Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/01/2019 Version 2019.2</td>
<td></td>
</tr>
<tr>
<td>Initial Xilinx release for the Xilinx Developer Forum.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
# Table of Contents

**Revision History** .......................................................................................................................... 2

**Chapter 1: Getting Started** ........................................................................................................... 5

- Overview ........................................................................................................................................ 5
- Migrating to the Vitis Software Platform from Xilinx SDK .......................................................... 6
- Create a Platform .......................................................................................................................... 7
- Create a Sample Application ........................................................................................................ 9
- Vitis IDE Extensions ..................................................................................................................... 12

**Chapter 2: Develop** ...................................................................................................................... 14

- Platform ........................................................................................................................................ 14
- Applications ................................................................................................................................... 21
- Using Custom Libraries in Application Projects ........................................................................... 39

**Chapter 3: Run, Debug, and Optimize** .......................................................................................... 41

- Run Application Project ............................................................................................................... 41
- Debug Application Project .......................................................................................................... 51
- Cross-Triggering ............................................................................................................................ 72
- Profile/Analyze ............................................................................................................................... 79
- Optimize ........................................................................................................................................ 84
- Packaging the System/Utilities ...................................................................................................... 110
- Other Xilinx Utilities .................................................................................................................... 113

**Chapter 4: Embedded Software Development Flow in Vitis** ........................................................ 118

- Overview ....................................................................................................................................... 118
- Creating a Platform Project .......................................................................................................... 120
- Customizing a Pre-Built Platform ............................................................................................... 123
- Adding Domains to a Platform Project ......................................................................................... 124
- Creating Applications from Domains in a Platform .................................................................... 125
- Managing Multiple Applications in a System Project ................................................................. 128
- Creating and Building Applications for XSA Exported from the Vivado Design Suite ......... 130
- Creating Multiple Domains for a Single Hardware ...................................................................... 134
- Changing a Referenced Domain .................................................................................................. 136
Changing and Updating the Hardware Specification ................................................................. 137
Debugging the Application on Hardware ............................................................................ 138
Running and Debugging Applications under a System Project Together ...................... 139
Creating a Bootable Image ................................................................................................... 139
Flash Programming ............................................................................................................... 141
Generating Device Tree ........................................................................................................ 142
Debugging an Application using the User-modified/Custom FSBL .................................. 142
Modifying the Domain Sources (Driver and Library Code)........................................... 147

Chapter 5: Bootgen Tool .................................................................................................. 151
Introduction ......................................................................................................................... 151
Boot Image Layout .............................................................................................................. 153
Creating Boot Images ......................................................................................................... 175
Using Bootgen Interfaces .................................................................................................... 185
Boot Time Security ............................................................................................................. 189
FPGA Support .................................................................................................................... 224
Use Cases and Examples .................................................................................................... 227
BIF Attribute Reference ....................................................................................................... 237
Command Reference .......................................................................................................... 267
Bootgen Utility ................................................................................................................... 280

Chapter 6: Xilinx Software Command-Line Tool ......................................................... 282
Introduction ......................................................................................................................... 282
Installing and Launching XSCT .......................................................................................... 284
XSCT Commands ................................................................................................................ 287
XSCT Use Cases .................................................................................................................. 390

Chapter 7: Embedded Drivers and Libraries .................................................................. 405

Appendix A: Additional Resources and Legal Notices .................................................. 406
Xilinx Resources .................................................................................................................. 406
Documentation Navigator and Design Hubs .................................................................... 406
Please Read: Important Legal Notices ............................................................................ 407
Chapter 1

Getting Started

Overview

The Vitis™ unified software platform is an integrated development environment (IDE) for the development of embedded software applications targeted towards Xilinx® embedded processors. The Vitis software platform works with hardware designs created with Vivado® Design Suite. The Vitis software platform is based on the Eclipse open source standard and 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.

- **System Project**: A system project groups together applications that run simultaneously on the device. Two applications for the same processor cannot sit together in a system project.

- **Platform**: The target platform or platform is a combination of hardware components (XSA) and software components (domains/BSPs, boot components such as U-Boot, and so on). Using this platform, you can create an application without creating the domain/BSP separately.
• **Platform Project**: A platform project groups hardware and domains/BSPs together. Boot components like FSBL and PMUFW are automatically generated in platform projects.

• **Domain (BSP)**: A domain or board support package (BSP) is a collection of software drivers and, optionally, the operating system 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 in the platform.

• **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. You must provide an initial workspace location when the Vitis software platform is launched. You can create multiple workspaces to more easily manage multiple software versions.

• **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 workspace can contain multiple software projects. Each software project must have a corresponding board support package.

### Document Scope and Audience

The purpose of this guide 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 guide 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

### 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 [Chapter 4: Embedded Software Development Flow in Vitis](#) 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.
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 re-created within 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 Unified Software Platform

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

1. Launch the Vitis™ software platform.
3. In the Project name field, enter a name for your platform project and click Next.
4. Select Create from hardware specification (XSA/DSA) and click Next.
5. In the XSA/DSA file field, browse and select the XSA file that you exported from the Vivado® Design Suite.
6. Select Operating system as **standalone** and Processor as **psu_cortexa53_0**.
7. Click **Finish**. The platform is generated with a single domain. 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 PMUFW.

9. In the platform view, click ![generate](image) to generate the platform. The Generation Successful message pops up.
Create a Sample Application

After installing the Vitis software platform, the next step is to create a software application project. Software application projects are your 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, as 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. To create a sample software application project, do the following.

1. Launch the Vitis software platform.
2. Select a workspace directory for your first project.
3. Click **Launch**. The welcome page appears.
5. Select **File → New → Vitis 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. Select the **Create from hardware** tab to create a new platform from an exported hardware design 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 do the following:

- 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. Open the Debug drop-down menu 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. Once you step through the `print()` function, you will see **Hello World** in the UART console.

![Image of a program running in Vitis IDE](image.png)

### Vitis IDE Extensions

The Vitis software platform has the following IDE extensions.

- **XSCT Console**: Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis software platform. 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.

- Creating platform projects and application projects
- Manage repositories
- Manage domain settings and add libraries to domains
- Set toolchain preferences
- Configure and build applications
- Download and run applications on hardware targets
- Create and flash boot images by running Bootgen and program_flash tools
• **Bootgen Utility:** 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. Bootgen comes with both a graphical user interface and a command line option. The tool is integrated into the Vitis software platform for generating basic boot images using a GUI, but the majority of Bootgen options are command line-driven.

  For more information on the Bootgen utility, see *Bootgen User Guide* (UG1283).

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

• **Repositories:** 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 workspace can point to multiple software repositories.

• **Program FPGA:** You can use the Program FPGA feature to program FPGA using bitstream.

• **Device Tree Generation:** Device tree (DT) is a data structure that describes hardware. This describes hardware that is readable by an operating system like Linux so that it does not need to hard code details of the machine. Linux uses the DT basically for platform identification, runtime configuration like bootargs, and device node population.
Develop

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

---

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 U-Boot and so on). Using this platform, you can create an application without creating the domain separately.

A platform project is a container for the hardware platform, runtime library, and settings for each processor, as well as the bootloader for the device. It can be as simple as a standalone domain for 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.

Creating a Platform

To create a new platform for application development in the Vitis integrated design environment (IDE), do the following:

1. Click File → New → Platform Project.
2. Click Specify to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.

4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, click to unselect the check box, then type or browse to the directory location.

5. From the **Hardware Platform** drop-down choose the appropriate platform for your application or click the **New** button to browse to an existing Hardware Platform.

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

7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.

8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.

9. Select **Project** → **Build Automatically** to automatically build the board support package.

10. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain. For details, see Using the Board Support Package Settings Page.

11. Click **OK** to accept the settings, build the platform, and close the dialog box.
Configuring a Domain/Board Support Package

There are various ways to launch the Board Support Package Settings dialog box.

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**.

![Board Support Package Settings](image)

### 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.

![Board Support Package Overview](image)
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 regenerates the domain/BSP sources.

If the Build Automatically 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.
**Adding a Domain to an Existing Platform**

**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.

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: You can give a custom Boot directory and *Bif* file for generation.

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. will generate them for you.

**Adding a Standalone 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](#).

2. Click the button.

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

4. Click OK.

**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](#).

2. Click the button.

3. Select the OS as **FreeRTOS** and a Processor of your choice.

4. Click OK.

**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 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 Domain for a New Hardware Platform

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 to that domain project are targeted (or referenced) to a single hardware project. Therefore, 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 software project to another hardware project:

1. Right-click the software domain project that your software application currently references.

2. Select Properties.

3. Select Project References.

4. Uncheck the reference to the current hardware project and select the hardware project which you would like to re-target.

5. Click OK. The Vitis software platform re-targets your domain project and application project to the new hardware project and starts rebuilding the projects.

   Alternatively, if you have two domain projects each targeting a different hardware project, you can re-target your software application to run or debug on a different hardware project by changing the domain reference accordingly.

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 tab and select the appropriate domain.

2. Click Reset BSP Sources.

3. Click Yes. This resets the sources for the domain/BSP selected.
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 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**.

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)**. Vitis 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 a 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 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 make files.
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**.
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 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 the **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 Template Application

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`. 

Chapter 2: Develop

UG1400 (v2019.2) October 1, 2019
Vitis Embedded Software Development
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 same as 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 application template name.

```tcl
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 write the description to provide application details.

```tcl
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.

```tcl
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.
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.

```bash
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.

```bash
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.

```bash
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.**
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 template, 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 Template Applications**

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 Tools → Repositories → Add.
   c. Select the repository folder, from the dialog box that appears.
      - **Note:** For standalone applications, parent folder that contains the applications should be sw_apps. Example: C:\temp\repo\sw_apps\custom_app_name. For Linux applications, parent folder that contains the applications should be sw_apps_linux. Example: C:\temp\repo\sw_apps_linux\custom_app_name.
e. Specify a project name. From the **OS Platform** drop down, select the OS supported by the user template application.

f. From the **Processor** drop down, select the processor supported by the user template application.

g. Click **Next**. The **Templates** page appears. This page lists all the available templates including the user template application created by you.

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:

```bash
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.

Once 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 table below:

**Table 1: Build Configurations**

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>Compiler Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>-O0 -g</td>
</tr>
<tr>
<td>Release</td>
<td>-O2</td>
</tr>
<tr>
<td>Profile</td>
<td>-O2 -g -pg</td>
</tr>
</tbody>
</table>

**Changing 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 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 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
- A64: 3001
- MicroBlaze: 3002

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

The final step in creating an executable from object files and libraries is linking. This 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. 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 app 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 Tools → 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 K 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 will be only one code section, .text, unless you specifically created other code sections. All the code sections will 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 doesn’t 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 K 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**

Once 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>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available Memory Regions</td>
<td>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.</td>
</tr>
<tr>
<td>Stack and Heap Sizes</td>
<td>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.</td>
</tr>
<tr>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Section to Memory Region Mapping</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 → Other**. The New Project dialog box appears.
2. Expand Xilinx and select **Library Project**.
3. Click **Next**. The **New Library Project** wizard appears.
4. Type a project name into the **Project Name** field.
5. 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.
6. The Library Type drop-down allows you to select the supported library types. You can choose to create a **Shared Library** or a **Static Library** project. The table below lists the flags set by the tool during the library project creation.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Toolchain</th>
<th>Standalone</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Static Library</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extra Compiler Flags</td>
<td>Archiver Flags</td>
</tr>
<tr>
<td>A9</td>
<td>Linaro</td>
<td><code>-mcpu=cortex-A9 -mfpu=vfpv3 -mfloat-abi=hard</code></td>
<td>None</td>
</tr>
<tr>
<td>A9</td>
<td>Code Sourcery</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>A53</td>
<td>Linaro</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>A53-32 Bit</td>
<td>Linaro</td>
<td><code>-march=armv7-a</code></td>
<td>None</td>
</tr>
<tr>
<td>R5</td>
<td>Linaro</td>
<td><code>-mcpu=cortex-r5</code></td>
<td>None</td>
</tr>
</tbody>
</table>
7. The OS Platform allows you to select which operating system you will be writing code for. The supported OS platforms are as follows:

- Linux: Shared libraries can be created only on the Linux OS platform.
- Standalone: Select this option if you plan to create a library for FreeRtos.

8. From the Processor 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 such as any Zynq PS.

9. Compiler type is 32-bit for all the processors except the psu_cortexa53 processor. You can also specify extra compiler settings under Advanced → Extra Compiler Flags on the wizard page.

10. Select your preferred language: Cor C++.

11. Click Finish to create your library project.

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

12. 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 in during the creation of the library project, a shared library (<library_name>.so) or a static library (<library_name>.a) is generated.

---

**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, refer 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.
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. The configurations for debugging, running, and profiling an application are similar; the differences between them are explained below. 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 are to select if **This is a self relocating application** and **Profiling Options** check-boxes.

**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.

• **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 (System Debugger) 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>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter configurations in closed projects</td>
<td>Filter out configurations that are associated with a project that is currently closed</td>
<td>On</td>
</tr>
<tr>
<td>Filter configurations in deleted or missing projects</td>
<td>Filter out configurations that are associated with a project that has been deleted or are simply no longer available</td>
<td>On</td>
</tr>
<tr>
<td>Apply windows working set</td>
<td>Applies the filtering from any working sets currently active to the visibility of configurations associated to 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.</td>
<td>On</td>
</tr>
<tr>
<td>Filter checked launch configuration types</td>
<td>Filter all configurations of the selected type regardless of the other filtering options. The checked options will not be displayed in the Run/Debug Configurations dialog box.</td>
<td>On</td>
</tr>
</tbody>
</table>

**Note:** To avoid any confusions, only configurations that are supported by the Vitis software platform are available by default.

| Delete configurations when associated project is deleted | Any launch configurations associated with a project being deleted will also be deleted if this option is enabled. Once 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.

![Target Connections View](image)

### Creating a New Target Connection

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

To create a 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 hw_server is running. 
6. In the Port field, type the port number on which hw_server is running. By default, 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 hw_server is running.

5. Specify the port number on which the hw_server is running. By default, 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 will be 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 hw_server manually on the remote host:
   
   a. Take a shell on the remote host.
   
   b. **Source the setup scripts.** `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 Tools → 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 named 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.

![Registers view](image.png)

**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 System Debugger.

```c
#define
main ()

/* main program, corresponds to procedures */
/* Main and Proc_0 in the Ada version */
{
    One_Fifty  Int_1_Loc;
    One_Fifty  Int_2_Loc;
    One_Fifty  Int_3_Loc;
    char       Ch_Index;
    Enumeration Enum_Loc;
    Str_30     Str_1_Loc;
    Str_30     Str_2_Loc;
    int        Run_Index;
    int        Number_Of_Runs;

    /* Initializations */
    init_platform();
```
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. Select Run → Debug Configurations.
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.

![Debug button](image1)

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

![Terminate button](image2)

**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. Click **Run → Debug Configurations**.

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:

19. Stepping through code might not work as expected due to the possible reordering of some instructions.

20. Some variables might be optimized out by the compiler and therefore 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 Xilinx C/C++ application 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 dropdown 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 `hw_server` with non-default port (for example: 3122) enables remote connections. Use the following command to launch `hw_server` on port 3122:

   ```
   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 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 SD card. For details on how to set up the kernel debug, refer [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 (新增).
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.

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

**Process/Thread Level Debugging**

The Debug view will be updated with the list of processes running on the Linux kernel, once the OS aware debugging is enabled. For details on how to enable OS aware debugging, refer to [Enabling OS Aware Debug](#). The processes list will be updated for the first time once the processor core is halted and dynamically thereafter (new processes will be added to the list and terminated processes will be 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()`. Everytime 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 like stepping in, stepping out, watching variables, stack trace can be performed. Target side path for a binary file does not include 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 will be updated with the list of processes running on the Linux kernel, once the OS aware debugging is enabled. The processes list will be updated for the first time once the processor core is halted and dynamically there after (new processes will be added to the list and terminated processes will be removed).

4. Click **Edit Source Lookup Path** to set the path mappings for the selected debug configuration. 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 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 *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 *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. Once 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 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. Select the Hardware Platform XML or HDF file, if it was not specified earlier. If you have not build hardware yet, you can select one of the pre-defined platforms from the drop-down. Alternatively, you can drag and drop an existing hardware specification XML/HDF file or search for one by clicking the **New** button and create a new hardware project. After completing the new hardware project creation, you are returned to the New Application Project dialog box.
   f. From the Processor 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.
   g. Select your preferred language: **C** or **C++**.
   h. Select the compiler: **64-bit** or **32-bit**.
i. From the Hypervisor Guest drop-down list, select **Yes** to create an application with a pre-defined linker script suitable to run the Xen.

j. 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`.

