Vitis Unified Software Platform Documentation

Embedded Software Development

UG1400 (v2025.1) July 8, 2025

AMD

Table of Contents

Getting Started with Vitis

Note: $XILINX_VITIS indicates the AMD Vitis™ installation directory. It is setup after you run `source <VITIS_INSTALL_DIR>/<VERSION>/settings.sh`.

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

Chapter 1

Navigating Content by Design Process

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

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

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

Vitis Software Platform Installation

For information on AMD Vitis™ Software Platform Release Notes, refer to *Vitis Software Platform Release Notes* (UG1742).

Installing the Vitis Software Platform

Ensure your system meets all requirements described in Installation Requirements in the *Vitis Software Platform Release Notes* (UG1742).

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

1. Go to the AMD Adaptive Computing Downloads Website.
2. Download the installer for your operating system.
3. Run the installer, `xsetup` on Linux, or `xsetup.exe` on Windows, which opens the Welcome screen. Extract the installer package.
4. Click **Next**.
5. If installing with the web installer, the Select Install Type screen is displayed. Enter your AMD user account credentials, and select **Download and Install Now**. Click **Next** to open the Accept License Agreements screen of the installer. Accept the terms and conditions by clicking each **I Agree** check box. Click **Next** to open the next screen (only required on the web installer).
6. The Select Product to Install screen is displayed.
   a. To install the full Vitis Software Platform for embedded software and application acceleration development, choose **Vitis** and click **Next** to open the Vitis Unified Software Platform screen of the installer. This full Vitis installation includes Vivado, v++ and AI Engine toolchains.
   b. If you only need to perform software development for embedded processors, and require a small installation footprint, choose **Vitis Embedded Development** and click **Next**. If you downloaded the Vitis Embedded installer, this is the only option available on the installer.
7. Customize your installation by selecting design tools and devices (this step is only for the full Vitis installation).

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

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

You can enable the devices required for your development.

IMPORTANT! *For the Vitis acceleration flow, the following device choice is required for installation:* ***Devices → Install devices for Alveo and Edge acceleration platforms***

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

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

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

   ***Note*:** This script is not required on Windows.

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

```
sudo installLibs.sh
```

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

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

Chapter 3

Getting Started with the Vitis Software Platform

Vitis Unified Software Platform Overview

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

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

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

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

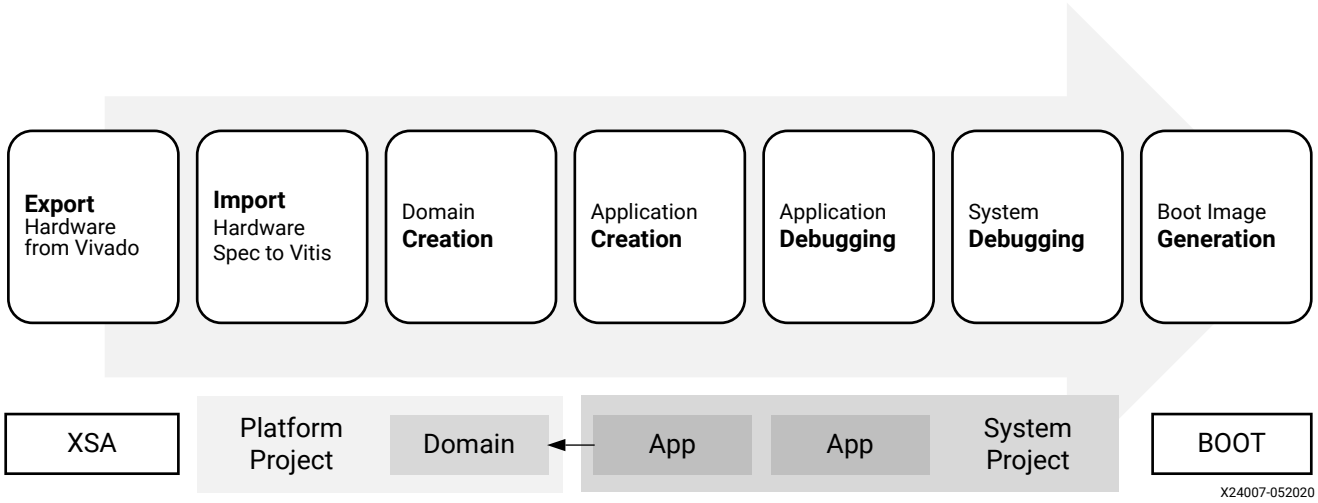
- Creating platforms from AMD Vivado™ generated hardware designs and generating BSP for software development
- Creating applications from example designs or empty template
- Configuring and building the platforms and applications
- Running, debugging, or profiling the applications on hardware
- Managing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

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

Vitis Software Development Workflow

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

Figure 1: **Embedded Software Application Development Workflow**



- Hardware engineers design the logic and export information required by software development from the AMD Vivado™ Design Suite to an XSA archive file.
- Software developers import XSA into the Vitis software platform by creating a platform. Platform was heavily used by application acceleration projects. To unify the Vitis workspace architecture for all kinds of applications, software development projects now migrate to platform and application architecture. A platform includes hardware specification and software environment settings.
- The software environment settings are called domains, which are also a part of a platform.
- Software developers create applications based on the platform and domains.
- Applications can be debugged in the Vitis IDE.
- In a complex system, several applications run at the same time and communicate with each other. So the system level verification needs to be done as well.

- After everything is ready, the Vitis IDE can help to create boot images which initialize the system and launch applications.

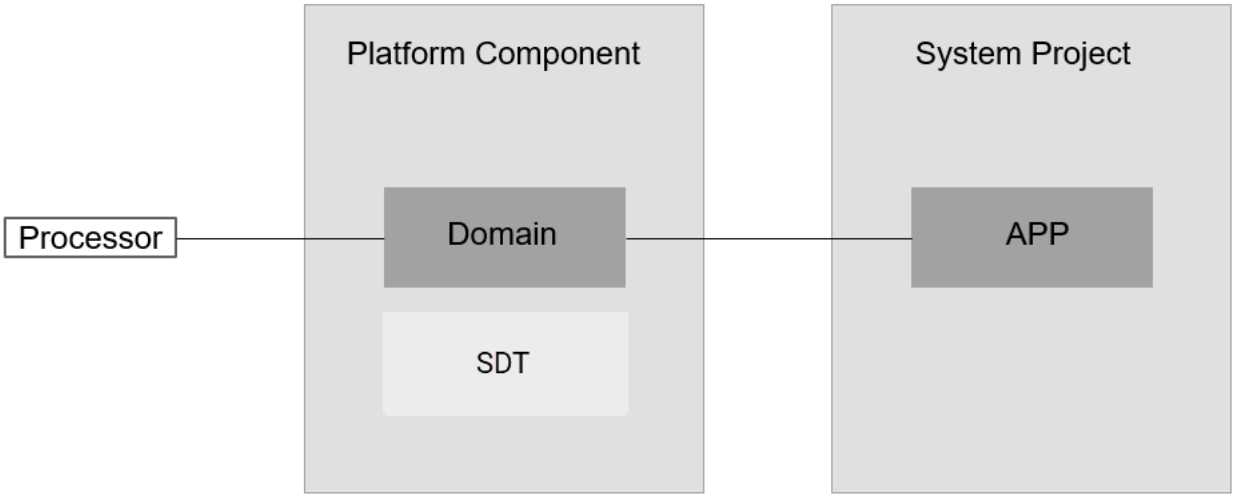
  ***Note*:**

  The Vitis Unified IDE offers support for two types of XSA and corresponding platforms: fixed XSA with a fixed platform and extensible XSA with an extensible platform. A fixed XSA, which includes a complete device image generated within Vivado, is employed to establish a fixed platform tailored for embedded development. On the other hand, the extensible XSA is used for acceleration development purposes. During the linking stage in V++, the comprehensive device image required for generating BOOT.bin for boot operations is created, specifically for acceleration development. So if you use extensible platform to create bare-metal application, you might encounter issue, for example "can not find corresponding devices." For more information about fixed platform and extensible platform, refer to Fixed Platforms versus Extensible Platforms in the *Embedded Design Development Using Vitis* (UG1701).

Workspace Structure in the Vitis Software Platform

There are two project types in Vitis unified workspace:

Figure 2: **Vitis Software Platform Project Types**



- **Workspace:** When you open the Vitis unified software platform, you create a workspace. A workspace is a directory location used by the Vitis unified software platform to store project data and metadata. An initial workspace location must be provided when the Vitis software platform is launched.
- **XSA:** XSAs are exported from the Vivado Design Suite. It has the hardware specifications like processor configuration properties, peripheral connection information, address map, and device initialization code. You have to provide the XSA when creating a platform project.

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

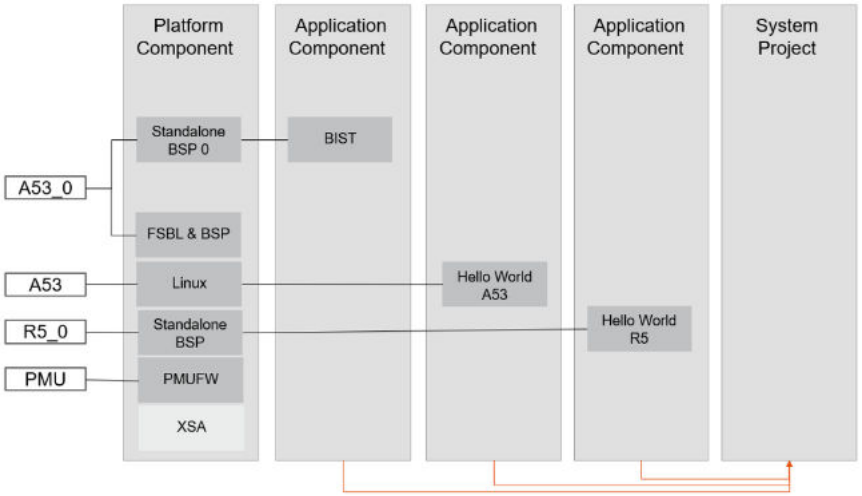
The Vitis platform has different configurations to support different use cases, outlined as follows:

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

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

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

Figure 3: **Vitis Software Development Workspace Example for Zynq UltraScale+ MPSoC**



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

Migrating from the Classic Vitis IDE to Vitis Unified IDE

The Vitis Unified IDE workspace is designed to ensure version control optimization. Hence, the workspace and project metadata is incompatible with classic IDE. To help you transition, two methods are available for migrating your projects.

Migrate manually (from 2025.1 Onward)

Starting with the 2025.1 release, the Vitis Classic IDE has been removed, and no migration utility is available in this version. If you did not migrate your workspace using the utility in a previous release (2023.2, 2024.1, or 2024.2), you will need to manually recreate your workspace in the Vitis Unified IDE.

To manually migrate your project, follow the steps below:

- Generate a new XSA file using the latest Vivado tools.
- Recreate your platform project in the Unified IDE based on the updated XSA file.
- Recreate your application project using the new platform.
- Import your application source code into the new project.
- Build your application and resolve any issues that arise during the build.

  Note: If you encounter any issues related to migration, refer to Standalone Application Component Migration Details

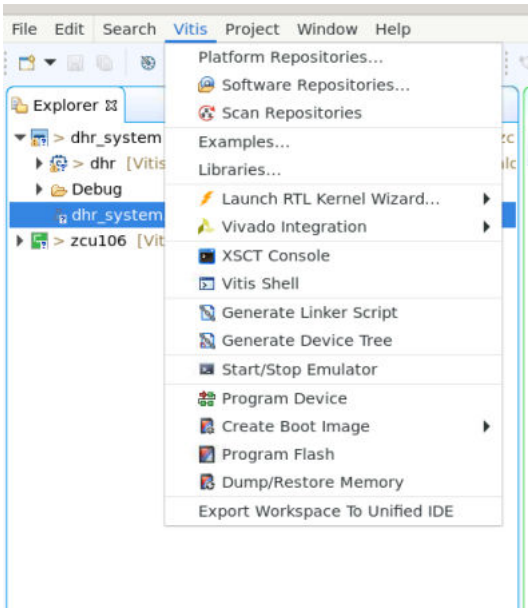
TIP: *For a cleaner and more controlled migration process, it is recommended to perform manual migration even if the utility is available in earlier releases.*

Migrate Projects Using Previous Version Utility

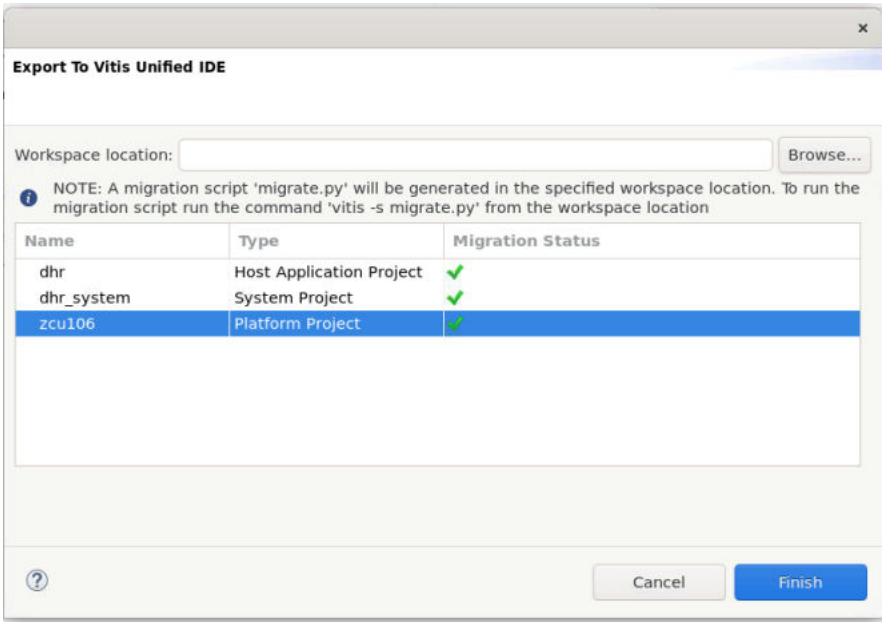
A migration utility is available in the Classic IDE (versions 2023.2, 2024.1, and 2024.2) to help you migrate your workspace to the Unified IDE. After migrating your project in the Classic IDE, you can then update your hardware platform and application to the current release.

To use this method, follow the steps below.

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



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



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

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

   Vitis would run the migration script. The running process takes some time. Once completed, you can see the migrated components and projects in this workspace directory. All the project resources are copied to the new location.
6. Open the migrated workspace with Vitis Unified IDE.
7. Open your platform project in the IDE and locate and open the platform Json file.
8. Click **Switch/Re-read XSA** to select the updated XSA file generated with the latest tool version.

   Note: This action regenerates the System Device Tree (SDT) and Board Support Package (BSP) for your applications.
9. Rebuild your platform and applications.

Table 1: **Supported Projects and Limitations when using this utility**

Project type	Limitation	Workaround
Platform	Local changes to BSP sources are not be migrated to the new workspace. A new BSP is created, and the settings are applied on the new BSP.	Copy the sources to the new BSP manually.
	Any embedded software repositories added to Vitis IDE cannot be migrated. A warning is shown in the wizard if you have any local sw repos.	All software repositories need to be migrated to lopper first. The path to the migrated repository can be added to the migration script (refer below screenshot for an example) to migrate the projects.
	IP drivers included in XSAs which were created with 2023.1 or older releases might encounter compilation errors.	Need to regenerate the XSA with 2024.2 release.
Embedded Application	Applications referring to platforms that are outside the current workspace cannot be migrated.	Migrate the platform first and update the application to use the new platform before migrating the application.
	If your application relies on the device ID, some modifications might be required in the application source code. This is because the Device ID in the xparameter.h file has is deprecated. For more details, refer to *Porting Guide for embeddedsw Components System Device Tree Based Build Flow* (UG1647)	Refer to Standalone Application Component Migration Details
	Debug Configurations cannot be taken over to Unified IDE	Create launch configuration in Vitis Unified IDE

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

```
vitis --classic
```

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

For more information on Vitis Unified IDE, refer to Launching the Vitis Unified IDE and Vitis Unified IDE View and Features.

Migrate manually (from 2025.1 Onward)

Starting with the 2025.1 release, the Vitis Classic IDE has been removed, and no migration utility is available in this version. If you did not migrate your workspace using the utility in a previous release (2023.2, 2024.1, or 2024.2), you will need to manually recreate your workspace in the Vitis Unified IDE.

To manually migrate your project, follow the steps below:

1. Generate a new XSA file using the latest Vivado tools.
2. Recreate your platform project in the Unified IDE based on the updated XSA file.
3. Recreate your application project using the new platform.

4. Import your application source code into the new project.
5. Build your application and resolve any issues that arise during the build.

> **TIP:** *For a cleaner and more controlled migration process, it is recommend performing manual migration even if the utility is available in earlier releases*

Standalone Application Component Migration Details

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

Unified IDE Features versus Classic IDE Features

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

Table 2: **Example table**

Features	Vitis Unified IDE	Vitis Classic IDE
Platform Creation	Target Platform	Target Platform
Application Development	Applications	Applications
Using Customer Libraries	Creating a Library Project	Using Customer Libraries in Application Projects
Run Application	Running the Application Component	Run Application Project
Debug Application	Debugging Application Component	Debug Application Project
Cross Trigger	Cross-Triggering	Cross Triggering
TCF Profiling	TCF Profiling	TCF Profiling
Gprof Profiling	No Support	Gprof Profiling
OS Aware Debug	No support	OS Aware Debug
Xen Aware Debug	XEN Aware Debug	Xen Aware Debug
OPtimize: Performance Analysis	No Support	Optimize Performance Analysis
Creatng Boot Image	Creating a Boot Image	Creating a Boot Image
Program Flash	Programming Flash	Program Flash
Multi-Cable and Multi-Device Support	Multi-Cable and Multi-Device Support	Multi-Cable and Multi-Device Support
Target Management	Target Connections	Target Connections
Version Control	Source Control	Version Control Git

Table 2: **Example table** *(cont'd)*

Features	Vitis Unified IDE	Vitis Classic IDE
User Managed Flow	User Managed Mode	No Support
Workspace Import and Export	Project Export and Import	Project Export and Import
Software Debugger	*Software Debugger Reference Guide* (UG1725)	Software Command-Line Tool
Python CLI	Section III: Vitis Python API	No Support

***Note*:** The feature link under the Vitis Classic IDE column directs you to the corresponding chapter in the 2023.1 version of this document.

Using the Vitis Unified IDE

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

This section contains the following chapters:

Launching the Vitis Unified IDE

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

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

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

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

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

   where `<workspace>` indicates a folder to hold all of the contents of your design project. The workspace is used to group together the source and data files that make up a design, or multiple designs, and stores the configuration of the tool for that workspace.

For other supported launch modes, see Vitis Unified IDE Launch Options.

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

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

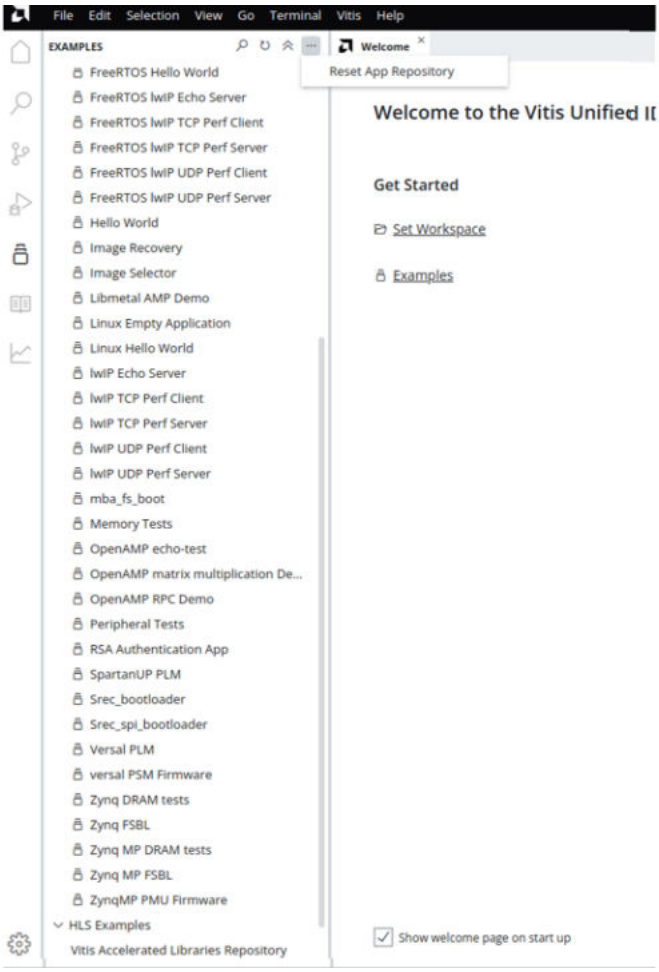
You can set up the platform repository path with the following example:

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

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

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

Figure 4: **Vitis Unified IDE Welcome Screen**



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

Vitis Unified IDE Launch Options

Launch Options

The Vitis Unified IDE supports the following modes:

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

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

- **Analysis Mode:** Analysis mode launches the tool directly into the Analysis View, letting you review the summary reports generated during the build, run, and debug processes.

  ```
  vitis -a
  ```

- **Interactive Mode:** Interactive mode lets you enter commands through the interactive Python shell, outside of the GUI, as described in Vitis Interactive Python Shell in the *Vitis Reference Guide* (UG1702).

  ```
  vitis -i
  ```

