>]	List of targets to be generated, or 'all' to generate all supported targets
[<objects>]	The objects for which target needs to be generate

Categories

Software

hsi::get_arrays

Description

Get a list of software arrays.

Syntax

```
get_arrays [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Array objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'array' objects of these types: 'sw_proc os driver lib sw_core'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Software

Description

Arrays are defined in MDD/MLDs. It contains any number of PARAMs and PROPERTYs which describes size, description of array and default values of elements in array.

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 array objects, "NAME", and "other config parameters" are some of the properties that can be used to filter results.

`-of_objects` <arg> - (optional) Get the arrays that are available in OS, Drivers, Libraries, Processor, Core, as returned by the get_os, get_drivers, get_libs, get_sw_processor, get_sw_cores commands.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_os` or `get_libs`, 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 arrays against the specified patterns. The default pattern is the wildcard `*` which gets a list of all arrays. More than one pattern can be specified to find multiple arrays 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 arrays present in all software cores(drivers/libs/os):

```
hsi::get_arrays
```

The following example gets a list of all arrays matching the name "mem_table":

```
hsi::get_arrays mem_table
```

The following example gets a list of arrays present in OS of current software design:

```
hsi::get_arrays -of_objects [get_os]
```

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

Name	Description
[`-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_comp_params

Description

Get a list of parmas.

Syntax

```
get_comp_params [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Property objects. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'param' objects of these types: 'driver sw_proc os node'.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<patterns>]	Match names against patterns Default: *

Categories

Software

Description

Get a list of params in drivers/os/nodes in the current software 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 `||`).

The following gets params that match NAME and VALUE within their name:

```
get_comp_params -filter {NAME == clock-names && VALUE == "ref_clk aper_clk"}
```

`-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 sotfware 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 params that include the specified driver in the software design:

```
hsi::get_comp_params -of_objects [get_drivers ps7_uart_0]
```

The following example gets a list of all params of OS:

```
hsi::get_comp_params -of_objects [get_os]
```

hsi::get_drivers

Description

Get a list of software driver instances.

Syntax

```
get_drivers [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Driver instance objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'driver' objects of these types: 'sw_design'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Software

Description

Get a list of driver instances in the current software design. Each instance is mapping to the IP instances in the loaded hardware design. Generic driver is assigned for the IPs which does not have drivers.

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 drivers, NAME and HW_INSTANCE are some of the properties you can use to filter results. The following gets drivers named gpio with a HW_INSTANCE of axi_gpio_0:

```
get_drivers * -filter {NAME==gpio && HW_INSTANCE == axi_gpio_0}
```

`-of_objects` <arg> - (optional) Get 'driver' objects of these types: 'sw_design'.

`-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 sotfware 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 drivers that include the specified Software design:

```
hsi::get_drivers
```

The following example gets a list of all driver instances of gpio driver:

```
hsi::get_drivers * -filter {NAME==gpio}
```

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
[-hier]	List of nodes in the current tree.
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get '' objects of these types: 'dtsNode dtsTree'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<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_fields

Description

Get a list of fields of a register.

Syntax

```
get_fields [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Register objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'field' objects of these types: 'register'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Software

Description

Get a list of fields of a register. By default, the command returns the fields of all registers, of all the cells.

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 array objects, "NAME", and "other config parameters" are some of the properties that can be used to filter results.

`-of_objects` <arg> - (optional) Get 'field' objects of these types: 'register'.

***Note*:** The -of_objects option requires objects to be specified using the get_registers command, 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 field names against the specified patterns. The default pattern is the wildcard `*` which gets a list of all fields. More than one pattern can be specified to find multiple fields 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 fields present in all registers of all the software driver cores:

```
hsi::get_fields
```

hsi::get_hw_designs

Description

Get a list of hardware designs opened.