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

l. 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.

m. 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.

n. 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, like 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 doesn't provide details about where the program sections would be relocated to. Due to this reason, you must provide additional information to the debugger, about the address to which program sections will be relocated to. 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 to which all the program sections will 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 to which 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 relocate 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)
x pdb file /data/0x3bf37000
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 0x3ff5d5f84 m88e1118_startup()+1005809712: drivers/net/phy/marvell.c, line 356
  4 0x3ff5d5154 zynq_gem_init()+1005809192: drivers/net/zynq_gem.c, line 402
  5 0x3ff7dc58 eth_init()+1005809720: net/eth.c, line 886
  6 0x3ff7e0ec net_loop()+1005809728: net/net.c, line 407
  7 0x3ff46330 netboot_common()+1005809972: common/cmd_net.c, line 230
  8 0x3ff46520 do_tftpbin()+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 0x3ff3af40 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 3248
```

Chapter 3: Run, Debug, and Optimize
Cross-Triggering

Cross-triggering is supported by the ECT module supplied by Arm. ECT provides a mechanism for multiple subsystems in a 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.
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. Select Run → Debug Configurations.

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

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 table below shows the trigger input and trigger output connections of each CTI.

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

<table>
<thead>
<tr>
<th>CTI Trigger Port</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CTI connected to ETB, TPIU</strong></td>
<td></td>
</tr>
<tr>
<td>Trigger Input 2</td>
<td>ETB full</td>
</tr>
<tr>
<td>Trigger Input 3</td>
<td>ETB acquisition complete</td>
</tr>
<tr>
<td>Trigger Input 4</td>
<td>ITM trigger</td>
</tr>
<tr>
<td>Trigger Output 0</td>
<td>ETB flush</td>
</tr>
<tr>
<td>Trigger Output 1</td>
<td>ETB trigger</td>
</tr>
<tr>
<td>Trigger Output 2</td>
<td>TPIU flush</td>
</tr>
<tr>
<td>Trigger Output 3</td>
<td>TPIU trigger</td>
</tr>
<tr>
<td><strong>FTM CTI</strong></td>
<td></td>
</tr>
<tr>
<td>Trigger Input 0</td>
<td>FTM trigger</td>
</tr>
<tr>
<td>Trigger Input 1</td>
<td>FTM trigger</td>
</tr>
<tr>
<td>Trigger Input 2</td>
<td>FTM trigger</td>
</tr>
<tr>
<td>Trigger Input 3</td>
<td>FTM trigger</td>
</tr>
<tr>
<td>Trigger Output 0</td>
<td>FTM trigger</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>CTI Trigger Port</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT 5</td>
<td>ETR TRIGIN</td>
</tr>
<tr>
<td>OUT 6</td>
<td>TPIU FLUSHIN</td>
</tr>
<tr>
<td>OUT 7</td>
<td>TPIU TRIGIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CTI 1 (soc_debug_fpd)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN 0</td>
<td>FTM</td>
</tr>
<tr>
<td>IN 1</td>
<td>FTM</td>
</tr>
<tr>
<td>IN 2</td>
<td>FTM</td>
</tr>
<tr>
<td>IN 3</td>
<td>FTM</td>
</tr>
<tr>
<td>IN 4</td>
<td>STM TRIGOUTSPTE</td>
</tr>
<tr>
<td>IN 5</td>
<td>STM TRIGOUTSW</td>
</tr>
<tr>
<td>IN 6</td>
<td>STM TRIGOUTHETE</td>
</tr>
<tr>
<td>IN 7</td>
<td>STM ASYNCOOUT</td>
</tr>
<tr>
<td>OUT 0</td>
<td>FTM</td>
</tr>
<tr>
<td>OUT 1</td>
<td>FTM</td>
</tr>
<tr>
<td>OUT 2</td>
<td>FTM</td>
</tr>
<tr>
<td>OUT 3</td>
<td>FTM</td>
</tr>
<tr>
<td>OUT 4</td>
<td>STM HWEVENTS</td>
</tr>
<tr>
<td>OUT 5</td>
<td>STM HWEVENTS</td>
</tr>
<tr>
<td>OUT 6</td>
<td></td>
</tr>
<tr>
<td>OUT 7</td>
<td>HALT SYSTEM TIMER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CTI 2 (soc_debug_fpd)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN 0</td>
<td>ATM 0</td>
</tr>
<tr>
<td>IN 1</td>
<td>ATM 1</td>
</tr>
<tr>
<td>IN 2</td>
<td></td>
</tr>
<tr>
<td>IN 3</td>
<td></td>
</tr>
<tr>
<td>IN 4</td>
<td></td>
</tr>
<tr>
<td>IN 5</td>
<td></td>
</tr>
<tr>
<td>IN 6</td>
<td></td>
</tr>
<tr>
<td>IN 7</td>
<td></td>
</tr>
<tr>
<td>OUT 0</td>
<td>ATM 0</td>
</tr>
<tr>
<td>OUT 1</td>
<td>ATM 1</td>
</tr>
<tr>
<td>OUT 2</td>
<td></td>
</tr>
<tr>
<td>OUT 3</td>
<td></td>
</tr>
<tr>
<td>OUT 4</td>
<td></td>
</tr>
<tr>
<td>OUT 5</td>
<td></td>
</tr>
<tr>
<td>OUT 6</td>
<td></td>
</tr>
<tr>
<td>OUT 7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CTI 0,1 (RPU)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN 0</td>
<td>DBGTRIGGER</td>
</tr>
<tr>
<td>IN 1</td>
<td>PMUIRQ</td>
</tr>
<tr>
<td>CTI Trigger Port</td>
<td>Signal</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>IN 2</td>
<td>ETMEXTOUT[0]</td>
</tr>
<tr>
<td>IN 3</td>
<td>ETMEXTOUT[1]</td>
</tr>
<tr>
<td>IN 4</td>
<td>COMMRX</td>
</tr>
<tr>
<td>IN 5</td>
<td>COMMTX</td>
</tr>
<tr>
<td>IN 6</td>
<td>ETM TRIGGER</td>
</tr>
<tr>
<td>IN 7</td>
<td>-</td>
</tr>
<tr>
<td>OUT 0</td>
<td>EDBGRQ</td>
</tr>
<tr>
<td>OUT 1</td>
<td>ETMEXTIN[0]</td>
</tr>
<tr>
<td>OUT 2</td>
<td>ETMEXTIN[1]</td>
</tr>
<tr>
<td>OUT 3</td>
<td>-(CTIIRQ, not connected)</td>
</tr>
<tr>
<td>OUT 4</td>
<td>-</td>
</tr>
<tr>
<td>OUT 5</td>
<td>-</td>
</tr>
<tr>
<td>OUT 6</td>
<td>-</td>
</tr>
<tr>
<td>OUT 7</td>
<td>DBGRESTART</td>
</tr>
<tr>
<td><strong>CTI 0,1,2,3 (APU)</strong></td>
<td></td>
</tr>
<tr>
<td>IN 0</td>
<td>DBGTRIGGER</td>
</tr>
<tr>
<td>IN 1</td>
<td>PMUIRQ</td>
</tr>
<tr>
<td>IN 2</td>
<td>-</td>
</tr>
<tr>
<td>IN 3</td>
<td>-</td>
</tr>
<tr>
<td>IN 4</td>
<td>ETMEXTOUT[0]</td>
</tr>
<tr>
<td>IN 5</td>
<td>ETMEXTOUT[1]</td>
</tr>
<tr>
<td>IN 6</td>
<td>ETMEXTOUT[2]</td>
</tr>
<tr>
<td>IN 7</td>
<td>ETMEXTOUT[3]</td>
</tr>
<tr>
<td>OUT 0</td>
<td>EDBGRQ</td>
</tr>
<tr>
<td>OUT 1</td>
<td>DBGRESTART</td>
</tr>
<tr>
<td>OUT 2</td>
<td>CTIIRQ</td>
</tr>
<tr>
<td>OUT 3</td>
<td>-</td>
</tr>
<tr>
<td>OUT 4</td>
<td>ETMEXTIN[0]</td>
</tr>
<tr>
<td>OUT 5</td>
<td>ETMEXTIN[1]</td>
</tr>
<tr>
<td>OUT 6</td>
<td>ETMEXTIN[2]</td>
</tr>
<tr>
<td>OUT 7</td>
<td>ETMEXTIN[3]</td>
</tr>
</tbody>
</table>
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 ... → 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.
Profiling Linux Applications with System Debugger

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

1. Create a 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 into 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:

\[ B = \log_2\left(\frac{H - L + S \times 4}{S \times 4}\right) \]

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.
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 platform 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:

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. The OS Platform allows you to select which operating system you will be writing code for. Select `freertos823_xilinx`.

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

5. Select the Hardware Platform XML or HDF file, if it was not specified earlier. If you have not built hardware yet, you can select one of the pre-defined platforms from the drop-down. Alternatively, you can drag and drop an existing hardware specification XML/HDF file or search for one by clicking the New button and create a new hardware project. After completing the new hardware project creation, you are returned to the New Application Project dialog box.

6. From the Processor 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 such as any Zynq PS.

7. Select your preferred language: C or C++.

8. Select the compiler: 64-bit or 32-bit.

9. Select a board support package or domain. You can opt to have the tools build a customizable domain for this application, or you can choose an existing domain.

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

11. 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.

12. 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.

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

14. Open **BSP Settings → Overview → FreeRTOS** and change the value of `enable_stm_event_trace` to true.

15. Click **Run → Debug Configurations**.

16. In the Debug Configurations dialog box, double-click **Xilinx C/C++ application** to create a launch configuration for the selected project.

17. Click **Debug**. Debugging begins, with the processors in the running state.

18. Debug the project using the system debugger on the required target.

19. Wait for project to be downloaded on to board and stop at `main()`.

20. Click **Window → Show View → Xilinx**. The Show View dialog box appears.

21. 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.
22. Click the start button in the Trace Session Manager view toolbar, to start the FreeRTOS trace collection.

23. Switch to the Debug view and resume the project.

24. Allow the project to run.

25. 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 milliseconds. 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.
Project 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 from the FileExport dialog box appears.
2. Expand Tracing and select Trace Package Export, main menu. The
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.
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 figure below.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Event Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00:00 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:01 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:02 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:03 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:04 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:05 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:06 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:07 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:08 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:09 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:10 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:11 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:12 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:13 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
<tr>
<td>09:00:14 AM</td>
<td>APM</td>
<td>HW Write Byte(s)</td>
</tr>
<tr>
<td>09:00:15 AM</td>
<td>APM</td>
<td>HW Write Transactions</td>
</tr>
</tbody>
</table>

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 will select the next matching event starting from the top currently displayed event. Wrapping will occur 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 will be dimmed.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Event type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00:04:04</td>
<td>PS</td>
<td>CPU Cycles=64*</td>
</tr>
<tr>
<td>0.00:04:17</td>
<td>PS</td>
<td>CPU Cycles=64, CPU Cycles=64*</td>
</tr>
<tr>
<td>0.00:04:45</td>
<td>PS</td>
<td>CPU Cycles=64*</td>
</tr>
</tbody>
</table>

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.

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.
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 table below lists the keys and the action performed when they are used in the Histogram view.

• **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.

![Colors View](image)
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, use 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.

APM Performance Graphs

APM metrics are displayed using the graphs.

APM Performance Counters

APM metrics displayed in tabular format.
FreeRTOS Analysis

FreeRTOS event trace displayed in different states.

<table>
<thead>
<tr>
<th>Write Transactions per ms</th>
<th>H0</th>
<th>H01</th>
<th>H02</th>
<th>H03</th>
<th>ACP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Write Latency</td>
<td>0.8</td>
<td>0.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Maximum Write Latency</td>
<td>12.0</td>
<td>12.0</td>
<td>12.0</td>
<td>12.0</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Average Write Latency</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
</tr>
<tr>
<td>Write Throughput (MB/sec)</td>
<td>169.8</td>
<td>169.8</td>
<td>169.8</td>
<td>169.8</td>
<td>169.8</td>
<td>169.8</td>
</tr>
</tbody>
</table>

| Read Transactions per ms  | 50.7| 15.7| 18.5| 19.3| 21.3| 23.3|
| Minimum Read Latency      | 0.8 | 0.8 | 0.0 | 0.0 | 0.0 | 0.0 |
| Maximum Read Latency      | 17.1| 17.1| 17.1| 17.1| 17.1| 17.1|
| Average Read Latency      | 26.3| 26.3| 26.3| 26.3| 26.3| 26.3|
| Read Throughput (MB/sec)  | 330.7| 330.7| 330.7| 330.7| 330.7| 330.7|

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.

![Configure Session](image)

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.
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>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS Clock Frequency (MHz)</td>
<td>The clock frequency of the Zynq-7000 SoC PS (specified in MHz).</td>
<td>666.7 MHz</td>
</tr>
<tr>
<td>PL Clock Frequency (MHz)</td>
<td>The clock frequency of the Zynq-7000 SoC PL (specified in MHz).</td>
<td>100.0 MHz</td>
</tr>
<tr>
<td>DDR Clock Frequency (MHz)</td>
<td>The clock frequency of the DDR memory (specified in MHz).</td>
<td>533.3 MHz</td>
</tr>
<tr>
<td>DDR Data Path Width</td>
<td>The bit width used in the DDR memory data path. Possible values are 16 and 32 bits.</td>
<td>32 bits</td>
</tr>
<tr>
<td>DDR Port 0 - Enable HPR</td>
<td>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.</td>
<td>Unchecked</td>
</tr>
<tr>
<td>DDR Port 1 - Enable HPR</td>
<td>This enables the usage of high priority reads on DDR port 1. This port is used by other masters via the central interconnect.</td>
<td>Unchecked</td>
</tr>
<tr>
<td>DDR Port 2 - Enable HPR</td>
<td>This enables the usage of high priority reads on DDR port 2. This port is used by HP2 and HP3.</td>
<td>Unchecked</td>
</tr>
<tr>
<td>DDR Port 3 - Enable HPR</td>
<td>This enables the usage of high priority reads on DDR port 3. This port is used by HP0 and HP1.</td>
<td>Unchecked</td>
</tr>
<tr>
<td>HPR/LPR Queue Partitioning</td>
<td>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.</td>
<td>HPR(0)/LPR(32)</td>
</tr>
<tr>
<td>LPR to Critical Priority Level</td>
<td>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.</td>
<td>2</td>
</tr>
<tr>
<td>HPR to Critical Priority Level</td>
<td>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.</td>
<td>15</td>
</tr>
<tr>
<td>Write to Critical Priority Level</td>
<td>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.</td>
<td>2</td>
</tr>
</tbody>
</table>

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>.hdf` into the Vitis software platform.
   c. Select **Run → Run Configurations**.
   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**.
   b. Select **Performance Analysis** and click the **New** button to create a performance analysis configuration.
   c. Select **Attach to running target** from the **Debug Type** dropdown list.
   d. Specify the PS clock frequency in the **PS Processor Frequency(MHz)** textbox.
   e. If you have an APM in your design, select **Enable APM Counters**.
   f. Click **Edit** to specify the hardware information of the APM, to be used for performance analysis, in the APM Hardware Information dialog box.
      1. Click the **New** icon to add a new row.
      2. Specify a unique identifier in the APM Id column.
      3. Specify the base address of the APM in the Base Address column.
      4. Specify the frequency of the clock connected to `s_axi_aclk` in the **Frequency(Hz)** column.
      5. Specify the number of slots in the Slots Count column.
      6. Click **OK** to save the details and close the **APM Hardware Information** dialog box.
   g. Click **Edit** to select the APM ports or slots that should be enabled for performance analysis, in the Configure APM dialog box.
   h. Click **OK** to save the details and close the **Configure APM** dialog box.
i. Click Run to open the Performance Analysis perspective.

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 #unique_144.

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**

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 ZCU702. Hence `-flash_type qspi_single` is used as an option in `program_flash`.

```
setws /tmp/wrk/workspace
createhw -name hw0 -hwspec /tmp/wrk/system.hdf
createapp -name fsbl -app {Zynq FSBL} -proc ps7_cortexa9_0 -hwproject hw0 -os standalone
createapp -name hello -app {Hello World} -proc ps7_cortexa9_0 -hwproject hw0 -os standalone
```
Other Xilinx 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 #unique_149.

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.

   **Import Type**
   
   Select an archive created by the Export Vitis Project wizard, or an Eclipse based project folder or zip file.

   - Vitis project exported zip file
   - Eclipse workspace or zip file

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

Specify projects to be imported.

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 device tree generator local path, 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.
Overview

The Vitis™ integrated development environment (IDE) is designed to develop embedded software applications for Xilinx® embedded processors. The Vitis unified software platform works with hardware designs created with the Vivado® Design Suite and is based on the Eclipse open source standard.

The Vitis software platform includes the following features:

- 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 FPGAs
- Bootable image creation
- Flash programming
- Scriptable command-line tool

Document Scope and Audience

The purpose of this guide is to familiarize software application developers and system software developers with the Vitis software platform by providing the following:

- Overview of the Vitis software platform and its features
- Software development in the Vitis software platform
New Concepts in the Vitis Software Platform

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.

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.
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 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.

Once the platform is generated, the dialog box shows the status of platform generation.

6. Optional: To see the hardware specification, switch from the Main tab to the Hardware Specification tab on the bottom left.
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 dialog box appears.
2. Provide a project name in the Project name field.
3. Click Next.
4. In the Platform Project dialog box, 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 to a platform with at least one domain.

1. 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-click Platform Project in the Explorer window 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 Application Project wizard, use the following steps:

1. Select File → New → Application Project to launch the Vitis 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.

![Platform selection screen](image)

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 view.
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 Explorer, 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 Vitis™ Application Project dialogue box.

2. Provide a project name.

3. Click **Next**.
4. Click **Create from Hardware** 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 Operating system 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 dialog 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.

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

**Exporting the DSA/XSA files from the Vivado Design Suite**

**Exporting the XSA for the Vitis Software Platform**

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

You can also run the following command in the Tcl console of the Vivado tool to generate the XSA file:

```plaintext
write_hw_platform -fixed -force -include_bit ./design_1.xsa
```

**Exporting the DSA for the Vitis Software Platform**

After you have generated the bitstream in the Vivado® tool, run the following commands in the Tcl console of the Vivado tool to generate the DSA file that is being used in Vitis.