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

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

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

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

  Note: Jupyter notebook is supported in Linux OS.

  ```
  vitis -j
  ```

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

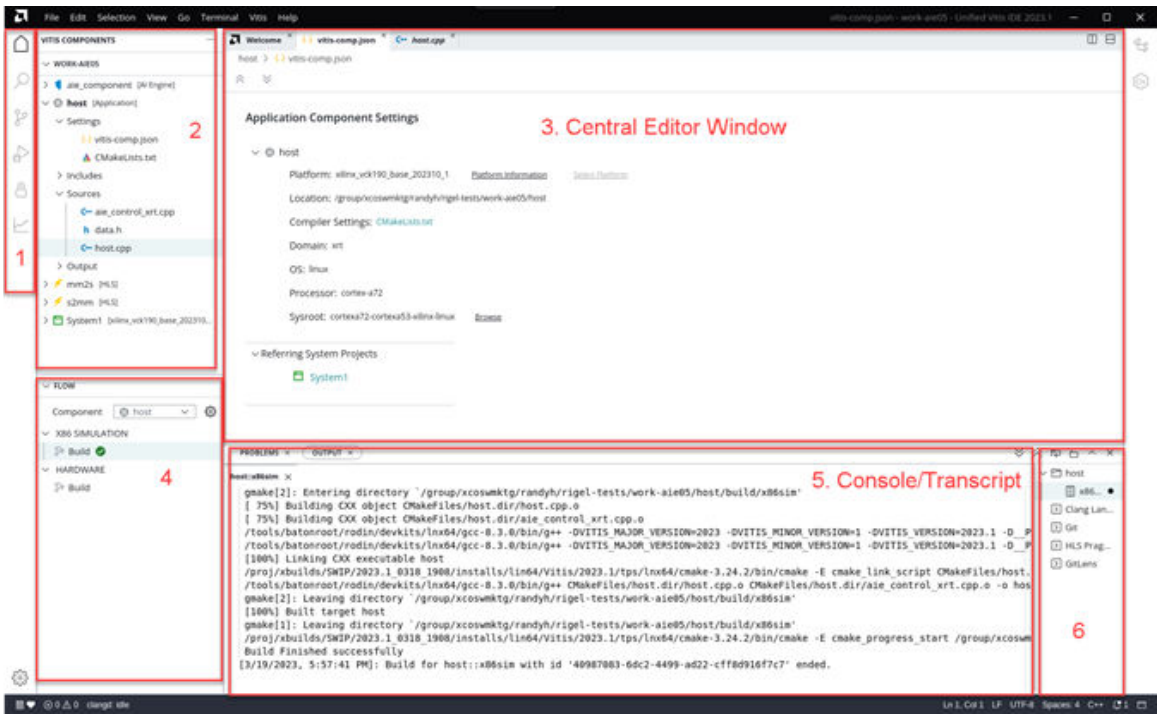
```
vitis -h

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

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

Vitis Unified IDE View and Features

Figure 5: **Vitis Unified IDE**

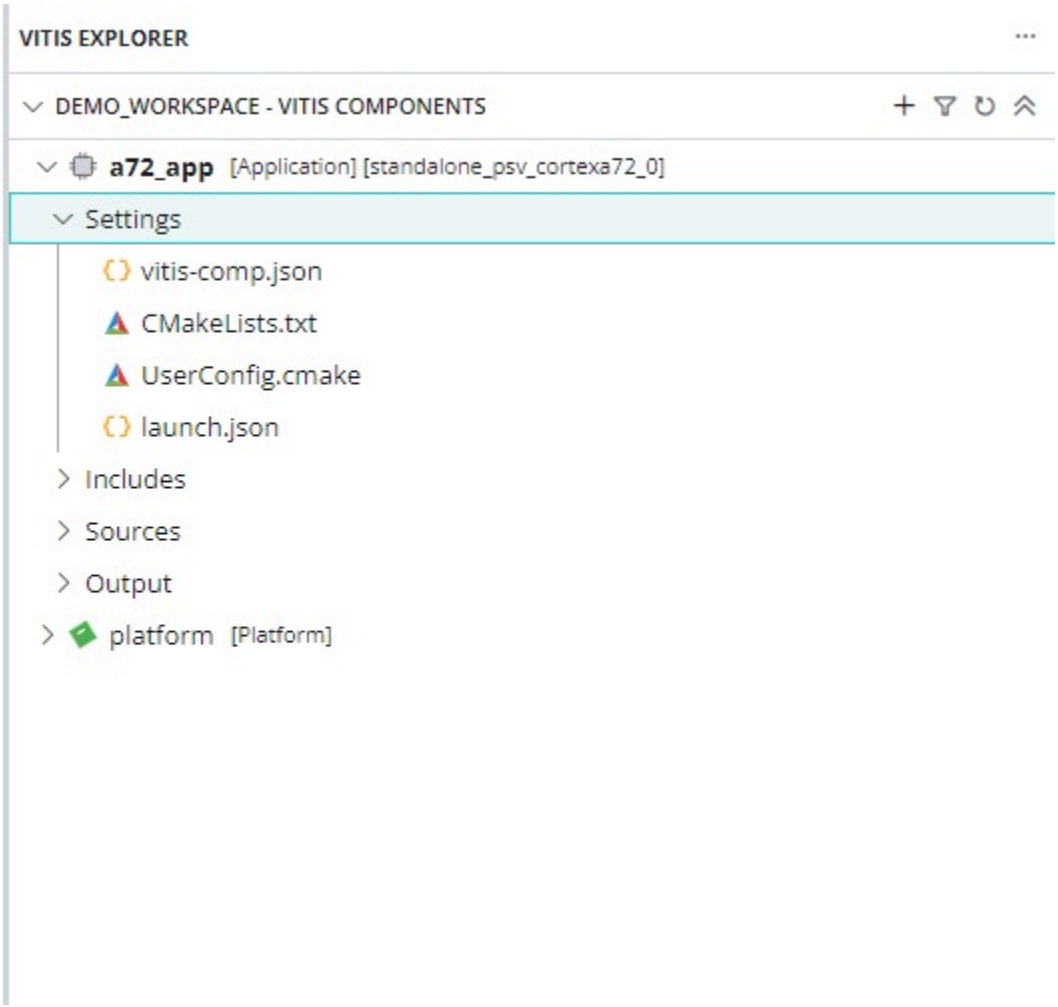


1. The Toolbar menu at the left of the screen provides easy access to major features of the tool: the Vitis Component Explorer, the Search function, Source Control, the Debug view, Examples, and the Analysis view.
2. Toolbar menu can customized based on your preference using View dropdown.
3. The Vitis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

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

Vitis Explorer View

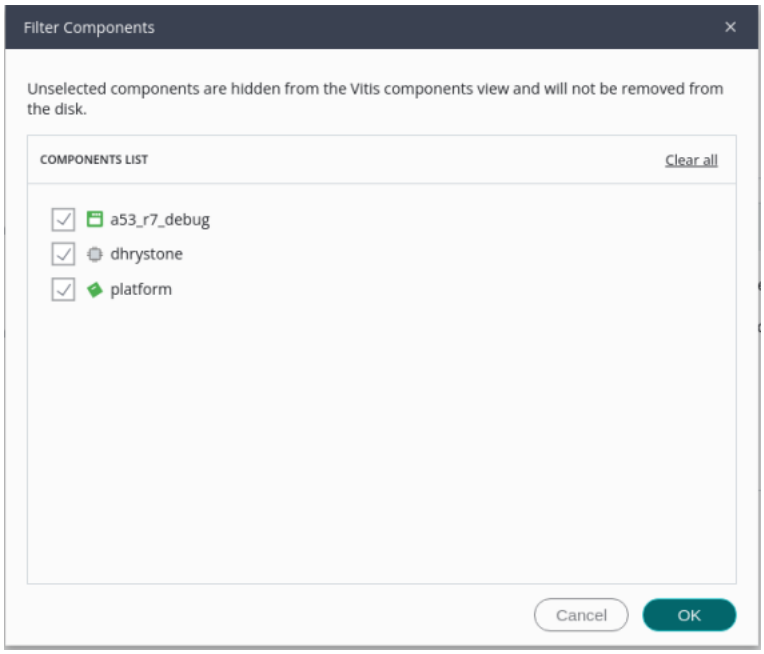
Figure 6: **Vitis Explorer View**



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

- **Via the toolbar menu:**
  - You can create a New Component.

- You can select Filter Components to hide or show the components of interest.



  - You can refresh the Component Explorer.
  - You can Collapse All to minimize the displayed content of the components and projects.

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

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

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

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

Deep JSON Based Component Display

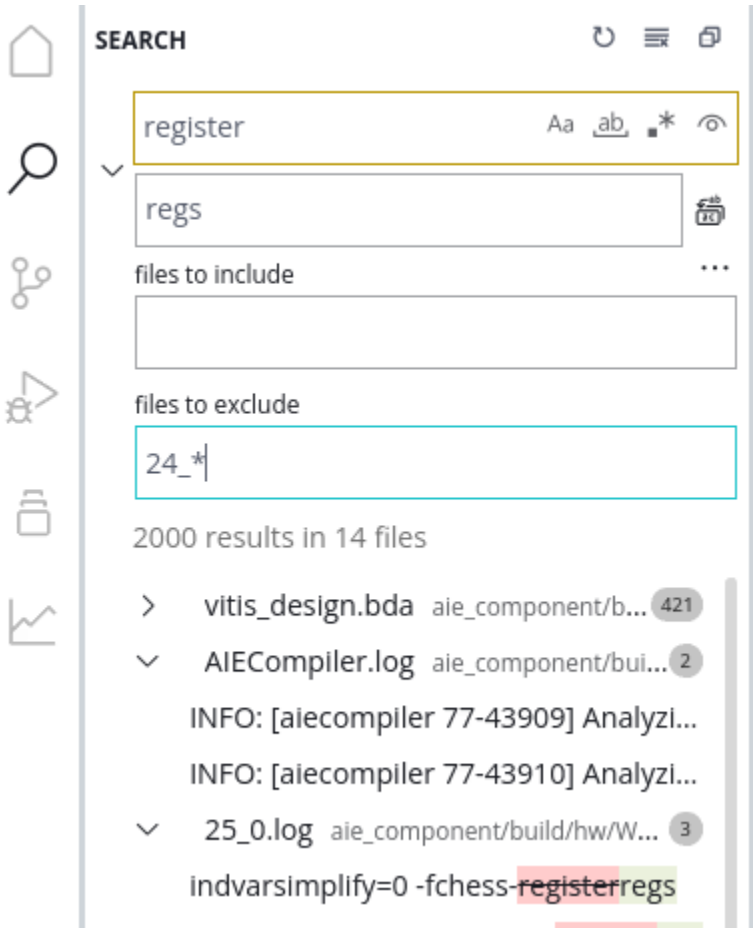
Vitis unified software platform supports displaying all components from the current workspace, no matter which sub directory they are in, in the Component View with unique names. You can directly access, select, edit, compile and use these components.

***Note*:** Notification are displayed to inform if components exist but are not supported.

Search View

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

Figure 7: **Search View**



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

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

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

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

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

Source Control

Source control and version control techniques are widely used in the software development flow. This chapter describes how to use Git integration in the Vitis Unified IDE.

Files Required for Source Control

The Vitis workspace contains both metadata and source/build files; however, not all of these files need to be tracked in git. The workspace root requires the `.gitignore` file. This file specifies the following files and folders to ignore when commiting your workspace to git:

- Build output directory
- Object files
- Dependent files
- Logs folder and log files
- Lock files

For more information on `.gitignore`, see https://git-scm.com/docs/gitignore

***Note*:** The '_ide' folders should be ignored as these folders will be automatically generated when a new workspace is loaded within the Vitis Unified IDE.

Therefore, the recommended files to track in git are as follows:

Workspace Folder

Table 3: **Files Under Workspace**

Files	Required for Source control
_ide/version.ini	Yes
_ide/setting.json	Depends on the user
_ide/worspace_journal.py	No
.gitignore	Yes

***Note*:** The `version.ini` file contains version information for your workspace and is used to manage compatibility across different IDE versions. It should be committed for source control

***Note*:** The `settings.json` file stores workspace-specific settings and preferences tailored to the workspace owner's environment. Because it reflects personal configurations, it is not recommended to commit this file to source control.

***Note*:** The .gitignore file specifies which files and directories Git should ignore in a project. It should be committed to source control to ensure all team members follow the same rules for excluding temporary, build, or user-specific files from version tracking.

***Note*:** The workspace journal file logs actions performed after opening the workspace. It serves as a reference for users to review the Python commands executed in the background. This file is not needed for source control.

Platform Folder

Table 4: **Files Under Platform Component Folder**

Files	Required for Source control
export	No
hw/hw.xsa	Yes
hw/sdt	No
resources	No
vitis-comp.json	Yes
.gitignore	Yes
psu_cortexa53_0 (BSP)	Yes
zynqmp_fsbl (BSP)	Yes

***Note*:** The `export/` folder contains the output files generated during platform creation. By default, this folder is excluded from version control through the `.gitignore` file. If you need to commit the contents of the `export/` folder (for example, for sharing or archiving purposes), use the `-f` (force) option with Git, as shown below:

```
git add export/* -f
```

***Note*:** The `hw/sdt/` folder contains the System Device Tree files used for BSP and device tree generation. This folder can be easily regenerated by clicking **Switch/Re-read XSA File** in the Platform Settings page. Therefore, it is not required to be committed to source control.

***Note*:** `psu_cortexa53_0 (BSP)` is the standalone Board Support Package (BSP) for the Zynq UltraScale+ MPSoC A53 core. For Versal devices, the APU core name differs — take note when selecting or creating BSPs.

Applications Folder

Table 5: **Files Under Application Component**

Files	Required for Source control
src	Yes
build	No

Table 5: **Files Under Application Component** *(cont'd)*

Files	Required for Source control
vitis-comp.json	Yes
compile_commands.json	No
_ide/launch.json	Depends on the user
.gitignore	Yes

***Note*:** The `build/` folder contains the output files generated during application building. By default, this folder is excluded from version control through the `.gitignore` file. If you need to commit the contents of the `export/` folder (for example, for sharing or archiving purposes), use the `-f` (force) option with Git, as shown below:

```
git add build/* -f
```

***Note*:** The `compile_commands.json` file contains the compiler commands used to build the application source files. It is automatically generated when the build process starts and is not required to be committed to source control.

***Note*:** The `launch.json` file, located under the `_ide/` folder, stores user-defined debug configurations. Because it reflects personal preferences and environment-specific settings, it is typically not recommended to commit this file to source control.

System Project Folder

Table 6: **Files Under System Project**

Files	Required for Source control
vitis-sys.json	Yes
_ide/launch.json	Depends on the user
.git ignore	Yes

***Note*:** The `launch.json` file, located under the `_ide/` folder, stores user-defined debug configurations. Because it reflects personal preferences and environment-specific settings, it is typically not recommended to commit this file to source control.

Source Control

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

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

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

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

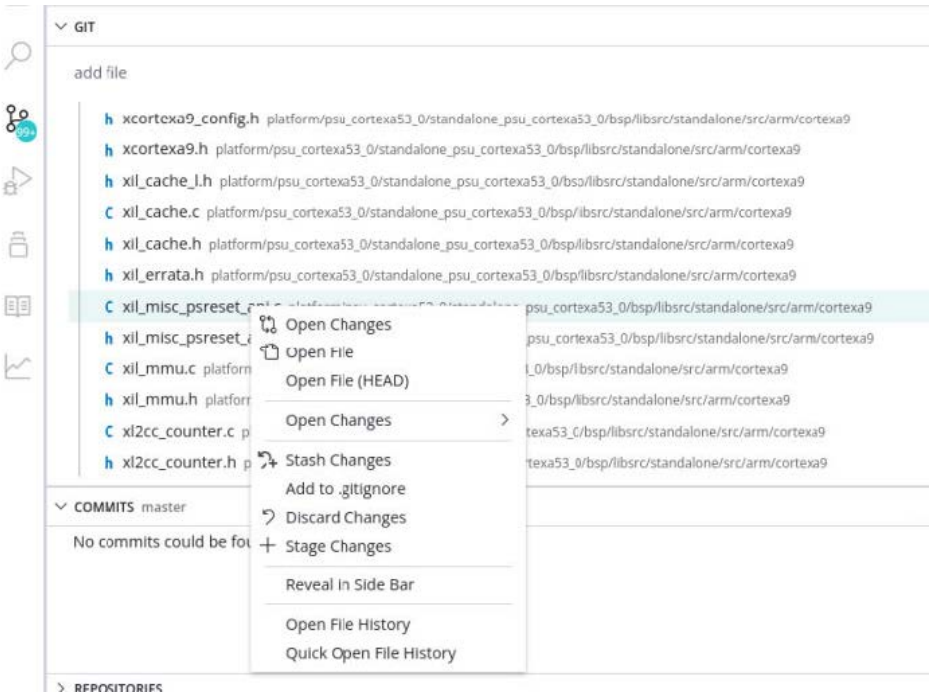
After you create a new component or project in the workspace, Vitis Unified IDE generates a `.gitignore` file. The `.gitignore` file can help you filter out the generated files so that it is easier to pick the files for source control. You can open the `.gitignore` file and edit this file if you have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

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

Add New File for Source Control

To add a file for source control, you can do the following: Right click the file you want to add and select **+ Stage Changes**.

Figure 8: **Add New File for Source Control**

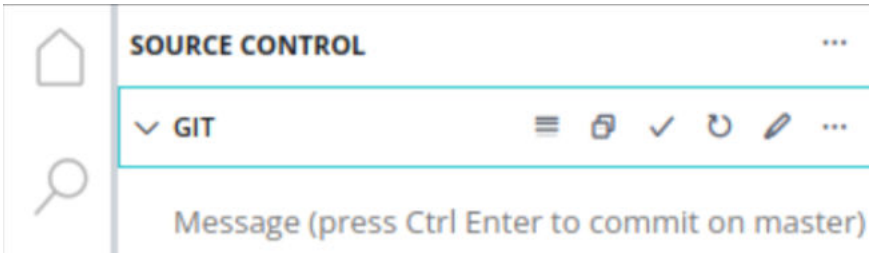


This GUI is equivalent to the following git command.

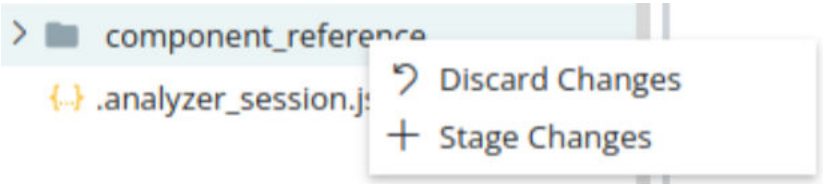
```
git add <file name>
```

Add Components for Source Control

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



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



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

   This GUI is equivalent to the following git command.

```
git add platform/
```

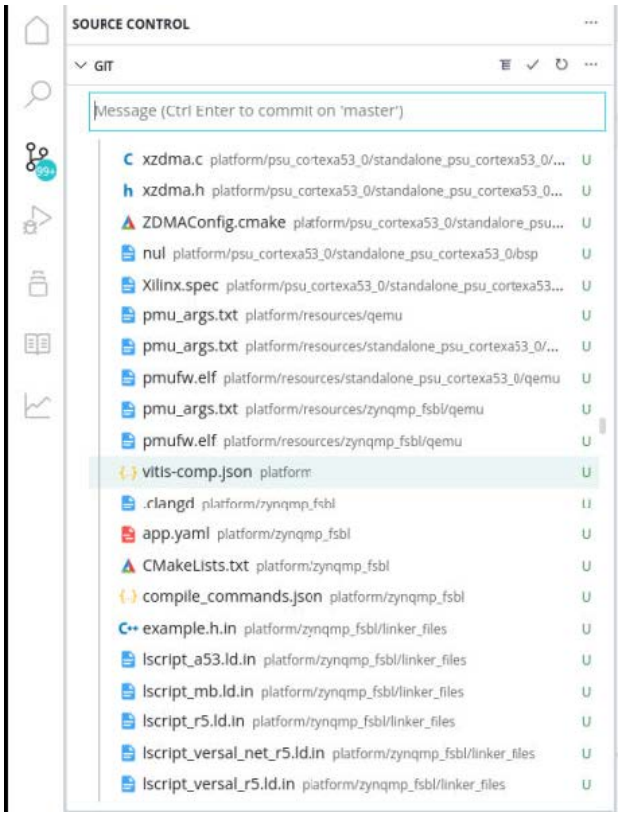
   ***Note*:** You need to select the components bound with system project together if the want to add system project for source control.

Commit the Changes

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

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

Figure 9: **Git View**



This GUI is equivalent to the following command:

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

Push the Project to the Remote Repository

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

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

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

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

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

  You can find your local project is in the remote repository.

Relative Path Support

A relative path is automatically selected if the imported sources or files share the same initial node in the path as the workspace. The following is an example:

- File Path: `/local/drive/source/app.cpp`
- Workspace Path: `/local/drive/workspace`

The relative path, based on the component location, is automatically selected and used during development:

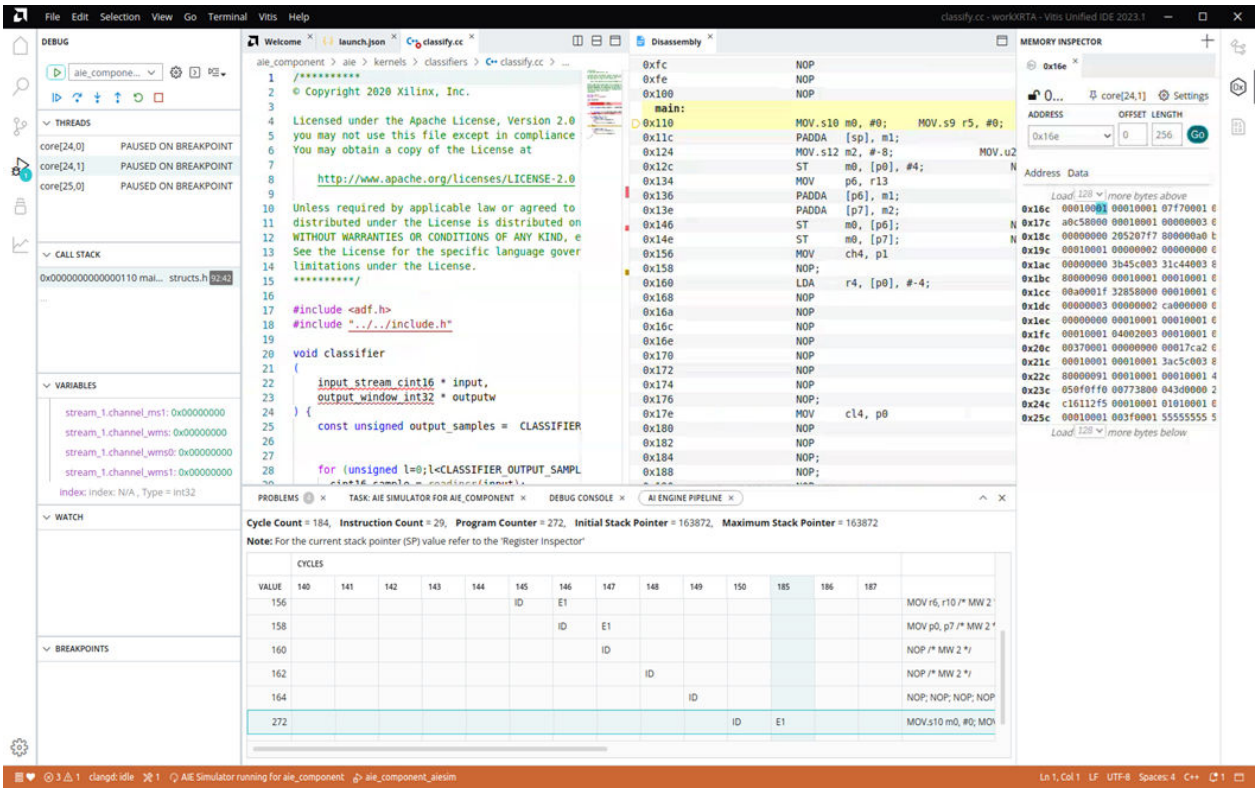
```
$COMPONENT_LOCATION/../../sources/app.cpp
```

***Note*:** When moving the workspace or committing workspace to source control tools, users should keep the relative path relationship.

Debug View

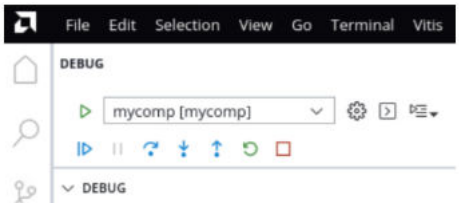
The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values. The AI Engine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

Figure 10: **AIE Component Debug View**



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

Figure 11: **Debug Control Panel**



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

- **Breakpoints:**

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

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

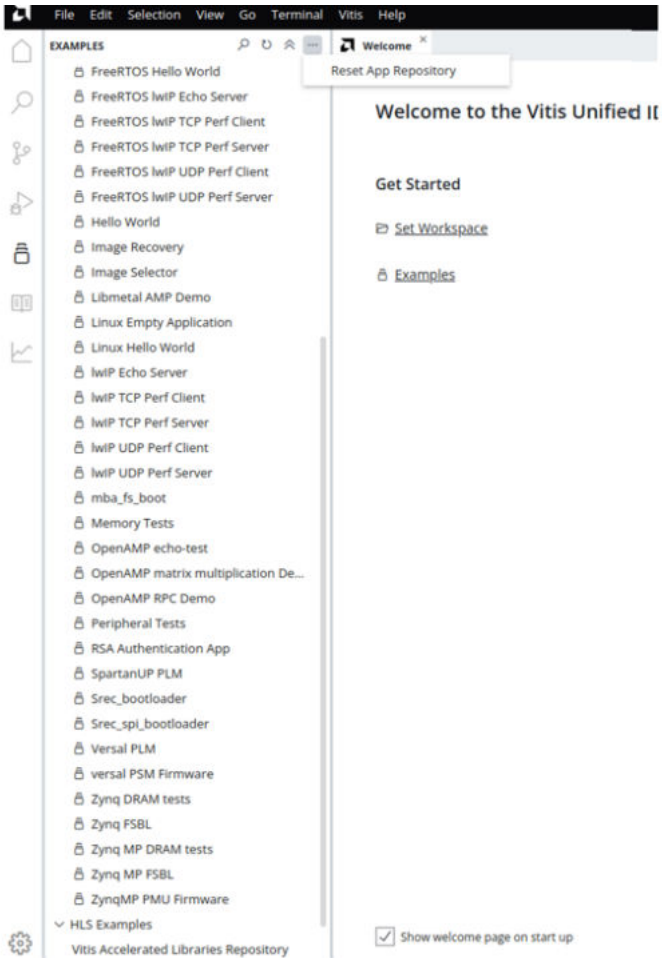
- **Source Code Editor:** The Source Code view is opened when Debug is launched from the Flow Navigator or Launch Configuration view.
- **Memory Inspector:** You can manually open the Memory Inspector. The Memory Inspector displays the content of specific memory addresses.
- **Register Inspector:** You can manually open the Register Inspector. The Viewing Registers shows the registers of the Cortex-A72 when a breakpoint is triggered in the Application Component source code, and the AI Engine when a breakpoint is triggered in the AI Engine kernel.
- **Disassembly View:** You can open the Disassembly view from the Source Code window right-click menu.
- **Debug Console:** Displays the transcript of the debug process, and any messages received from the tested application.

***Note*:** All the prints from MDM (MicroBlaze/ V Debug Module) UART are shown in Debug Console.

Example View

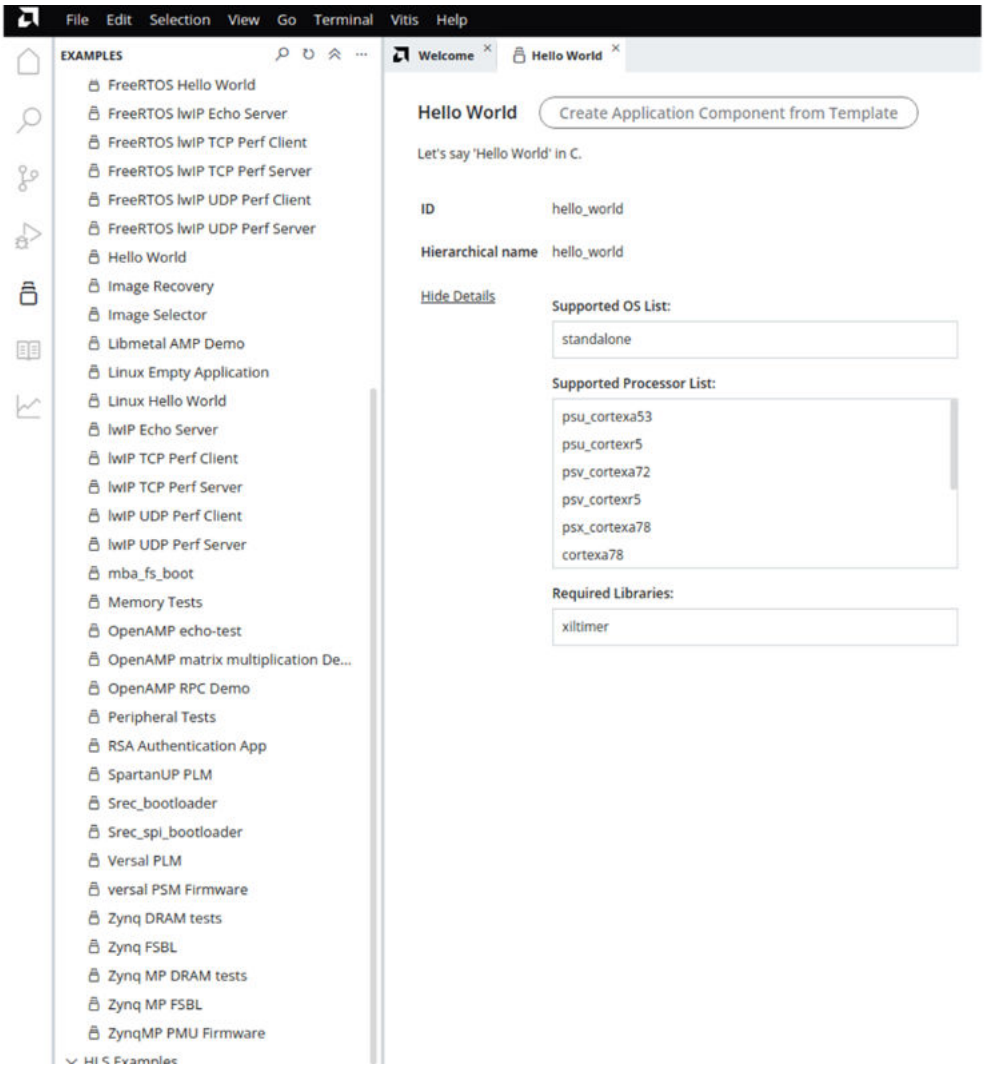
You can access the Examples View from the left side panel or from **View → Examples**, or by using **Ctrl + Shift + R** shortcut keys. In this view, you can create example System projects to explore the tool, and manage example repositories.

Figure 12: **Examples View**



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

Figure 13: **Example View Supported Devices**



Code View and Smart Editor

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

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

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

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

Figure 14: **GUI Format**

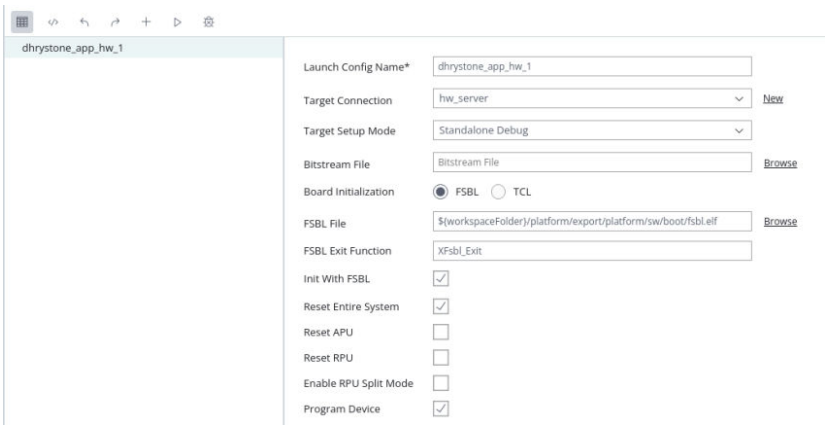
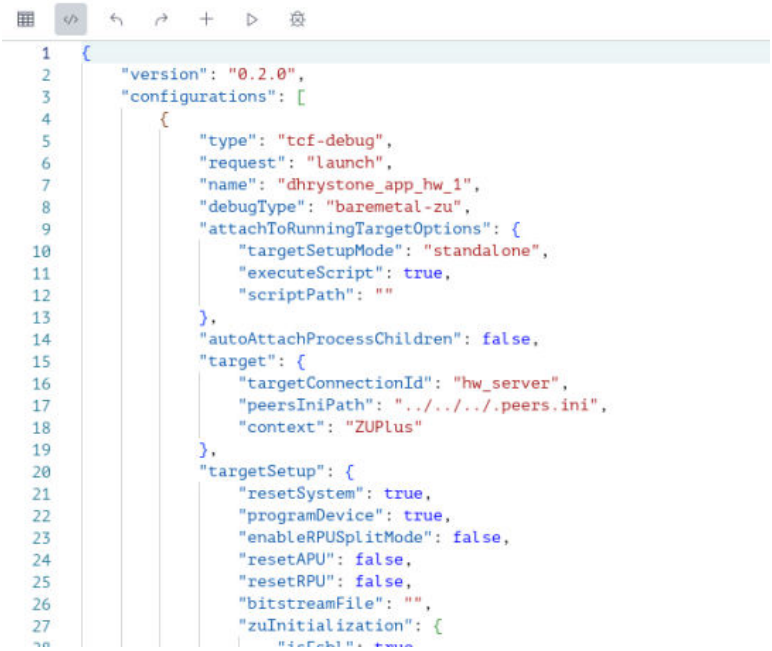


Figure 15: **Text Format**



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

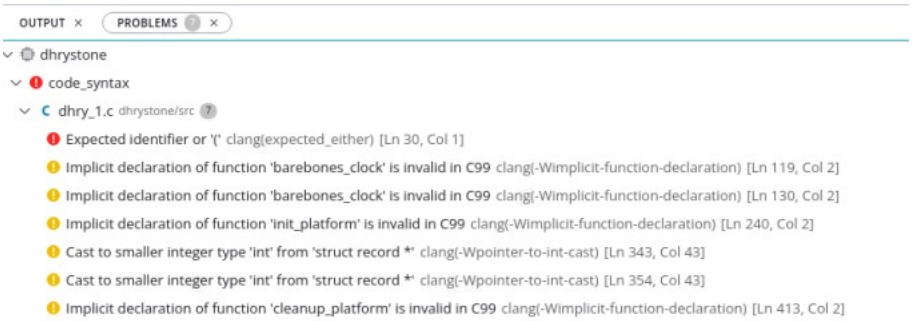
Issue and Source Code Cross-probing

You can quickly jump to source code from errors in output console by pressing **Ctrl** and left-clicking an issue simultaneously.

Problem View

All the warning and error messages from AMD Vitis™ output channel would displayed in problem pane. You can quickly navigate to the source of the error or warning in problem pane by clicking the error or warning message.

Figure 16: **Problem View**



Parallel Compiling

The Vitis Unified IDE provides fast responses to actions. It employs non-blocking build commands that allow you continue working and run multiple builds at the same time.

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

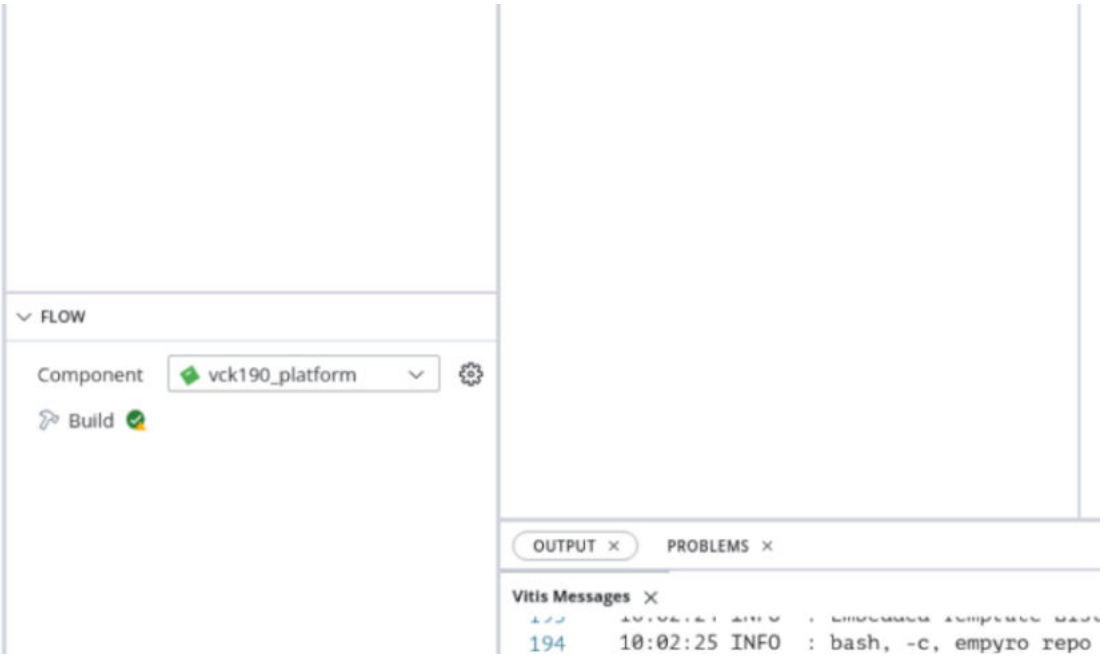
The application building job can also launch multiple components in parallel. Building jobs for the referenced components can be launched in parallel.

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

Notification for File Change

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

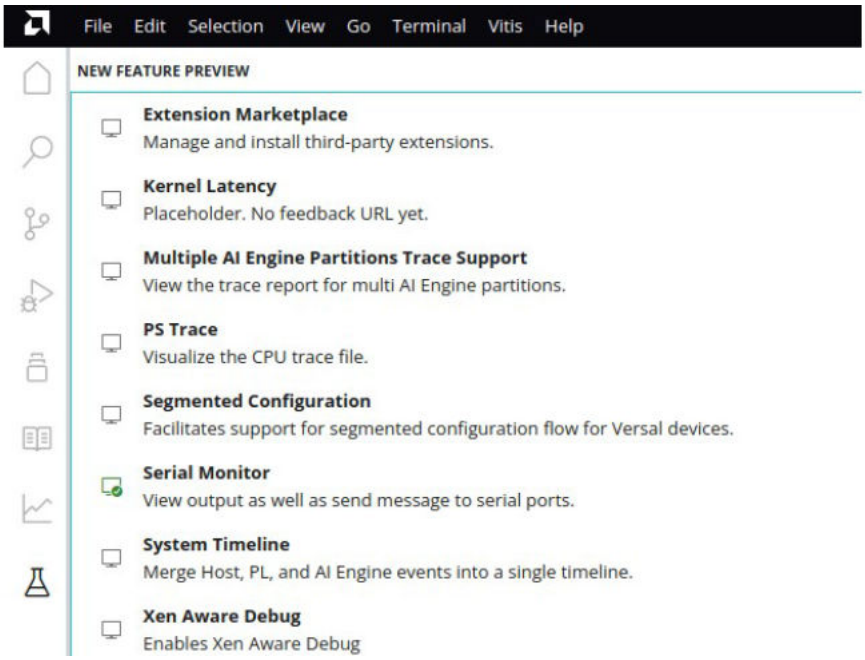
Figure 17: **Out-of-date Notification for Components**



New Feature Preview

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

Figure 18: **New Feature Preview**



Before using this feature, you need to enable it. Click **Enable** to activate the feature. You can also click **Feedback** to give your insights.

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

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

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

Workspace Journal

The Workspace Journal is a log file that records back end commands corresponding to the actions triggered by the user after opening the workspace. It is a good approach to check the corresponding commands used by GUI and recreate the workspace as well. To open the Journal file, go to menu **Vitis**-> **Workspace Journal**.

You can run following command to recreate the projects with the journal file in the current workspace:

```
vitis -s workspace_journal.py
```

***Note*:** In `workspace_journal.py`, relative paths are used whenever possible to improve portability. However, there are certain instances where absolute paths are necessary. Prior to executing the script, ensure that the specified source and destination paths in `workspace_journal.py` are correctly configured and accessible.

External Lopper Support

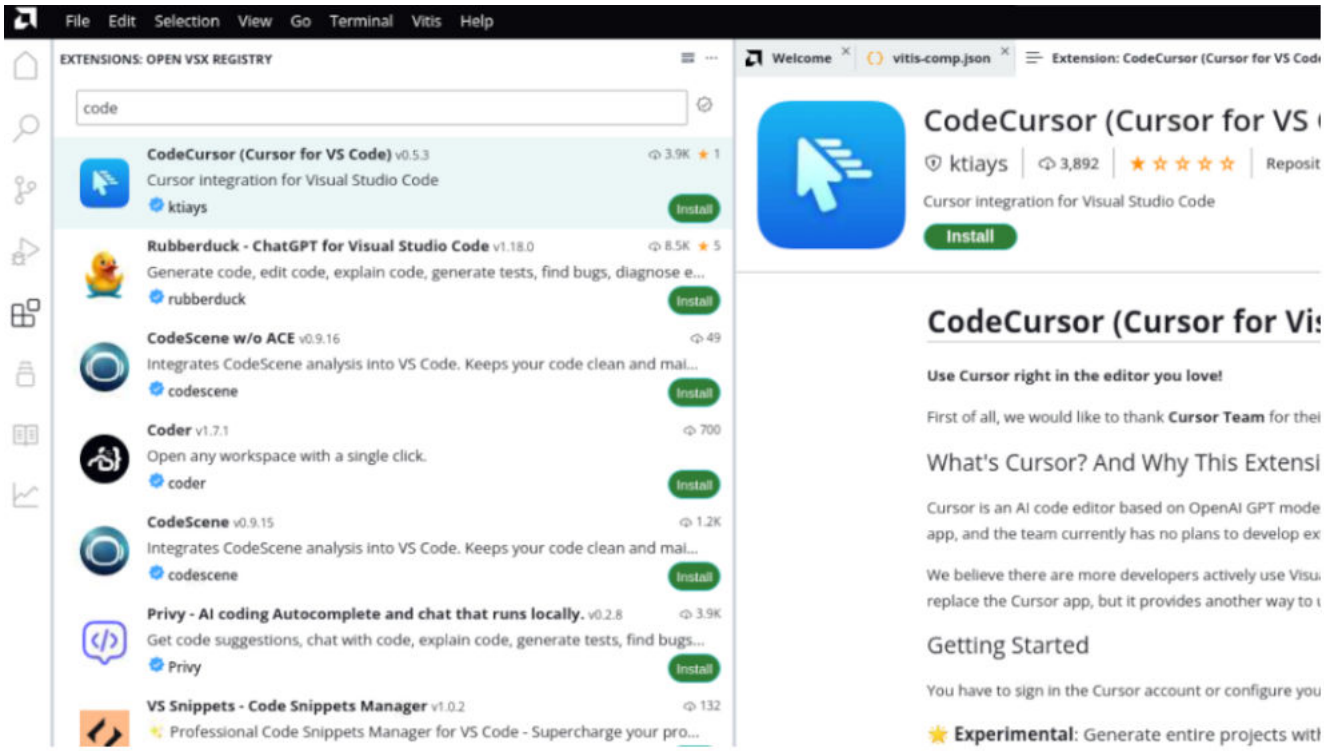
Lopper is a backend Python utility used by AMD Vitis™ to extract hardware metadata from the system device tree. Although included in the Vitis installation, Lopper is also available on GitHub. To allow developers to make changes and apply bug fixes to Lopper without modifying the Vitis installation, an environment variable is provided to enable the use of an external Lopper. To use an external version of Lopper, run the following command:

```
export VITIS_LOPPER_INSTALL_LOC="path/to/Lopper"
```

Extensions

You can explore and install extensions directly within the Vitis Unified IDE. Access the Extensions view by clicking the dedicated **Extensions** icon along the left edge of the Vitis Unified IDE. This will show you a list of recommended, installed or built-in extensions on the Extension Marketplace.

Figure 19: **Extensions**



Each extension in the list includes a brief description. You can select the extension item to display the extension's details.

***Note*:** The extension marketplace feature serves as a convenient portal for exploring and installing extensions from Open VSX. Open VSX is an open-source registry for VS Code extensions. This service is operated by the Eclipse Foundation. We do not conduct screenings for the quality or safety of any extensions.

IMPORTANT! *By using extensions from the extension marketplace, you understand and accept the risks involved. We are not liable for any damages, losses, or inconveniences arising from the use of this feature. Carefully consider these factors before accessing and utilizing any extensions.*

***Note*:** This is a preview feature that is currently in the early access phase. This means that the feature is still under development and may not be fully functional or stable. By accessing and using this feature, you acknowledge and accept the following:

- Limited functionality: This feature can have limited functionality compared to fully released features. Be aware that your experience with this feature may not be optimal.
- Potential instability: This feature is still being tested and refined. As a result, it may be prone to crashes, errors, or unexpected behavior. We recommend using this feature with caution and understand that it may not always work as intended.
- Feedback and improvements: We encourage you to report any issues, bugs, or suggestions you encounter while using this feature. Your input will help us enhance their performance and stability.

- No guarantees: This feature is provided on an "as-is" basis, without any warranties or guarantees of any kind, whether expressed or implied. We do not guarantee that this feature will be released in their current form. We reserve the right to modify, suspend, or discontinue this feature without prior notice.
- Use at your own risk: By using this feature, you understand and accept the risks involved. We shall not be held liable for any damages, losses, or inconveniences arising from the use of this feature.

Serial Monitor

The Serial Monitor extension provides a serial monitor to view output as well as send message to serial ports. This is often useful when testing or debugging programs on embedded devices.

You can open the serial monitor from the Vitis menu and selecting **Serial Monitor**.

Segmented Configuration

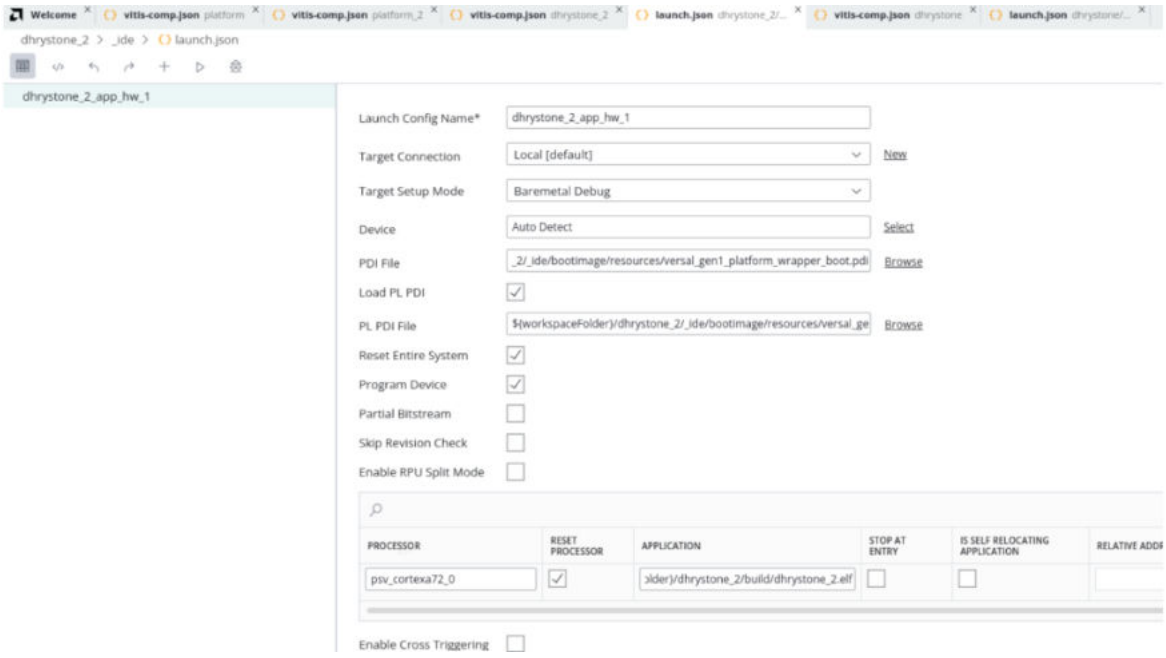
The Segmented Configuration Flow is a development process for AMD Versal™ that allows you to separately download or program the boot PDI (CIPS and NOC initialization) and PL PDI (FPGA side initialization image) components. It is beneficial for:

- Quick software boot
- Minimizing local boot flash requirements
- Loading PL PDI Image by runtime request

Even though Segmented Configuration is usually used in Linux designs, this GUI feature provides capabilities to run or debug standalone applications if the hardware has already been configured as Segmented Configuration.

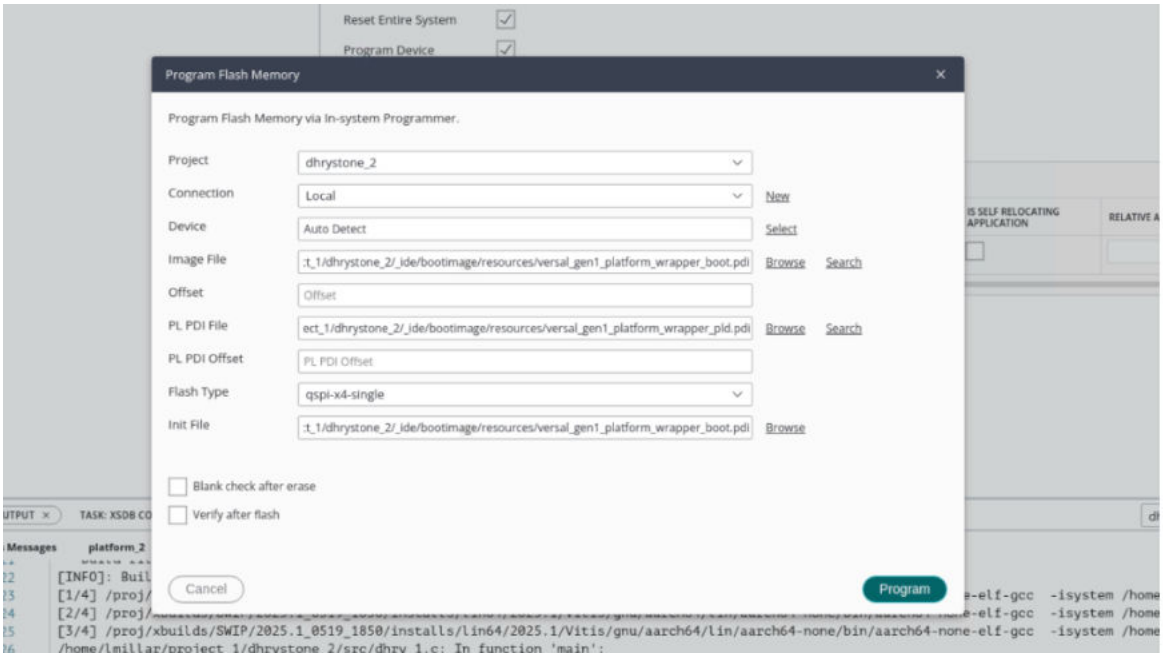
After enabling this feature, one option is provided to select the PL PDI when editing launch configuration for either running or debugging for standalone applications.

Figure 20: **Select PL PDI**



The Program Flash utility provides an additional input box for assigning PL PDI in case you wish to program all contents to Flash.

Figure 21: **Program Flash utility**



You can select to program or download the PL PDI accordingly.

***Note*:** This is a preview feature that is currently in the early access phase. This means that the feature is still under development and may not be fully functional or stable. By accessing and using this feature, you acknowledge and accept the following:

- Limited functionality: This feature can have limited functionality compared to fully released features. Be aware that your experience with this feature may not be optimal.
- Potential instability: This feature is still being tested and refined. As a result, it may be prone to crashes, errors, or unexpected behavior. We recommend using this feature with caution and understand that it may not always work as intended.
- Feedback and improvements: We encourage you to report any issues, bugs, or suggestions you encounter while using this feature. Your input will help us enhance their performance and stability.
- No guarantees: This feature is provided on an "as-is" basis, without any warranties or guarantees of any kind, whether expressed or implied. We do not guarantee that this feature will be released in their current form. We reserve the right to modify, suspend, or discontinue this feature without prior notice.
- Use at your own risk: By using this feature, you understand and accept the risks involved. We shall not be held liable for any damages, losses, or inconveniences arising from the use of this feature.

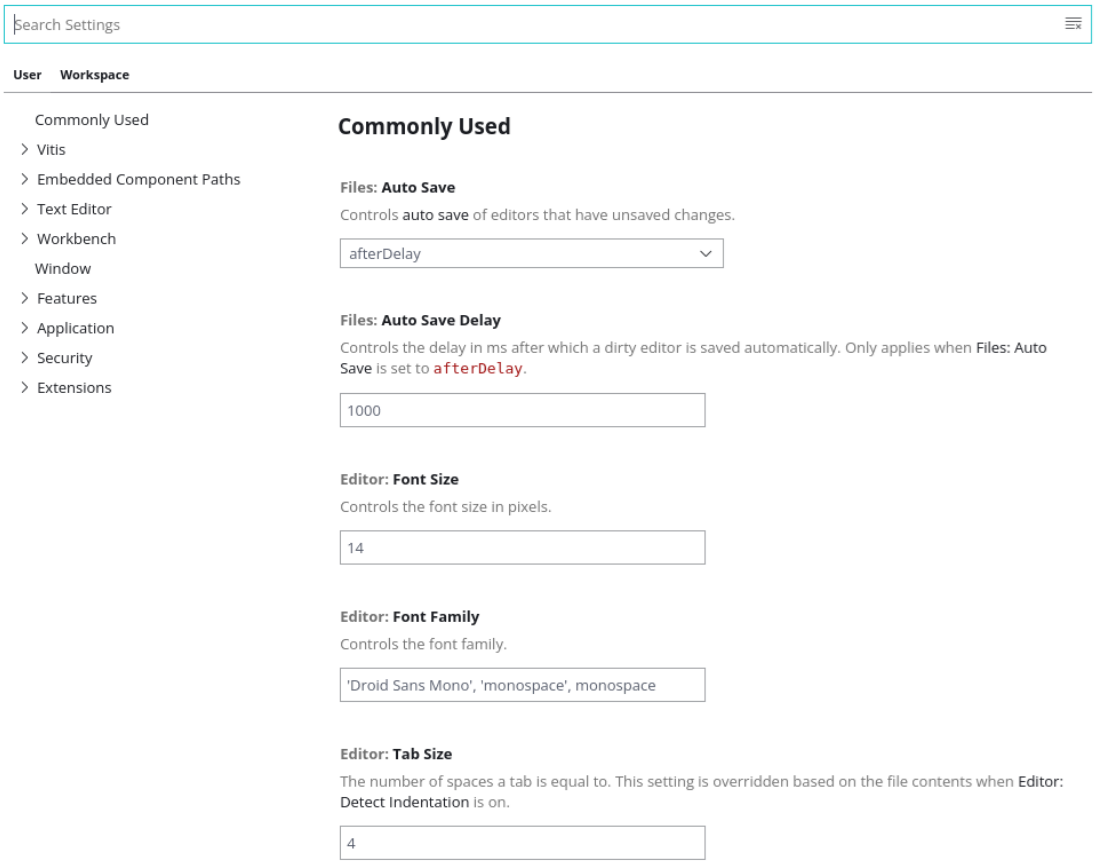
Preferences

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

Settings

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

Figure 22: **Common Preferences**



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

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

Keyboard Shortcuts, Command Palette, and Quick Find

Keyboard Shortcuts

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

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

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

Command Palette

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

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

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

Quick Find

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

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

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

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

Develop

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

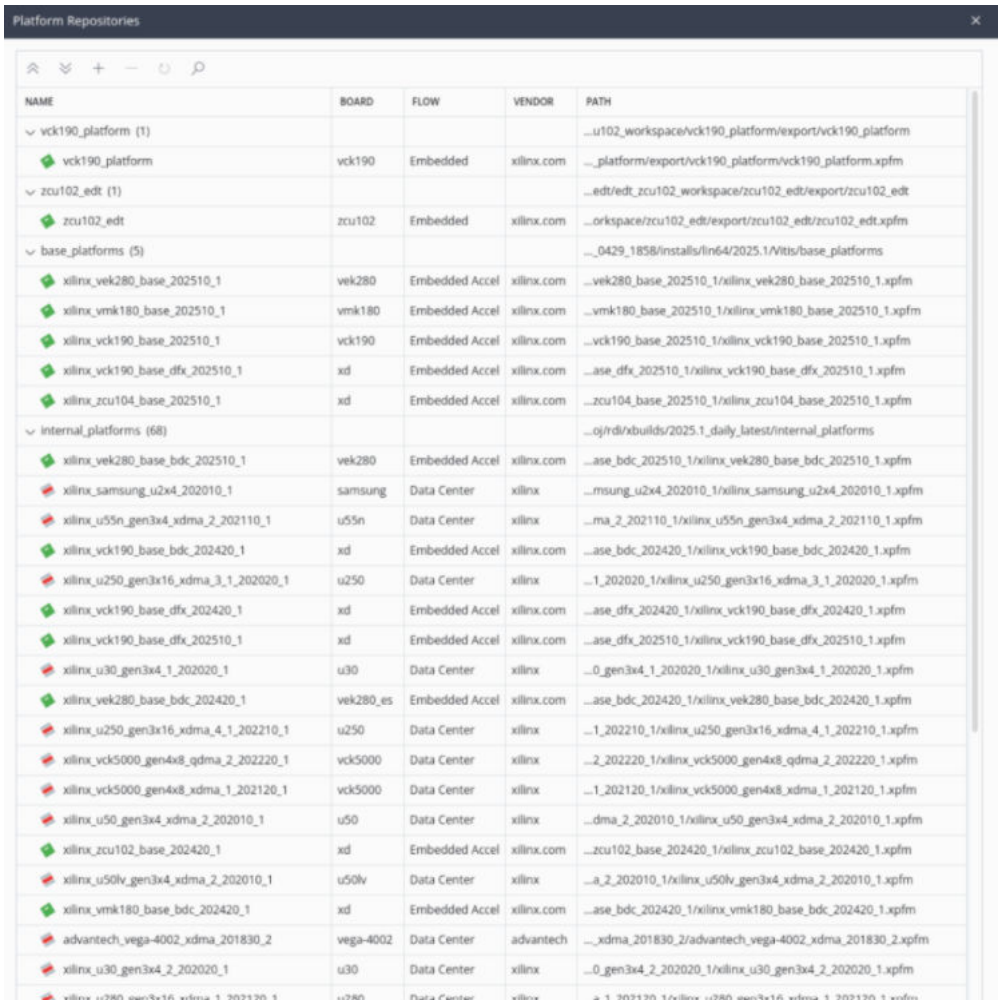
Managing Platforms and Platform Repositories

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

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

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

Figure 23: **Platform Repositories**



Target Platform

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

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

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

Creating a Hardware Design (XSA File)

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

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

The generic steps are as follows:

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

Creating a Platform Component from XSA

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

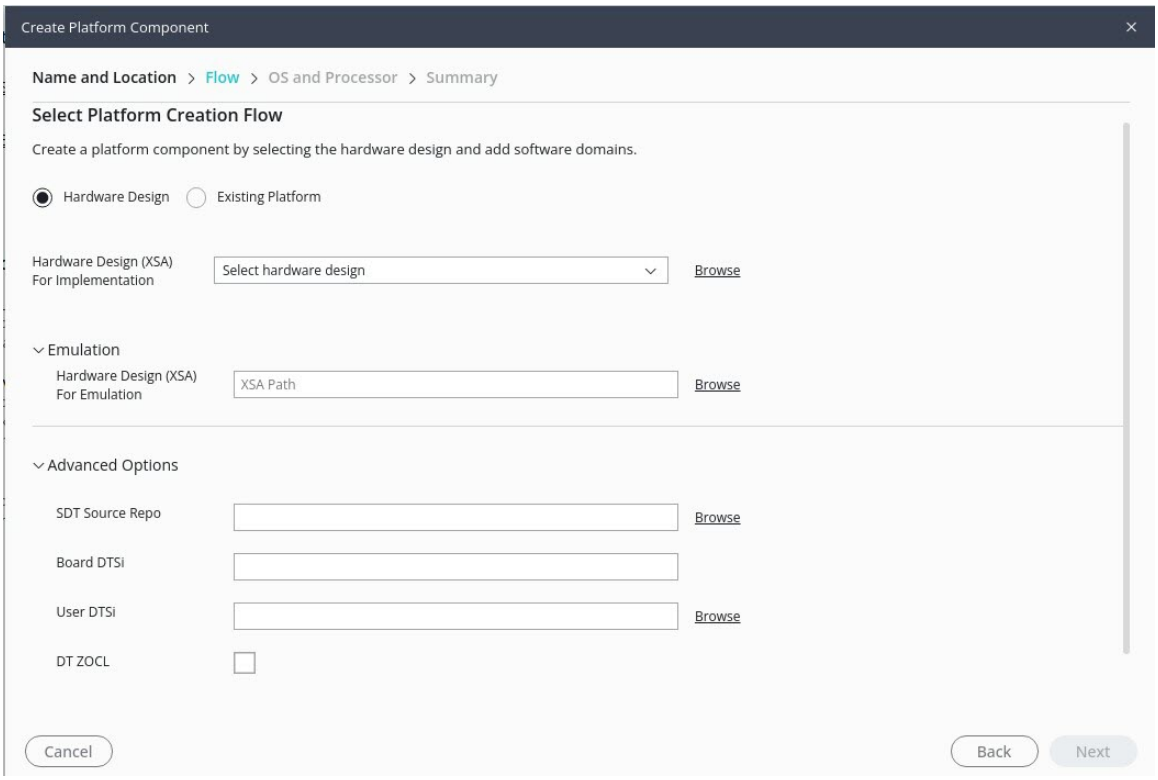
1. Click the File option in the Vitis Unified IDE and select **New Component → Platform**.

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

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

Figure 24: **Create Platform Component**



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

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

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

Note: SDTgen is a built-in tool within the Vitis Unified IDE. It is responsible for reading the XSA file and generating the SDT (system device tree) files. For more information on SDT, refer to System Device Tree. Another built-in tool, Lopper, then parses the SDT and generates the corresponding BSP (Board Support Package) file. To facilitate user debugging of this process and to take advantage of the SDT flow, Advanced options are introduced. These options allow you to switch to a custom version SDTgen tool. The following are the uses of the three advanced options:

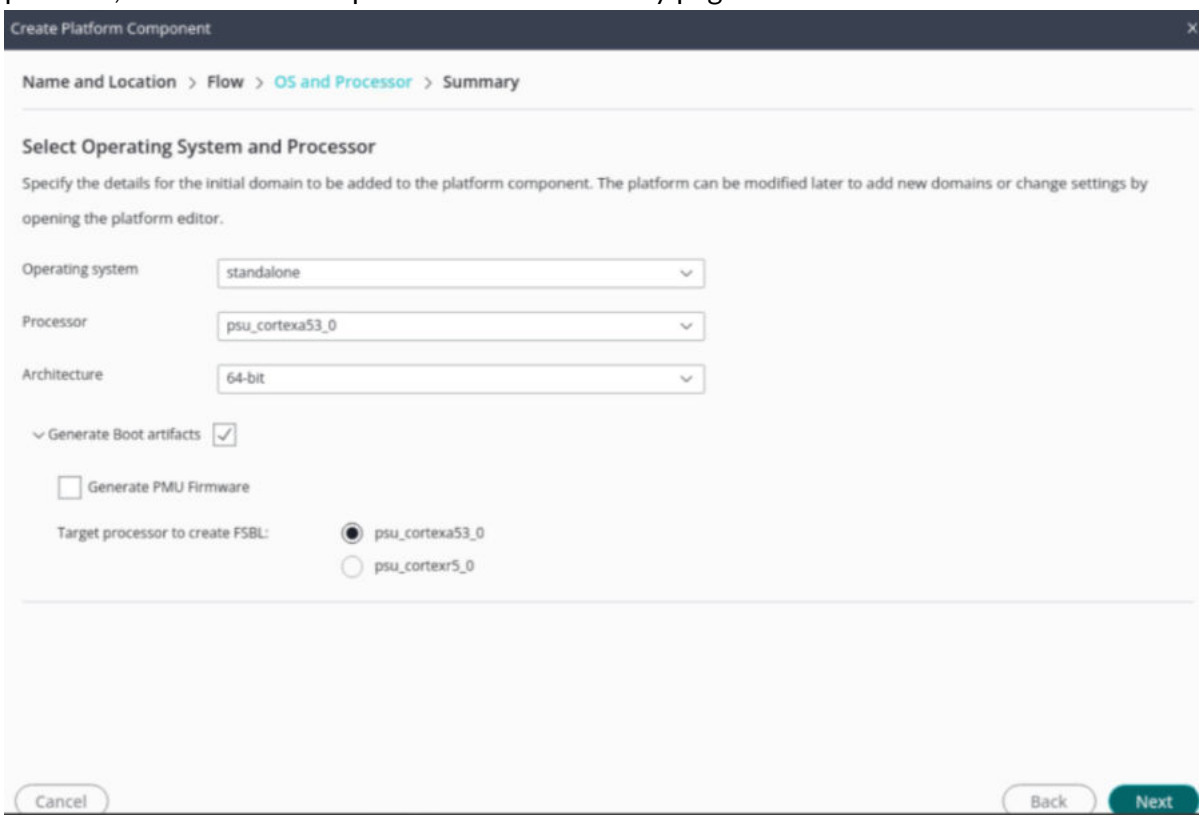
- *SDT Source REPO*: input customer version SDTgen REPO

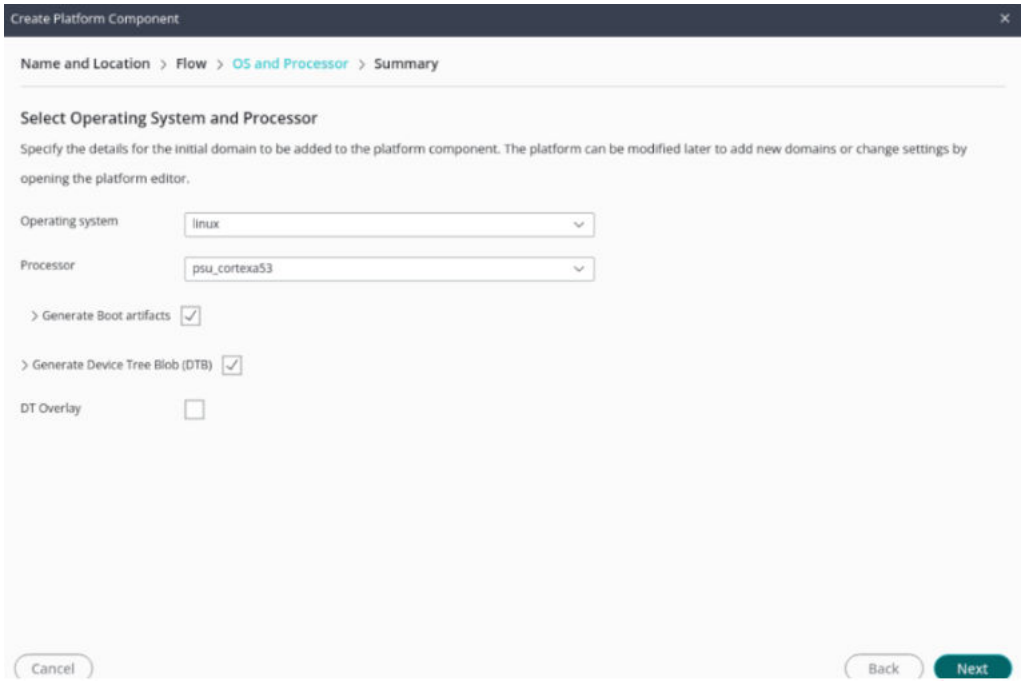
  Note: If you encounter issues related to the system device tree or the SDTgen tool, you can use the updated version of the SDTgen tool to help debug the issue. For more information on the system device tree, refer to System Device Tree Repo.

- *Board DTS*: input the board name to obtain the corresponding board DTS file. For more information, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144) for the corresponding board name.

. *User DTS*: to specify the user device tree file (DTSI)

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





***Note*:** When you enable the **Generate Boot Artifacts** and **Generate PMU Firmware** options, the tool automatically generates both the FSBL (First Stage Boot Loader) and PMU (Platform Management Unit) firmware components required for your platform if the domain is *Standalone*. Observe that the Generate Boot Artifacts option is not available for Versal adaptive SoC, as only the PDI file is required.

***Note*:** When you enable the **Generate Device Tree Blob** (DTB) option, the tool automatically generates the Device Tree Blob for your platform if the domain is Linux.

***Note*:** DT Overlay is only for extensible platform or fixed XSA generated by V++ tool to generate device tree blob overlay.

For option settings, refer to the following table.

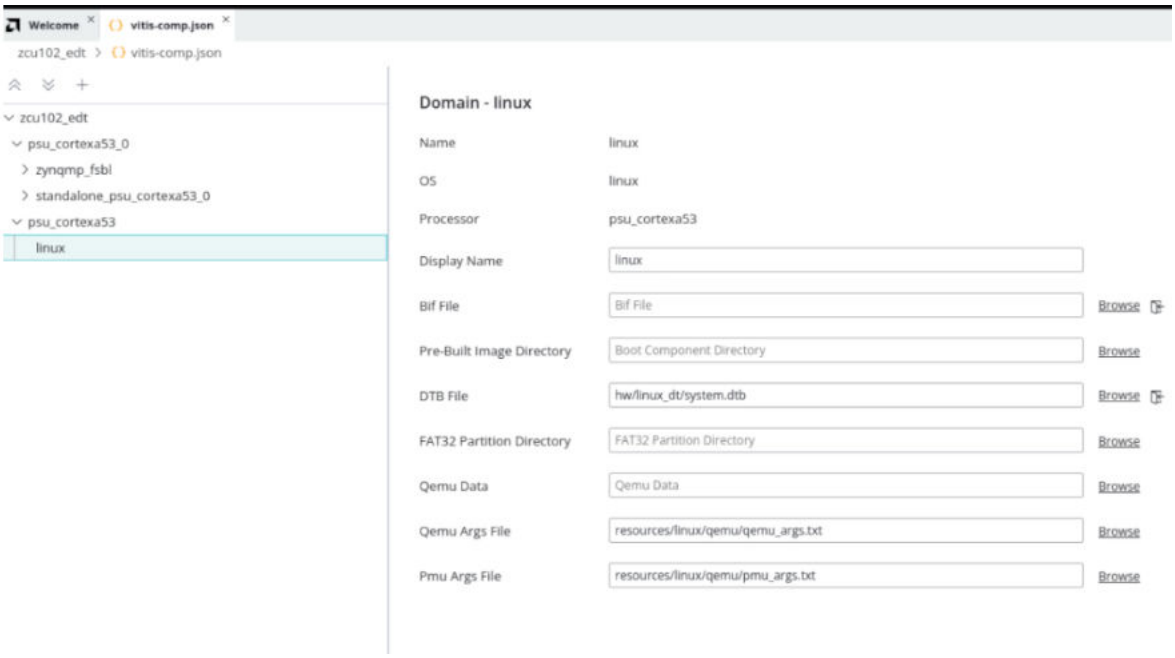
Table 7: **Options Setting for Different Device Family**

Device Family	Generate Boot Artifacts	PMU	DTB	DTB Overlay
Zynq	The FSBL is necessary for device initialization and should be enabled.	No option	DTB is only required for Linux system	The DTB overlay is required only for Linux systems when loading certain devices after the system is up and running
ZynqMP	The FSBL is necessary for device initialization and should be enabled.	The PMU is not required for the standalone domain but is essential for Linux systems to manage high-security functions.	DTB is only required for Linux system	The DTB overlay is required only for Linux systems when loading certain devices after the system is up and running
Versal	No option	No option	DTB is only required for Linux system	The DTB overlay is required only for Linux systems when loading certain devices after the system is up and running

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

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

Figure 25: **Domain - linux**



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

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

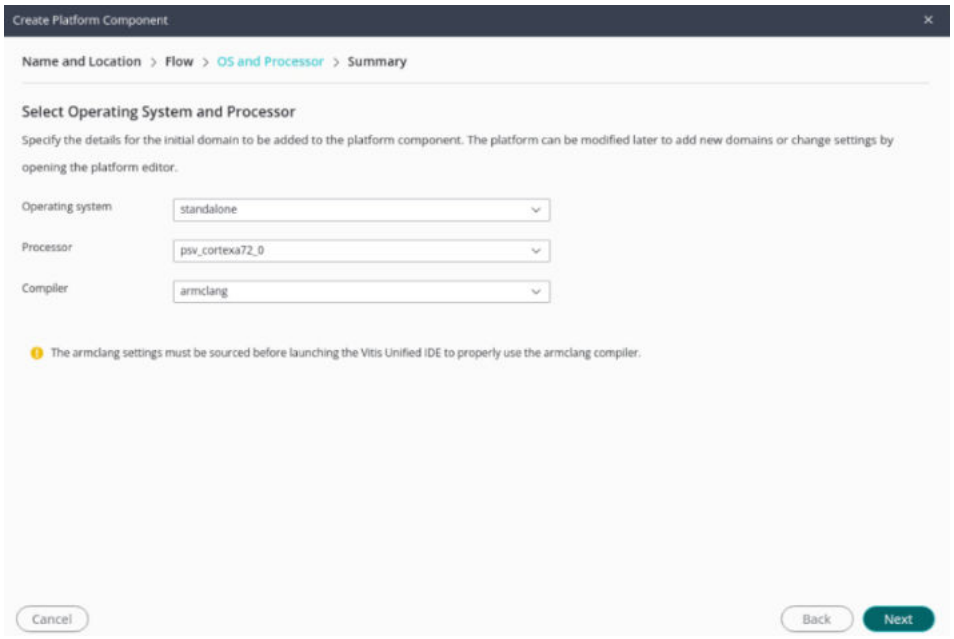
After confirming the settings, you can build the Platform component by selecting it in the Flow Navigator and selecting the **Build** command. After compilation is completed, the tool populates the platform and its related software and hardware components in the `Output` folder of the Platform component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

Note the following compiler support for third-party toolchains:

- ARMClang support for Versal A72 and R5 processors
- ARMCC support for the Zynq A9 processor
- IAR support for the Zynq MP R5 processor and Zynq A9 processor

IMPORTANT! *These compilers are not included with the Vitis Unified IDE installation. Therefore, you will need to install the compiler yourself and add it to the `$PATH` environment variable. The following is an example of selecting the 'ARMCLANG' compiler when creating a platform for the vck190.*

Figure 26: **OS and Processor Summary**



If using the CLI flow, the command looks like the following:

```
platform = client.create_platform_component(name = "platform",hw_design
= "vck190",os = "standalone",cpu = "psv_cortexa72_0",domain_name
= "standalone_psv_cortexa72_0",generate_dtb = False,advanced_options =
advanced_options,compiler = "armclang")
```

TIP: *Open the workspace journal to see the respective CLI commands executed in the Vitis Unified IDE.*

***Note*:** The flow is the same for all compilers.

Customizing a Pre-Built Platform

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

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

4. Click **Next**.
5. In the Platform Component page, select **Existing platform**. Select the desired platform and click **Next** to review the platform component summary.

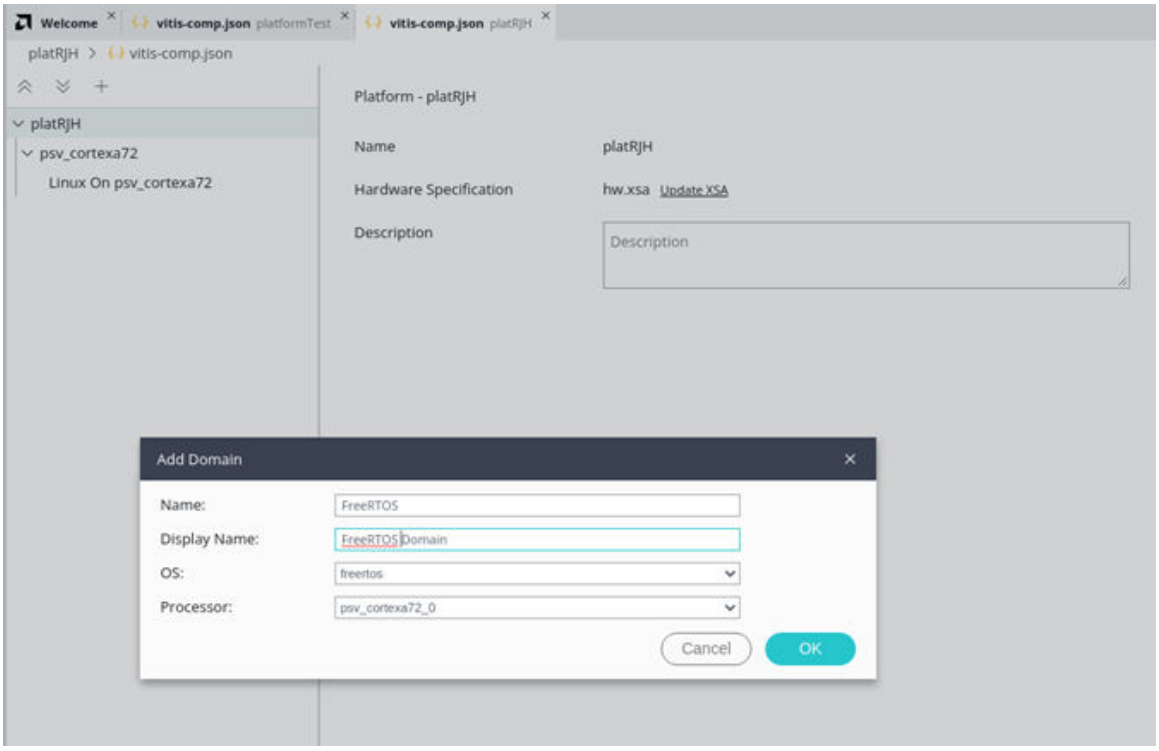
   Note: Make sure the platform to customize is visible in the platform repository list. If it is not, click **+** to add the platform path to the platform repository.
6. Click **Finish**. You can now modify the new platform in the workspace as any other platform.

Adding a Domain to an Existing Platform

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

1. In the current workspace, expand the Platform component in the Component Explorer to open the Settings folder and select the `vitis-comp.json` file.

   Note: To create a Platform component refer to Creating a Platform Component from XSA.
2. Click the **+** button to Add Domain.



3. Specify the Name and Display name.
4. For the OS select Linux, FreeRTOS, Standalone.
5. Select from the available Processors. The selection of Processor changes based on the selected OS.

6. Click **OK** to add the domain to the Platform.

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

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

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

Configuring a Domain

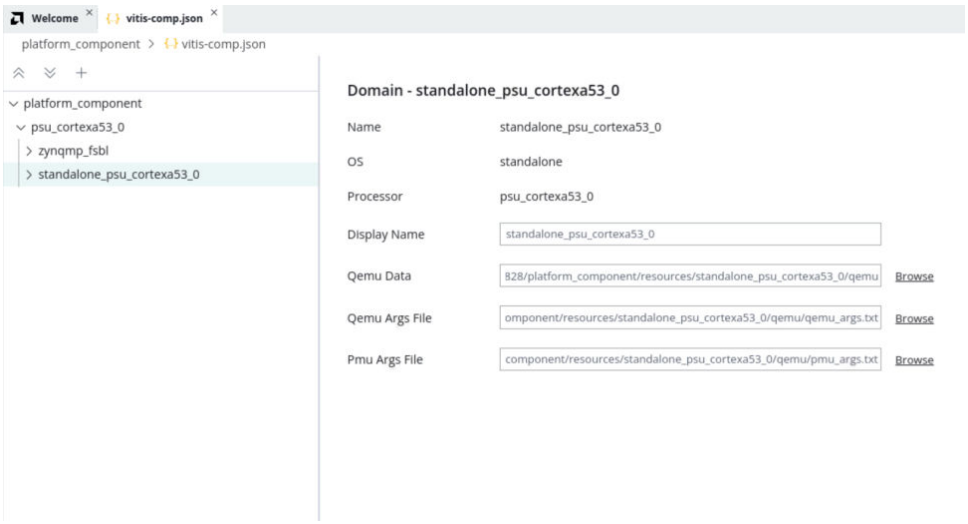
There are different kinds of domains, the standalone domain being the most frequently used. Each domain has an associated BSP which can be configured extensively. Additionally, the domain overview page includes extra settings for the domain.

Domain Overview Page

Standalone and FreeRTOS Domain

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

Figure 27: **Standalone Domain Overview Page**

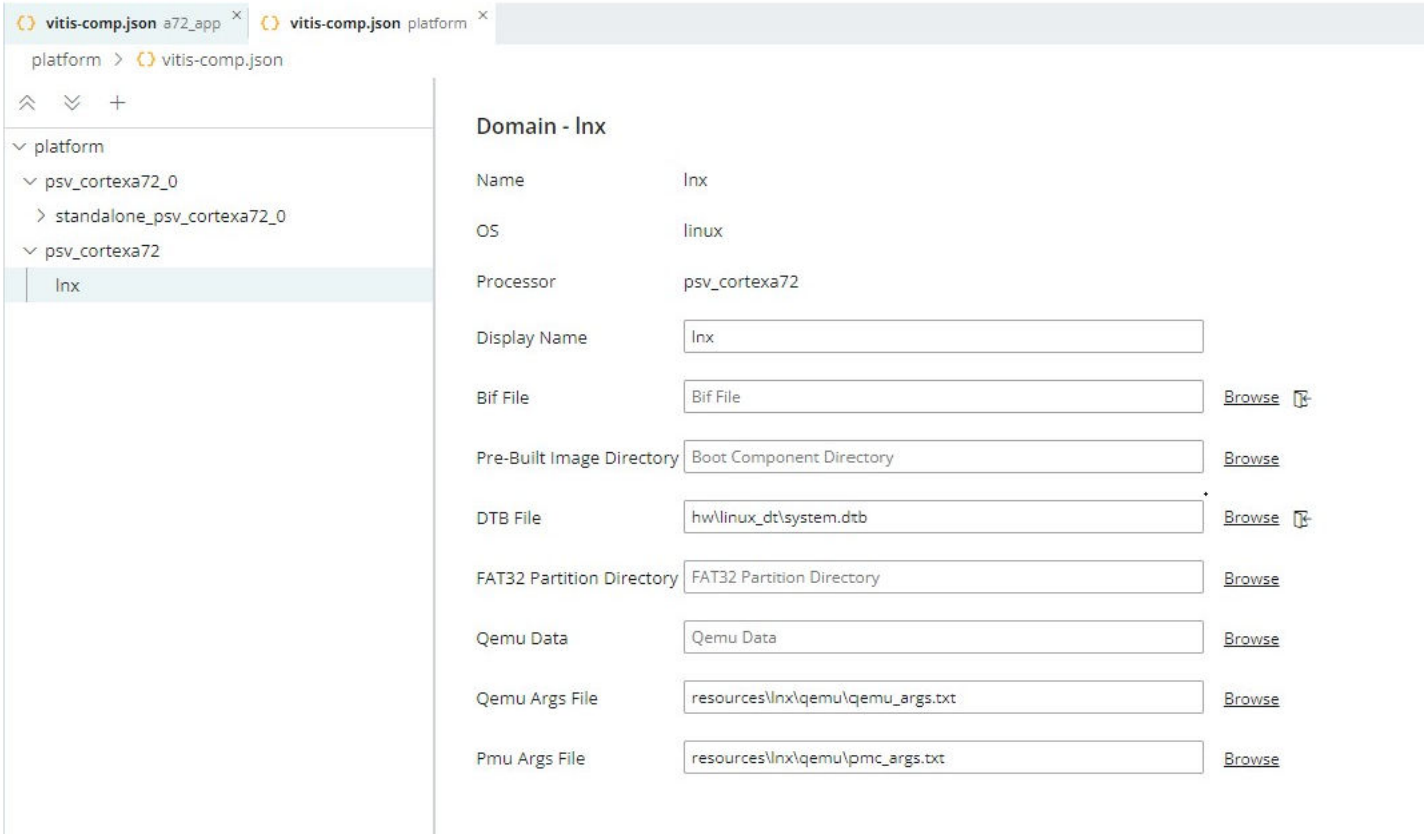


Linux Domain

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

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

Figure 28: **Linux Domain Overview Page**



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

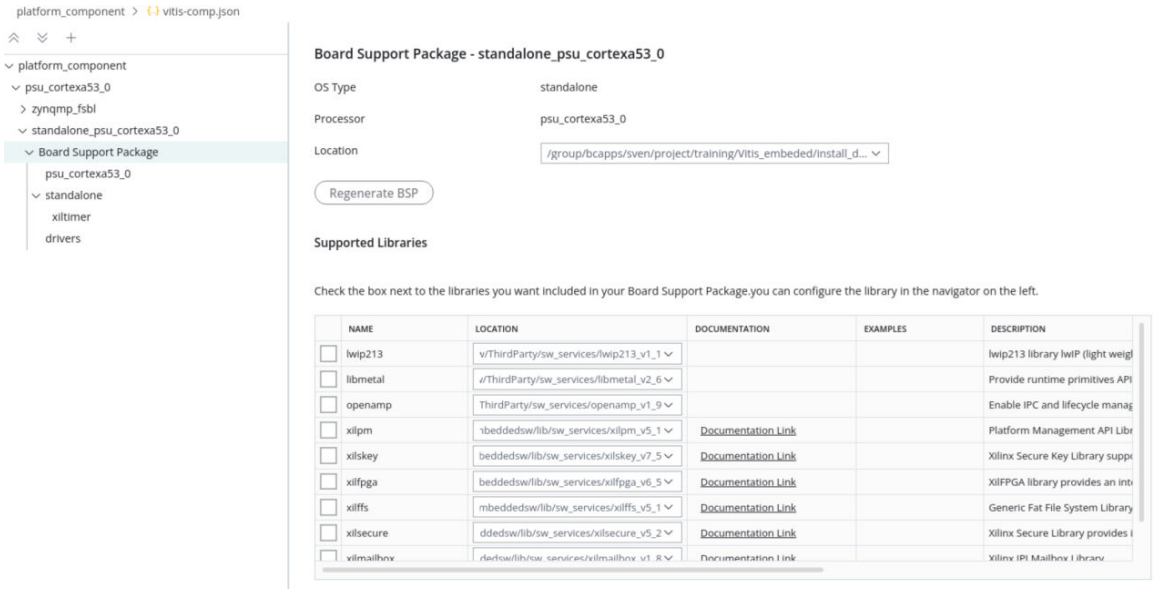
Board Support Package Settings Page

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

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

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

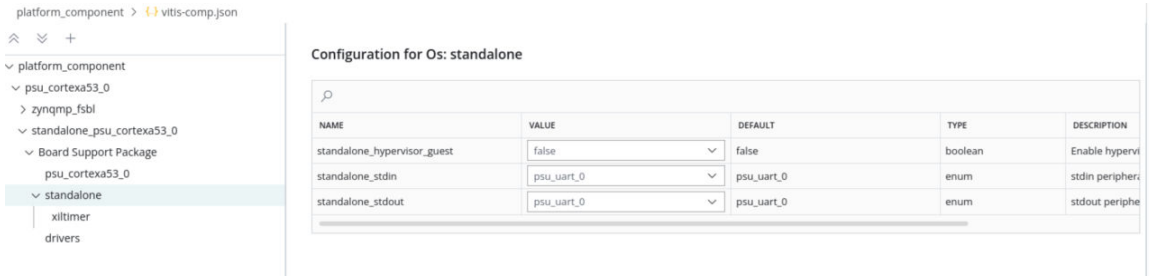
Figure 29: **Overview**



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

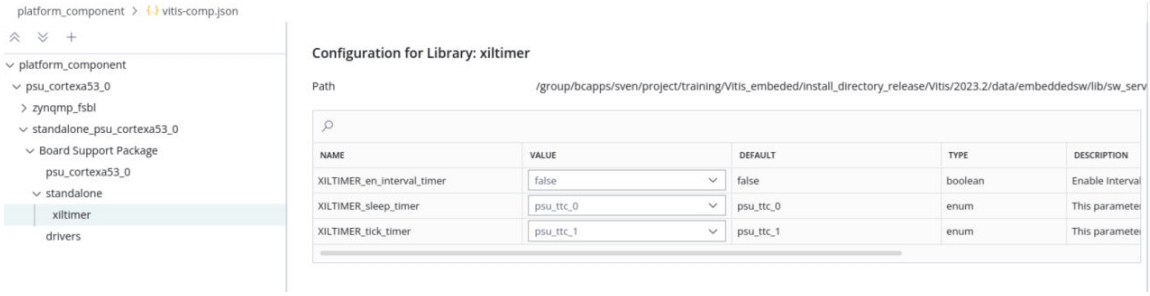
Figure 30: **Processor Parameter Configuration**

Figure 31: **OS Parameter Configuration**



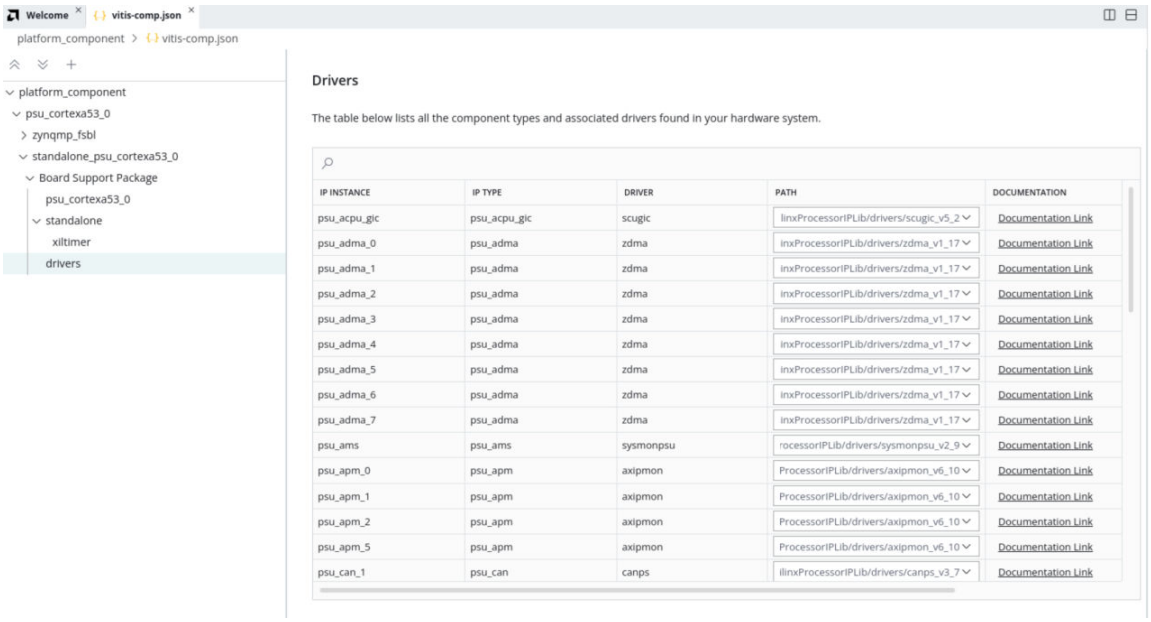
The library settings page enables you to configure the parameters of each library enabled in the library page.

Figure 32: **Library Configuration**



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

Figure 33: **Drivers**

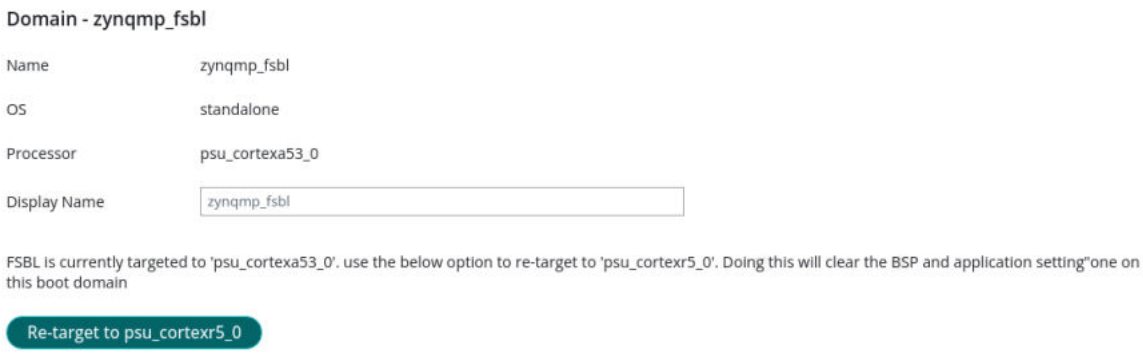


The build settings section lists the toolchain selected to build the BSP as well as some extra configuration settings.

Switching FSBL Targeting Processor

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

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



Modifying Source Code for FSBL

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

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

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

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

Modifying the Domain Sources (Driver and Library Code)

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

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

Creating a Software Repository

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

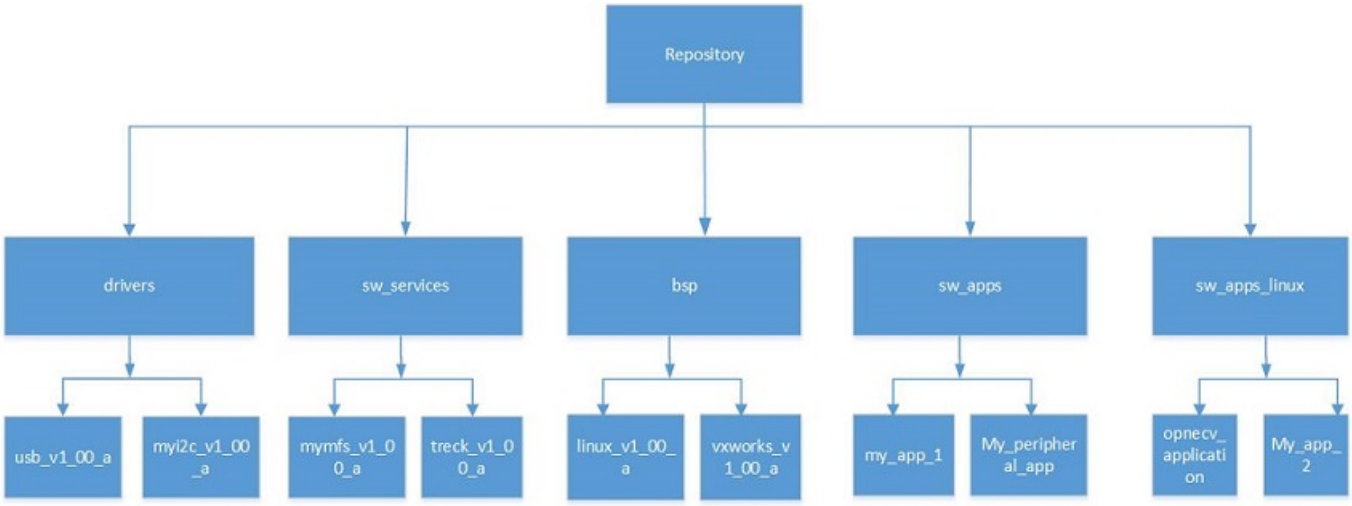
Your Vitis software platform workspace can point to multiple software repositories. The scope of the software repository can be global (available across all workspaces) or local (available only to the current workspace). Components found in any local software repositories added to a Vitis software platform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

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

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

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

Figure 34: **Repository Structure**



Adding the Software Repository

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

Resetting BSP Sources for a Domain

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

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

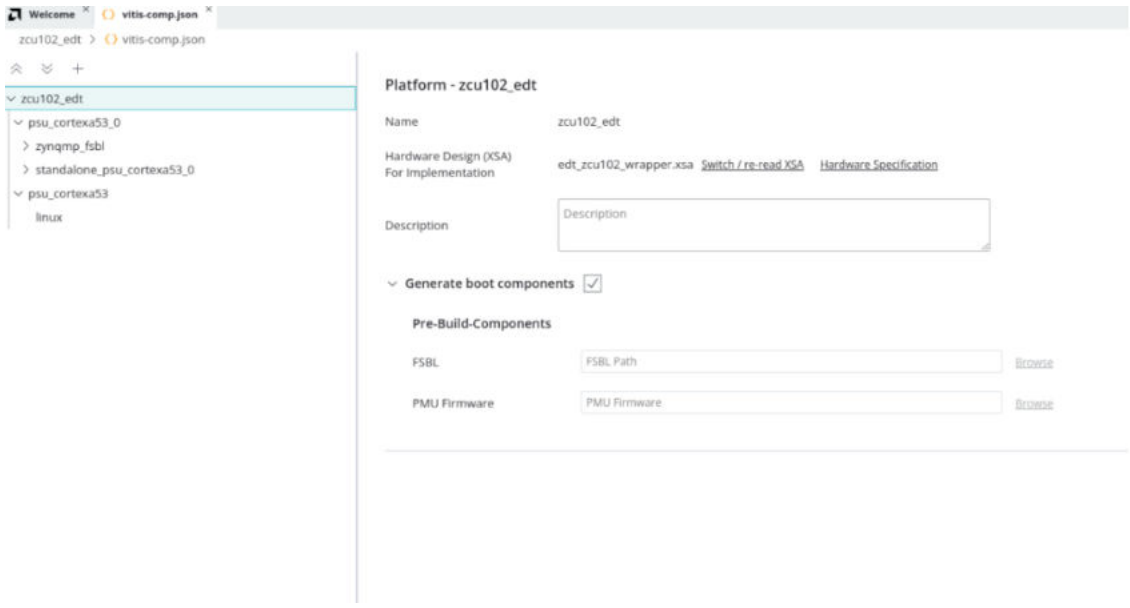
***Note*:** Only the source files are reverted back to their original state. The settings however, are retained.

Updating the Hardware Specification

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

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

1. Right-click the platform component in the component view, and expand Settings, open the **vitis-comp.json**.
2. Click **Switch / Re-Read XSA**.



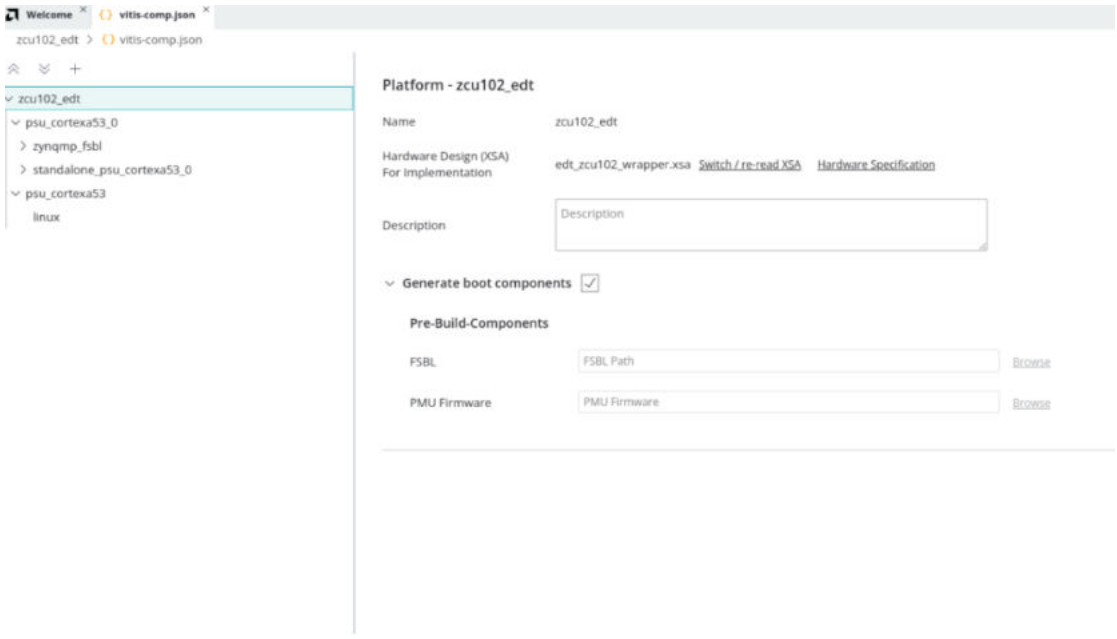
3. Specify the source hardware specification file in pop-up window.
4. Click **open** to select the XSA file.

Reading Hardware Specification

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

Execute the following steps to read the hardware information.

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



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

Applications

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

- **Targeted for Embedded Systems:** Embedded applications are intended to run on embedded hardware systems, which are typically resource-constrained and designed for specific tasks. These systems can range from small microcontrollers to more complex devices like embedded FPGAs or SoCs (System on Chips).

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

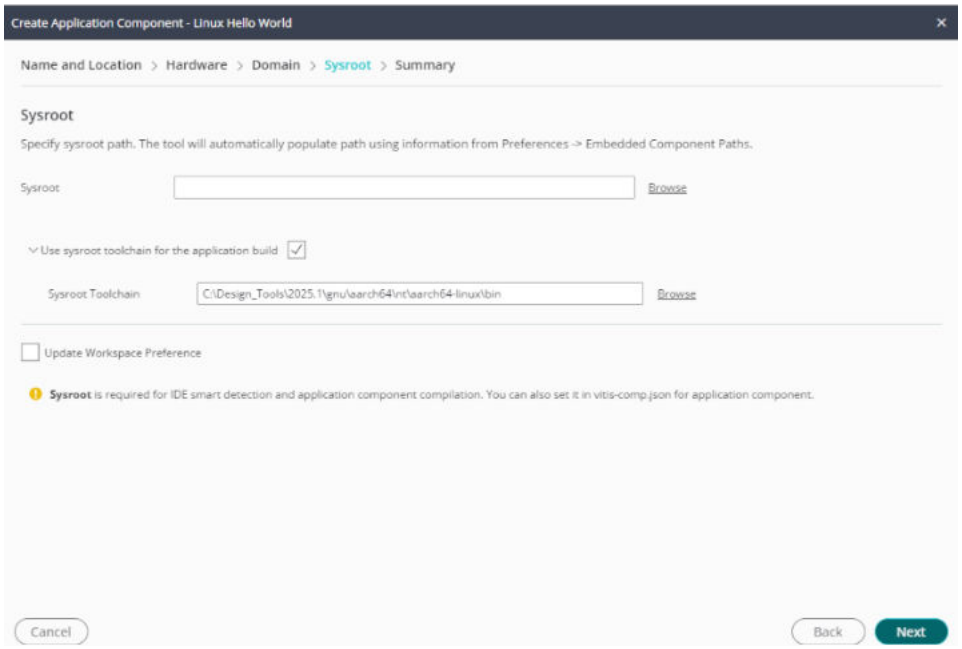
Creating application component

Creating an Application Component from Examples

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

1. Go to **View → Examples**, or click **File → New → Component From Example**, or click the toolbar on the left side or use **Ctrl + Shift + R** shortcut keys to open the Example view.
2. Click the example in the example list.
3. Click **Show details** to the requirement of the template.
4. Click **Create Application Component from Template**. Refer to Creating an Application Component for details.

When creating any Linux application component, you can select a third party sysroot toolchain in the GUI by selecting the checkbox and providing the path as seen below.



If `use sysroot toolchain` is disabled, then the default gcc compiler will be used. Furthermore, if the user passes a path that does not have any gcc or g++ compilers, then Vitis will throw an error.

For the CLI flow, add the following arguments to your `create_app_component` command:

```
use_sysroot_toolchain=True
sysroot_toolchain=<TOOLCHAIN_PATH>
```

Creating an Application Component

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

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

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

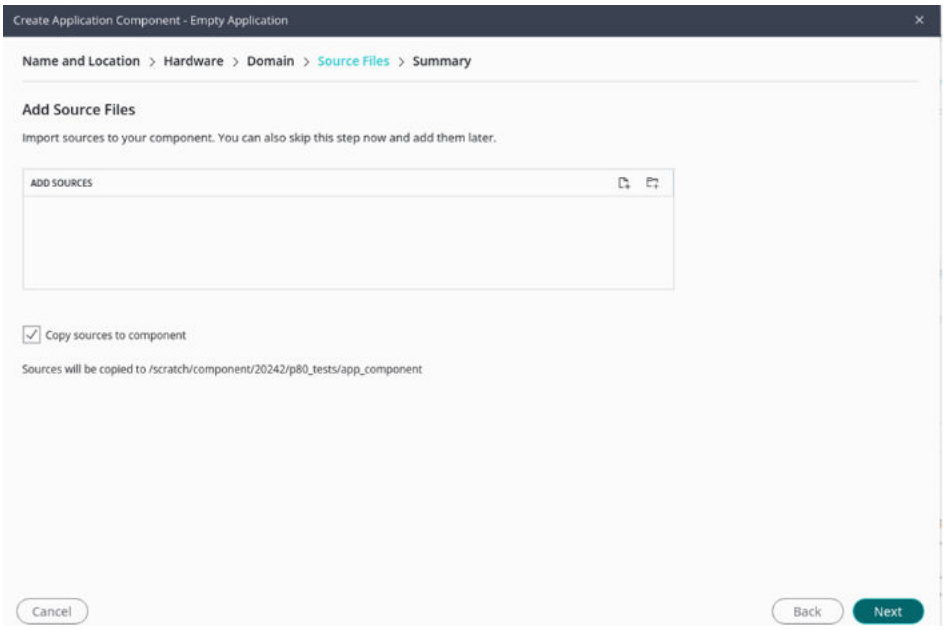
1. Input the Application component name and specify the location. Click **Next**
2. Select a fixed platform and select **Next**.

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

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

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

4. Click **Add files** or **Add folders** to import source files or folder for the application.



   ***Note*: Copy Source to Component** is selected by default. Deselect this option if you prefer to reference the external source file directly, rather than copying it into the workspace.

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

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

Creating a System Project

A system project can be created by following steps:

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

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

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

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

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

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

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

Managing Multiple Applications in a System Project Component

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

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

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

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

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

Building Projects

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

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

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

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

Build Configuration Settings

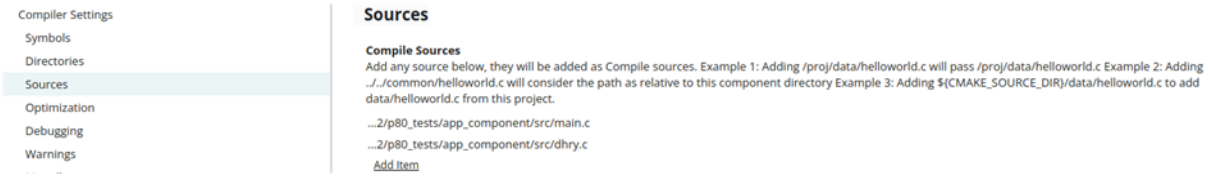
You could modify the build configurations manually. Each build configuration can customize:

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

Manage Source Code

This section explains how to manage the source code.

1. Click on your application component in the Component view and expand **Settings**. Open the `vitis-comp.json` file, and click on `UserConfig.cmake` to configure the compiler. Alternatively, you can hover your mouse over the application component, a settings button appears—click it to open the `vitis-comp.json` file. You can also open the `UserConfig.cmake` from the Settings folder directory.
2. Under Compiler Settings, select **Sources** section.



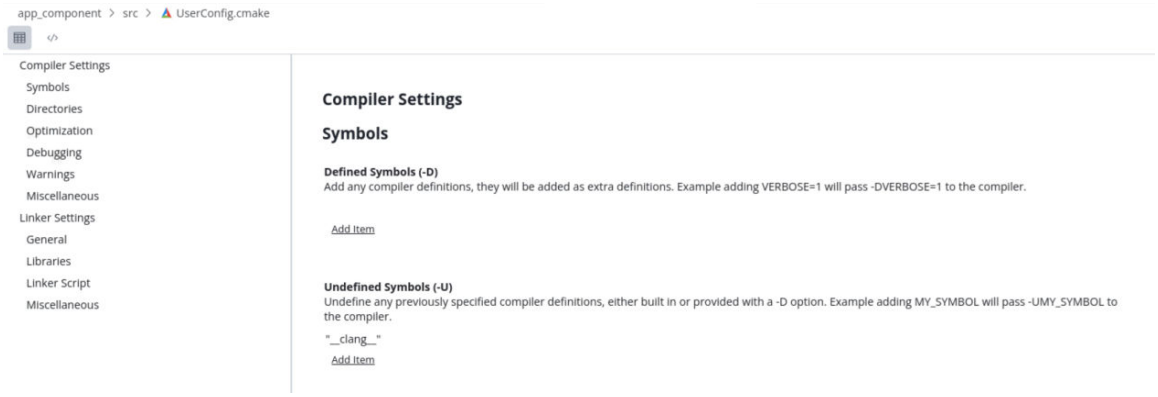
3. Click **Add Item** to add source code for your application.
4. You can also hove your mouse over the name of source file and select to remove it.

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

Adding Symbols or Definitions

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

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



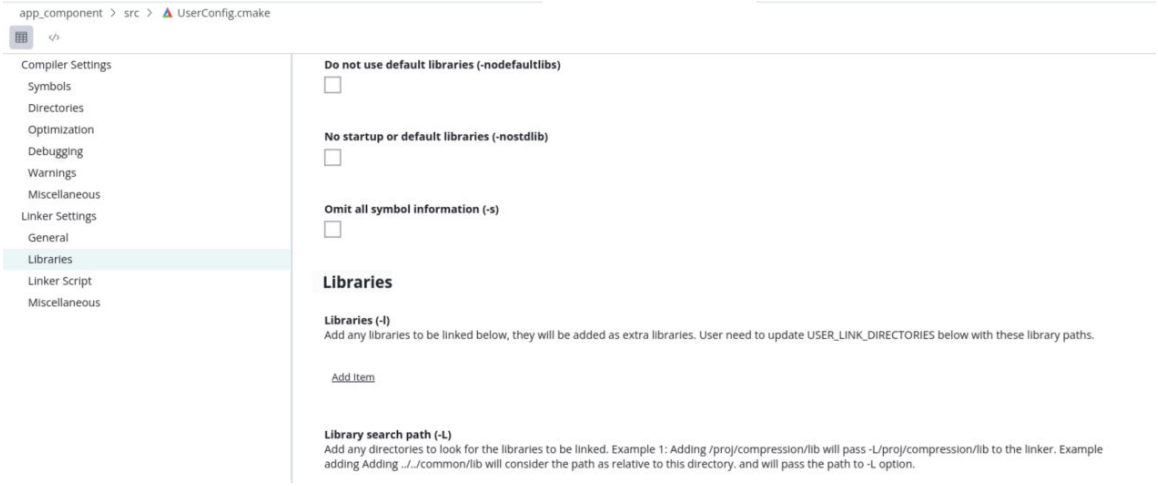
***Note*:** `UserConfig.cmake` supports GUI format and text format. You can click these icons **</>** to access the source editor to view the source code and modify the settings in text format.

Adding Libraries and Library Paths

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

To set properties for your Application project:

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



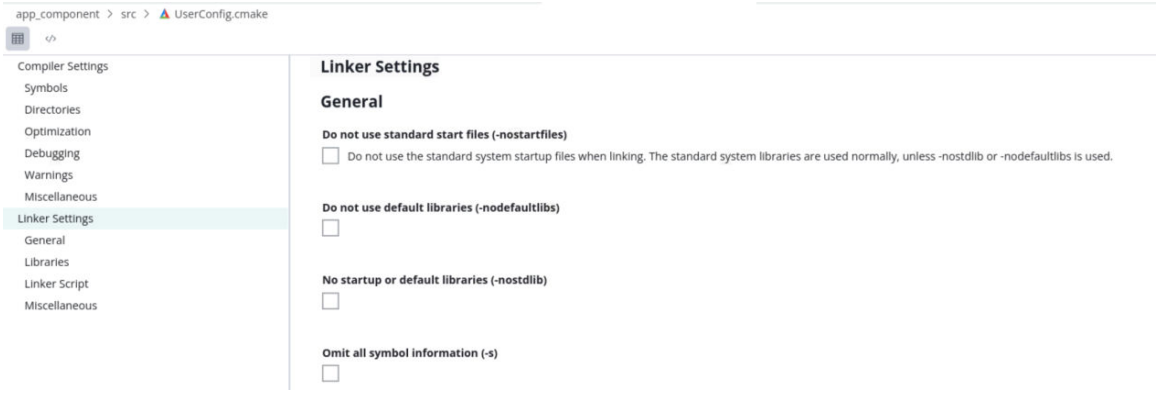
***Note*:** `UserConfig.cmake` supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

Specifying the Linker Options

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

To set properties for your application component:

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



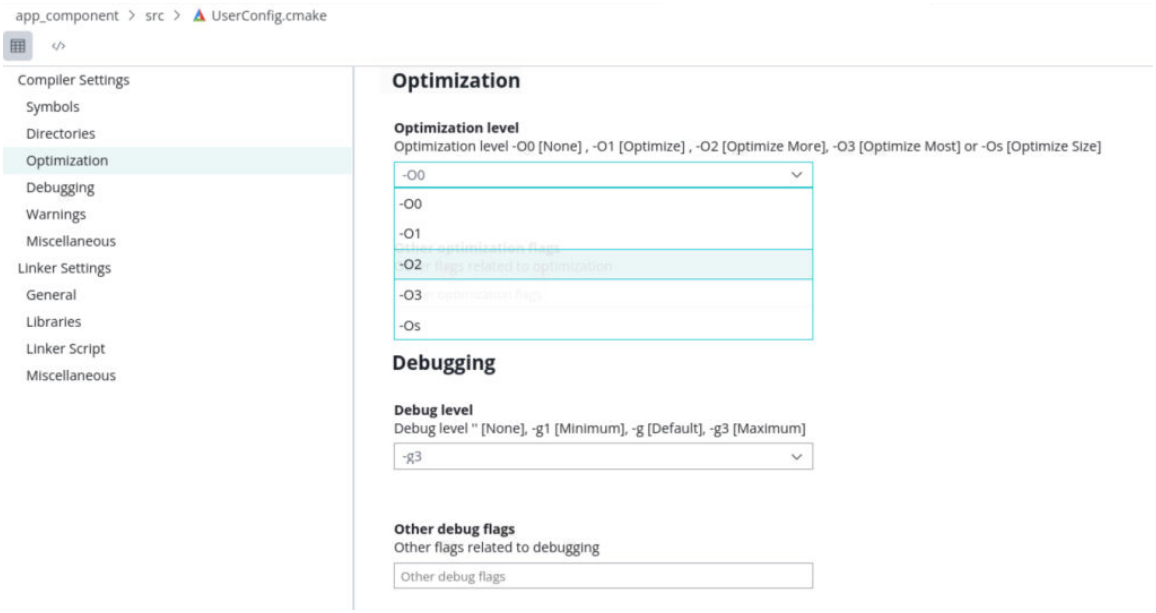
***Note*: UserConfig.cmake** supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

Specifying Debug and Optimization Compiler Flags

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

To set properties for your project:

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



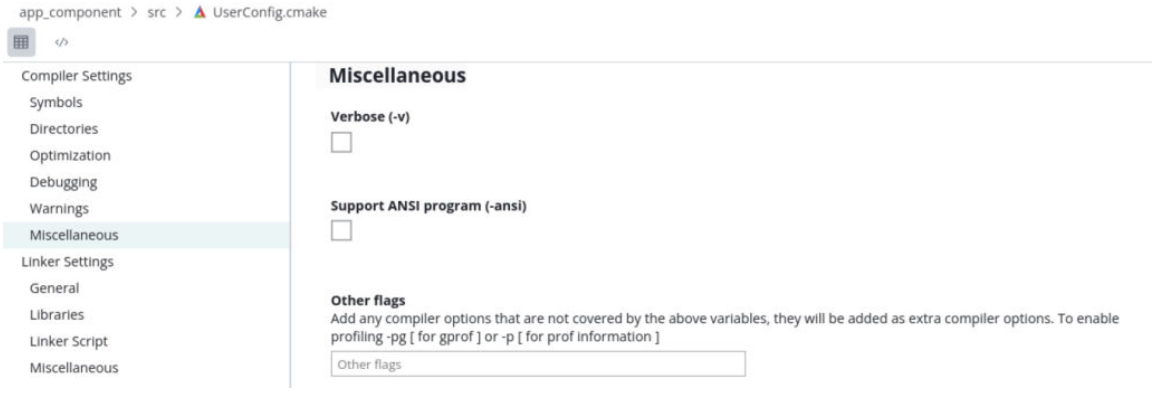
Note: `UserConfig.cmake` supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

Specifying Miscellaneous Compiler Flags

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

To set properties for your project:

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



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

Linker Scripts

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

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

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

***Note*:**

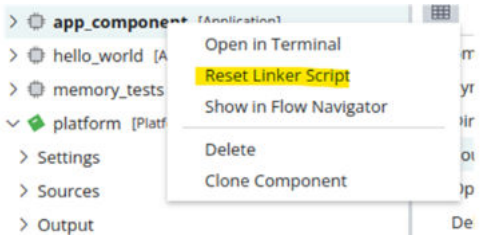
- For multiprocessor systems, each processor runs a different ELF file, and each ELF file requires its own linker script. Ensure that the two ELF files do not overlap in memory.
- The default linker always points to the DDR memory address available in memory. If you are creating an application under a given hardware/domain project, the memory overlaps for the applications.

Reset Linker Script for an Application

***Note*:** You can edit the linker script by double clicking it. The script is available at `<application_name>/sources/src/lscript.ld`.

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

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



3. Click **OK**.

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

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

Manually Adding the Linker Script

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

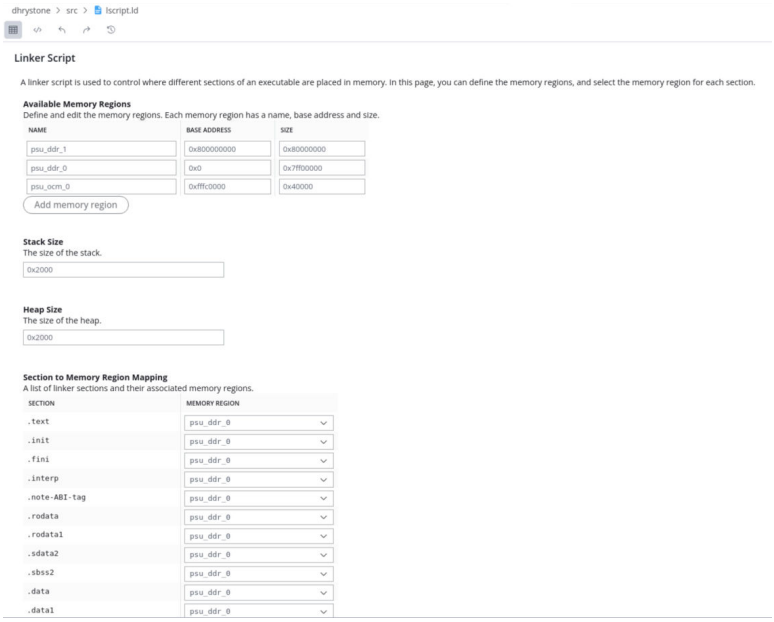
1. Select the application component in the Component View and then expand Settings. Right click **UserConfig.cmake** to open it.
2. Right click **Linker Script** to go to linker script setting section.
3. Click on **Browse** to select your own linker script file.

Modifying a Linker Script

There are multiple ways to update the linker script.

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

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



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

The linker script editor provides the following functionality.

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

Table 8: **Linker Script Editor Functionality**

Name	Function
Add Memory Regions	This section lists the memory regions specified in the linker script. You can add a new region by clicking on the Add button to the right. You can modify the name, base address and size of each defined memory region.

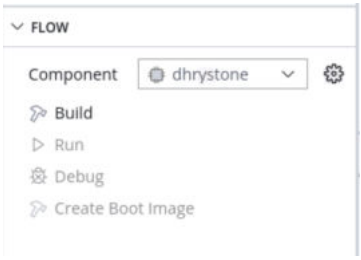
Table 8: **Linker Script Editor Functionality** *(cont'd)*

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

Building the Application Component

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

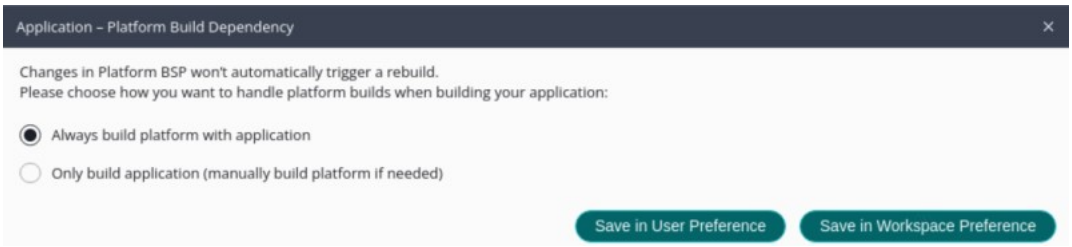
1. Go to flow navigator



2. Select the application component by clicking the drop-down button in the component field.

   ***Note*:** Component is automatically selected in the flow navigator with the component in the component view.

3. Click **Build** to build your application component and select the application build preference



   ***Note*:** Changes made to the platform BSP will not automatically trigger a rebuild. A pop-up message is displayed the first time the application is built. You should either enable the option to always build the platform before building the application or manually build the platform. This behavior can be configured in the workspace or user preferences.

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

Reading ELF Disassembly

This task is for you to view the disassembling code.

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

Creating a Library Project

A static library is a collection of object files that are linked into a program during the compile time. A platform is required before creating an application.

To create a library component, select **File → New Component → Static Library**.