Syntax

```
get_hw_designs [-regexp] [-filter <arg>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Hardware design 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 design names against patterns Default: *

Categories

Hardware

Description

Gets a list of hardware designs open in the current HSM session that match a specified search pattern. The default command gets a list of all open hardware designs in the 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 "Hardware design" object you can use the"NAME" property to filter results. The following gets hardware designs that do NOT contain the "design" substring within their name:

```
get_hw_designs * -filter {NAME !~ "*design*"}
```

`-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 designs against the specified patterns. The default pattern is the wildcard `*` which gets all hardware designs. More than one pattern can be specified to find multiple designs based on different search criteria.

Examples

Get all open hardware designs in the current session:

```
hsi::get_hw_designs
```

hsi::get_hw_files

Description

Get a list of hardware design supporting files.

Syntax

```
get_hw_files [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Hardware design supporting file objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'hw_file' objects of these types: 'hw_design'.
[-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 hardware handoff files in the current hardware 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 "file" objects you can use the "TYPE" property to filter results.

`-of_objects` <args> - (optional) Get the files that are associated with the specified fileset objects. The default is to search all filesets. When `-compile_order` and `-used_in` are specified, the `-of_objects` switch will only accept a single fileset, or a single sub-design such as an IP core, Block Design, or DSP design. A sub-design is also known as a composite file.

***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 files against the specified patterns. The default pattern is the wildcard `*` which gets all files in the project or of_objects. More than one pattern can be specified to find multiple files based on different search criteria.

Examples

The following example returns the bit files in the design that are used for programming FPGA:

```
hsi::get_hw_files -filter {TYPE == bit}
```

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
[-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 interface nets from all levels of hierarchical cells
[-of_objects]	Get 'intf_net' objects of these types: 'hw_design cell bus_intf'.
[-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 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
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression

Name	Description
[-hierarchical]	Get interface pins from all levels of hierarchical cells
[-of_objects]	Get 'bus_intf' objects of these types: 'hw_design cell 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 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 ilter 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_libs

Description

Get a list of software libraries.

Syntax

```
get_libs [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose]
[<patterns>...]
```

Returns

Library objects. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'lib' objects of these types: 'sw_design'.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[`<patterns>`]	Match cell names against patterns Default: *

Categories

Software

Description

Get a list of libraries in the current software 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 libraries, "NAME", "VERSION" and "other config parameters" are some of the properties you can useto filter results. The following gets software libraries that which are named xilrsa and are version 1.0:

```
get_libs * -filter {NAME == xilrsa && VERSION == "1.0"}
```

`-of_objects <arg>` - (optional) Get the subsystem cells that are connected to the specified pin or net objects, as returned by the `get_nets` and `get_pins`, or `get_intf_pins` commands.

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 subsystem 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 libraries:

```
hsi::get_libs
```

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
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-hierarchical`]	Get memory ranges from all levels of hierarchical cells
[`-of_objects`]	Get 'mem_range' objects of these types: 'cell'.
[`-quiet`]	Ignore command errors

Name	Description
[-verbose]	Suspend message limits during command execution
[<patterns>]	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

Name	Description
[-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
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'node' objects of these types: 'driver sw_proc os node'.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[`<patterns>`]	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_slcr_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 sotfware 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_os

Description

Get OS in the software design.

Syntax

```
get_os [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose]
[<patterns>...]
```

Returns

OS object. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'os' objects of these types: 'sw_design'.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[`<patterns>`]	Match design names against patterns Default: *

Categories

Software

Description

Returns OS object on success and nothing if the command fails.

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.

Examples

The following example returns OS object of the current software design:

```
hsi::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 "".

id="p_czj_cvr_4r">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::get_sw_cores

Description

Get a list of software cores like driver, library and operating system.

Syntax

```
get_sw_cores [-regexp] [-filter <arg>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Software core objects. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions

Name	Description
[-filter]	Filter list with expression
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match swcore name and versions against patterns Default: *

Categories

Software

Description

Get a list of software cores defined in the software repository of the current session, based on the specified search pattern. The default is to return all software cores defined in the repository.

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 sw_cores, "NAME", "CORE_STATE" and "TYPE" are some of the properties you can use to filter results.

`-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 IP core definitions in the IP catalog against the specified search patterns. The default pattern is the wildcard `*` which gets a list of all IP cores in the catalog. More than one pattern can be specified to find multiple core definitions 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 a list of all software cores with TYPE property matching the specified pattern:

```
get_sw_cores -filter {TYPE == "driver"}
```

The following example returns a list of all software cores with REPOSITORY property matching the specified pattern:

```
get_sw_cores -filter {REPOSITORY=="C:/user/repo"}
```

hsi::get_sw_designs

Description

Get a list of software designs opened.