```plaintext
set_property dsa.board_id zcu102 [current_project]
set_property dsa.name zcu102 [current_project]
write_dsa -fixed -force zcu102.dsa
write_dsa -fixed -force -include_bit zcu102.dsa
```
Creating Multiple Domains for a Single Hardware

In the Vitis™ software platform, 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 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 ![new domain](image)

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 ![new domain](image) 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 Editor 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. Select Platform Project in the Project Explorer view.
2. Right-click Platform Project and select Update Hardware Specification.
3. Specify the source hardware specification file in the Update hardware specification for test dialog box.
4. 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 window, 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 **System Project** in Explorer, select **Run as** or **Debug as**, then select **Launch on Hardware**.

2. Open the XSCT console to see the detailed commands and logs.

Creating a Bootable Image

When a system project is selected, by running build, 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 boot 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 the Project Explorer view.

2. Right-click the application and select **Create Boot Image** to open the Create Boot Image window.

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 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 device tree generator local path, 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 an FSBL Application and Editing FSBL Sources

1. Select File → New → Application Project.
2. Provide a name for your project in the project name field.

3. Select the **Create from DSA** option. This will list all the hardware designs available for this installation.

4. Select **zcu102.dsa**.
   You can browse to your DSA and select it.

5. Click **OK** to see a list of supported CPUs and operating systems.

6. Select **cortexa53_0** from the CPU drop-down list if you want to debug the A53 application.

7. Select **standalone** from the Operating System drop-down list.
   A list of supported application projects for this operating systems and processor combinations is displayed.
8. Select **Zynq MP FSBL** template from the list and click **Finish**.

After the project is created, you can modify FSBL sources based on your requirement. The FSBL sources are available at `<project name>/src/`.

### Modifying the BSP Settings of the FSBL Application

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

1. Double-click `<app_name>.prj` and click **Navigate to BSP Settings**.
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. Click **Build Project** to build the FSBL application project.

### 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 had created and generated in **Creating a Platform Project**. Click **Next**.
4. Provide the system configuration and software details for your project and click **Next**.
5. Select a template to create your project. In this case, select "Hello World."
6. Click **Finish** to build this application project.
Debugging the “Hello World” Application using the Modified FSBL

1. Right-click the application project and select **Debug As → Debug Configurations**.
2. Double-click **Launch on Hardware (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. Browse to the modified FSBL .elf file path for FSBL File.
   By default, debug/run will initialize the target using FSBL generated as part of the platform. It is recommended that you use the custom FSBL generated in Creating an FSBL Application and Editing FSBL Sources.
6. Click **Debug** to switch the perspective. 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. 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. The diagram below gives an overview of how a repository can be structured. Software components in your repository must belong to one of three 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.
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 platform 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 one hardware design file and click **OK**.
   A list of supported CPUs and operating systems is displayed.

5. Select **cortexa53_0** from the CPU drop-down list if you want to debug the A53 application.

6. Select **standalone** from the Operating system drop-down list.
   A list of supported application projects for this operating system and processor combination is displayed.

7. 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)**.
Chapter 5

Bootgen Tool

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.

For more information about Vitis, see Vitis help.

Boot Time Security

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

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

- Zynq-7000 SoC Technical Reference Manual (UG585)

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

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

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

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

Zynq-7000 SoC Boot and Configuration

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

BootROM on Zynq-7000 SoC

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

Optionally, you can execute the FSBL directly from a Quad-SPI or NOR device in a non-secure environment. The master boot device holds one or more boot images. A boot image is made up of the boot header and the first stage boot loader (FSBL). Additionally, a boot image can have programmable logic (PL), a second stage boot loader (SSBL), and an embedded operating system and applications; however, these are not accessed by the BootROM. The BootROM execution flow is affected by the boot mode pin strap settings, the Boot Header, and what it discovers about the system. The BootROM can execute in a secure environment with encrypted FSBL, or a non-secure environment. The supported boot modes are:
- JTAG mode is primarily used for development and debug.
- NAND, parallel NOR, Serial NOR (Quad-SPI), and Secure Digital (SD) flash memories are used for booting the device. The *Zynq SoC Technical Reference Manual (UG585)* provides the details of these boot modes. See *Zynq-7000 Boot and Configuration AR#52538* for answers to common boot and configuration questions.

**Zynq-7000 SoC Boot Image Layout**

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

![Figure 1: Boot Header](image)

**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**
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 2: Zynq-7000 SoC Boot Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1F</td>
<td>Arm® Vector table</td>
<td>Filled with dummy vector table by Bootgen (Arm Op code 0xEAFFFFFE, which is a branch-to-self infinite loop intended to catch uninitialized vectors.</td>
</tr>
<tr>
<td>0x20</td>
<td>Width Detection Word</td>
<td>This is required to identify the QSPI flash in single/dual stacked or dual parallel mode. 0xAA995566 in little endian format.</td>
</tr>
<tr>
<td>0x24</td>
<td>Header Signature</td>
<td>Contains 4 bytes ‘X’,’N’,’L’,’X’ in byte order, which is 0x584c4e58 in little endian format.</td>
</tr>
<tr>
<td>0x28</td>
<td>Key Source</td>
<td>Location of encryption key within the device: 0x3A5C3C5A: Encryption key in BBRAM. 0xA5C3C5A3: Encryption key in eFUSE. 0x00000000: Not Encrypted.</td>
</tr>
<tr>
<td>0x30</td>
<td>Source Offset</td>
<td>Location of FSBL (bootloader) in this image file.</td>
</tr>
<tr>
<td>0x34</td>
<td>FSBL Image Length</td>
<td>Length of the FSBL, after decryption.</td>
</tr>
<tr>
<td>0x38</td>
<td>FSBL Load Address (RAM)</td>
<td>Destination RAM address to which to copy the FSBL.</td>
</tr>
<tr>
<td>0x3C</td>
<td>FSBL Execution address (RAM)</td>
<td>Entry vector for FSBL execution.</td>
</tr>
<tr>
<td>0x40</td>
<td>Total FSBL Length</td>
<td>Total size of FSBL after encryption, including authentication certificate (if any) and padding.</td>
</tr>
<tr>
<td>0x44</td>
<td>QSPI Configuration Word</td>
<td>Hard coded to 0x00000001.</td>
</tr>
<tr>
<td>0x48</td>
<td>Boot Header Checksum</td>
<td>Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.</td>
</tr>
<tr>
<td>0x4c-0x97</td>
<td>User Defined Fields</td>
<td>76 bytes</td>
</tr>
<tr>
<td>0x98</td>
<td>Image Header Table Offset</td>
<td>Pointer to Image Header Table (word offset).</td>
</tr>
<tr>
<td>0x9C</td>
<td>Partition Header Table Offset</td>
<td>Pointer to Partition Header Table (word offset).</td>
</tr>
</tbody>
</table>
**Zynq-7000 SoC Register Initialization Table**

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

**Table 3: Zynq-7000 SoC Register Initialization Table**

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

**Zynq-7000 SoC Image Header Table**

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

**Table 4: Zynq-7000 SoC Image Header Table**

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

**Zynq-7000 SoC Image Header**

The Image Header is an array of structures containing information related to each image, such as an ELF file, bitstream, data files, and so forth. Each image can have multiple partitions, for example an ELF can have multiple loadable sections, each of which forms a partition in the boot image. The table will also contain the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC device.
Table 5: Zynq-7000 SoC Image Header

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

Zynq-7000 SoC Partition Header

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

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

Table 6: Zynq-7000 SoC Partition Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Encrypted Partition length</td>
<td>Encrypted partition data length.</td>
</tr>
<tr>
<td>0x04</td>
<td>Unencrypted Partition length</td>
<td>Unencrypted data length.</td>
</tr>
<tr>
<td>0x08</td>
<td>Total partition word length</td>
<td>The total partition word length comprises the encrypted information length with padding, the expansion length, and the authentication length.</td>
</tr>
<tr>
<td></td>
<td>(Includes Authentication Certificate)</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td>Destination load address.</td>
<td>The RAM address into which this partition is to be loaded.</td>
</tr>
<tr>
<td>0x10</td>
<td>Destination execution address.</td>
<td>Entry point of this partition when executed.</td>
</tr>
<tr>
<td>0x14</td>
<td>Data word offset in Image</td>
<td>Position of the partition data relative to the start of the boot image.</td>
</tr>
</tbody>
</table>
Table 6: Zynq-7000 SoC Partition Header (cont’d)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18</td>
<td>Attribute Bits</td>
<td>See Zynq-7000 SoC Partition Attribute Bits</td>
</tr>
<tr>
<td>0x1C</td>
<td>Section Count</td>
<td>Number of sections in a single partition.</td>
</tr>
<tr>
<td>0x20</td>
<td>Checksum Word Offset</td>
<td>Location of the corresponding checksum word in the boot image.</td>
</tr>
<tr>
<td>0x24</td>
<td>Image Header Word Offset</td>
<td>Location of the corresponding Image Header in the boot image.</td>
</tr>
<tr>
<td>0x28</td>
<td>Authentication Certification Word Offset</td>
<td>Location of the corresponding Authentication Certification in the boot image.</td>
</tr>
<tr>
<td>0x2C-0x38</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3C</td>
<td>Header Checksum</td>
<td>Sum of the previous words in the Partition Header.</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Partition Attribute Bits

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

Table 7: Zynq-7000 SoC Partition Attribute Bits

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:18</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
</tbody>
</table>
| 17:16     | Partition owner           | 0: FSBL  
1: UBOOT  
2 and 3: reserved |
| 15        | RSA signature present     | 0: No RSA authentication certificate  
1: RSA authentication certificate |
| 14:12     | Checksum type             | 0: None  
1: MD5  
2-7: reserved |
| 11:8      | Reserved                  | Not used                                                   |
| 7:4       | Destination device        | 0: None  
1: PS  
2: PL  
3: INT  
4-15: Reserved |
| 3:2       | Reserved                  | Not used                                                   |
| 1:0       | Reserved                  | Not used                                                   |
**Zynq-7000 SoC Authentication Certificate**

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

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

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

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

**Zynq-7000 SoC Authentication Certificate Header**

The following table describes the Zynq®-7000 SoC Authentication Certificate Header.
Table 9: Zynq-7000 SoC Authentication Certificate Header

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

**Zynq-7000 SoC Boot Image Block Diagram**

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

Introduction

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

During initial boot, the following steps occur:

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

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

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

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

Zynq UltraScale+ MPSoC Boot Image

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

<table>
<thead>
<tr>
<th>Boot Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Initialization Table</td>
</tr>
<tr>
<td>PUF Helper Data (Optional)</td>
</tr>
<tr>
<td>Image Header Table</td>
</tr>
<tr>
<td>Image Header 1</td>
</tr>
<tr>
<td>Image Header 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Image Header n</td>
</tr>
<tr>
<td>Partition Header 1</td>
</tr>
<tr>
<td>Partition Header 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Partition Header n</td>
</tr>
<tr>
<td>Header Authentication Certificate (Optional)</td>
</tr>
<tr>
<td>Partition 1 (FSBL)</td>
</tr>
<tr>
<td>PMU FW (Optional)</td>
</tr>
<tr>
<td>AC (Optional)</td>
</tr>
<tr>
<td>Partition 2</td>
</tr>
<tr>
<td>AC (Optional)</td>
</tr>
<tr>
<td>Partition n</td>
</tr>
<tr>
<td>AC (Optional)</td>
</tr>
</tbody>
</table>

**Zynq UltraScale+ MPSoC Boot Header**

**About the Boot Header**

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

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

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

Table 10: Zynq UltraScale+ MPSoC Device Boot Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1F</td>
<td>Arm® vector table</td>
<td>XIP ELF vector table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xEAFFFFFE: for Cortex™-R5F and Cortex A53 (32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x14000000: for Cortex A53 (64-bit)</td>
</tr>
<tr>
<td>0x20</td>
<td>Width Detection Word</td>
<td>This field is used for QSPI width detection. 0xAA995566 in little endian format.</td>
</tr>
<tr>
<td>0x24</td>
<td>Header Signature</td>
<td>Contains 4 bytes ‘X’, ‘N’, ‘L’, ‘X’ in byte order, which is 0x584c4e58 in little endian format.</td>
</tr>
<tr>
<td>0x28</td>
<td>Key Source</td>
<td>0x00000000 (Un-Encrypted)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA5C3C5A5 (Black key stored in eFUSE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA5C3C5A7 (Obfuscated key stored in eFUSE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x3A5C3C5A (Red key stored in BBRAM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA5C3C5A3 (eFUSE RED key stored in eFUSE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA35C7CA5 (Obfuscated key stored in Boot Header)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA3A5C3C5 (USER key stored in Boot Header)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xA35C7C53 (Black key stored in Boot Header)</td>
</tr>
<tr>
<td>0x2C</td>
<td>FSBL Execution address (RAM)</td>
<td>FSBL execution address in OCM or XIP base address.</td>
</tr>
<tr>
<td>0x30</td>
<td>Source Offset</td>
<td>If no PMUFW, then it is the start offset of FSBL. If PMUFW, then start of PMUFW.</td>
</tr>
<tr>
<td>0x34</td>
<td>PMU Image Length</td>
<td>PMU FW original image length in bytes. (0-128KB).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If size &gt; 0, PMUFW is prefixed to FSBL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If size = 0, no PMUFW image.</td>
</tr>
<tr>
<td>0x38</td>
<td>Total PMU FW Length</td>
<td>Total PMUFW image length in bytes.(PMUFW length + encryption overhead)</td>
</tr>
<tr>
<td>0x3C</td>
<td>FSBL Image Length</td>
<td>Original FSBL image length in bytes. (0-250KB).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 0, XIP bootimage is assumed.</td>
</tr>
<tr>
<td>0x40</td>
<td>Total FSBL Length</td>
<td>FSBL image length + Encryption overhead of FSBL image + Auth. Cert., + 64byte alignment + hash size (Integrity check).</td>
</tr>
</tbody>
</table>
### Table 10: Zynq UltraScale+ MPSoC Device Boot Header (cont’d)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x44</td>
<td>FSBL Image Attributes</td>
<td>See Bit Attributes.</td>
</tr>
<tr>
<td>0x48</td>
<td>Boot Header Checksum</td>
<td>Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.</td>
</tr>
<tr>
<td>0x4C-0x68</td>
<td>Obfuscated/Black Key Storage</td>
<td>Stores the Obfuscated key or Black key.</td>
</tr>
<tr>
<td>0x6C</td>
<td>Shutter Value</td>
<td>32-bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time.</td>
</tr>
<tr>
<td>0x70 -0x94</td>
<td>User-Defined Fields (UDF)</td>
<td>40 bytes.</td>
</tr>
<tr>
<td>0x98</td>
<td>Image Header Table Offset</td>
<td>Pointer to Image Header Table. (word offset)</td>
</tr>
<tr>
<td>0x9C</td>
<td>Partition Header Table Offset</td>
<td>Pointer to Partition Header. (word offset)</td>
</tr>
<tr>
<td>0xA0-0xA8</td>
<td>Secure Header IV</td>
<td>IV for secure header of bootloader partition.</td>
</tr>
<tr>
<td>0x0AC-0xB4</td>
<td>Obfuscated/Black Key IV</td>
<td>IV for Obfuscated or Black key.</td>
</tr>
</tbody>
</table>

### Zynq UltraScale+ MPSoC Boot Header Attribute Bits

### Table 11: Zynq UltraScale+ MPSoC Boot Header Attribute Bits

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Offset</th>
<th>Width</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>31:16</td>
<td>16</td>
<td>0x0</td>
<td>Reserved. Must be 0.</td>
</tr>
<tr>
<td>BHDR RSA</td>
<td>15:14</td>
<td>2</td>
<td>0x0</td>
<td>0x3: RSA Authentication of the boot image will be done, excluding verification of PPK hash and SPK ID. All Others Others : RSA Authentication will be decided based on eFuse RSA bits.</td>
</tr>
<tr>
<td>Reserved</td>
<td>13:12</td>
<td>2</td>
<td>0x0</td>
<td>NA</td>
</tr>
<tr>
<td>CPU Select</td>
<td>11:10</td>
<td>2</td>
<td>0x0</td>
<td>0x0: R5 Single 0x1: A53 Single 0x2: R5 Dual 0x3: Reserved</td>
</tr>
<tr>
<td>Hashing Select</td>
<td>9:8</td>
<td>2</td>
<td>0x0</td>
<td>0x0, 0x1 : No Integrity check 0x3: SHA3 for BI integrity check</td>
</tr>
</tbody>
</table>
Table 11: Zynq UltraScale+ MPSoC Boot Header Attribute Bits (cont’d)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Offset</th>
<th>Width</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUF-HD</td>
<td>7:6</td>
<td>2</td>
<td>0x0</td>
<td>0x3: PUF HD is part of boot header. All other: PUF HD is in eFuse</td>
</tr>
<tr>
<td>Reserved</td>
<td>5:0</td>
<td>6</td>
<td>0x0</td>
<td>Reserved for future use. Must be 0.</td>
</tr>
</tbody>
</table>

Zynq UltraScale+ MPSoC Register Initialization Table

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

Table 12: Zynq UltraScale+ MPSoC Register Initialization Table

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

Zynq UltraScale+ MPSoC PUF Helper Data

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

Table 13: Zynq UltraScale+ MPSoC PUF Helper Data

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

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

Table 14: Zynq UltraScale+ MPSoC Device Image Header Table

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

Zynq UltraScale+ MPSoC Image Header

About Image Headers

The Image Header is an array of structures containing information related to each image, such as an ELF file, bitstream, data files, and so forth. Each image can have multiple partitions, for example an ELF can have multiple loadable sections, each of which form a partition in the boot image. The table will also contain the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the Zynq® UltraScale+™ MPSoC.
### Table 15: Zynq UltraScale+ MPSoC Device Image Header

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

### Zynq UltraScale+ MPSoC Partition Header

#### About the Partition Header

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

### Table 16: Zynq UltraScale+ MPSoC Device Partition Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Encrypted Partition Data Word Length</td>
<td>Encrypted partition data length.</td>
</tr>
<tr>
<td>0x04</td>
<td>Un-encrypted Data Word Length</td>
<td>Unencrypted data length.</td>
</tr>
<tr>
<td>0x08</td>
<td>Total Partition Word Length (Includes Authentication Certificate. See Authentication Certificate.</td>
<td>The total encrypted + padding + expansion + authentication length.</td>
</tr>
<tr>
<td>0x0C</td>
<td>Next Partition Header Offset LO</td>
<td>Location of next partition header (word offset).</td>
</tr>
<tr>
<td>0x10</td>
<td>Destination Execution Address</td>
<td>The lower 32-bits of executable address of this partition after loading.</td>
</tr>
<tr>
<td>0x14</td>
<td>Destination Execution Address HI</td>
<td>The higher 32-bits of executable address of this partition after loading.</td>
</tr>
<tr>
<td>0x18</td>
<td>Destination Load Address LO</td>
<td>The lower 32-bits of RAM address into which this partition is to be loaded.</td>
</tr>
<tr>
<td>0x1C</td>
<td>Destination Load Address HI</td>
<td>The higher 32-bits of RAM address into which this partition is to be loaded.</td>
</tr>
</tbody>
</table>
### Table 16: Zynq UltraScale+ MPSoC Device Partition Header (cont’d)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>Actual Partition Word Offset</td>
<td>The position of the partition data relative to the start of the boot image. (word offset)</td>
</tr>
<tr>
<td>0x24</td>
<td>Attributes</td>
<td>See Zynq UltraScale+ MPSoC Partition Attribute Bits</td>
</tr>
<tr>
<td>0x28</td>
<td>Section Count</td>
<td>The number of sections associated with this partition.</td>
</tr>
<tr>
<td>0x2C</td>
<td>Checksum Word Offset</td>
<td>The location of the checksum table in the boot image. (word offset)</td>
</tr>
<tr>
<td>0x30</td>
<td>Image Header Word Offset</td>
<td>The location of the corresponding image header in the boot image.</td>
</tr>
<tr>
<td>0x34</td>
<td>Partition Number/ID</td>
<td>Partition ID.</td>
</tr>
<tr>
<td>0x3C</td>
<td>Header Checksum</td>
<td>A sum of the previous words in the Partition Header.</td>
</tr>
</tbody>
</table>

### Zynq UltraScale+ MPSoC Partition Attribute Bits

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

### Table 17: Zynq® UltraScale+™ MPSoC Device Partition Attribute Bits

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Vector Location</td>
<td>Location of exception vector.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: LOVEC (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: HIVEC</td>
</tr>
<tr>
<td>22:20</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Early Handoff</td>
<td>Handoff immediately after loading:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: No Early Handoff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Early Handoff Enabled</td>
</tr>
<tr>
<td>18</td>
<td>Endianness</td>
<td>0: Little Endian</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Big Endian</td>
</tr>
<tr>
<td>17:16</td>
<td>Partition Owner</td>
<td>0: FSBL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: U-Boot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 and 3: Reserved</td>
</tr>
<tr>
<td>15</td>
<td>RSA Authentication Certificate present</td>
<td>0: No RSA Authentication Certificate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: RSA Authentication Certificate</td>
</tr>
<tr>
<td>14:12</td>
<td>Checksum Type</td>
<td>0: None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-2: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: SHA3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-7: Reserved</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>11:8</td>
<td>Destination CPU</td>
<td>0: None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: A53-0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: A53-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: A53-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4: A53-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5: R5-0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6: R5-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 R5-lockstep</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8: PMU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9-15: Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Encryption Present</td>
<td>0: Not Encrypted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Encrypted</td>
</tr>
<tr>
<td>6:4</td>
<td>Destination Device</td>
<td>0: None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: PS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: PL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3-15: Reserved</td>
</tr>
<tr>
<td>3</td>
<td>ASX Exec State</td>
<td>0: AARCH64 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: AARCH32</td>
</tr>
<tr>
<td>2:1</td>
<td>Exception Level</td>
<td>0: EL0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: EL3</td>
</tr>
<tr>
<td>0</td>
<td>Trustzone</td>
<td>0: Non-secure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Secure</td>
</tr>
</tbody>
</table>

**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 18: Zynq UltraScale+ MPSoC Device Authentication Certificates

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

Note: FSBL Signature is calculated as follows:

\[
\text{FSBL Signature} = \text{RSA-4096 (SSK, Padding \ || \ SHA-384 (PMUFW \ || \ FSBL \ || \ Padding \ || \ Authentication Header \ || \ UDF \ || \ PPK \ || \ SPK \ || \ SPK Signature)}
\]

Zynq UltraScale+ MPSoC Authentication Certification Header

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

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:20</td>
<td>Reserved</td>
<td>0</td>
</tr>
<tr>
<td>19:18</td>
<td>SPK/User eFuse Select</td>
<td>01: SPK eFuse&lt;br&gt; 10: User eFuse</td>
</tr>
<tr>
<td>17:16</td>
<td>PPK Key Select</td>
<td>0: PPK0&lt;br&gt; 1: PPK1</td>
</tr>
<tr>
<td>15:14</td>
<td>Authentication Certificate Format</td>
<td>00: PKCS #1 v1.5</td>
</tr>
<tr>
<td>13:12</td>
<td>Authentication Certificate Version</td>
<td>00: Current AC</td>
</tr>
<tr>
<td>11</td>
<td>PPK Key Type</td>
<td>0: Hash Key</td>
</tr>
<tr>
<td>10:9</td>
<td>PPK Key Source</td>
<td>0: eFUSE</td>
</tr>
<tr>
<td>8</td>
<td>SPK Enable</td>
<td>1: SPK Enable</td>
</tr>
</tbody>
</table>
**Zynq UltraScale+ MPSoC Secure Header**

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

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:4</td>
<td>Public Strength</td>
<td>0 : 2048b, 1 : 4096, 2:3 : Reserved</td>
</tr>
<tr>
<td>3:2</td>
<td>Hash Algorithm</td>
<td>1: SHA3/384, 2:3 Reserved</td>
</tr>
<tr>
<td>1:0</td>
<td>Public Algorithm</td>
<td>0: Reserved, 1: RSA, 2: Reserved, 3: Reserved</td>
</tr>
</tbody>
</table>

**AES**

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

**AES with Key rolling**

<table>
<thead>
<tr>
<th></th>
<th>Partition#0 (FSBL)</th>
<th></th>
<th>Partition#1</th>
<th></th>
<th>Partition#2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypted Using</td>
<td>Contents</td>
<td>Encrypted Using</td>
<td>Contents</td>
<td>Encrypted Using</td>
<td>Contents</td>
</tr>
<tr>
<td>Secure Header</td>
<td>Key0</td>
<td>IV0</td>
<td>-</td>
<td>IV1</td>
<td>Key0</td>
<td>IV0+0x01</td>
</tr>
<tr>
<td>Block #0</td>
<td>Key0</td>
<td>IV1</td>
<td>Key 2</td>
<td>IV2</td>
<td>Key1</td>
<td>IV1</td>
</tr>
<tr>
<td>Block #1</td>
<td>Key2</td>
<td>IV2</td>
<td>Key 3</td>
<td>IV3</td>
<td>Key2</td>
<td>IV2</td>
</tr>
<tr>
<td>Block #2</td>
<td>Key3</td>
<td>IV3</td>
<td>Key 4</td>
<td>IV4</td>
<td>Key3</td>
<td>IV3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Chapter 5: Bootgen Tool
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 4: Zynq UltraScale+ MPSoC Device Boot Image Block Diagram

- Boot Header 0x000-0xEC0
- Image Header Table
- Image Headers (IH1-IHn)
- Partition Header 1
- Partition Header n
- Header AC
- BootLoader (FSBL and PMUFW (opt))
- BootLoader AC
- Partition 1
- Partition 1 AC
- Partition(n)
- Partition(n) AC
- Header AC
- AC Header
- SPK ID Header
- User Defined Field
- PPK
- SPK Header
- SPK Signature
- BH Signature
- Partition Signature
- BootLoader AC
- AC Header
- SPK ID BootLoader
- User Defined Field
- PPK
- SPK BootLoader
- SPK Signature
- BH Signature
- Partition Signature
- Partition 1 AC
- AC Header
- SPK ID Partition 1
- User Defined Field
- PPK
- SPK Partition 1
- SPK Signature
- BH Signature
- Partition Signature
- Partition n AC
- AC Header
- SPK ID Partition(n)
- User Defined Field
- PPK
- SPK Partition(n)
- SPK Signature
- BH Signature
- Partition Signature
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 or PLM Image
- 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.

- 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
BIF Syntax and Supported File Types

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

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

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

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

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

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

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

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

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

**Bootgen Supported Files**

The following table lists the Bootgen supported files.

*Table 19: Bootgen Supported Files*

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bin</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>.bit/.rbt</td>
<td>Bitstream</td>
<td>Strips the BIT file header.</td>
</tr>
<tr>
<td>.dtb</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>image.gz</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>.elf</td>
<td>Executable Linked File (ELF)</td>
<td>Symbols and headers removed.</td>
</tr>
<tr>
<td>.int</td>
<td>Register initialization file</td>
<td></td>
</tr>
<tr>
<td>.nky</td>
<td>AES key</td>
<td></td>
</tr>
<tr>
<td>.pk1/.pub/.pem</td>
<td>RSA key</td>
<td></td>
</tr>
<tr>
<td>.sig</td>
<td>Signature files</td>
<td>Signature files generated by bootgen or HSM.</td>
</tr>
</tbody>
</table>
**BIF Syntax with Sub-systems**

The following example shows the detailed manner in which you can write a BIF while grouping the partitions together to form sub-systems.