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

   ***Note*:** If you do not have an existing fixed platform, refer the chapter Creating a Platform Component from XSA to create a platform. If your target platform is not in the list, you can add it by clicking the '+' button.
3. Select the target processor domain for the library, and select **Next**.
4. Click **Add Files** or **Add Folders** to import the source file for the library, and select **Next**.

   ***Note*:** You can import source files after library component is created by right clicking the SRC folder and import the source.
5. Review the summary and click **Finish**.

After Library component is created, you can select to build it in **Flow Navigator**. For more information on compiler setting, refer to Build Configuration Settings.

***Note*:** Linker support for Library component is not provided. Omit Linker setting if you check the build configuration settings.

Using Custom Libraries in Application Projects

This task provides instructions on how to use custom libraries..

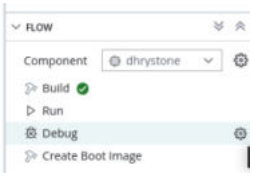
You can configure the libraries to be linked with their application by setting the parameters in the `UserConfig.cmake` file. For more information, refer to Adding Libraries and Library Paths.

Run, Debug, and Optimize

Launch Configurations

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

Figure 35: **Flow Window**



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

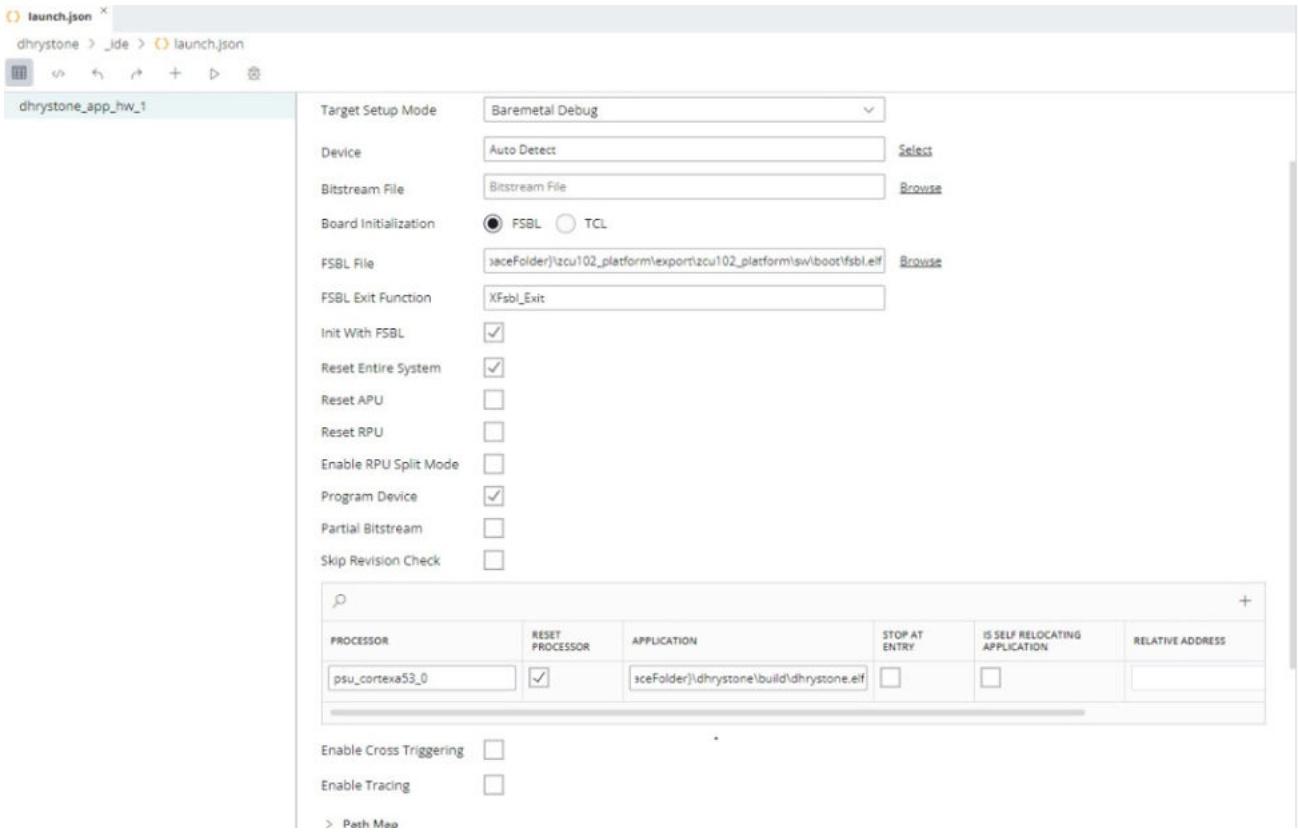
The `launch.json` file is opened to let you edit the launch configurations. The Launch Configuration editor has a toolbar menu at the top, as shown in the following figure. The specific contents of the launch configuration vary depending on the component or the project selected.

Launch Configurations

The commands in the Launch Configuration toolbar include:

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

Figure 36: **Launch Configurations**



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

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

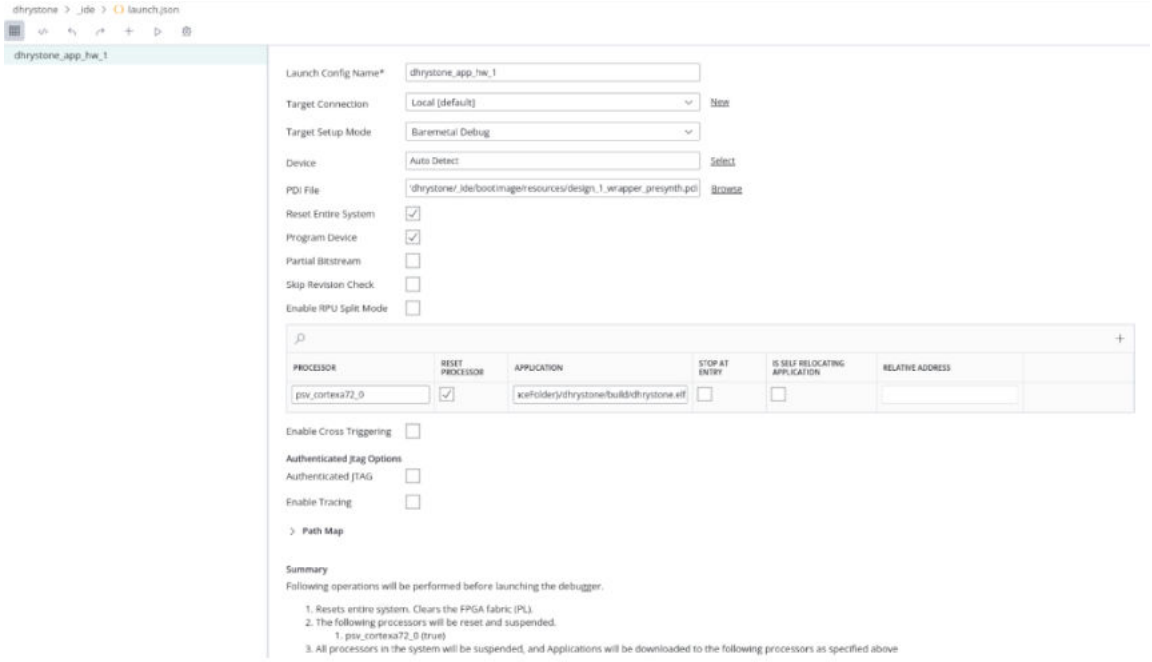
Main Page

The main view has the following options:

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

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

Figure 37: **Launch Configuration Window**



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

Application Part

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

Figure 38: **Debug Confrontation Application Part**

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

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

Target Connections

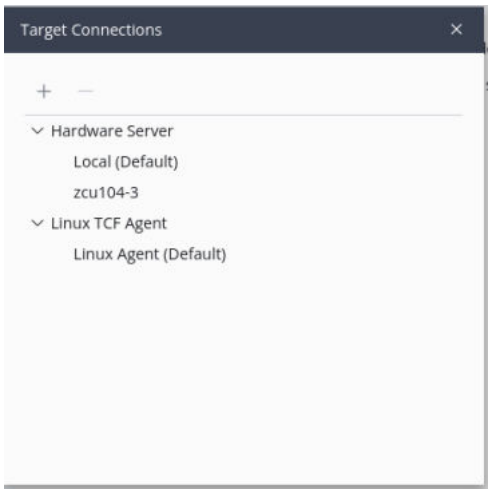
The Target Connections dialog box ( ) allows you to configure multiple remote targets. It shows connected targets and gives you an option to add or delete target connections.

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

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

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

Figure 39: **Target Connections**



Creating a New Target Connection

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

To create new target connection:, do the following.

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

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

Setting Custom JTAG Frequency

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

To set the JTAG frequency, execute the following steps.

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

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

Establishing a Target Connection

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

To use a remote board to establish a target connection:

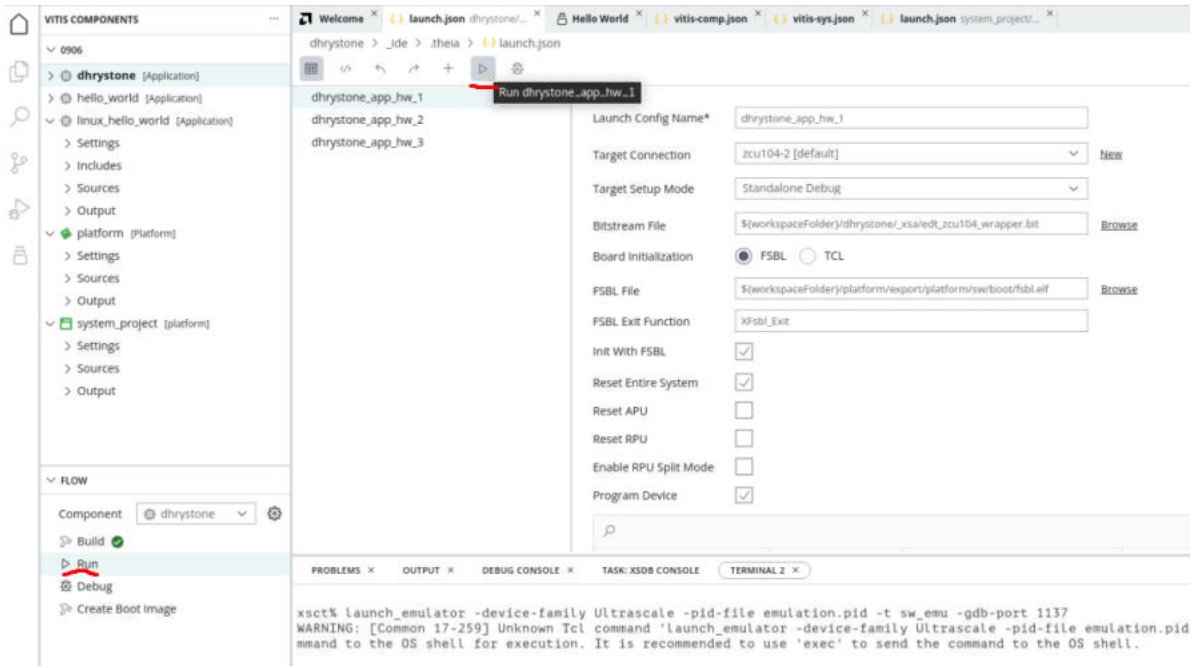
1. Ensure that the target is connected to the remote host.
2. Launch the `hw_server` manually on the remote host:
   a. Take a shell on the remote host.
   b. Source the setup scripts by using `C:/Xilinx/<version>Vitis/settings64.bat` (or) `<Vitis_local_install_path>/<version>Vitis/settings64.csh`.
3.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

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

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

Running the Application Component

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



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

Debugging Application Component

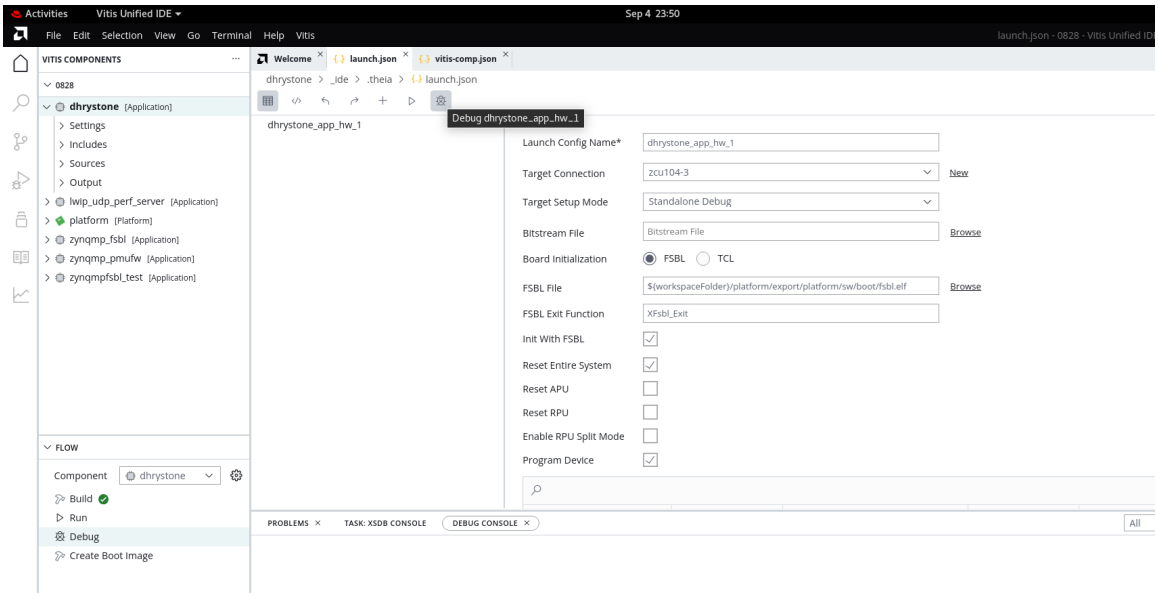
Starting Debug

The following chapters the steps to start the debug process.

Debugging Standalone Application Component

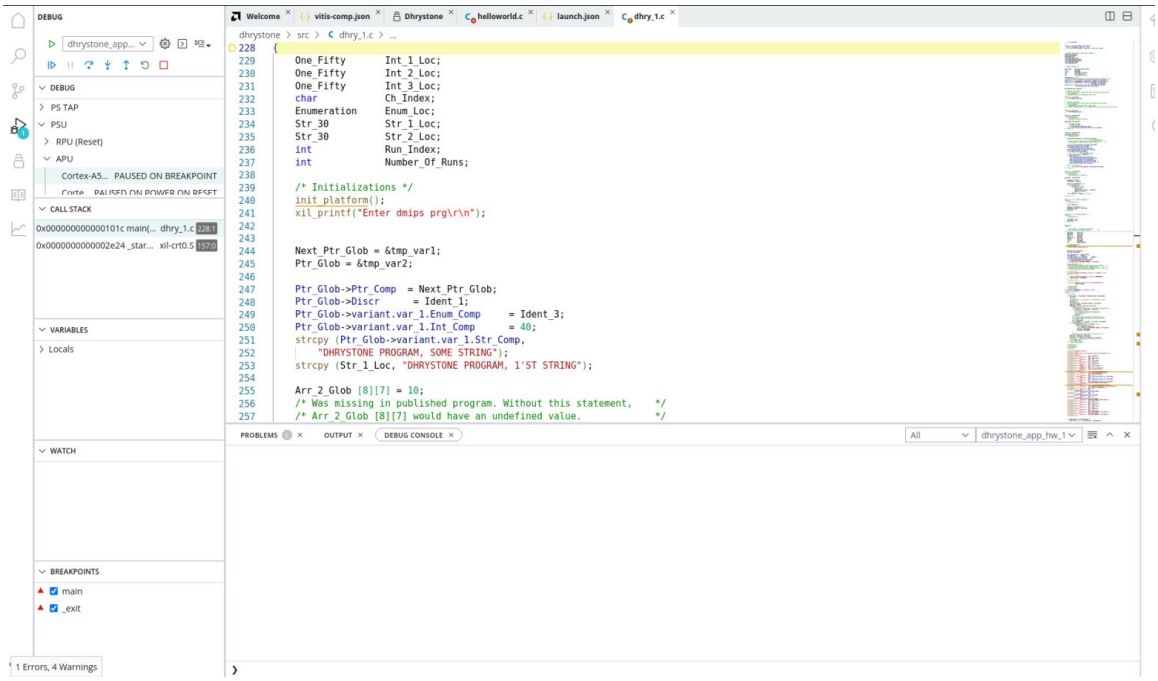
This topic describes how to use the Debugger to debug standalone applications.

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



5. The debug view is displayed.

Figure 40: **Debug View**



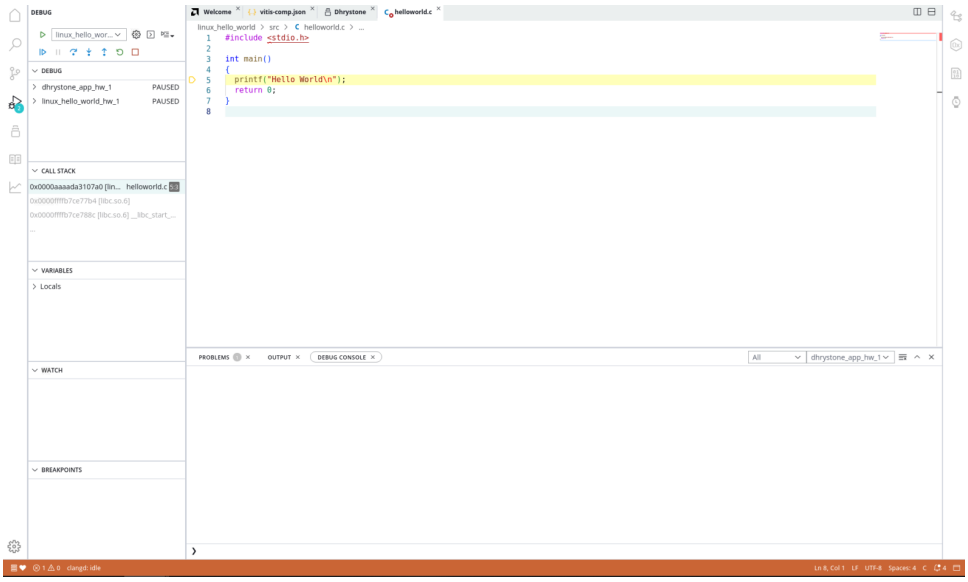
Debugging Linux Application Component

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

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

The debug view is displayed.

Figure 41: **Debug View**



Debugging Standalone Application Component on QEMU

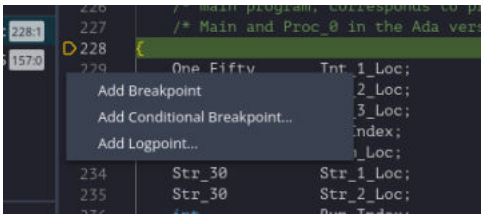
This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Setting Conditional Breakpoints

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

To add conditional breakpoints, execute the following steps.

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



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

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

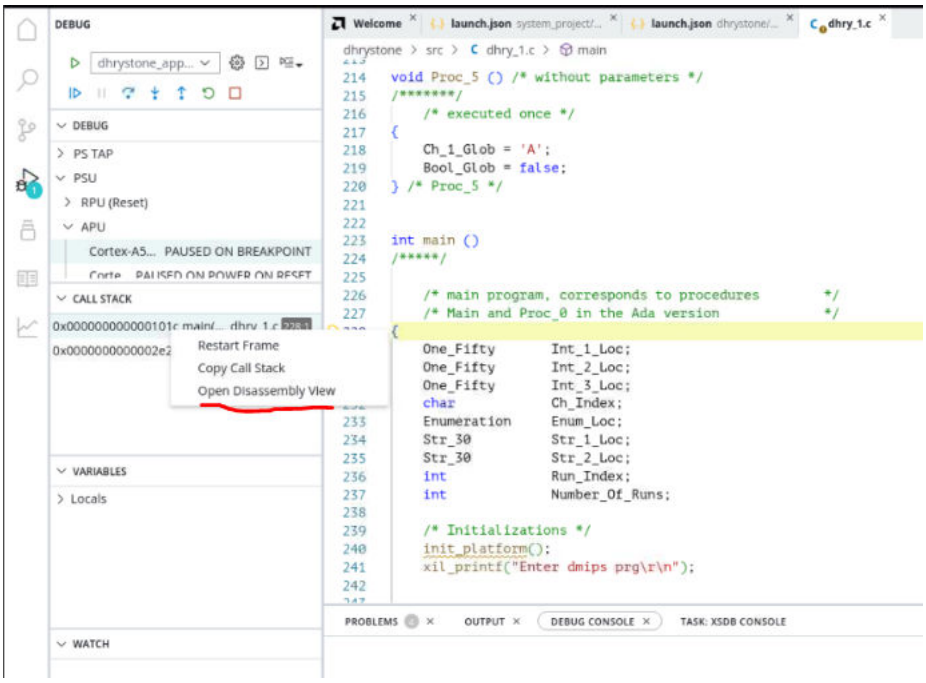
- Right click on an existing breakpoint
- Select **Edit Breakpoint**.

- Add the expression to change it to a conditional breakpoint.

Viewing Disassembly Code

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

Figure 42: **Viewing Disassembly Code**



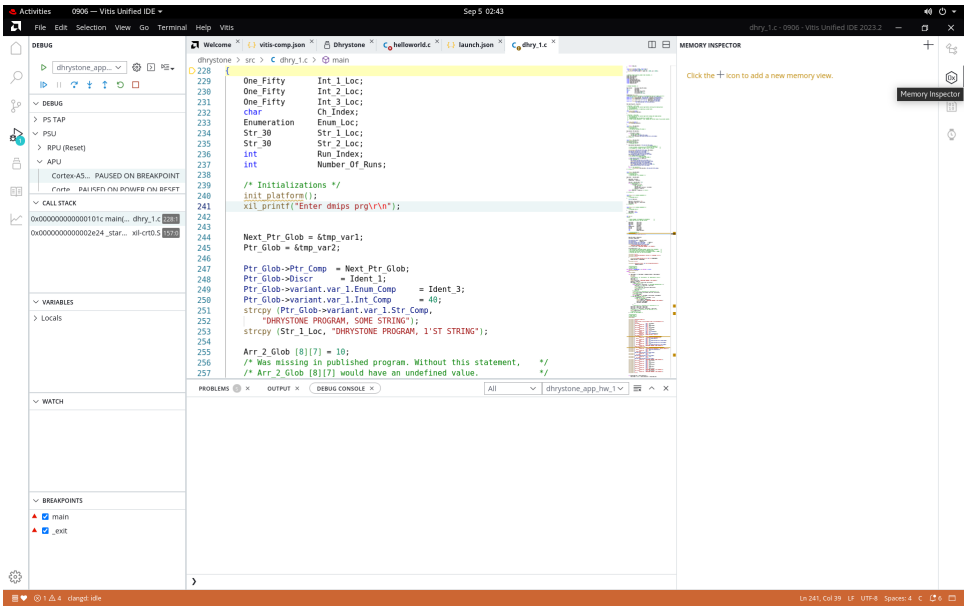
Next, select the Open Disassembly to view disassembly code is displayed.

Viewing Memory

Viewing the Value of a Certain Memory address

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

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



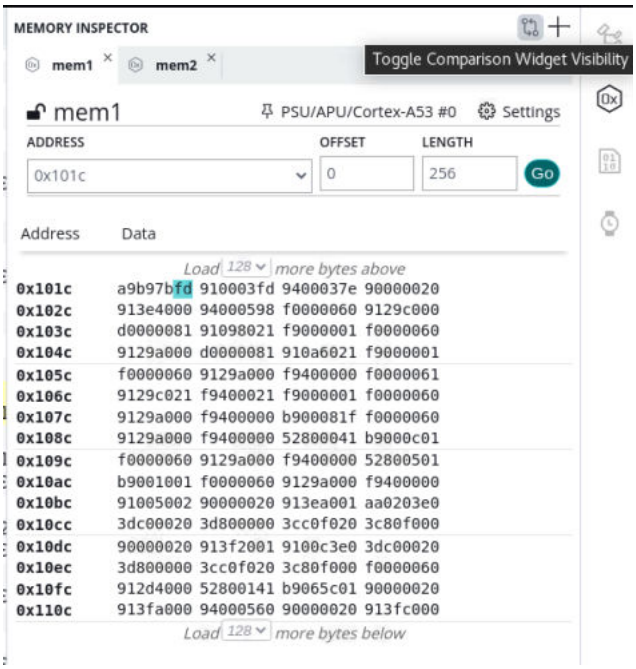
2. Click **+** icon to add a new memory view.
3. Set the memory address, offset, and length. Click **Go**.

The memory value is displayed in the Memory view.

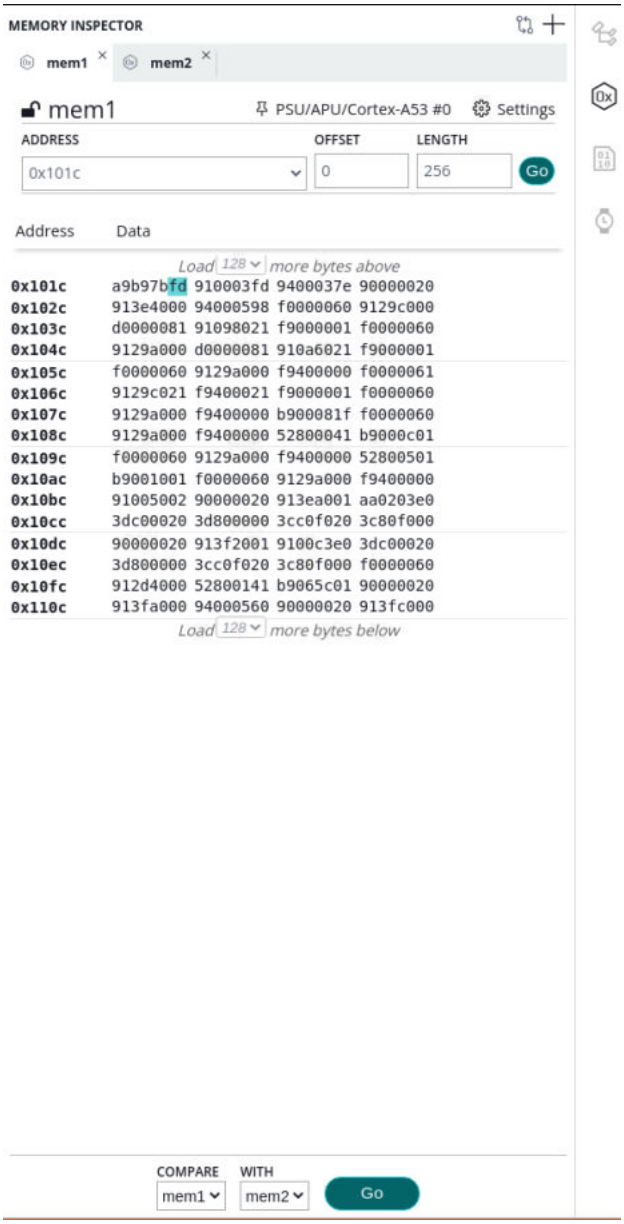
Comparing the Memory Value of Two Addresses

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

1. Refer to Viewing the Value of a Certain Memory address to create two memory inspectors. Set the difference of the two inspectors.



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



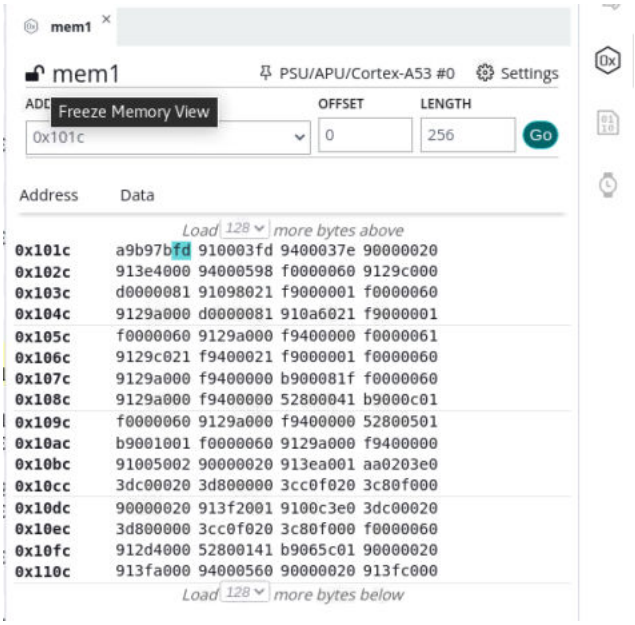
4. The comparison result is displayed in main view.

Freezing Memory Value

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

1. Refer to Viewing the Value of a Certain Memory address to create a new memory inspector.

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



The value can be viewed until the tool is closed.

Viewing Registers

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

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

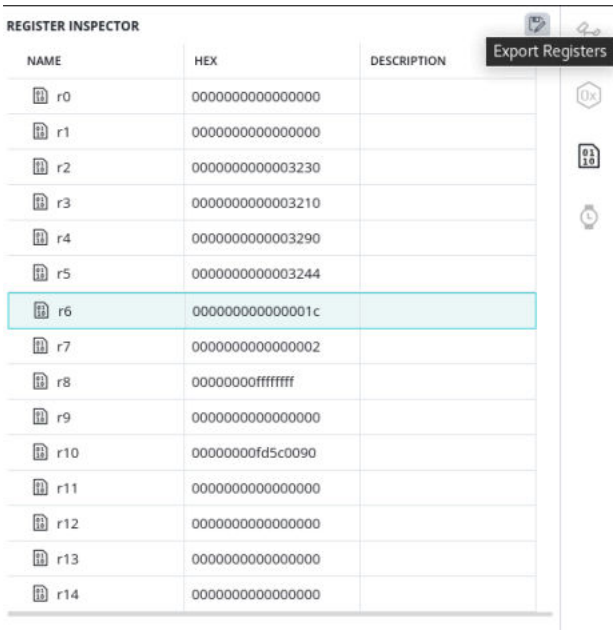
Figure 43: **Register Inspector Window**

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

Exporting Registers from the Vitis IDE

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

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

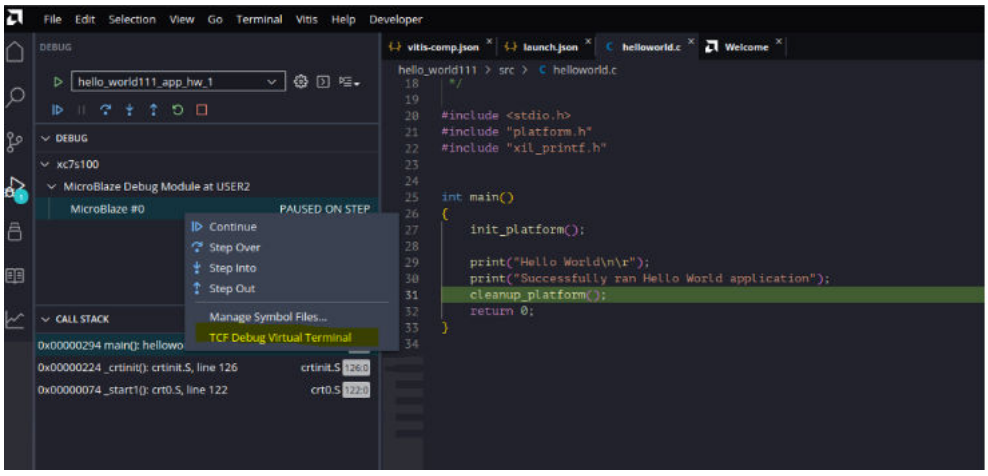


4. Add the following information on the next screen:
   - **Location:** Provide the location where you want to save the register dump file.
   - **Select registers/groups to export:** Select the list of registers to be dumped. You can uncheck any registers that are not required.
5. Click **OK** to dump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

Using Virtual UART Terminal

TCF Virtual UART terminal is support for MDM terminal. Right click the MicroBlaze™ core or MicroBlaze™ V core and select **TCF Debug Virtual Terminal**.

Figure 44: **TCF Debug Virtual Terminal**

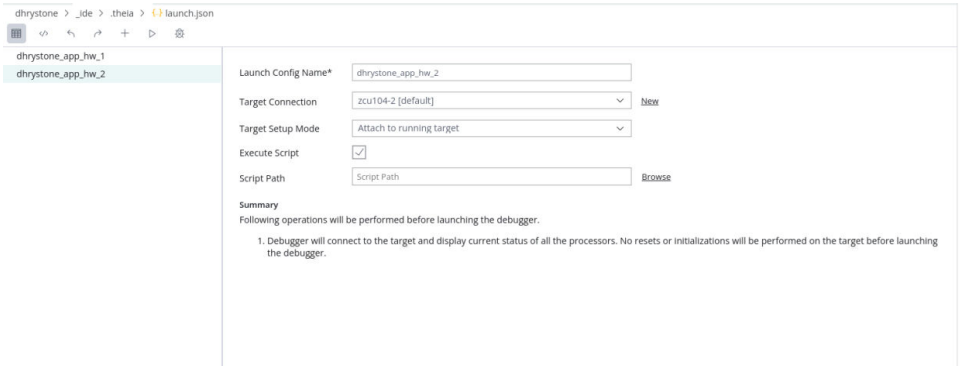


Debugging an Application Component Already Running On a Target Device

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