Syntax

```
get_sw_designs [-regexp] [-filter <arg>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Software design 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 design names against patterns Default: *

Categories

Software

Description

Gets a list of software designs open in the current HSM session that match a specified search pattern. The default command gets a list of all open software designs in the active 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 software designs you can use "NAME" to filter results.

`-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 designs against the specified patterns. The default pattern is the wildcard `*` which gets all software designs. More than one pattern can be specified to find multiple designs based on different search criteria.

Examples

The following example gets all open Software designs in the current session:

```
get_sw_designs
```

hsi::get_sw_interfaces

Description

Get a list of software interfaces.

Syntax

```
get_sw_interfaces [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Software interface objects. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'interface' objects of these types: 'sw_proc os driver lib sw_core'.
[`-quiet`]	Ignore command errors

Name	Description
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Software

Description

Specifies the interfaces implemented by the library or driver and describes the interface functions and header files used by the library or driver.

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 `||`).

sw_interface, "NAME", and "other config parameters" are some of the properties you can use to filter results.

`-of_objects` <arg> - (optional) Get the software interfaces that are available in OS, Drivers, Libraries, Processor, Core, as returned by the `get_os`, `get_drivers`, `get_libs`, `get_sw_processor`, `get_sw_cores` commands.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_os` or `get_libs`, 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 interfaces against the specified patterns. The default pattern is the wildcard `*` which gets a list of all software interfaces. More than one pattern can be specified to find multiple software interfaces 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 interfaces present in all software cores (drivers/libs/os)

```
get_sw_interfaces
```

The following example gets a list of all software interfaces matching the name "stdout"

```
get_sw_interfaces stdout
```

The following example gets a list of software interfaces present in OS of current software design.

```
get_sw_interfaces -of_objects [get_os]
```

hsi::get_sw_processor

Description

Get processor of the software design.

Syntax

```
get_sw_processor [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-
verbose] [<patterns>...]
```

Returns

Processor object. Returns nothing if the command fails.

Usage

Name	Description
[`-regexp`]	Patterns are full regular expressions
[`-filter`]	Filter list with expression
[`-of_objects`]	Get 'sw_proc' objects of these types: 'sw_design'.
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
[<patterns>]	Match design names against patterns Default: *

Categories

Software

Description

Returns the processor object of the active software design or nothing if the command fails.

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.

Examples

The following example returns the software processor of the current software design:

```
get_sw_processor
```

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
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<file>]	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
```

hsi::open_sw_design

Description

Open a software design from disk file.

Syntax

```
open_sw_design [-quiet] [-verbose] [<file>]
```

Returns

Software design object. Returns nothing if the command fails.

Usage

Name	Description
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<file>]	Software design file to open

Categories

Software

Description

Open a software design in the Hardware Software Interface.

If successful, this command returns software design object representing the opened Software 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.

name - The name of the software design to open.

Examples

Open the specified software design in the current session:

```
open_sw_design sw_design_1
```

hsi::set_repo_path

Description

Set a list of software repository paths.

Syntax

```
set_repo_path [-quiet] [-verbose] <paths>...
```

Returns

Returns nothing

Usage

Name	Description
[`-quiet`]	Ignore command errors
[`-verbose`]	Suspend message limits during command execution
<paths>	List of software repository paths separated by spaces

Categories

Software

Description

Loads the software cores available in the repository path. Users can specify multiple repository paths.

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.

<paths> List of software repository paths separated by spaces.

Examples

The following example loads the user software cores in the current session:

```
set_repo_path C:/users/user_driver_repo
```

Section V

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)

Section VI

Embedded Drivers and Libraries

The embedded drivers and libraries are hosted on the Xilinx® wiki. You can access them with the following links:

- Embedded Driver Documentation
- Embedded Library Documentation

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

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

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

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

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

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

Please Read: Important Legal Notices

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

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright

© Copyright 2019-2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.