```xml
/* single line comments allowed */
new_bif: /* This is not an attribute, user defined name to identify BIF file */
{
    id = 0x5 /* PDI ID */
    image
    {
        name = pmc_subsys, id = 0x1c000001 /* Image name (max 16 characters) & Image ID */
        partition
        {
            id = 0x11, type = bootloader, /* Partition ID & Partition Type */
            file = /path/to/plm.elf
        }
    }
    partition
    {
        type = pmcdata, load = 0xf2000000.
        file = /path/to/pmc_cdo.bin
    }
}
image
{
    name = pl_subsys, id = 0x1c000009 /* Image name (max 16 characters) & Image ID */
    partition
    {
        id = 0x33 /* Partition ID */
        file = /path/to/system.cfi
    }
    partition
    {
        id = 0x44 /* Partition ID */
        file = /path/to/system.npi
    }
}
image
{
    name = apu_subsys, id = 0x1c000003 /* Image name (max 16 characters) & Image ID */
    partition
    {
        id = 0x55, type = cdo, /* Partition ID */
        file = /path/to/apu_cdo.bin
    }
    partition
    {
```
The following example shows how you can write a BIF in a concise manner by grouping the partitions together to form sub-systems.

```plaintext
new_bif_short:
{
  id = 0x5
  
  image
  {
    name = pmc_subsys, id = 0x1c000001  // Image name (max 16 characters) & Image ID
    [ id = 0x11, type = bootloader, file = /path/to/plm.elf ]
    [ type = pmcdata, load = 0xf2000000, file = /path/to/pmc_cdo.bin ]
  }

  image
  {
    name = pl_subsys, id = 0x1c000009  // Image name (max 16 characters) & Image ID
    [ id = 0x33, file = /path/to/system.cfi ]
    [ id = 0x44, file = /path/to/system.npi ]
  }

  image
  {
    name = apu_subsys, id = 0x1c000003  // Image name (max 16 characters) & Image ID
    [ id = 0x55, type = cdo, file = /path/to/ps_cdo.bin ]
    [ id = 0x66, core = a72-0, file = /path/to/a72.elf ]
  }
}
```

**Attributes**

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

- **zynq**: Zynq-7000 SoC device
- **zynqmp**: Zynq® UltraScale+™ MPSoC device
- **fpga**: Any 7 series and above devices
### Table 20: Bootgen Attributes and Description

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aeskeyfile</strong> &lt;aes_key_filepath&gt;</td>
<td>The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file needs to written to eFUSE or 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.</td>
<td>All</td>
</tr>
<tr>
<td><strong>alignment</strong> &lt;byte&gt;</td>
<td>Sets the byte alignment. The partition will be padded to be aligned to a multiple of this value. This attribute cannot be used with offset.</td>
<td>zynq, zynqmp</td>
</tr>
</tbody>
</table>
| **auth_params** <options> | Extra options for authentication:  
- **ppk_select**: 0=1, 1=2 of two PPKs supported.  
- **spk_id**: 32-bit ID to differentiate SPKs.  
- **spk_select**: To differentiate spk and user efuses. Default will be spk-efuse.  
- **header_auth**: To authenticate headers when no partition is authenticated. | zynqmp |
| **authentication** <option> | Specifies the partition to be authenticated.  
- Authentication for Zynq is done using RSA-2048.  
- Authentication for Zynq UltraScale+ MPSoCs is done using RSA-4096.  
The arguments are:  
- **none**: Partition not encrypted.  
- **rsa**: Partition encrypted using RSA algorithm. | All |
| **bh_keyfile** <filename> | 256-bit obfuscated key or black key to be stored in the Boot Header. This is only valid when [keysrc_encryption]=bh_gry_key or [keysrc_encryption]=bh_blk_key. | zynqmp |
| **bh_key_iv** <filename> | Initialization vector used when decrypting the obfuscated key or a black key. | zynqmp |
| **bhsignature** <filename> | Imports Boot Header signature into authentication certificate. This can be used if you do not want to share the secret key PSK. You can create a signature and provide it to Bootgen. The file format is boohader.sba384.sig. | zynqmp |
| **blocks** <block sizes> | Specify block sizes for key-rolling feature in Encryption. Each module is encrypted using its own unique key. The initial key is stored at the key source on the device, while keys for each successive blocks are encrypted (wrapped) in the previous module. | zynqmp |
## Table 20: Bootgen Attributes and Description (cont’d)

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot_device &lt;options&gt;</td>
<td>Specifies the secondary boot device. Indicates the device on which the</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>partition is present. Options are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qspi32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qspi24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd-Is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mmc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>usb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ethernet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pcie</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sata</td>
<td></td>
</tr>
<tr>
<td>bootimage &lt;filename.bin&gt;</td>
<td>Specifies that the listed input file is a boot image that was created by</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>Bootgen.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>bootloader &lt;partition&gt;</td>
<td>Specifies the partition is a bootloader (FSBL). This attribute is specified</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>along with other partition BIF attributes.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>bootvector &lt;vector_values&gt;</td>
<td>Specifies the vector table for execute in place (XIP).</td>
<td></td>
</tr>
<tr>
<td>checksum &lt;options&gt;</td>
<td>Specifies that the partition needs to be checksummed. This option is not</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>supported along with more secure features like authentication and encryption.</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>Checksum algorithms are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>md5: for Zynq®-7000 SoC devices.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For Zynq® UltraScale+™ MPSoC, options are none: No checksum operation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sha3: Sha3 checksum.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zynq devices do not support checksum for bootloaders.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zynq UltraScale+ MPSoC devices do support checksum operation for bootloaders.</td>
<td></td>
</tr>
<tr>
<td>destination_cpu &lt;device_core&gt;</td>
<td>Specifies the core on which the partition needs to be executed.</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>a53-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a53-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a53-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a53-3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r5-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r5-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r5-lockstep</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pmu</td>
<td></td>
</tr>
<tr>
<td>destination_device &lt;device_type&gt;</td>
<td>This specifies if the partition is targeted for PS or PL. The options are:</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>ps: the partition is targeted for PS (default).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pl: the partition is targeted for PL, for bitstreams.</td>
<td></td>
</tr>
<tr>
<td>Option/Attribute</td>
<td>Description</td>
<td>Used By</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>early_handoff</td>
<td>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.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>encryption &lt;option&gt;</td>
<td>Specifies the partition to be encrypted. Encryption algorithms are: zynq uses AES-CBC, and zynqmp uses AES-GCM. The partition options are:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>none: Partition not encrypted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aes: Partition encrypted using AES algorithm.</td>
<td></td>
</tr>
<tr>
<td>exception_level&lt;options&gt;</td>
<td>Exception level for which the core should be configured. Options are:</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>el-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>el-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>el-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>el-3</td>
<td></td>
</tr>
<tr>
<td>familykey</td>
<td>Specifies the family key.</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>fpga</td>
<td></td>
</tr>
<tr>
<td>fsbl_config</td>
<td>Specifies the sub-attributes used to configure the bootimage. Those sub-attributes are:</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>bh_auth_enable: RSA authentication of the boot image is done excluding the verification of PPK hash and SPK ID.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>auth_only: boot image is only RSA signed. FSBL should not be decrypted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>opt_key: Operational key is used for block-0 decryption. Secure Header has the opt key.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pfhd_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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>puf4kmode: PUF is tuned to use in 4k bit configuration. (Default is 12k bit). shutter = &lt;value&gt;32 bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time.</td>
<td></td>
</tr>
<tr>
<td>headersignature&lt;signature_file&gt;</td>
<td>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.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>zynqmp</td>
<td></td>
</tr>
<tr>
<td>hivec</td>
<td>Specifies the location of exception vector table as hivec (Hi-Vector). The default value is lovec (Low-Vector). This is applicable with A53 (32 bit) and R5 cores only.</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>hivec: exception vector table at 0xFFFF0000. lovec: exception vector table at 0x00000000.</td>
<td></td>
</tr>
<tr>
<td>init &lt;filename&gt;</td>
<td>Register initialization block at the end of the Bootloader, built by parsing the init (.int) file specification. A maximum of 256 address-value init pairs are allowed. The init files have a specific format.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>zynqmp</td>
<td></td>
</tr>
</tbody>
</table>
### Table 20: Bootgen Attributes and Description (cont’d)

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
</table>
| **keysr src_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   |
| `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   |
<p>| <code>pid &lt;ID&gt;</code> | Specifies the Partition ID. PID can be a 32-bit value (0 to 0xFFFFFFFF).                                                                                                                                       | zynqmp        |
| <code>pmufw_image &lt;image_name&gt;</code> | PMU firmware image to be loaded by BootROM, before loading the FSBL.                                                                                                                                       | zynqmp        |
| <code>ppkfile &lt;key filename&gt;</code> | Primary Public Key (PPK). Used to authenticate partitions in the boot image. See <a href="#">Using Authentication</a> for more information.                                                                           | zynq zynqmp   |
| <code>presign &lt;sig_filename&gt;</code> | Partition signature (.sig) file.                                                                                                                                                                           | All           |
| <code>pskfile &lt;key filename&gt;</code> | Primary Secret Key (PSK). Used to authenticate partitions in the boot image. See the <a href="#">Using Authentication</a> for more information.                                                                        | zynq zynqmp   |
| <code>puf_file &lt;filename&gt;</code> | PUF helper data file. PUF is used with black key as encryption key source. PUF helper data is of 1544 bytes.1536 bytes of PUF HD + 4 bytes of HASH + 3 bytes of AUX + 1 byte alignment. | zynqmp        |
| <code>reserve</code> | Reserves the memory, which is padded after the partition.                                                                                                                                                 | zynq zynqmp   |
| <code>spkfile &lt;filename&gt;</code> | Keys used to authenticate partitions in the boot image. See <a href="#">Using Authentication</a> for more information. SPK - Secondary Public Key                                                                          | All           |</p>
<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>split &lt;options&gt;</strong></td>
<td>Splits the image into parts, based on the mode. Split options are:</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>slaveboot: Supported for zynqmp only. Splits as follows:</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>Boot Header + Bootloader</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Image and Partition Headers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rest of the partitions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>normal: Supported for both zynq and zynqmp. Splits as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bootheader + Image Headers + Partition Headers + Bootloader</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Partition1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Partition2 and so on</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Along with the split mode, output format can also be specified as <code>bin</code> or</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>mcs</code>.</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>The option split mode normal is same as the command line option</td>
<td></td>
</tr>
<tr>
<td></td>
<td>split. This command line option is deprecated.</td>
<td></td>
</tr>
<tr>
<td><strong>spk_select &lt;SPK_ID&gt;</strong></td>
<td>Specify an SPK ID in user eFUSE.</td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>spksignature &lt;signature_file&gt;</strong></td>
<td>Imports the SPK signature into an Authentication Certificate. See Using Authentication This can be used in case the user does not want to share the secret key PSK. The user can create a signature and provide it to Bootgen.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>sskfile &lt;key filename&gt;</strong></td>
<td>Secondary Secret Key (SSK) key authenticates partitions in the Boot Image. The primary keys authenticate the secondary keys; the secondary keys authenticate the partitions.</td>
<td>All</td>
</tr>
<tr>
<td><strong>startup=&lt;address&gt;</strong></td>
<td>Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>trustzone= &lt;option&gt;</strong></td>
<td>The trustzone options are:</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>secure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nonsecure</td>
<td></td>
</tr>
<tr>
<td><strong>udf_bh &lt;data_file&gt;</strong></td>
<td>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.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>udf_data &lt;data_file&gt;</strong></td>
<td>Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>xip_mode</strong></td>
<td>Indicates eXecute In Place (XIP) for FSBL to be executed directly from QSPI flash.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>big_endian</strong></td>
<td>To specify the binary file is in big endian format</td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>aarch32_mode</strong></td>
<td>To specify the binary file that is to be executed in 32-bit mode.</td>
<td>zynqmp</td>
</tr>
</tbody>
</table>
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 21: Bootgen Command and Descriptions

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description and Options</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch &lt;type&gt;</td>
<td>Xilinx® device architecture: Options:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>zynq (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>zynqmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fpga</td>
<td></td>
</tr>
<tr>
<td>bif_help</td>
<td>Prints out the BIF help summary.</td>
<td>All</td>
</tr>
<tr>
<td>dual_qspi_mode &lt;configuration&gt;</td>
<td>Generates two output files for dual QSPI configurations:</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>parallel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stacked &lt;size&gt;</td>
<td>zynqmp</td>
</tr>
<tr>
<td>efuseppkbits &lt;PPK_filename&gt;</td>
<td>Generates a PPK hash for eFUSE.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>encrypt &lt;options&gt;</td>
<td>AES Key storage in device. Options are:</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>bbram (default)</td>
<td>fpga</td>
</tr>
<tr>
<td></td>
<td>efuse</td>
<td></td>
</tr>
<tr>
<td>encryption_dump</td>
<td>Generates encryption log file, aes_log.txt.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>fill &lt;hex_byte&gt;</td>
<td>Specifies the fill byte to use for padding.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>generate_hashes</td>
<td>Output SHA2/SHA3 hash files with padding in PKCS#1v1.5 format.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>generate_keys &lt;key_type&gt;</td>
<td>Generate the authentication keys. Options are:</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>pem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rsa</td>
<td></td>
</tr>
<tr>
<td></td>
<td>obfuscatedkey</td>
<td></td>
</tr>
<tr>
<td>h, help</td>
<td>Prints out help summary.</td>
<td>All</td>
</tr>
<tr>
<td>image &lt;filename(.bif)&gt;</td>
<td>Provides a boot image format (.bif) file name.</td>
<td>All</td>
</tr>
<tr>
<td>log&lt;level_type&gt;</td>
<td>Generates a log file at the current working directory with following message types:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>warning (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td></td>
</tr>
<tr>
<td></td>
<td>debug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td></td>
</tr>
<tr>
<td>Commands</td>
<td>Description and Options</td>
<td>Used by</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>nonbooting</td>
<td>Create an intermediate boot image.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>o &lt;filename&gt;</td>
<td>Specifies the output file. The format of the file is determined by the filename extension. Valid extensions are:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>.bin (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.mcs</td>
<td></td>
</tr>
<tr>
<td>p &lt;partname&gt;</td>
<td>Specify the part name used in generating the encryption key.</td>
<td>All</td>
</tr>
<tr>
<td>padimageheader &lt;option&gt;</td>
<td>Pads the image headers to force alignment of following partitions. Options are:</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>1 (default)</td>
<td></td>
</tr>
<tr>
<td>process_bitstream &lt;option&gt;</td>
<td>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.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>read &lt;options&gt;</td>
<td>Used to read boot headers, image headers, and partition headers based on the options.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>bh: To read boot header from PDI in human readable form</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>iht: To read image header table from PDI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ih: To read image headers from PDI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pht: To read partition headers from PDI</td>
<td></td>
</tr>
<tr>
<td>split &lt;options&gt;</td>
<td>Splits the boot image into partitions and outputs the files as .bin or .mcs. *Bootheader + Image Headers + Partition Headers + Fsb1.elf *Partition1.bit *Partition2.elf</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>spksignature &lt;filename&gt;</td>
<td>Generates an SPK signature file.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>verify</td>
<td>This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image will be verified against the available partitions.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zynqmp</td>
</tr>
<tr>
<td>verify_kdf</td>
<td>This option is used to validate the Counter Mode KDF used in bootgen for generation AES keys.</td>
<td>zynqmp</td>
</tr>
</tbody>
</table>
Table 21: Bootgen Command and Descriptions (cont’d)

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description and Options</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>w &lt;option&gt;</td>
<td>Specifies whether to overwrite the output files:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>on (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>off</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The <code>-w</code> without an option is interpreted as <code>-w on</code>.</td>
<td></td>
</tr>
<tr>
<td>zynqmpes1</td>
<td>Generates a boot image for ES1 (1.0). The default padding scheme is ES2 (2.0).</td>
<td>zynqmp</td>
</tr>
</tbody>
</table>

Boot Time Security

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

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

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

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

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

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

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

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

**Zynq UltraScale+ MPSoC Device Security**

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

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

**Using Encryption**

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

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

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

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

**Encryption Process**

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

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

**Figure 6: Decryption Process Diagram**
**Encrypting Zynq-7000 Device Partitions**

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

**Example BIF File**

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

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

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

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

**Key Generation**

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

*Figure 7: Sample Key File*