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

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



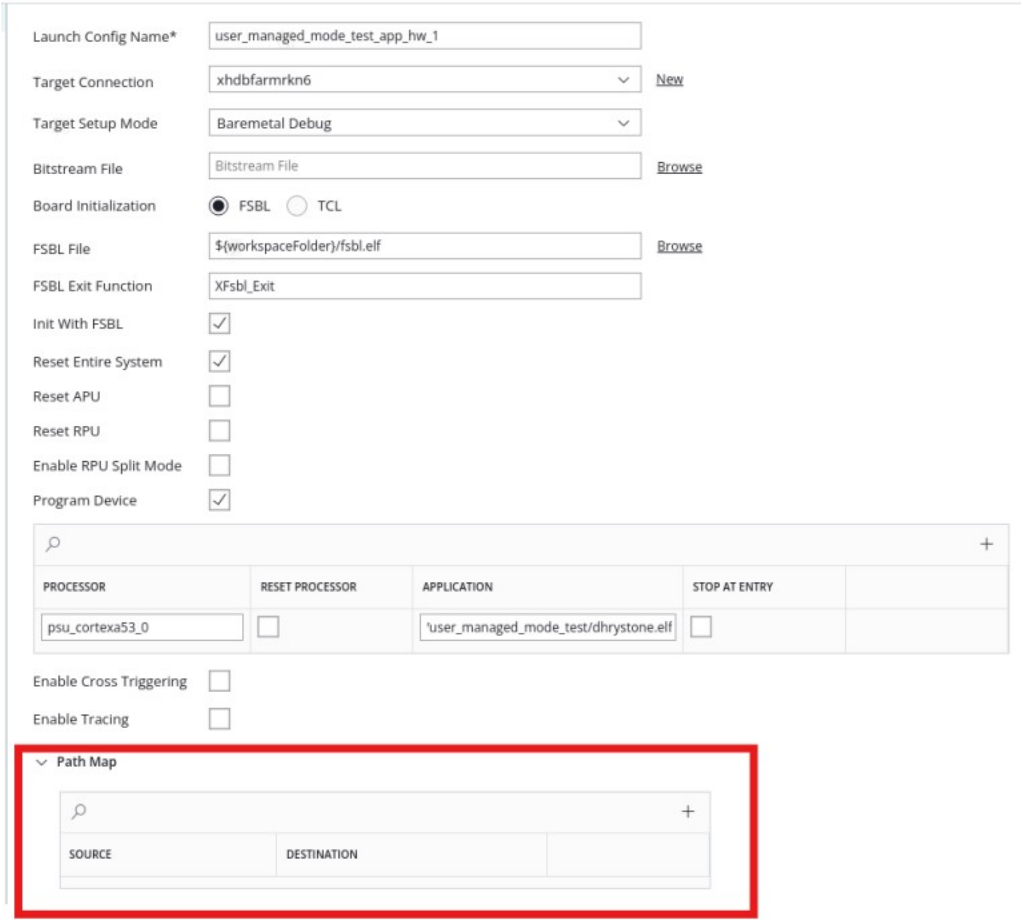
3. Follow Starting Debug to start debug the application.

Source Code Remap

You can remap the source code during debugging in user-managed mode when the application source path has changed or ELF location is changed. Follow the steps below:

***Note*:** An error indicating "cannot find source" appears if the source code path or ELF location has been changed.

1. Go to debug launch configuration, expand Path Map



2. Click **+** to add source map.
   a. SOURCE: It refers to the original path of the source file. You can hover over the source file name in the CALL STACK view to see this path.
   b. DESTINATION: the current file path of the source code
3. Then click **OK** and start debug.

Debugging an Application on the Emulator (QEMU)

***Note*:** This function is not supported in the Vitis Unified IDE. Switch to the classic Vitis IDE (available in version 2024.2 and older) and refer to the 2024.2 documentation if you wish to access this function.

Running and Debugging Application Components under a System Project Together

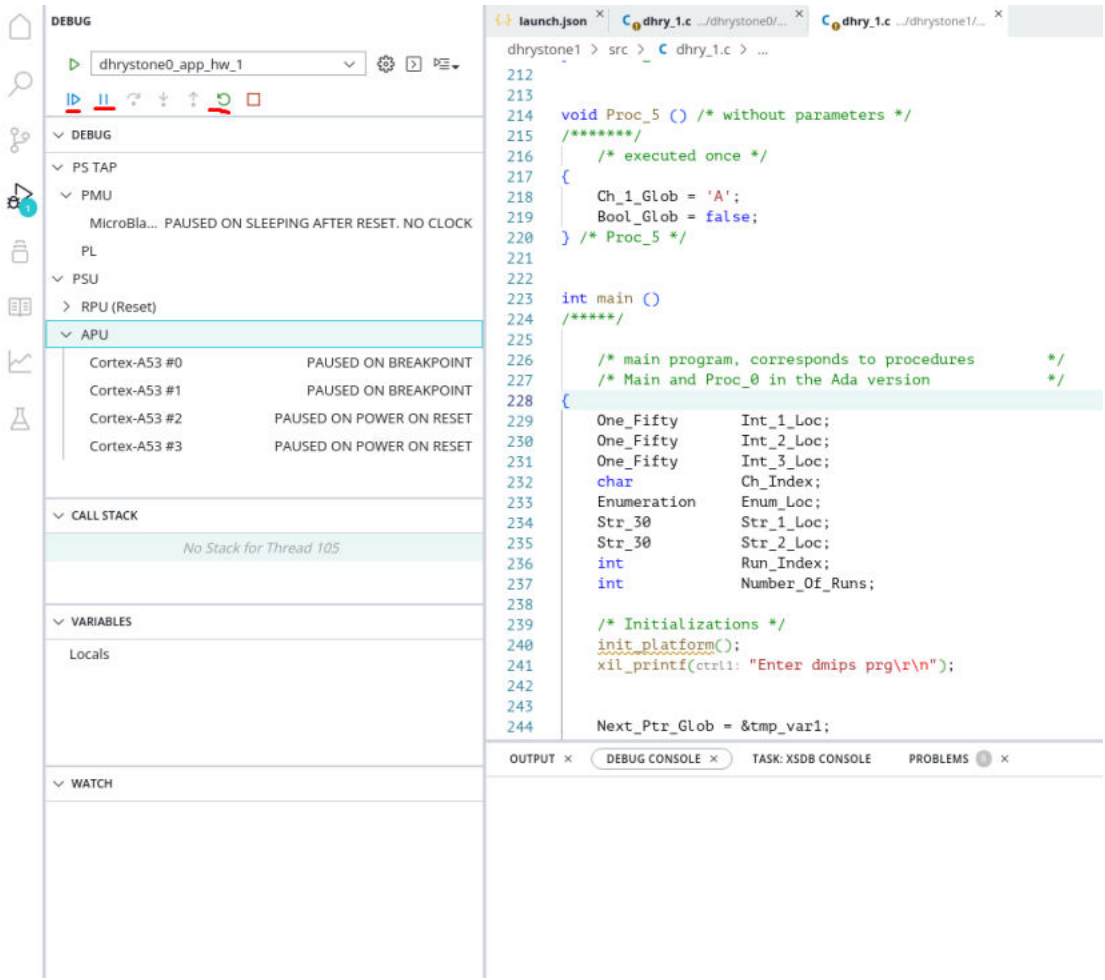
Each application component under a system project can run standalone. Application components under a system project can be launched together as well. The Vitis Unified IDE can download them one by one and launch them one after another. In debug mode, all applications stop at `main()`. The following steps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

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

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

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

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



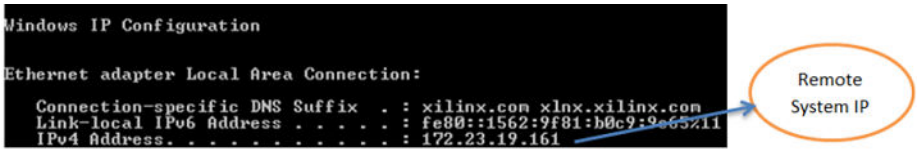
Debugging on a Remote Board

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

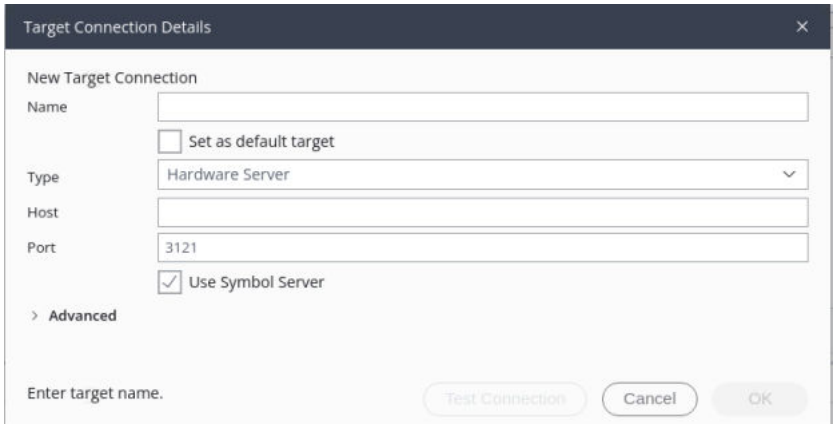
1. Setting up the Remote System Environment:
   a. Run `hw_server` with a non-default port (for example: 3122) to enable remote connections. Use the following command to launch the `hw_server` on port 3122:

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

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



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



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

   l. Click the **debug** button.

***Note*:** For target connection creation, refer to: Target Connections.

OS Aware Debugging

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

Xen Aware Debugging

Xen is an open-source hypervisor that allows multiple virtual machines to run on a single physical machine. Xen-aware debugging refers to the debugging techniques and tools specifically designed for debugging software running on the Xen hypervisor. With Xen-aware debug support enabled, you can conduct Xen-aware debug.

To enable Xen-aware debug, follow the steps below:

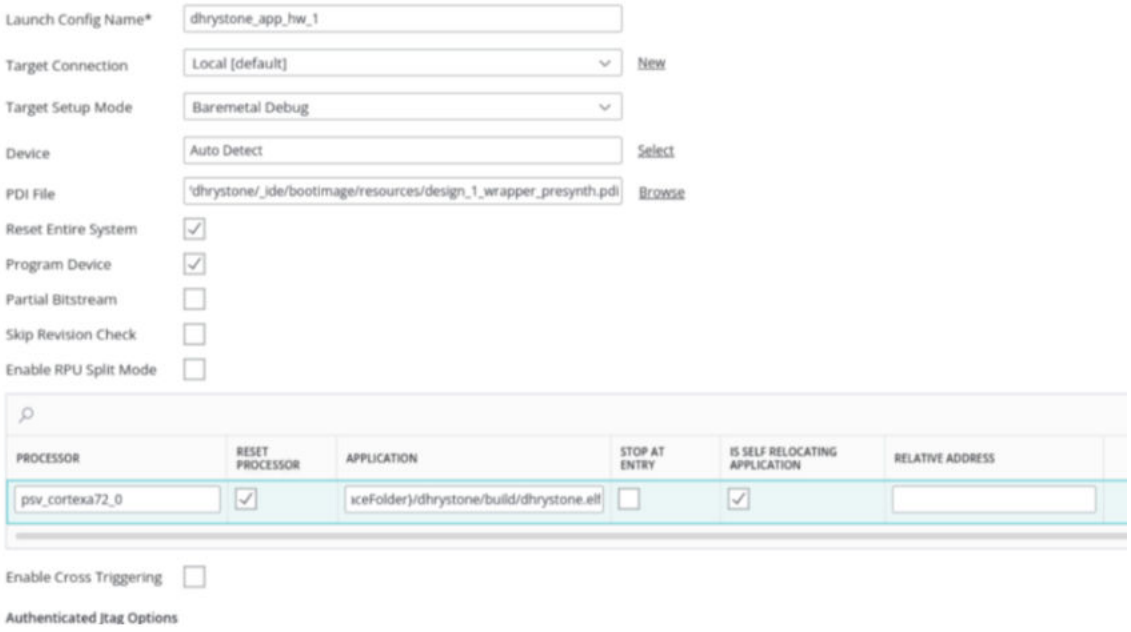
1. Refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144) to prepare the environment for booting a pre-built Linux as dom0.
2. Start the Vitis Unified IDE.
3. Create and build the application.
4. Launch the Debug configuration.
5. Click **New Launch Configuration**.
6. Select the **Target Connection** and set the target mode as **Attach to running process**.
7. Click the **Debug** button to start the debug session.
8. Right click on APU Cortex A72 #0 and click on **Manage symbol files** in the Debug view. A dialog box for adding symbols file is displayed.
9. Click the **+** button to add symbols. Select the symbol file `xen-syms` generated in the PetaLinux project in the first step, enable all the following options and click the **Ok** button. This step maps the symbols of the Xen hypervisor. Xen is now enabled and can be observed in the Debug view.
10. Right click on VCPU #0 and click **Manage Symbol files** and add the `vmlinux` symbol file generated in the PetaLinux project in the first step for DOM0. Do not enable the **OS Awareness** option in this step.
11. Select VCPU #0 and **Pause** in the Debug view.
12. Check the Call Stack view to ensure all the symbol files are mapped, then use the step-in and step-over options for further debugging.

Debugging Self-Relocating Programs

In some resource-constrained embedded designs, users often require the ability to load and relocate their applications. A self-relocating program is a program that relocates its own code and data sections during runtime. The debug information available in the ELF file does not specify where the program sections will be relocated to. Therefore, you must provide the debugger with the relative address to which the program sections will be relocated. You can do this either through the Vitis Unified IDE or CLI.

To perform the process in the Vitis Unified IDE, update the `launch.json` file of the application you want to debug. Check the box **Is Self Relocating Application** and provide the relative address as shown in the screenshot below.

Figure 45: **Debugging Self-Relocating Programs**



To perform the process for XSDB, use the following memmap command to provide the address to which the program sections are allocated: `memmap -reloc <addr> -file <path-to-elf>`

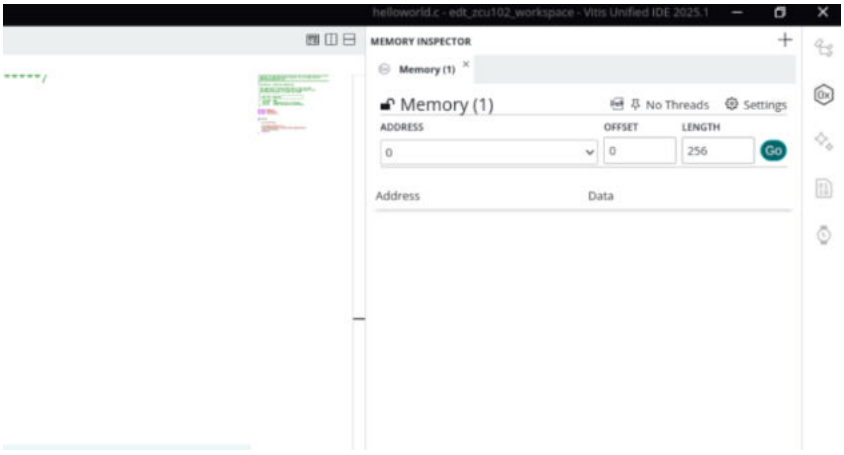
Debugging an Application Project Using the Emulator (Command-Line Flow)

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

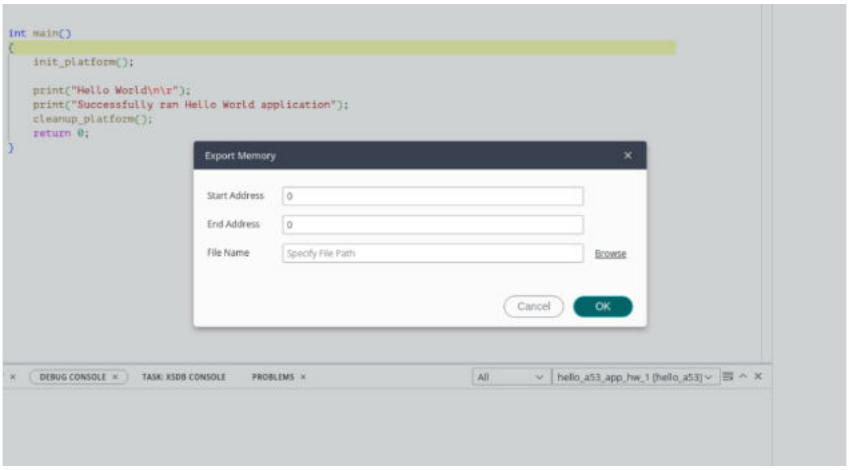
Export Memory Function

In the Vitis Unified IDE, you can export a specific memory section to an s record file during a debug session so that you can examine the memory contents during runtime. Follow the below steps for exporting a memory section during an active debug session.

1. Once the debug session has been launched select the memory inspector view on the right hand side.



2. Select the 'Export Memory' icon in the Memory Inspector View. The Export Memory Wizard appears.



3. Enter your preferred start address, end address and output file path.

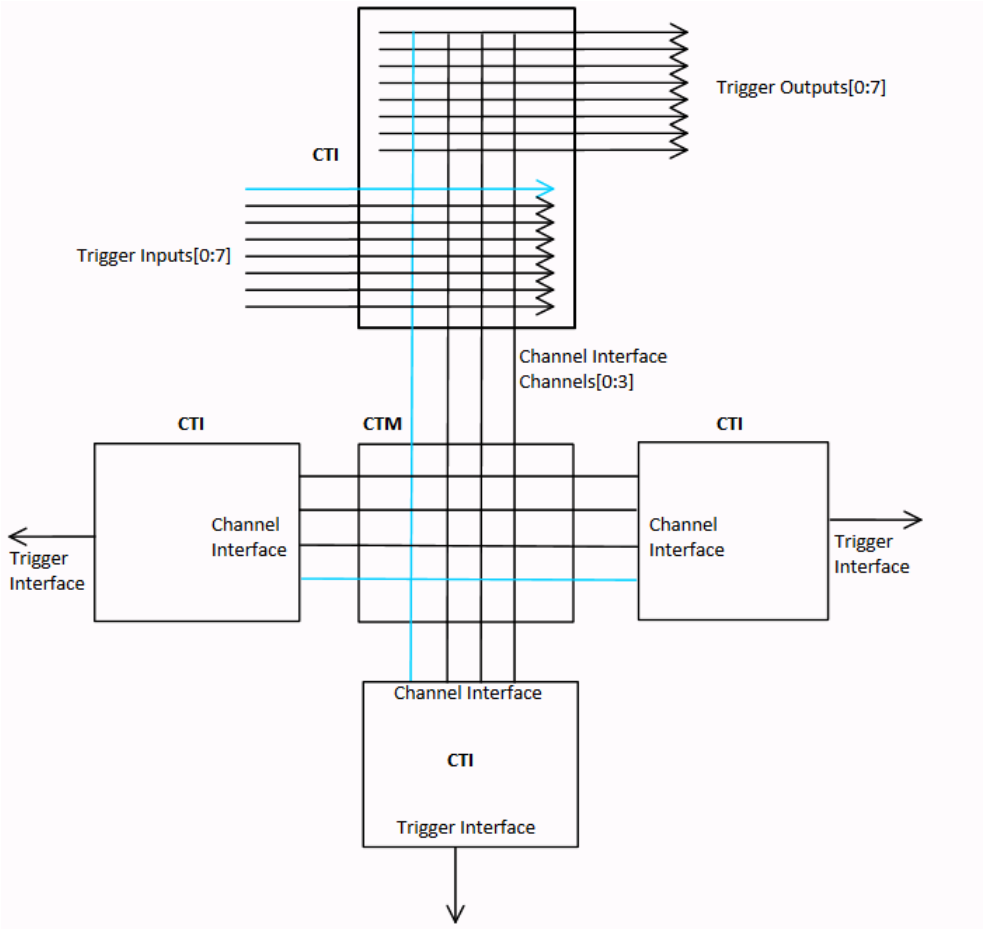
Cross-Triggering

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

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

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

Figure 46: **CTIs and CTM in a Generic Setup**



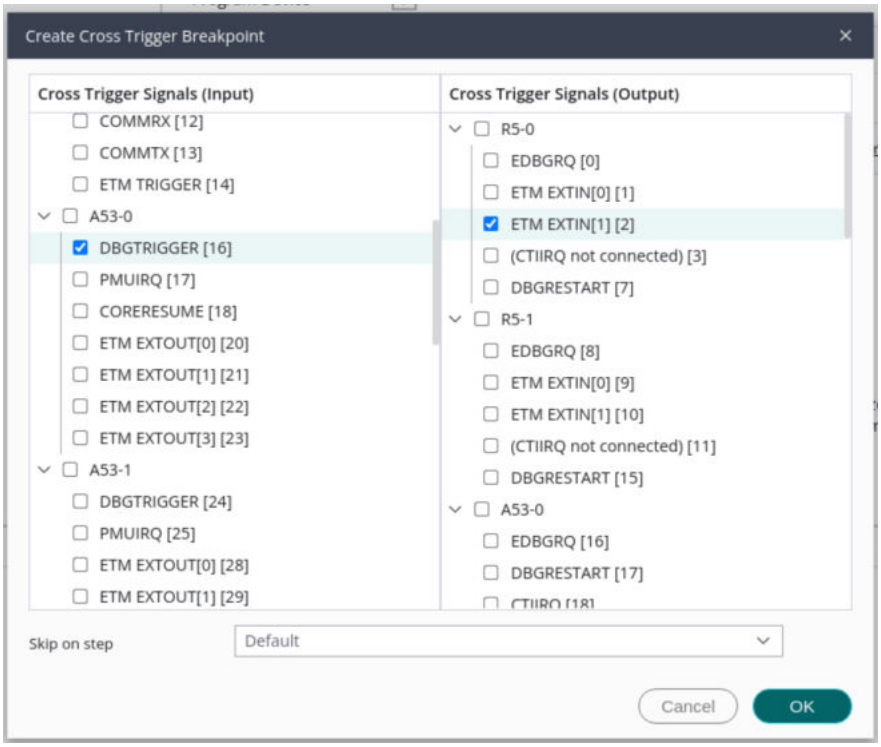
CTM forms an event broadcasting network with multiple channels. A CTI listens to one or more channels for an event, maps a received event into a trigger, and sends the trigger to one or more CoreSight components connected to the CTI. A CTI also combines and maps the triggers from the connected CoreSight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

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

Enable Cross-Triggering

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

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



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

Cross-Triggering in Zynq Devices

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

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

Table 9: **CTI Trigger Ports in Zynq Devices**

CTI Trigger Port	Signal
CTI connected to ETB, TPIU	
Trigger Input 2	ETB full
Trigger Input 3	ETB acquisition complete
Trigger Input 4	ITM trigger
Trigger Output 0	ETB flush
Trigger Output 1	ETB trigger
Trigger Output 2	TPIU flush
Trigger Output 3	TPIU trigger
FTM CTI	
Trigger Input 0	FTM trigger
Trigger Input 1	FTM trigger

Table 9: **CTI Trigger Ports in Zynq Devices** *(cont'd)*

CTI Trigger Port	Signal
Trigger Input 2	FTM trigger
Trigger Input 3	FTM trigger
Trigger Output 0	FTM trigger
Trigger Output 1	FTM trigger
Trigger Output 2	FTM trigger
Trigger Output 3	FTM trigger
CPU0/1 CTIs	
Trigger Input 0	CPU DBGACK
Trigger Input 1	CPU PMU IRQ
Trigger Input 2	PTM EXT
Trigger Input 3	PTM EXT
Trigger Input 4	CPU COMMTX
Trigger Input 5	CPU COMMTX
Trigger Input 6	PTM TRIGGER
Trigger Output 0	CPU debug request
Trigger Output 1	PTM EXT
Trigger Output 2	PTM EXT
Trigger Output 3	PTM EXT
Trigger Output 4	PTM EXT
Trigger Output 7	CPU restart request