```
Device      xc7z020clg484;
Key 0        f878b838d8589818e868a828c8488808
Key StartCBC 5C9D95ECBFE8A1F12A8EB312362C596
Key HMAC     000011122223334444555566667777
```

**Encrypting Zynq MPSoC Device Partitions**

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

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

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

```plaintext
image:
[  
    [fsbl_config] opt_key 
    [keysrc_encryption] bbram_red_key 
    [bootloader,  
     destination_cpu = a53-0,  
     encryption = aes,  
     aeskeyfile = aes_p1.nky|fsbl.elf

    [destination_cpu = a53-3, 
     encryption = aes,  
     aeskeyfile = aes_p2.nky|hello.elf

    ]
]
```

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

**Figure 8: Operational Key**

```
Device         xczu9eg;
Key 0          9C42D9B74B633132F57C381D5CA4C7DF0829382CDBC455CDA08ECA62EB11D19D;
IV 0           42D3818AC135A365E5BD316;
Key Opt        36AD83215CA7299F80E4F3A85AC9ACD27D1F50773E24B59067B4BA75A3A62;
```

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

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

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

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

```plaintext
image:
[
  [keys_encryption] bbram_red_key
  
  [boiloder,
   destination_cpu = a53-0,
   encryption = aes,
   aeskeyfile = aes_p1.nky,
   blocks = 1024(2);2048;4096(2);8192(2);4096;2048;1024
  ]
  fsbl.elf
  
  [destination_cpu = a53-3,
   encryption = aes,
   aeskeyfile = aes_p2.nky,
   blocks = 4096(1);1024
  ]
  hello.elf
]
```

**Note:**

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

**Gray/Obfuscated Keys**

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

```plaintext
image:
[
  [keysrsrc_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.
Obfuscated Key Generation

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

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

The command to generate the Obfuscated key is:

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

Black/PUF Keys

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

```plaintext
image:
[
    [puf_file] pufdata.txt
    [bh_key_iv] black_iv.txt
    [bh_keyfile] black_key.txt
    [fsbl_config] puf4kmode, shutter=0x0100005E, pufhd_bh
    [keysrsrc_encryption] bh_blk_key
    [
        bootloader,
        destination_cpu = a53-0,
        encryption = aes,
        aeskeyfile = aes_pl.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:

```plaintext
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 an authentication certificate in two ways:

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

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

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

For device-specific Authentication information, see the following:

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

## Signing

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

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

**Table 22: Supported File Formats for Authentication Keys**

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Description</th>
<th>Supported File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPK</td>
<td>Primary Public Key</td>
<td>This key is used to authenticate a partition. It should always be specified when authenticating a partition.</td>
<td>*.txt *.pem *.pub *.pk1</td>
</tr>
<tr>
<td>PSK</td>
<td>Primary Secret Key</td>
<td>This key is used to authenticate a partition. It should always be specified when authenticating a partition.</td>
<td>*.txt *.pem *.pk1</td>
</tr>
<tr>
<td>SPK</td>
<td>Secondary Public Key</td>
<td>This key, when specified, is used to authenticate a partition.</td>
<td>*.txt *.pem *.pub *.pk1</td>
</tr>
</tbody>
</table>
Table 22: Supported File Formats for Authentication Keys (cont'd)

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

**Verifying**

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

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

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

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

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

**Zynq UltraScale+ MPSoC Authentication Support**

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

**NIST SHA-3 Support**

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

The generated signature uses the Keccak-SHA3 or NIST-SHA3 based on following table:
<table>
<thead>
<tr>
<th>Which Authentication Certificate (AC)?</th>
<th>Signature</th>
<th>SHA Algorithm and SPK eFUSE</th>
<th>Secret Key used for Signature Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header AC (loader by FSBL/FW)</td>
<td>SPK Signature</td>
<td>If SPKID eFUSEs, then Keecak; If User eFUSE, then NIST</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK\text{header}</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always Nist</td>
<td>SSK\text{header}</td>
</tr>
<tr>
<td>BootLoader AC (loaded by ROM)</td>
<td>SPK Signature</td>
<td>Always Keecak; Always SPKID eFUSE for SPK</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK\text{Bootloader}</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always Keecak</td>
<td>SSK\text{Bootloader}</td>
</tr>
<tr>
<td>Partition AC (loaded by FSBL FW)</td>
<td>SPK Signature</td>
<td>If SPKID eFUSEs then Keecak; If User eFUSE then NIST</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK\text{Partition}</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always NIST</td>
<td>SSK\text{Partition}</td>
</tr>
</tbody>
</table>

**Examples**

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

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

```plaintext
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.
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.

```plaintext
the_ROM_image:
[
  [auth_params]ppk_select = 0
  [pskfile]psk.pem
  [sskfile]ssk1.pem
  [bootloader, authunciation = 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 13: Sample RSA Private Key File

BIF Example

A sample BIF file, `generate_pem.bif`:

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

The command to generate keys is, as follows:

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

**PPK Hash for eFUSE**

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

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

**BIF File Example**

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

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

Command

The command to generate PPK hash for eFUSE programming is:

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

**Using HSM Mode**

In current cryptography, all the algorithms are public, so it becomes critical to protect the private/secret key. The hardware security module (HSM) is a dedicated crypto-processing device that is specifically designed for the protection of the crypto key lifecycle, and increases key handling security, because only public keys are passed to the Bootgen and not the private/secure keys. A *Standard* mode is also available; this mode does not require passing keys.

In some organizations, an Infosec staff is responsible for the production release of a secure embedded product. The Infosec staff might use a HSM for digital signatures and a separate secure server for encryption. The HSM and secure server typically reside in a secure area. The HSM is a secure key/signature generation device which generates private keys, signs the partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys reside in the HSM only.
Bootgen in HSM mode uses only RSA public keys and the signatures that were created by the HSM to generate the boot image. The HSM accepts hash values of partitions generated by Bootgen and returns a signature block, based on the hash and the secret RSA key.

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

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

Using Advanced Key Management Options

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

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

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

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

To create a boot image using HSM mode, it is similar to a boot image created 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**

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

```
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**

Trusted individual distributes the SPK Signature to the development teams.

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

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

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

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

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

Stage2:

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

The Bootgen command is:

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

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

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

Stage 0
- Generate SPK Hash
- SPK hash

Stage 2a
- Encrypt FSBL
- nky Key

Stage 2b
- Generate SPK signature
- Insert FSBL Signature
- PPK & SPK

Stage 3a
- Generate FSBL Hash
- FSBL hash

Stage 3b
- Generate FSBL Hash

Stage 4a
- Generate FSBL Signature
- SSK

Stage 4b
- Generate Header Table Signature
- Header Table hash

Stage 5a
- PPK & SPK
- Generate FSBL Signature
- SPK signature

Stage 5b
- Generate Header Table Signature
- Header Table hash

Stage 6
- Generate Header Table signature
- Header Table hash

Stage 7
- Generate Header Table signature
- Header Table hash

Stage 8
- Generate Boot Image
- Final Boot Image

Partitions with Authenticated Certificates
- Generate Header Table Signatures
- Header Table hash
To create a boot image using HSM mode for a Zynq®-7000 SoC 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.

```text
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**

The following is the snippet from the BIF file. The following is the snippet from the BIF file.

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

The following is the Bootgen command:

```text
bootgen.exe -image stage0.bif -w on -generate_hashes
```

**Stage 1: Sign the SPK Hash**

Encrypt the partitions.

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

The output of this command is: `secondary.pub.sha256.sig`

**Stage 2: Encrypt using AES**

Encrypt using the following snippet in the BIF file.

```text
Stage 2:
  {
    [aeskeyfile]my_efuse.nky
    [bootloader,encryption=aes]zynq_fsbl_0.elf
  }
```

The Bootgen command is:

```text
bootgen -image stage2a.bif -w -o fsbl_e.bin -encrypt efuse
```

The output is the following encrypted file: `fsbl_e.bin`. 
Stage 3: Generate Partition Hashes

Stage 3a: Generate the boot header hash using the following BIF file:

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

The Bootgen command is:

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

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

Stage 3b: Generate the boot header hash using the following BIF file:

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

The Bootgen command is:

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

The output is the following hash file: `system.bit.0.sha256`

Stage 4: Sign the Hashes

Stage 4a: Sign the FSBL partition hash using the following BIF file:

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

The output is the following hash file: `system.bit.0.sha256.sig`

Stage 4b: Sign the bitstream hash using the following BIF file:

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

The output is the following signature file: `system.bit.0.sha256.sig`. 
Stage 5: Insert Partition Signatures into Authentication Certificates

Stage 5a: Insert the FSBL signature with the following BIF file:

```
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:

```
bootgen.exe -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 with the following BIF file:

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

The Bootgen command is:

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

The authenticated output file is `system_e_ac.bin`.

Stage 6: Generate Header Table Hash

The BIF file and the Bootgen command look like the following:

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

The output hash file is: `ImageHeaderTable.sha256`.

Stage 7: Generate Header Table Signature

Generate the header table signature using the following command:

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

The output hash file is: `ImageHeaderTable.sha256.sig`.
Stage 8: Combine Partitions, Insert Header Table Signature

Enter the following in a BIF file:

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

The Bootgen command is:

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

The output file is the final boot image: `final.bin`.

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

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

- **Stage 0**: Generate SPK Hash
- **Stage 1**: Generate SPK Signature
- **Stage 2a**: Encrypt FSBL
  - **Stage 2b**: Generate SPK Hash
  - **Stage 2c**: Generate BH Signature
- **Stage 3**: Generate BH signature
- **Stage 4**: Generate partition signatures
- **Stage 5**: Generate partition hashes
- **Stage 6**: Generate partition signatures
- **Stage 7a**: PPK & SPK Encrypt FSBL with auth certificate
- **Stage 7b**: Generate Header Table hash
- **Stage 7c**: Generate Header Table signature
- **Stage 8**: Generate Header Table hash
- **Stage 9**: Generate Header Table signature
- **Stage 10**: Generate Header Table signature

Insert FSBL Signature

Final bootimage
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.

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

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

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

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

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

```

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

The following is the snippet from the BIF file.

```yaml
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:

```bash
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.

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

The bootgen command is:

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

Stage 2b: Encrypt Bitstream

Generate the following BIF file entry:

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

The Bootgen command is:

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

Generate the boot header hash using the following BIF file:

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

The Bootgen command is:

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

Stage 4: Sign Boot Header Hash

Generate the boot header hash with the following OpenSSL command:

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

Stage 5: Get Partition Hashes

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

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

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

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

The Bootgen command is:

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

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

**Stage 6: Sign Partition Hashes**

Create the following files using OpenSSL:

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

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

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

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

The Bootgen command is as follows:

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

```bash
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:

```bash
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:

```bash
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 stage7.d.bif -o bl31_ac.bin -nonbooting -w on -log error
```

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

Enter the following in a BIF file:

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

The Bootgen command is:

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

### Stage 9: Sign Header Table Hash

Generate the following files using OpenSSL:

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

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

Enter the following in a BIF file:

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

The Bootgen command is:

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

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

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

Encryption and Authentication

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

Encryption Example

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

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

The BIF file looks like the following:

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

Authentication Example

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

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

The BIF file is as follows:

```
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 17: AES Key Sample](image)

HSM Mode

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

Standard Mode

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

```
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:
**Stage 0: Authenticate with dummy key**

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

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

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

**Stage 1: Generate hashes**

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

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

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

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

**Stage 3: Update the RSA certificate with Actual Signature**

The Stage3.bif is as follows:

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

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

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

### Use Cases and Examples

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

Simple Application Boot on Different Cores

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

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

PMUFW Load by BootROM

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

the-ROM_image:
{
    [pmufw_image] pmu_fw.elf
    [bootloader, destination_cpu=a53-0] fsbl_a53.elf
    [destination_cpu=r5-0] app_r5.elf
}

PMUFW Load by FSBL

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

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 \texttt{efuse\_gry\_key} stored in eFUSE.

\begin{verbatim}
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
  ]
\end{verbatim}

**Encryption Flow: Gray Key stored in Boot Header**

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

\begin{verbatim}
the\_ROM\_image:
  [
    [keysrc\_encryption] bh\_gry_key
    [bh\_keyfile] bhkey.txt
    [bh\_key\_iv] bh\_key\_iv.txt
    
    [bootloader,
     encryption=aes,
     aeskeyfile=aes0.nky,
     destination\_cpu=a53-0
    ] ZynqMP_Fsbl.elf
    
    [destination\_cpu=a53-0,
     encryption=aes,
     aeskeyfile=aes1.nky
    ] App_A53_0.elf
  ]
\end{verbatim}
**Operational Key**

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

```
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:

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

Example aes.nky for stage1:

```
Device xc7z020clg484;
Key 0 AD00C023E2384C9039EA984D49A8C819456A98C124AE890ACEF002100128932;
IV 0 F7F8FDE08674A28DC6ED8E37;
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBADF457D613508F;
```

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

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

Example of stage2.bif:

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

Example aes-opt.nky for stage2:

```
Device xc7z020clg484;
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBADF457D613508F;
```

**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 F7F8FDE08674A28D6ED8E37;
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
}
```

**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 F7F8FDE08674A28D6ED8E37;
3. Use Bootgen to stitch the above example to form a complete image:

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

**Example of stage3.bif:**

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

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

### Single Partition Image

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

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

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

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

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

### Only Authentication Use Case

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

1. Create a single partition image that is authenticated at U-Boot.

   *Note: If you provide an elf file, it should not contain multiple loadable sections. If your elf file contains multiple loadable sections, you should convert the input to the .bin format and provide the .bin as input in bif. An example bif is as follows:*

<table>
<thead>
<tr>
<th>the_ROM_image:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>[pskfile]rsa4096_private1.pem</td>
</tr>
<tr>
<td>[sskfile]rsa4096_private2.pem</td>
</tr>
<tr>
<td>[auth_params] ppk_select=1;spk_id=0x12345678</td>
</tr>
<tr>
<td>[authentication = rsa]Data.bin</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

2. Once 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:
{
    [bootherloader,
        destination_cpu=a53-0,
        xip_mode
    ] mpsoc_qspi_xip.elf
}

See xip_mode for more information about the command.

BIF Attribute Reference

aarch32_mode

Syntax

[aarch32_mode] <partition>

Description

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

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

Arguments

Specified partition.
Example

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

aeskeyfile

**Syntax**

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

**Description**

The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file must be written to eFUSE or BBRAM. If the key file is not present in the path specified, a new key is generated by Bootgen, which is used for encryption.

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

**Arguments**

Specified file name.

**Return Value**

None

**Zynq-7000 SoC Example**

The partitions `fsbl.elf` and `hello.elf` are encrypted using keys in `test.nky`.

```
all:
{
    [keysrc_encryption] bbram_red_key
    [aeskeyfile] test.nky
    [bootloader, encryption=aes] fsbl.elf
    [encryption=aes] hello.elf
}
```
Sample key (.nky) file - test.nky

<table>
<thead>
<tr>
<th>Device</th>
<th>xc7z020clg484;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 0</td>
<td>8177B12032A7DEEE35D0F71A7FC399027BF....D608C58;</td>
</tr>
<tr>
<td>Key StartCBC</td>
<td>952FD2DF1DA543C46CDDE4F8811506228;</td>
</tr>
<tr>
<td>Key HMAC</td>
<td>123177B12032A7DEEE35D0F71A7FC3990BF....127BD89;</td>
</tr>
</tbody>
</table>

**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

```yaml
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

```yaml
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

```yaml
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

```yaml
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
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
]

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
```

**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
```

**blocks**

**Syntax**

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

**Description**

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

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

Example

Sample BIF - test.bif
all:
    [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

```yaml
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**

```yaml
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**

```yaml
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

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

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

### bootloader

**Syntax**

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

**Description**

Identifies an ELF file as the FSBL.

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

**Arguments**

Specified file name.

**Example**

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

### bootvectors

**Syntax**

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

**Description**

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

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

checksum

Syntax

[checksum = <options>] <partition>

Description

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

Arguments

- none: No checksum operation.
- MD5: MD5 checksum operation for Zynq®-7000 SoC devices. In these devices, checksum operations are not supported for bootloaders.
- SHA3: Checksum operation for Zynq® UltraScale+™ MPSoC devices.

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

```bash
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

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

Syntax

[early_handoff] <partition>

Description

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

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

Example

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

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:

```plaintext
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:

```plaintext
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**

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

**Description**

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

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

Normal mode splits as follows:

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

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

- slaveboot
- normal
- bin
- mcs

Example

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

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

spkfile

Syntax

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

Description

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

Arguments

Specified file name.

Example

```all:
[
  [pskfile] primarykey.pem
  [spkfile] secondarykey.pem
  [sskfile] secondarykey.pem
  [bootloader, authentication=rsa] fsbl.elf
  [authentication=rsa] hello.elf
]
```
Note: The secret key file contains the public key component of the key. You need not specify public key (SPK) when the secret key (SSK) is mentioned.

spksignature

Code Example

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

Description

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

Example

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

trustzone

Syntax

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

Description

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

- secure
- nonsecure (default)
Example

```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

```
123456789abcdef85072696e636530300301440408706d616c6164000508
266431530102030405060708090a0b0c0d0e0f101112131415161718191a1b
1c1d1
```
**udf_data**

**Syntax**

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

**Description**

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

**Arguments**

Specified file name.

**Example**

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

**xip_mode**

**Syntax**

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

**Description**

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

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

**Arguments**

Specified partition.
Example

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

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

Command Reference

**arch**

**Syntax**

```plaintext
-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**

```plaintext
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>
efuseppkbhits

Syntax

```
bootgen -image test.bif -o boot.bin -efuseppkbhits 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.

eencrypt

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 BRAM.

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

```makefile
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 24: Zynq: SHA-2 (256-bytes)

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

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

Table 25: ZynqMP: SHA-3 (384-bytes)

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

Example

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

Bootgen generates the following hash files with the specified BIF:

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

generate_keys

Syntax

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

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

Note: For more information on generating encryption keys, see Key Generation.

Authentication Key Generation Example

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

Examples

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

Obfuscated Key Generation Example

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

Command:

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

The Sample BIF file is shown in the following example:

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

Arguments

- rsa
- pem
- obfuscated

image

Syntax

```
-image <BIF_filename>
```
Description

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

Arguments

bif_filename

Example

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

The Sample BIF file is shown in the following example:

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

log

Syntax

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

Description

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

Arguments

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

Syntax

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

Description

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

Example

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

O

Syntax

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

Description

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

Outputs

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

Example

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

The boot image is output in an MCS format.
Syntax

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

Description

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

padimageheader

Syntax

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

Description

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

Arguments

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

Image or Partition Header Lengths

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

process_bitstream

Syntax

-process_bitstream <bin|mcs>
Description

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

Arguments

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

Returns

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

read

Syntax

-read [options]

Description

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

Arguments

- bh: To read boot header from PDI in human readable form
- iht: To read image header table from PDI
- ih: To read image headers from PDI.
- pht: To read partition headers from PDI
- bootgen -arch zynqmp -read BOOT.bin

spkssignature

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.

\textbf{split}

\textbf{Syntax}

\begin{verbatim}
bootgen -arch zynq -image test.bif -split bin
\end{verbatim}

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

\textbf{Example}

\begin{verbatim}
the_ROM_image:
[
    [bootloader] Fsbl.elf
    Partition1.bit
    Partition2.elf
]
\end{verbatim}

\textbf{verify}

\textbf{Syntax}

\begin{verbatim}
bootgen -arch zynqmp -verify boot.bin
\end{verbatim}
Description

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

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

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

verify_kdf

Syntax

```
bootgen -arch zynqmp -verify_kdf testVec.txt
```

Description

The format of the `testVec.txt` file is as below.

```
L = 256
KI = d54b6fd94f7cf98fd955517f937e9927f9536caeb148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4af9526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acdbbdc4f6142d7b8515bad09ec7f7af
```

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
```

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.

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.
Chapter 5: Bootgen Tool

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 19: Example Output**
Chapter 6

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 guide 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/DSA) 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 26: Hardware Requirements*

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Speed</td>
<td>2.2 GHz minimum or higher; Hyper-threading (HHT) or multicore recommended.</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Pentium 4, Intel Core Duo, or Xeon Processors; SSE2 minimum</td>
</tr>
<tr>
<td>Memory/RAM</td>
<td>2 GB or higher</td>
</tr>
<tr>
<td>Display Resolution</td>
<td>1024×768 or higher at normal size (96 dpi)</td>
</tr>
<tr>
<td>Disk Space</td>
<td>Based on the components selected during the installation</td>
</tr>
</tbody>
</table>

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 27: Software Requirements*

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Supported Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>• Windows 7 SP1 (64-bit)</td>
</tr>
<tr>
<td></td>
<td>• Windows 8.1 (64-bit)</td>
</tr>
<tr>
<td></td>
<td>• Windows 10 Pro (64-bit)</td>
</tr>
</tbody>
</table>
Table 27: Software Requirements (cont’d)

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Supported Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td></td>
</tr>
<tr>
<td>• Red Hat Enterprise Linux:</td>
<td></td>
</tr>
<tr>
<td>○ 6.6-6.9 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>○ 7.0-7.1 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>• CentOS:</td>
<td></td>
</tr>
<tr>
<td>○ 6.7-6.8 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>○ 7.2-7.3 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>• SUSE Linux Enterprise:</td>
<td></td>
</tr>
<tr>
<td>○ 11.4 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>○ 12.2 (64-bit)</td>
<td></td>
</tr>
<tr>
<td>• Ubuntu Linux 16.04.2 LTS (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Additional library installation required.

---

**Installing and Launching XSCT**

The Xilinx® Software Command-Line Tool (XSCT) can be installed either as a part of the Vitis IDE installer or as a separate command-line tool only installation. XSCT is available for the following platforms:

- Microsoft Windows
- Linux

The following sections explain the installation process for each of these platforms.

### Installing and Launching XSCT on Windows

XSCT can be installed using the Windows executable installer. The installer executable bears the name `Xilinx_vitis_<version>_Win64.EXE`, where `<version>` indicates the Vitis IDE version number.

**Note:** Installing XSCT on Microsoft Windows operating system might require administrator rights. In addition, your project workspace needs to be set up in any folder that you can fully access.

1. To install XSCT, double-click the Windows installer executable file.
2. The installer accepts your login credentials and allows you to select specific tool components. The client then automatically downloads only what you have selected and installs it on your local machine.
3. In the Select Edition to Install window, select the **Xilinx Software Command-Line Tool (XSCT)** option to install XSCT as a separate command-line tool only. Alternatively, you can also select the **Vitis IDE** option to install XSCT as a part of the Vitis IDE, an Eclipse-based integrated development environment.

4. Unless you choose otherwise, XSCT is installed in the **C:\Xilinx** directory.

5. To launch XSCT on Windows, select **Start → Programs → Xilinx Design Tools → Vitis <version>** and then select **Vitis**. Where **Vitis <version>** indicates the Vitis version number.

6. You can also launch XSCT from the command line.

    ```
    cd C:\Xilinx\vitis\<version>\bin
    xsct.bat
    ```

7. To view the available command-line options, issue the `help` command at the XSCT command prompt.

    ```
    ***** Xilinx Software Commandline Tool (XSCT)
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
    xsct% help
    Available Help Categories
    breakpoints   - Target Breakpoints/Watchpoints.
    connections   - Target Connection Management.
    device        - Device Configuration System.
    download      - Target Download FPGA/BINARY.
    hsi           - HSI commands.
    jtag          - JTAG Access.
    memory        - Target Memory.
    miscellaneous - Miscellaneous.
    petalinux     - Petalinux commands.
    projects      - Vitis Projects.
    registers     - Target Registers.
    reset         - Target Reset.
    running       - Program Execution.
    streams       - Jtag UART.
    svf           - SVF Operations.
    tfile         - Target File System.
    Type "help" followed by above "category" for more
details or
the commands
    help" followed by the keyword "commands" to list all
    xsct%
    ```

**Installing and Launching XSCT on Linux**

Xilinx Software Command-line Tool (XSCT) can be installed using the small self-extracting web install executable binary distribution file. The installer file bears the name **Xilinx_vitis_<version>_Lin64.BIN**, where **<version>** indicates the Vitis IDE version number.
Note: The procedure for installing XSCT on Linux depends on which Linux distribution you are using. Ensure that the installation folder has the appropriate permissions. In addition, your project workspace needs to be set up in any folder that you can fully access.

1. To install XSCT, launch the terminal and change the permission of the self-extracting binary executable.

   ```
   $ chmod +x Xilinx_vitis_<version>_Lin64.BIN
   ```

2. Start the installation process or run the `.BIN` file.

   ```
   ./Xilinx_vitis_<version>_Lin64.BIN
   ```

3. The installer accepts your login credentials and allows you to select specific tool components. The client then automatically downloads only what you have selected and installs it on your local machine.

4. In the Select Edition to Install window, select the Xilinx Software Command-Line Tool (XSCT) option to install XSCT as a separate command-line tool only. Alternatively, you can also select the Vitis option to install XSCT as a part of Vitis, an Eclipse-based integrated development environment.

5. Unless you choose otherwise, XSCT is installed in the `/opt/Xilinx` directory.

6. To launch XSCT on Linux, select Applications → Other and then select Xilinx Software Command Line Tool `<version>`. Where `<version>` is the version number of the XSCT.

7. You can also launch XSCT from the command line.

   ```
   cd /opt/Xilinx/vitis/<version>/bin
   ./xsct
   ```

8. To view the available command-line options, issue the `help` command at the XSCT command prompt.

   ```
   xsct% help
   Available Help Categories
   breakpoints  - Target Breakpoints/Watchpoints.
   connections  - Target Connection Management.
   device       - Device Configuration System.
   download     - Target Download FPGA/BINARY.
   hsi          - HSI commands.
   jtag         - JTAG Access.
   memory       - Target Memory.
   miscellaneous - Miscellaneous.
   petalinux    - Petalinux commands.
   projects     - Vitis Projects.
   registers    - Target Registers.
   reset        - Target Reset.
   running      - Program Execution.
   streams      - Jtag UART.
   ```
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 hw_server/TCF agent.

**Syntax**

```
connect [options]
```

Allows users to connect to a server, list connections or switch between connections.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-host &lt;host name/ip&gt;</code></td>
<td>Name/IP address of the host machine</td>
</tr>
<tr>
<td><code>-port &lt;port num&gt;</code></td>
<td>TCP port number</td>
</tr>
<tr>
<td><code>-url &lt;url&gt;</code></td>
<td>URL description of hw_server/TCF agent</td>
</tr>
<tr>
<td><code>-list</code></td>
<td>List open connections</td>
</tr>
<tr>
<td><code>-set &lt;channel-id&gt;</code></td>
<td>Set active connection</td>
</tr>
<tr>
<td><code>-new</code></td>
<td>Create a new connection, even one exist to the same url</td>
</tr>
<tr>
<td><code>-xvc-url &lt;url&gt;</code></td>
<td>Open Xilinx Virtual Cable connection</td>
</tr>
<tr>
<td><code>-symbols</code></td>
<td>Launch symbol server to enable source level debugging for remote connections</td>
</tr>
</tbody>
</table>

**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 hw_server/TCF agent on host localhost and port 3121.

```
connect -url tcp:localhost:3121
```

Identical to previous example.

**disconnect**

Disconnect from hw_server/TCF agent.

**Syntax**

```
disconnect
```

Disconnect from active channel.

```
disconnect <channel-id>
```

Disconnect from specified channel.

**Returns**

Nothing, if the connection is closed. Error string, if invalid channel-id is specified.

**targets**

List targets or switch between targets.

**Syntax**

```
targets [options]
```

List available targets.

```
targets <target id>
```

Select <target id> as active target.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-set</td>
<td>Set current target to entry single entry in list. This is useful in combination with -filter option. An error will be generate if list is empty or contains more than one entry.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-regexp</td>
<td>Use regexp for filter matching</td>
</tr>
<tr>
<td>-nocase</td>
<td>Use case insensitive filter matching</td>
</tr>
<tr>
<td>-filter &lt;filter-expression&gt;</td>
<td>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 ==, !=, &gt;=, &lt;=, &lt;, &gt;, &amp;&amp; and</td>
</tr>
<tr>
<td>-target-properties</td>
<td>Returns a Tcl list of dict’s containing target properties.</td>
</tr>
<tr>
<td>-index &lt;index&gt;</td>
<td>Include targets based on jtag scan chain position. This is identical to specifying -filter {jtag_device_index==&lt;index&gt;}.</td>
</tr>
<tr>
<td>-timeout &lt;sec&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-architecture &lt;name&gt;</td>
<td>Specify default architecture is remote server does not provide it.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-defs</td>
<td>Read register definitions instead of values</td>
</tr>
<tr>
<td>-no-bits</td>
<td>Does not show bit fields along with register values. By default, bit fields are shown, when available</td>
</tr>
</tbody>
</table>

**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)**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rwr r8 0x0</code></td>
<td>Write 0x0 to register r8.</td>
</tr>
<tr>
<td><code>rwr usr r8 0x0</code></td>
<td>Write 0x0 to register r8 in group usr.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-addr &lt;address&gt;</td>
<td>Resume execution from address specified by &lt;address&gt;</td>
</tr>
<tr>
<td>-block</td>
<td>Block until the target stops or a timeout is reached</td>
</tr>
<tr>
<td>-timeout &lt;sec&gt;</td>
<td>Timeout value in seconds</td>
</tr>
</tbody>
</table>
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.

\textit{stpi}

Execute a machine instruction.

Syntax

\texttt{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.

\textit{nxti}

Step over a machine instruction.

Syntax

\texttt{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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-add &lt;expression&gt;</code></td>
<td>Add the <code>&lt;expression&gt;</code> 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.</td>
</tr>
<tr>
<td><code>-defs [expression]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-dict [expression]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-remove [expression]</code></td>
<td>Remove the expression from auto expression list. Only expressions previously added to the list through <code>-add</code> option can be removed. When the expression name is not specified, all the expressions in the auto expression list are removed.</td>
</tr>
<tr>
<td><code>-set &lt;expression&gt;</code></td>
<td>Set the value of a variable. It is not possible to set the value of an expression which includes constants or operators.</td>
</tr>
</tbody>
</table>

**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)**

```plaintext
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-defs</td>
<td>Return the variable definitions like address, type, size and RW flags.</td>
</tr>
<tr>
<td>-dict [expression]</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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)**

```sh
locals Int_Loc
```

Return the value of the local variable Int_Loc.

```sh
locals
```

Return the values of all the local variables in the current stack frame.

```sh
locals -defs
```

Return definitions of all the local variables in the current stack frame.

```sh
locals Int_Loc 23
```
Set the value of the local variable `Int_Loc` to 23.

```python
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.

```python
locals tmp_var.var1.array1[0]
```

Return the value of the element at index 0 in `array1`. `array1` is a member of `var1` struct, which is in turn a member of `tmp_var` struct.

**backtrace**

Stack back trace.

**Syntax**

```plaintext
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**

```plaintext
profile [options]
```

Configure and run the GNU profiler. The profiling needs to enabled while building bsp and application to be profiled.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-freq &lt;sampling-freq&gt;</code></td>
<td>Sampling frequency.</td>
</tr>
<tr>
<td><code>-scratchaddr &lt;addr&gt;</code></td>
<td>Scratch memory for storing the profiling related data. It needs to be assigned carefully, as it should not overlap with the program sections.</td>
</tr>
<tr>
<td><code>-out &lt;file-name&gt;</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-low &lt;addr&gt;</td>
<td>Low address of the profiling address range.</td>
</tr>
<tr>
<td>-high &lt;addr&gt;</td>
<td>High address of the profiling address range.</td>
</tr>
<tr>
<td>-freq &lt;value&gt;</td>
<td>Microblaze clock frequency in Hz. Default is 100MHz.</td>
</tr>
<tr>
<td>-count-instr</td>
<td>Count no. of executed instructions. By default no. of clock cycles of executed instructions are counted.</td>
</tr>
<tr>
<td>-cumulate</td>
<td>Cumulative profiling. Profiling without clearing the profiling buffers.</td>
</tr>
<tr>
<td>-start</td>
<td>Enable and start profiling.</td>
</tr>
<tr>
<td>-stop</td>
<td>Disable/stop profiling.</td>
</tr>
<tr>
<td>-out &lt;filename&gt;</td>
<td>Output profiling data to file. &lt;filename&gt; Name of the output file for writing the profiling data. If file name is not specified, profiling data is written to gmon.out.</td>
</tr>
</tbody>
</table>
**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-start</td>
<td>Enable and start trace. After starting trace the execution of the program is captured for later output.</td>
</tr>
<tr>
<td>-stop</td>
<td>Stop and output trace.</td>
</tr>
<tr>
<td>-con</td>
<td>Execute the command and output trace. The -con option is only available with embedded trace.</td>
</tr>
<tr>
<td>-stp</td>
<td></td>
</tr>
<tr>
<td>-nxt</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-out &lt;filename&gt;</td>
<td>Output trace data to a file. <em>&lt;filename&gt;</em> Name of the output file for writing the trace data. If not specified, data is output to standard output.</td>
</tr>
<tr>
<td>-level &lt;level&gt;</td>
<td>Set the trace level to &quot;full&quot;, &quot;flow&quot;, &quot;event&quot;, or &quot;cycles&quot;. If not specified, &quot;flow&quot; is used.</td>
</tr>
<tr>
<td>-halt</td>
<td>Set to halt program execution when the trace buffer is full. If not specified, trace is stopped but program execution continues.</td>
</tr>
<tr>
<td>-save</td>
<td>Set to enable capture of load and get instruction new data value.</td>
</tr>
<tr>
<td>-low &lt;addr&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-high &lt;addr&gt;</td>
<td>Set external trace data format to &quot;mdm&quot;, &quot;ftm&quot;, or &quot;tpiu&quot;. If format is not specified, &quot;mdm&quot; is used. The &quot;ftm&quot; and &quot;tpiu&quot; formats are output by Zynq-7000 PS. Only used with external trace.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-force</td>
<td>Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.</td>
</tr>
<tr>
<td>-size &lt;access-size&gt;</td>
<td>&lt;access-size&gt; 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.</td>
</tr>
<tr>
<td>-value</td>
<td>Return a Tcl list of values, instead of displaying the result on console.</td>
</tr>
<tr>
<td>-bin</td>
<td>Return data read from the target in binary format.</td>
</tr>
<tr>
<td>-file &lt;file-name&gt;</td>
<td>Write binary data read from the target to &lt;file-name&gt;.</td>
</tr>
<tr>
<td>-address-space &lt;name&gt;</td>
<td>Access specified memory space instead default memory space of current target. For ARM DAP targets, address spaces DPR, APR and AP&lt;n&gt; 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 &quot;-address-space APR&quot; and &quot;-address-space AP&lt;n&gt;&quot;, respectively. The APR address range is 0x0 - 0xffff, where the higher 8 bits select an AP and lower 8 bits are the register address for that AP.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-unaligned-access</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
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 specified by <address>. If <num> is greater than the size of the <values> list, the last word in the list is filled at the remaining address locations.

```
mwr [options] -bin -file <file-name> <address> [num]
```

*Read* <num> data values from a binary file and write to active target memory address specified by <address>. If <num> is not specified, all the data from the file is written sequentially from the address specified by <address>.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-force</td>
<td>Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.</td>
</tr>
<tr>
<td>-bypass-cache-sync</td>
<td>Do not flush/invalidate CPU caches during memory write. Without this option, debugger flushes/invalidates caches to make sure caches are in sync.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-size &lt;access-size&gt;</td>
<td>&lt;access-size&gt; 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 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.</td>
</tr>
<tr>
<td>-bin</td>
<td>Read binary data from a file and write it to target address space.</td>
</tr>
<tr>
<td>-file &lt;file-name&gt;</td>
<td>File from which binary data is read to write to target address space.</td>
</tr>
<tr>
<td>-address-space &lt;name&gt;</td>
<td>Access specified memory space instead default memory space of current target. For ARM DAP targets, address spaces DPR, APR and AP&lt;exp&lt;n&gt;&gt; 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 &quot;-address-space APR&quot; and &quot;-address-space AP&lt;exp&lt;n&gt;&gt;&quot;, respectively. The APR address range is 0x0 - 0xffff, where the higher 8 bits select an AP and lower 8 bits are the register address for that AP.</td>
</tr>
<tr>
<td>-unaligned-accesses</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
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 specified and only one symbol file exists in target's memory map, then that symbol file is used. If no symbol file is specified and multiple symbol files exist in target’s memory map, then an error is thrown.
Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-disable</td>
<td>Disable OS awareness for a symbol file. If this option is not specified, OS awareness is enabled.</td>
</tr>
<tr>
<td>-fast-exec</td>
<td>Enable fast process start. New processes will not be tracked for debug and are not visible in the debug targets view.</td>
</tr>
<tr>
<td>-fast-step</td>
<td>Enable fast stepping. Only the current process will be resynced after stepping. All other processes will not be resynced when this flag is turned on.</td>
</tr>
</tbody>
</table>

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 `<symbol-file>` and turn on fast-exec and fast-step modes.

```
osa -disable -file <symbol-file>
```

Disable OSA for `<symbol-file>`.

**memmap**

Modify Memory Map.

**Syntax**

```
memmap <options>
```

Add/remove a memory map entry for the active target.

Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-addr &lt;memory-address&gt;</td>
<td>Address of the memory region that should be added/removed from the target’s memory map.</td>
</tr>
<tr>
<td>-size &lt;memory-size&gt;</td>
<td>Size of the memory region.</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-flags &lt;protection-flags&gt;</td>
<td>Protection flags for the memory region. <code>&lt;protection-flags&gt;</code> 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 <code>&lt;protection-flags&gt;</code> is 0x3 (Read/Write Access).</td>
</tr>
<tr>
<td>-list</td>
<td>List the memory regions added to the active target’s memory map.</td>
</tr>
<tr>
<td>-clear</td>
<td>Specify whether the memory region should be removed from the target’s memory map.</td>
</tr>
<tr>
<td>-relocate-section-map &lt;addr&gt;</td>
<td>Relocate the address map of the program sections to <code>&lt;addr&gt;</code>. This option should be used when the code is self-relocating, so that the debugger can find the debug symbol info for the code. <code>&lt;addr&gt;</code> is the relative address, to which all the program sections are relocated.</td>
</tr>
<tr>
<td>-osa</td>
<td>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.</td>
</tr>
<tr>
<td>-properties &lt;dict&gt;</td>
<td>Specify advanced memory map properties.</td>
</tr>
<tr>
<td>-meta-data &lt;dict&gt;</td>
<td>Specify meta-data of advanced memory map properties.</td>
</tr>
</tbody>
</table>

### 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)

```bash
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.

```bash
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-clear</td>
<td>Clear uninitialized data (bss).</td>
</tr>
<tr>
<td>-keepsym</td>
<td>Keep previously downloaded elfs in the list of symbol files. Default behavior is to clear the old symbol files while downloading an elf.</td>
</tr>
<tr>
<td>-force</td>
<td>Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.</td>
</tr>
<tr>
<td>-bypass-cache-sync</td>
<td>Do not flush/invalidate CPU caches during elf download. Without this option, debugger flushes/invalidates caches to make sure caches are in sync.</td>
</tr>
<tr>
<td>-relocate-section-map <code>&lt;addr&gt;</code></td>
<td>Relocate the address map of the program sections to <code>&lt;addr&gt;</code>. This option should be used when the code is self-relocating, so that the debugger can find debug symbol info for the code. <code>&lt;addr&gt;</code> is the relative address, to which all the program sections are relocated.</td>
</tr>
<tr>
<td>-vaddr</td>
<td>Use vaddr from the elf program headers while downloading the elf. This option is valid only for elf files.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-force</code></td>
<td>Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.</td>
</tr>
<tr>
<td><code>-vaddr</code></td>
<td>Use vaddr from the elf program headers while verifying the elf data. This option is valid only for elf files.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-file &lt;bitstream-file&gt;</code></td>
<td>Specify file containing bitstream.</td>
</tr>
<tr>
<td><code>-partial</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-no-revision-check</code></td>
<td>Disable bitstream vs silicon revision revision compatibility check.</td>
</tr>
<tr>
<td><code>-skip-compatibility-check</code></td>
<td>Disable bitstream vs FPGA device compatibility check.</td>
</tr>
<tr>
<td><code>-state</code></td>
<td>Return whether the FPGA is configured.</td>
</tr>
<tr>
<td><code>-config-status</code></td>
<td>Return configuration status.</td>
</tr>
<tr>
<td><code>-ir-status</code></td>
<td>Return IR capture status.</td>
</tr>
<tr>
<td><code>-boot-status</code></td>
<td>Return boot history status.</td>
</tr>
<tr>
<td><code>-timer-status</code></td>
<td>Return watchdog timer status.</td>
</tr>
<tr>
<td><code>-cor0-status</code></td>
<td>Return configuration option 0 status.</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cor1-status</td>
<td>Return configuration option 1 status.</td>
</tr>
<tr>
<td>-wbstar-status</td>
<td>Return warm boot start address status.</td>
</tr>
</tbody>
</table>

**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 Configuration value.

### Target Reset

The following is a list of reset commands:

- **rst**

**rst**

Target Reset.

**Syntax**

```
rst [options]
```

Reset the active target.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-processor</td>
<td>Reset the active processor target.</td>
</tr>
<tr>
<td>-cores</td>
<td>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.</td>
</tr>
<tr>
<td>-system</td>
<td>Reset the active System.</td>
</tr>
<tr>
<td>-srst</td>
<td>Generate system reset for active target. With JTAG this is done by generating a pulse on the SRST pin on the JTAG cable associated with the active target.</td>
</tr>
<tr>
<td>-por</td>
<td>Generate power on reset for active target. With JTAG this is done by generating a pulse on the POR pin on the JTAG cable associated with the active target.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-ps</td>
<td>Generate PS only reset on Zynq MP. This is supported only through MicroBlaze PMU target.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-addr &lt;breakpoint-address&gt;</td>
<td>Specify the address at which the Breakpoint should be set.</td>
</tr>
<tr>
<td>-file &lt;file-name&gt;</td>
<td>Specify the <code>&lt;file-name&gt;</code> in which the Breakpoint should be set.</td>
</tr>
<tr>
<td>-line &lt;line-number&gt;</td>
<td>Specify the <code>&lt;line-number&gt;</code> within the file, where Breakpoint should be set.</td>
</tr>
<tr>
<td>-type &lt;breakpoint-type&gt;</td>
<td>Specify the Breakpoint type <code>&lt;breakpoint-type&gt;</code> can be one of the values below: auto = Auto - Breakpoint type is chosen by hw_server/TCF agent. This is the default type hw = Hardware Breakpoint sw = Software Breakpoint</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mode &lt;breakpoint-mode&gt;</td>
<td>Specify the access mode that will trigger the breakpoint. &lt;breakpoint-mode&gt; 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</td>
</tr>
<tr>
<td>-enable &lt;mode&gt;</td>
<td>Specify initial enablement state of breakpoint. When &lt;mode&gt; is 0 the breakpoint is disabled, otherwise the breakpoint is enabled. The default is enabled.</td>
</tr>
<tr>
<td>-ct-input &lt;list&gt; -ct-output &lt;list&gt;</td>
<td>Specify input and output cross triggers. &lt;list&gt; 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.</td>
</tr>
<tr>
<td>-skip-on-step &lt;value&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-properties &lt;dict&gt;</td>
<td>Specify advanced breakpoint properties.</td>
</tr>
<tr>
<td>-meta-data &lt;dict&gt;</td>
<td>Specify meta-data of advanced breakpoint properties.</td>
</tr>
<tr>
<td>-target-id &lt;id&gt;</td>
<td>Specify a target id for which the breakpoint should be set. A breakpoint can be set for all the targets by specifying the &lt;id&gt; 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.</td>
</tr>
</tbody>
</table>

**Note(s)**

- Breakpoints can be set in XSDB before connecting to hw_server/TCF agent. If there is an active target when a Breakpoint is set, the Breakpoint 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)**

```plaintext
bpadd -addr 0x100000
```

Set a Breakpoint at address 0x100000. Breakpoint type is chosen by hw_server/TCF agent.

```plaintext
bpadd -addr &main
```
Set a function Breakpoint at main. Breakpoint type is chosen by hw_server/TCF agent.

```bash
bpadd -file test.c -line 23 -type hw
```

Set a Hardware Breakpoint at test.c:23.

```bash
bpadd -target-id all 0x100
```

Set a breakpoint for all targets, at address 0x100.

```bash
bpadd -target-id 2 test.c:23
```

Set a breakpoint for target 2, at line 23 in test.c.

```bash
bpadd -addr &fooVar -type hw -mode 0x3
```

Set a Read_Write Watchpoint on variable fooVar.

```bash
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**

```bash
bpremove <id-list> | -all
```

Remove the Breakpoints/Watchpoints specified by `<id-list>` or remove all the breakpoints when `-all` option is used.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-all</td>
<td>Remove all breakpoints.</td>
</tr>
</tbody>
</table>

**Returns**

Nothing, if the breakpoint is removed successfully. Error string, if the breakpoint specified by `<id>` is not set.

**Example(s)**

```bash
bpremove 0
```
Remove Breakpoint 0.

\texttt{bpremove 1 2}

Remove Breakpoints 1 and 2.

\texttt{bpremove -all}

Remove all Breakpoints.

\textbf{bpenable}

Enable Breakpoints/Watchpoints.

**Syntax**

\texttt{bpenable <id-list> | -all}

Enable the Breakpoints/Watchpoints specified by \texttt{<id-list>} or enable all the breakpoints when \"-all\" option is used.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-all</td>
<td>Enable all breakpoints.</td>
</tr>
</tbody>
</table>

**Returns**

Nothing, if the breakpoint is enabled successfully. Error string, if the breakpoint specified by \texttt{<id>} is not set.

**Example(s)**

\texttt{bpenable 0}

Enable Breakpoint 0.

\texttt{bpenable 1 2}

Enable Breakpoints 1 and 2.

\texttt{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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-all</td>
<td>Disable all breakpoints.</td>
</tr>
</tbody>
</table>

**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 Breakpoints/Watchpoints along with brief status for each Breakpoint and the target on which it is set.
Returns
List of breakpoints.

**bpstatus**
Print Breakpoint/Watchpoint status.

**Syntax**
```
bpstatus <id>
```
Print the status of a Breakpoint/Watchpoint specified by `<id>`. Status includes the target information for which the Breakpoint is active and also 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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-start</code></td>
<td>Start the Jtag Uart terminal. This is the default option.</td>
</tr>
<tr>
<td><code>-stop</code></td>
<td>Stop the Jtag Uart terminal.</td>
</tr>
</tbody>
</table>
**Option** | **Description**
--- | ---
-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 running this command.

**Returns**
Socket port number.

**readjtaguart**
Start/Stop reading from Jtag_uart.

**Syntax**

```plaintext
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-start</td>
<td>Start reading the Jtag Uart output.</td>
</tr>
<tr>
<td>-stop</td>
<td>Stop reading the Jtag Uart output.</td>
</tr>
<tr>
<td>-handle &lt;file-handle&gt;</td>
<td>Specify the file handle to which the data should be redirected. If no file handle is given, data is printed on stdout.</td>
</tr>
</tbody>
</table>

**Note(s)**
- Select a MDM or ARM/MicroBlaze processor target before running 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)**

```plaintext
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-hw</td>
<td>HW design file.</td>
</tr>
</tbody>
</table>
## Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-list</code></td>
<td>Return a list of open designs for the targets.</td>
</tr>
<tr>
<td><code>-mem-ranges [list {start1 end1} {start2 end2}]</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

## 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.hdf Load the HW design named design.hdf and set memory map for all the child processors of APU target.
targets -filter {name =~ "xc7z045"}; loadhw design.hdf Load the HW design named design.hdf 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-target-id &lt;id&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-user &lt;bscan number&gt;</td>
<td>Specify user bscan port number.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-target-id &lt;id&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-user &lt;bscan number&gt;</td>
<td>Specify user bscan port number.</td>
</tr>
<tr>
<td>-which &lt;instance&gt;</td>
<td>Specify MicroBlaze instance number.</td>
</tr>
</tbody>
</table>

Returns

Nothing, if successful.

Example(s)

```
mb_drwr 1 0x282 10
```
Write to MB Control Reg.

**mdm_drrd**

Read from MDM Debug Register.

**Syntax**

```markdown
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-target-id &lt;id&gt;</code></td>
<td>Specify a target id representing MicroBlaze Debug Module or MicroBlaze instance to access. If this option is not used and <code>-user</code> option is not specified, then the current target is used.</td>
</tr>
<tr>
<td><code>-user &lt;bscan number&gt;</code></td>
<td>Specify user bscan port number.</td>
</tr>
</tbody>
</table>

**Returns**

Register value, if successful.

**Example(s)**

```bash
mdm_drrd 0 32
```

Read XMDC ID Reg.

**mb_drrd**

Read from MicroBlaze Debug Register.

**Syntax**

```markdown
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-target-id &lt;id&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-user &lt;bscan number&gt;</td>
<td>Specify user bscan port number.</td>
</tr>
<tr>
<td>-which &lt;instance&gt;</td>
<td>Specify MicroBlaze instance number.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-all</td>
<td>Include values for all contexts in result.</td>
</tr>
<tr>
<td>-context [context]</td>
<td>Specify context of value to get or set. The default context is &quot;&quot; which represent the global default. Not all options support context specific values.</td>
</tr>
<tr>
<td>-target-id &lt;id&gt;</td>
<td>Specify target id or value to get or set. This is an alternative to the -context option.</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-server</td>
<td>Get the TCF server build version, for the active connection.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-host &lt;addr&gt;</td>
<td>Limits the network interface on which to listen for incoming connections.</td>
</tr>
<tr>
<td>-port &lt;port&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

Returns

Server details are displayed 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 incoming connections on this host.

`xsdbserver stop`

Stop XSDB command server.

Syntax

```
xsdbserver stop
```

Stop XSDB command server listener and disconnect connected client if any.

Returns

Nothing, if the server is closed successfully. Error string, if the server has not been started already.
**xsdbserver disconnect**

Disconnect active XSDB server connection.

**Syntax**

`xsdbserver disconnect`

Disconnect current XSDB server connection.

**Returns**

Nothing, if the connection is closed. Error string, if there is no active connection.

**xsdbserver version**

Return XSDB command server version

**Syntax**

`xsdbserver version`

Return XSDB command server protocol version.

**Returns**

Server version if there is an active connection. Error string, if there is no active connection.

**JTAG Access**

The following is a list of jtag commands:

- `jtag targets`
- `jtag sequence`
- `jtag device_properties`
- `jtag lock`
- `jtag unlock`
- `jtag claim`
- `jtag disclaim`
- `jtag frequency`
- `jtag skew`
- `jtag servers`
**jtag targets**

List JTAG targets or switch between JTAG targets.

**Syntax**

jtag targets

List available JTAG targets.

jtag targets <target id>

Select <target id> as active JTAG target.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-set</td>
<td>Set current target to entry single entry in list. This is useful in combination with -filter option. An error will be generated if list is empty or contains more than one entry.</td>
</tr>
<tr>
<td>-regexp</td>
<td>Use regexp for filter matching.</td>
</tr>
<tr>
<td>-nocase</td>
<td>Use case insensitive filter matching.</td>
</tr>
<tr>
<td>-filter &lt;filter-expression&gt;</td>
<td>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 ==, !=, &lt;, &gt;, &lt;=, &gt;=, &lt;, &gt;, &amp;&amp; and</td>
</tr>
<tr>
<td>-target-properties</td>
<td>Returns a Tcl list of dictionaries containing target properties.</td>
</tr>
<tr>
<td>-open</td>
<td>Open all targets in list. List can be shorted by specifying target-ids and using filters.</td>
</tr>
<tr>
<td>-close</td>
<td>Close all targets in list. List can be shorted by specifying target-ids and using filters.</td>
</tr>
<tr>
<td>-timeout &lt;sec&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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)**

```bash
jtag targets
```

List all targets.

```bash
jtag targets -filter {name == "arm_dap"}
```

List targets with name "arm_dap".

```bash
jtag targets 2
```

Set target with id 2 as the current target.

```bash
jtag targets -set -filter {name =~ 'arm*'}
```

Set current target to target with name starting with "arm".

```bash
jtag targets -set -filter {level == 0}
```

List Jtag cables.

**jtag sequence**

Create JTAG sequence object.

**Syntax**

```bash
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:

```bash
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.

```bash
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 Capture TDO data during shift and return from sequence run command. -state <new-state> State to enter after shift is complete. The default is RESET.

sequence delay usec

Generate 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

\textit{jtag device\_properties}
Get/set device properties.

Syntax
jtag device\_properties idcode
Get JTAG device properties associated with \(<\text{idcode}>\).

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)
\begin{verbatim}
jtag device\_properties 0x4ba00477
\end{verbatim}
Return Tcl dict containing device properties for idcode 0x4ba00477.

\begin{verbatim}
jtag device\_properties \{idcode 0x4ba00477 mask 0xffffffff name dap irlen 4\}
\end{verbatim}
Set device properties for idcode 0x4ba00477.

\textit{jtag lock}
Lock JTAG scan chain.

Syntax
jtag lock [\text{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 attempt 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 enumerate or connect to available JTAG ports.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-list</code></td>
<td>List opened servers. This is the default if no other option is given.</td>
</tr>
<tr>
<td><code>-format</code></td>
<td>List format of supported server strings.</td>
</tr>
<tr>
<td><code>-open &lt;server&gt;</code></td>
<td>Specifies server to open.</td>
</tr>
<tr>
<td><code>-close &lt;server&gt;</code></td>
<td>Specifies server to close.</td>
</tr>
</tbody>
</table>

**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)

```bash
jtag servers
```
List opened servers and number of associated ports.

```bash
jtag servers -open xilinx-xvc:localhost:10200
```
Connect to XVC server on host localhost port 10200

```bash
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 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-offset &lt;seek&gt;</td>
<td>File offset to read from</td>
</tr>
</tbody>
</table>

Returns

Read data

`tfile write`

Write file handle

Syntax

`tfile write <handle>`

Write to specified file handle

Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-offset &lt;seek&gt;</td>
<td>File offset to write to</td>
</tr>
</tbody>
</table>

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>

Returns

tfile rmdir
Remove directory

Syntax

tfile rmdir <path>

Returns

tfile mkdir
Create directory

Syntax

tfile mkdir <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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-scan-chain <code>{list of idcode-irlength pairs}</code></td>
<td>List of idcode-irlength pairs. This can be obtained from xsdb command - jtag targets</td>
</tr>
<tr>
<td>-device-index <code>&lt;index&gt;</code></td>
<td>This is used to select device in the jtag scan chain.</td>
</tr>
<tr>
<td>-cpu-index <code>&lt;processor core&gt;</code></td>
<td>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.</td>
</tr>
<tr>
<td>-out <code>&lt;filename&gt;</code></td>
<td>Output SVF file.</td>
</tr>
<tr>
<td>-delay <code>&lt;tcks&gt;</code></td>
<td>Delay in ticks between AP writes.</td>
</tr>
<tr>
<td>-linkdap</td>
<td>Generate SVF for linking DAP to the jtag chain for ZynqMP Silicon versions 2.0 and above.</td>
</tr>
<tr>
<td>-bscan <code>&lt;user port&gt;</code></td>
<td>This is used to specify user bscan port to which MDM is connected.</td>
</tr>
<tr>
<td>-mb-chunksize <code>&lt;size in bytes&gt;</code></td>
<td>This used to specify the chunk size in bytes for each transaction while downloading. Supported only for Microblaze processors.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-jreg-name &lt;jtag-register-name&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-hex</td>
<td>Format the return data in hexadecimal.</td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-set &lt;path-list&gt;</td>
<td>Set the repository path and load all the software cores available. Multiple repository paths can be specified as Tcl list.</td>
</tr>
<tr>
<td>-get</td>
<td>Get the repository path(s).</td>
</tr>
<tr>
<td>-scan</td>
<td>Scan the repositories. Used this option to scan the repositories, when some changes are done.</td>
</tr>
<tr>
<td>-os</td>
<td>Return a list of all the OS from the repositories.</td>
</tr>
<tr>
<td>-libs</td>
<td>Return a list of all the libs from the repositories.</td>
</tr>
<tr>
<td>-drivers</td>
<td>Return a list of all the drivers from the repositories.</td>
</tr>
<tr>
<td>-apps</td>
<td>Return a list of all the applications from the repositories.</td>
</tr>
<tr>
<td>-add-platforms &lt;platform-name&gt;</td>
<td>Add the platform specified by &lt;platform-name&gt; to the repository.</td>
</tr>
</tbody>
</table>

**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)**

```bash
repo -set <repo-path>
```

Set the repository path to the path specified by `<repo-path>`.

```bash
repo -os
```

Return a list of OS from the repo.

```bash
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. Please refer to sub-command help for details.

**Returns**

Depends on the sub-command. Please refer to sub-command help for details.

**Example(s)**

Please 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-desc &lt;description&gt;</td>
<td>Add a Brief description about the platform.</td>
</tr>
<tr>
<td>-updatehw &lt;hw-spec&gt;</td>
<td>Update the platform to use a new hardware specification file specified by &lt;hw-spec&gt;.</td>
</tr>
<tr>
<td>-samples &lt;samples-dir&gt;</td>
<td>Make the application template specified in &lt;samples-dir&gt;, part of the platform. This option can only be used for acceleratable application. &quot;repo -apps &lt;platform-name&gt;&quot; can be used to list the application templates available for the given platform-name.</td>
</tr>
<tr>
<td>-make-local</td>
<td>Make the referenced SW components local to the platform.</td>
</tr>
<tr>
<td>-fsbl-target &lt;processor-type&gt;</td>
<td>Processor-type for which the existing fsbl has to be regenerated. This option is valid only for ZU+.</td>
</tr>
<tr>
<td>-create-boot-bsp</td>
<td>Generate boot components for the platform.</td>
</tr>
<tr>
<td>-remove-boot-bsp</td>
<td>Remove all the boot components generated during platform creation.</td>
</tr>
<tr>
<td>-fsbl-elf &lt;fsbl.elf&gt;</td>
<td>Prebuilt fsbl.elf to be used as boot component when &quot;remove-boot-bsp&quot; option is specified.</td>
</tr>
<tr>
<td>-pmufw-elf &lt;pmufw.elf&gt;</td>
<td>Prebuilt pmufw.elf to be used as boot component when &quot;remove-boot-bsp&quot; option is specified.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;software-platform name&gt;</td>
<td>Name of the software platform to be generated.</td>
</tr>
<tr>
<td>-desc &lt;description&gt;</td>
<td>Brief description about the software platform.</td>
</tr>
<tr>
<td>-hw &lt;handoff-file&gt;</td>
<td>Hardware description file to be used to create the platform.</td>
</tr>
<tr>
<td>-out &lt;output-directory&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-prebuilt</td>
<td>Mark the platform to be built from already built sw artifacts. This option should be used only if you have existig software platform artifacts.</td>
</tr>
<tr>
<td>-proc &lt;processor&gt;</td>
<td>The processor to be used; the tool will create default domain.</td>
</tr>
<tr>
<td>-samples &lt;samples-directory&gt;</td>
<td>Make the samples in &lt;samples-directory&gt;, part of the platform.</td>
</tr>
<tr>
<td>-os &lt;os&gt;</td>
<td>The os to be used; the tool will create default domain. This works in combination with -proc option.</td>
</tr>
<tr>
<td>-xpfm &lt;platform-path&gt;</td>
<td>Existing platform from which the projects have to be imported and made part of the current platform.</td>
</tr>
<tr>
<td>-no-boot-bsp</td>
<td>Mark the platform to build without generating boot components.</td>
</tr>
</tbody>
</table>

**Returns**

Empty string, if the platform is created successfully. Error string, if the platform cannot be created.

**Example(s)**

```bash
platform create -name 'zcu102_test' -hw zcu102
```

Defines a software platform for a pre-defined hardware description file.

```bash
platform create -name 'zcu102_test' -hw zcu102 -proc psu_cortexa53_0 -os standalone
```

Defines a software platform for a pre-defined hardware description file. Create a default domain with standalone os running on psu_cortexa53_0.

```bash
platform create -xpfm /path/zc702.xpfm
```

This will create a platform project for the platform pointed by the xpfm file.

```bash
platform create -name 'ZC702Test' -hw /path/zc702.xsa
```
Defines a software platform for a hardware description file.

**platform fsbl**

Configure fsbl

**Syntax**

```
platform fsbl
```

Configure extra compiler and linker flags for fsbl.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-extra-compiler-flags &lt;flags&gt;</code></td>
<td>Set extra compiler flags for fsbl to the flags specified by <code>&lt;flags&gt;</code>.</td>
</tr>
<tr>
<td><code>-extra-linker-flags &lt;flags&gt;</code></td>
<td>Set extra linker flags for fsbl to the flags specified by <code>&lt;flags&gt;</code>.</td>
</tr>
<tr>
<td><code>-report</code></td>
<td>Return a table of extra compiler and linker flags set for fsbl.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-domains &lt;domain-list&gt;</td>
<td>List of domains which need to be built and added to the repository. Without this option, all the domains that are part of the platform are built.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-extra-compiler-flags &lt;value&gt;</td>
<td>Set extra compiler flag for pmufw with the provided value.</td>
</tr>
<tr>
<td>-extra-linker-flags &lt;value&gt;</td>
<td>Set extra linker flag for pmufw with the provided value.</td>
</tr>
<tr>
<td>-report</td>
<td>Return the list of the flags set to pmufw.</td>
</tr>
</tbody>
</table>

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. Please refer to sub-command help for details.

**Returns**

Depends on the sub-command. Please refer to sub-command help for details.
Example(s)
Please 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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-display-name &lt;display name&gt;</td>
<td>Display name of the domain.</td>
</tr>
<tr>
<td>-desc &lt;description&gt;</td>
<td>Brief description about the domain.</td>
</tr>
<tr>
<td>-image &lt;location&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-sw-repo &lt;repositories-list&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-mss &lt;mss-file&gt;</td>
<td>Use mss from specified by &lt;mss-file&gt;, instead of generating mss file for the domain.</td>
</tr>
<tr>
<td>-prebuilt-data &lt;directory-name&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-readme &lt;file-name&gt;</td>
<td>Add README file for the domain, with boot instructions, etc.</td>
</tr>
<tr>
<td>-inc-path &lt;include-path&gt;</td>
<td>Additional include path which should be added while building the application created for this domain.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-lib-path &lt;library-path&gt;</td>
<td>Additional library search path which should be added to the linker settings of the application created for this domain.</td>
</tr>
<tr>
<td>-sysroot &lt;sysroot-dir&gt;</td>
<td>The Linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build.</td>
</tr>
<tr>
<td>-boot &lt;boot-dir&gt;</td>
<td>Directory to generate components after Linux image build.</td>
</tr>
<tr>
<td>-bif &lt;file-name&gt;</td>
<td>Bif file used to create boot image for Linux boot.</td>
</tr>
<tr>
<td>-gemu-args &lt;file-name&gt;</td>
<td>File with all PS QEMU args listed. This is used to start PS QEMU.</td>
</tr>
<tr>
<td>-pmuqemu-args &lt;file-name&gt;</td>
<td>File with all PMC QEMU args listed. This is used to start PMU QEMU.</td>
</tr>
<tr>
<td>-pmcqemu-args &lt;file-name&gt;</td>
<td>File with all pmcqemu args listed. This is used to start pmcqemu.</td>
</tr>
<tr>
<td>-gemu-data &lt;data-dir&gt;</td>
<td>Directory which has all the files listed in file-name provided as part of qemu-args and pmuqemu-args options.</td>
</tr>
</tbody>
</table>

**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)**

```bash
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;domain-name&gt;</td>
<td>Name of the domain.</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-display-name &lt;display_name&gt;</code></td>
<td>The name to be displayed in the report for the domain.</td>
</tr>
<tr>
<td><code>-desc &lt;description&gt;</code></td>
<td>Brief description about the domain.</td>
</tr>
<tr>
<td><code>-proc &lt;processor&gt;</code></td>
<td>Processor core to be used for creating the domain. For SMP Linux, this can be a Tcl list of processor cores.</td>
</tr>
<tr>
<td><code>-os &lt;os&gt;</code></td>
<td>OS type. Default type is standalone.</td>
</tr>
<tr>
<td><code>-support-app &lt;app-name&gt;</code></td>
<td>Create a domain with BSP settings needed for application specified by <code>&lt;app-name&gt;</code>. This option is valid only for standalone domains. &quot;repo -apps&quot; command can be used to list the available application.</td>
</tr>
<tr>
<td><code>-auto-generate-linux</code></td>
<td>Generate the Linux artifacts automatically.</td>
</tr>
<tr>
<td><code>-image &lt;location&gt;</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-sysroot &lt;sysroot-dir&gt;</code></td>
<td>The linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build.</td>
</tr>
</tbody>
</table>

### Returns

Empty string, if the domain is created successfully. Error string, if the domain cannot be created.

### Example(s)

```bash
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_domain 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. Please refer to sub-command help for details.

**Returns**

Depends on the sub-command. Please refer to sub-command help for details.

**Example(s)**

Please 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-append &lt;param&gt; &lt;value&gt;</code></td>
<td>Append the given value to the parameter.</td>
</tr>
</tbody>
</table>

Returns

Nothing, if the parameter is set/Appended successfully. Current value of the parameter if `<value>` is not specified. Error string, if the parameter cannot be set/Appended.

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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-lib &lt;lib-name&gt;</code></td>
<td>Return the configurable parameters of Library in BSP.</td>
</tr>
<tr>
<td><code>-os</code></td>
<td>Return the configurable parameters of OS in BSP.</td>
</tr>
<tr>
<td><code>-proc</code></td>
<td>Return the configurable parameters of processor in BSP.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;lib-name&gt;</td>
<td>Library to be removed from bsp settings.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-driver &lt;driver-name&gt;</td>
<td>Driver to be assigned to an IP.</td>
</tr>
<tr>
<td>-ip &lt;ip-name&gt;</td>
<td>IP instance for which the driver has to be added.</td>
</tr>
<tr>
<td>-ver &lt;version&gt;</td>
<td>Driver version.</td>
</tr>
</tbody>
</table>

**Returns**

Nothing, if the driver is set successfully. Error string, if the driver cannot be set.

**Example(s)**

```bash
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**

```bash
bsp setlib [options]
```

Add the library to the bsp settings of active domain.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;lib-name&gt;</td>
<td>Library to be added to the bsp settings.</td>
</tr>
<tr>
<td>-ver &lt;version&gt;</td>
<td>Library version.</td>
</tr>
</tbody>
</table>

**Returns**

Nothing, if the library is set successfully. Error string, if the library cannot be set.

**Example(s)**

```bash
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ver &lt;version&gt;</td>
<td>OS version.</td>
</tr>
</tbody>
</table>

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. Please 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name <code>&lt;project-name&gt;</code></td>
<td>Name of the library project to be built.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name <code>&lt;project-name&gt;</code></td>
<td>Name of the library project to be clean built.</td>
</tr>
</tbody>
</table>

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>
```

- `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>
```

- `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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-name &lt;project-name&gt;</code></td>
<td>Project name that should be created.</td>
</tr>
<tr>
<td><code>-type &lt;library-type&gt;</code></td>
<td><code>&lt;library-type&gt;</code> can be 'static' or 'shared'</td>
</tr>
<tr>
<td><code>-platform &lt;platform-name&gt;</code></td>
<td>Name of the platform. Use &quot;repo -platforms&quot; to list available pre-defined platforms.</td>
</tr>
<tr>
<td><code>-domain &lt;domain-name&gt;</code></td>
<td>Name of the domain. Use &quot;platform report &lt;platform-name&gt;&quot; to list the available domains in a platform.</td>
</tr>
<tr>
<td><code>-sysproj &lt;system-project&gt;</code></td>
<td>Name of the system project. Use &quot;sysproj list&quot; to know the available system projects in the workspace.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-switch &lt;path&gt;</td>
<td>Close existing workspace and switch to new workspace.</td>
</tr>
</tbody>
</table>
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. Please refer to sub-command help for details.
Returns
Depends on the sub-command. Please 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;app-name&gt;</td>
<td>Name of the application to be built.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;app-name&gt;</td>
<td>Name of the application to be clean built.</td>
</tr>
</tbody>
</table>

Returns

Nothing. Build log will be printed on the console.

Example(s)

```bash
app clean -name helloworld
```

Clean helloworld application.

**app config**

Configure C/C++ build settings of the application

Syntax


```bash
app config -name <app-name> <param-name>
```

Get the value of configuration parameter `<param-name>` for the application specified by `<app-name>`.

```bash
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name</td>
<td>Name of the application.</td>
</tr>
<tr>
<td>-set</td>
<td>Set the configuration parameter value to new <code>&lt;value&gt;</code>,</td>
</tr>
<tr>
<td>-get</td>
<td>Get the configuration parameter value.</td>
</tr>
<tr>
<td>-add</td>
<td>Append the new <code>&lt;value&gt;</code> to configuration parameter value. Add option is not supported for compiler-optimization</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-info</td>
<td>Displays more information like possible values and possible operations about the configuration parameter. A parameter name must be specified when this option is used.</td>
</tr>
<tr>
<td>-remove</td>
<td>Remove <code>&lt;value&gt;</code> from the configuration parameter value. Remove option is not supported for assembler-flags, build-config, compiler-misc, compiler-optimization, linker-misc and linker-script.</td>
</tr>
</tbody>
</table>

**Returns**

Depends on the arguments specified. `<none>` List of parameters available for configuration and description of each parameter.

<parameter name>: Parameter value, or error, if unsupported parameter is specified.

<parameter name><parameter value>: Nothing if the value is set successfully, or error, if unsupported parameter is specified.

**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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;application-name&gt;</td>
<td>Name of the application to be created.</td>
</tr>
<tr>
<td>-platform &lt;platform-name&gt;</td>
<td>Name of the platform. Use &quot;repo -platforms&quot; to list available pre-defined platforms.</td>
</tr>
<tr>
<td>-domain &lt;domain-name&gt;</td>
<td>Name of the domain. Use &quot;platform report &lt;platform-name&gt;&quot; to list the available system configurations in a platform.</td>
</tr>
<tr>
<td>-hw &lt;hw-spec&gt;</td>
<td>HW specification file exported from Vivado (XSA).</td>
</tr>
<tr>
<td>-sysproj &lt;system-project&gt;</td>
<td>Name of the system project. Use &quot;sysproj list&quot; to know available system projects in the workspace.</td>
</tr>
<tr>
<td>-proc &lt;processor&gt;</td>
<td>Processor core for which the application should be created.</td>
</tr>
<tr>
<td>-template &lt;application template&gt;</td>
<td>Name of the template application. Default is &quot;Hello World&quot;. Use &quot;repo -apps&quot; to list available template applications.</td>
</tr>
<tr>
<td>-os &lt;os-name&gt;</td>
<td>OS type. Default type is standalone.</td>
</tr>
<tr>
<td>-lang &lt;programming language&gt;</td>
<td>Programming language can be c or c++.</td>
</tr>
</tbody>
</table>

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.

```bash
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.

```bash
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.

```bash
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;application-name&gt;</td>
<td>Name of the application to be switched.</td>
</tr>
<tr>
<td>-platform &lt;platform-name&gt;</td>
<td>Name of the new Platform. Use &quot;platform -list&quot; to list the available platforms.</td>
</tr>
<tr>
<td>-domain &lt;domain-name&gt;</td>
<td>Name of the new domain. Use &quot;domain -list&quot; to list available domain in the active platform.</td>
</tr>
</tbody>
</table>

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. Please 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;sysproj-name&gt;</td>
<td>Name of the system project to be built.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;sysproj-name&gt;</td>
<td>Name of the application to be clean built.</td>
</tr>
</tbody>
</table>

Returns

Nothing, if the application is cleaned successfully. 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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name &lt;project-name&gt;</td>
<td>Application Project to which the sources should be imported.</td>
</tr>
<tr>
<td>-path &lt;source-path&gt;</td>
<td>Path from which the source files should be imported. If <code>&lt;source-path&gt;</code> is a file, it is imported to application project. If <code>&lt;source-path&gt;</code> is a directory, all the files/subdirectories from the <code>&lt;source-path&gt;</code> are imported to application project. All existing source files will be overwritten in the application, and new files will be copied. Linker script will not be copied to the application directory, unless -linker-script option is used.</td>
</tr>
<tr>
<td>-linker-script</td>
<td>Copies the linker script as well.</td>
</tr>
</tbody>
</table>

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

---

**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*
- *Modifying BSP 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`.

```bash
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.

```bash
# Connect to hw_server
xsdb% conn -url TCP:xhdbfarmc7:3121
tcfchan#0
xsdb% Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x1005a4 (Hardware Breakpoint)
xsd% Info: ARM Cortex-A9 MPCore #1 (target 3) Stopped at 0xffffffff (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

# Once 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 bootmodes 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)
xbdb% 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_tbl10: 0xfffd8000 - 0xfffd800f
section .mmu_tbl11: 0xfffd9000 - 0xfffdafff
section .mmu_tbl12: 0xfffdff000 - 0xfffe0fff
section .init_array: 0xfffd0000 - 0xfffd0007
section .fini_array: 0xfffe0008 - 0xfffe047
section .sdata: 0xfffe048 - 0xfffe07f
section .bsa: 0xfffe080 - 0xfffe197f
section .bsa: 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: 00000000000000000004 r4: 0000000000000000000f r5: 000000000000000000ff
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 AP 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 0xfffffelfc (Suspended)
Info: ARM Cortex-A9 MPCore #1 (target 3) Stopped at 0xfffffffe18 (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.hdf
```
# 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, .data: 0x00103824 - 0x001042db
section, .rodata: 0x001042dc - 0x0010e898
section, .init_array: 0x0010e899 - 0x0010e89c
section, .fini_array: 0x0010e89d - 0x0010e89f
section, .data: 0x0010e8a0 - 0x0010e898
section, .heap: 0x0010e898 - 0x0010e89f
section, .stack: 0x0010e898 - 0x0010e89f

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

# 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
      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 an example XSCT script to generate an SVF file:

```bash
# 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}
```
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.hdf
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.hdf
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.

```tcl
connect
source ps7_init.tcl
targets -set -filter {name =~ 'APU'}
loadhw system.hdf
stop
ps7_init
targets -set -nocase -filter {name =~ 'ARM*#0'}
 rst -processor
dow <app>.elf
readjtaguart
c
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.

```tcl
connect
source ps7_init.tcl
targets -set -filter {name =~ 'APU'}
loadhw system.hdf
stop
ps7_init
targets -set -nocase -filter {name =~ 'ARM*#0'}
 rst -processor
dow <app>.elf
set fp [open uart.log w]
readjtaguart -handle $fp
c
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.

```bash
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.

```bash
bsp setosversion -ver 6.6
```
Assigning a driver to an IP.

```bash
bsp setdriver -ip ps7_uart_1 -driver generic -ver 2.0
```

Removing a library (removes xilrsa library from the domain/BSP).

```bash
bsp removelib -name xilrsa
```

**Editing FSBL/PMUFW Source File**

The following example shows you how to edit FSBL/PMUFW source files.

```bash
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.

```bash
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
```
Embedded Drivers and Libraries

The embedded drivers and libraries are hosted on a wiki.

**Embedded Driver Documentation**

The embedded driver documentation is hosted on this wiki.

**Embedded Library Documentation**

The embedded library documentation is hosted on this wiki.
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 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.