Cross-Triggering in Zynq UltraScale+ MPSoCs

In Zynq UltraScale+ MPSoCs, ECT is configured with four broadcast channels, nine CTIs, and a CTM. The following table 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 10: **CTI Trigger Ports in Zynq UltraScale+ MPSoCs**

CTI Trigger Port	Signal
CTI 0 (soc_debug_fpd)	
IN 0	ETF 1 FULL
IN 1	ETF 1 ACQCOMP
IN 2	ETF 2 FULL
IN 3	ETF 2 ACQCOMP
IN 4	ETR FULL
IN 5	ETR ACQCOMP
IN 6	-

Table 10: **CTI Trigger Ports in Zynq UltraScale+ MPSoCs** *(cont'd)*

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

Table 10: **CTI Trigger Ports in Zynq UltraScale+ MPSoCs** *(cont'd)*

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

Cross-Triggering in Versal Devices

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

Table 11: **CTI Trigger Ports in Versal Devices**

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

Table 11: **CTI Trigger Ports in Versal Devices** *(cont'd)*

CTI Trigger Port	Signal
OUT 6	ETM EXTIN[2]
OUT 7	ETM EXTIN[3]
CTI p (pmc_debug). XSDB IDs = 48-55	
IN 0	ATM TRIGOUT[0]
IN 1	-
IN 2	-
IN 3	-
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ATM TRIGIN[0]
OUT 1	-
OUT 2	-
OUT 3	-
OUT 4	-
OUT 5	-
OUT 6	-
OUT 7	-
CTI 0d (soc_debug_lpd). XSDB IDs = 56-63	
IN 0	ATM0 TRIGOUT[0]
IN 1	ATM0 TRIGOUT[1]
IN 2	ATM0 TRIGOUT[2]
IN 3	ATM0 TRIGOUT[3]
IN 4	ATM0 TRIGOUT[4]
IN 5	-
IN 6	-
IN 7	-
OUT 0	ATM0 TRIGIN[0]
OUT 1	ATM0 TRIGIN[1]
OUT 2	ATM0 TRIGIN[2]
OUT 3	ATM0 TRIGIN[3]
OUT 4	ATM0 TRIGIN[4]
OUT 5	7
OUT 6	-
OUT 7	-
CTI 1a (APU). XSDB IDs = 64-71	
IN 0	ELA 1a CTTRIGOUT[0]
IN 1	ELA 1a CTTRIGOUT[1]

Table 11: **CTI Trigger Ports in Versal Devices** *(cont'd)*

CTI Trigger Port	Signal
IN 2	ETF 1a FULL
IN 3	ETF 1a ACQCOMP
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ELA 1a CTTRIGIN[0]
OUT 1	ELA 1a CTTRIGIN[1]
OUT 2	ETF 1a FLUSHIN
OUT 3	ETF 1a TRIGIN
OUT 4	PMUSNAPSHOT[0]
OUT 5	PMUSNAPSHOT[1]
OUT 6	-
OUT 7	-
CTI 1b (soc_debug_fpd). XSDB IDs = 72-79	
IN 0	STM TRIGOUTSPTE
IN 1	STM TRIGOUTSW
IN 2	STM TRIGOUTHETE
IN 3	STM ASYNCOUT
IN 4	ETF 1 FULL
IN 5	ETF 1 ACQCOMP
IN 6	ETR FULL
IN 7	ETF ACQCOMP
OUT 0	STM HWEVENTS
OUT 1	STM HWEVENTS
OUT 2	TPIU FLUSHIN
OUT 3	TPIU TRIGIN
OUT 4	ETF 1 FLUSHIN
OUT 5	ETF 1 TRIGIN
OUT 6	ETR FLUSHIN
OUT 7	ETR TRIGIN
CTI 1c (soc_debug_fpd). XSDB IDs = 80-87	
IN 0	pl_ps_trigger[0]
IN 1	pl_ps_trigger[1]
IN 2	pl_ps_trigger[2]
IN 3	pl_ps_trigger[3]
IN 4	-
IN 5	-
IN 6	-

Table 11: **CTI Trigger Ports in Versal Devices** *(cont'd)*

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

Use Cases

FPGA to CPU Triggering

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

PTM to CPU Triggering

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

CPU to CPU Triggering

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

XSDB Cross-Triggering Commands

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

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

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

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

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

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

Profile/Analyze

TCF Profiling

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

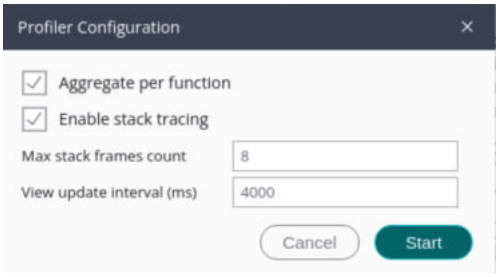
1. Start the Vitis Unified IDE and select the application component you wish to profile.
2. Go to the Flow Navigator and create a debug launch configuration.

   ***Note*:** Refer to Launch Configurations and Creating a New Target Connection to create a debug launch configuration.

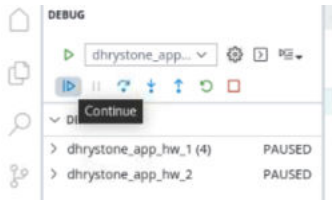
   ***Note*:** Regarding the target connection, if the application component is a Linux application, select the TCF agent. If it a standalone application, select HW server.
3. After configuration is finished, click **Debug** to start the debug session.

   ***Note*:** Refer to Starting Debug to start the debug session.
4. Click the **TCF Profiler** button to start profiling view.

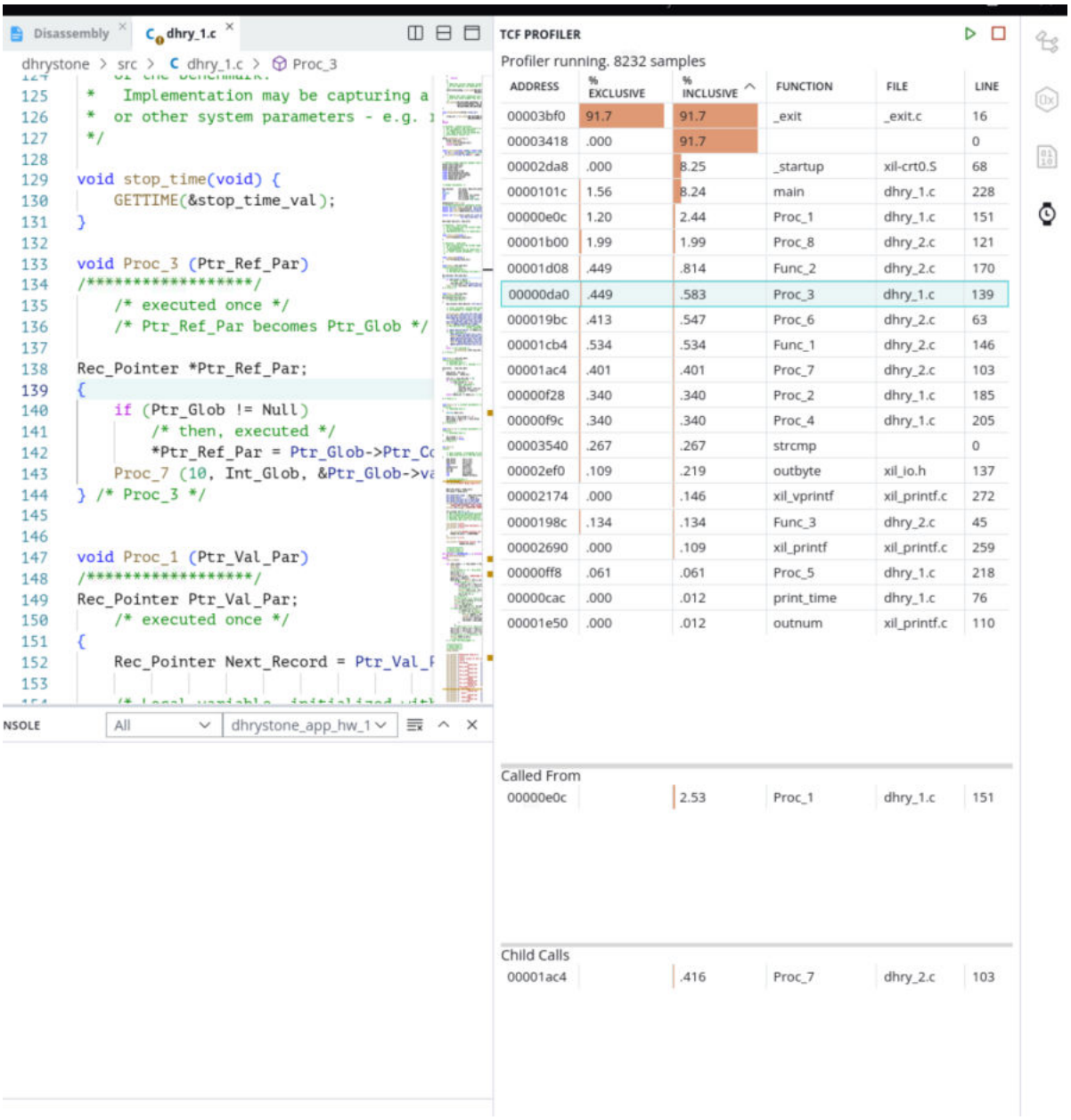
5. Click the **Start** button to begin the profiling. Select the **Enable stack tracing** option to show the stack trace for each address in the sample data. You can also set the frame count and update the interval according to your requirements.
   - Specify the **Max stack frames count** for the maximum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



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



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



- Clicking the function in the profiler view displays the Called From and Child Calls in the bottom right corner of the Profiler view.

- The view supports cross-probe between the function name and source code. Clicking the function jumps to the source code in the Source Code view.

gprof Profiling

***Note*:** This function is not supported in the Vitis Unified IDE. Switch to the classic Vitis IDE (available in version 2024.2 and older) and refer to the 2024.2 documentation if you wish to access this function.

Non-Intrusive Profiling for MicroBlaze Processors

This functionality is not supported in the Vitis Unified IDE GUI, although you can use XSDB to achieve the same result. Use the XSDB or Python XSDB tool to run non-intrusive profiling for MicroBlaze and MicroBlaze RISC V.

The XSDB command is `mbprofile`, or execute `mbprofile -help` for more details.

FreeRTOS Analysis Using STM

***Note*:** This function is not supported in the Vitis Unified IDE. Switch to the classic Vitis IDE (available in version 2024.2 and older) and refer to the 2024.2 documentation if you wish to access this function.

Creating a Boot Image

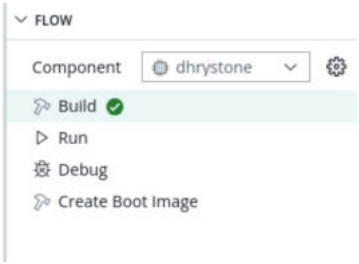
AMD FPGAs and system-on-chip (SoC) devices typically have multiple hardware and software binaries used to boot them to the function they were designed for. These binaries can include FPGA bitstreams, firmware images, bootloaders, operating systems, and user-chosen applications that can be loaded in both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

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

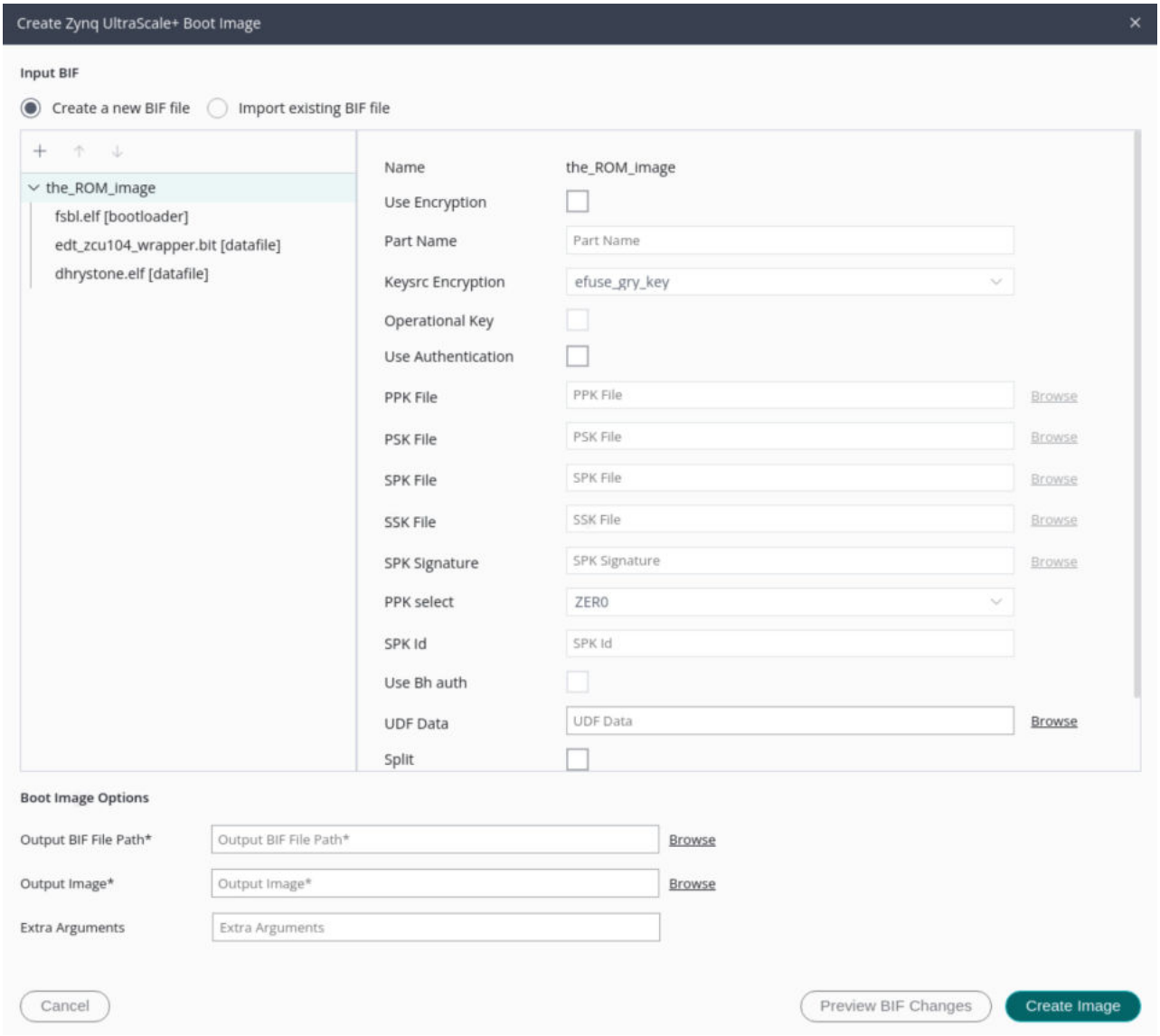
There are two ways to create a boot image: from the Flow Navigator or from scratch.

Create boot image from Flow Navigator (faster)

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



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



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

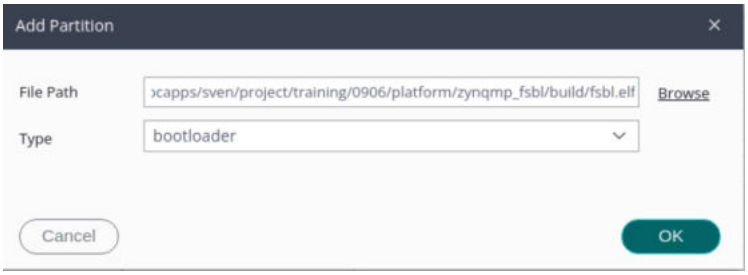
Create a boot image from scratch

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

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

ZYNQ/ZYNQMP

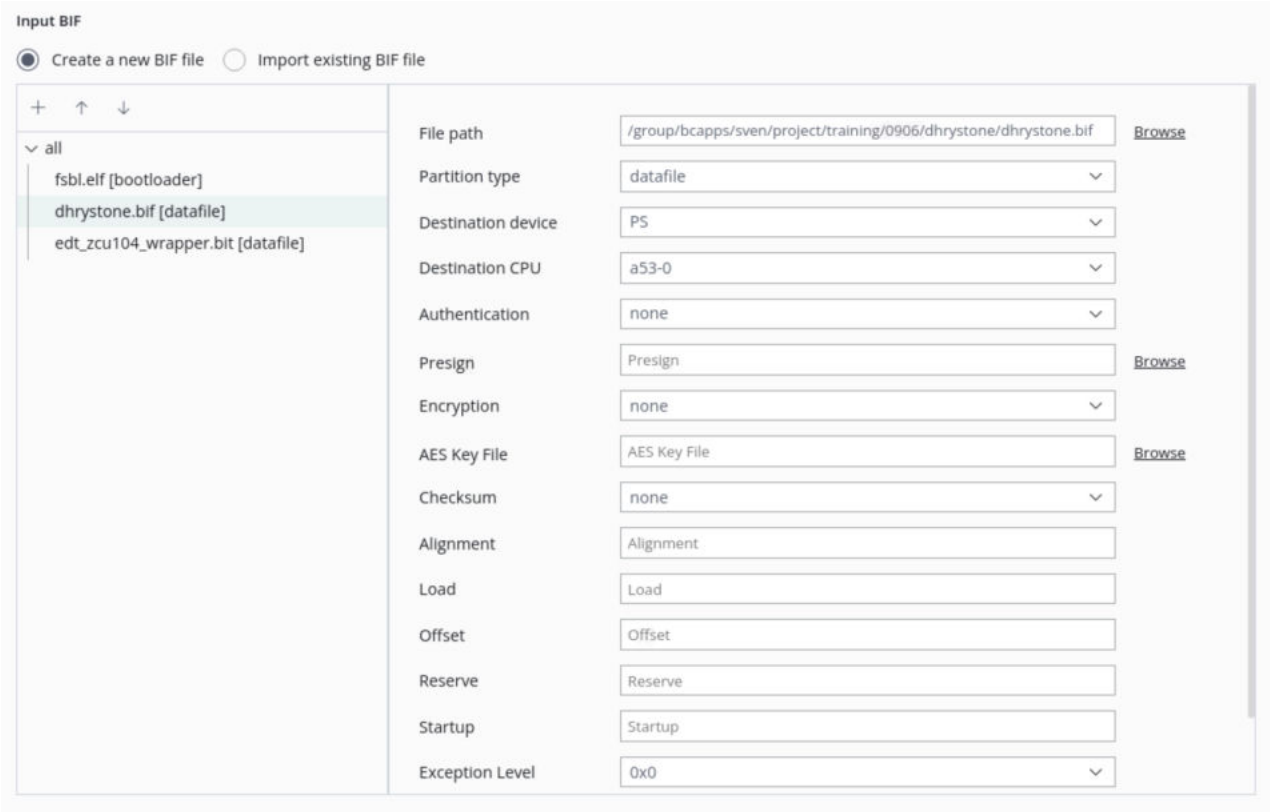
1. If you do not have an existing bif file, select Create a new BIF file.
2. Click **+** to add the image partition.



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

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

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



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

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

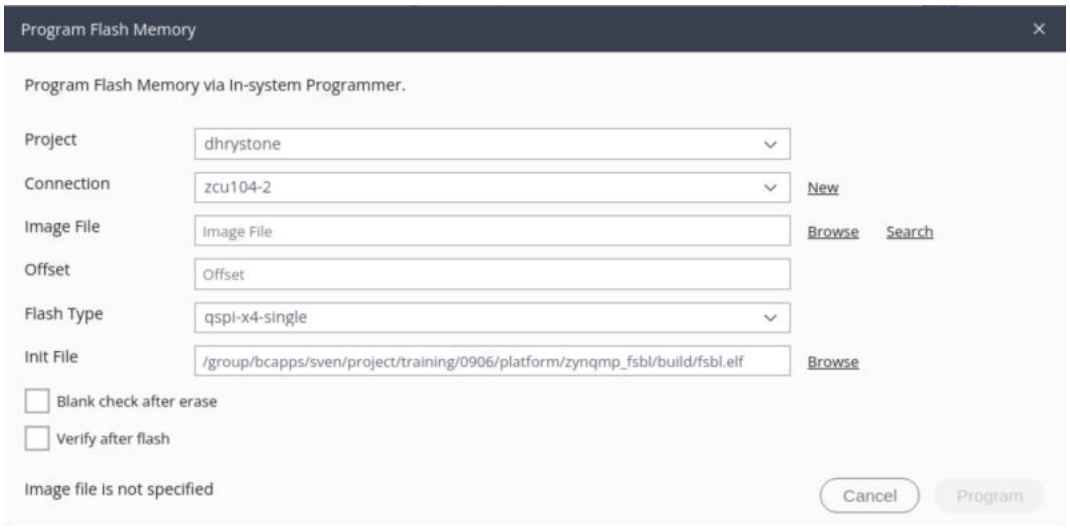
Programming Flash

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

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

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

Figure 47: **Program Flash Window**



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

- **Project:** Select the system project you plan to use. The application component is be automatically selected in the Component View.
- **Connection:** Select the connection to the hardware server.
- **Image File:** Select the file to write to the flash memory.
  - Zynq devices:
    - Supported file formats for qspi flash types are BIN or MCS formats.
    - Supported file formats for nor and nand types is BIN format.
  - Non-Zynq devices:
    - Supported types for flash parts in non Zynq devices are BIT, ELF, SREC, MCS, BIN.
- **Offset:** Specify the offset relative to the Flash Base Address, where the file should be programmed.

  ***Note*:** Offset is not required for MCS files.
- **Flash Type:** Select a flash type.
  - Zynq devices:
    - qspi_single
    - qspi_dual_parallel
    - qspi_dual_stacked
    - nand_8

    - nand_16
    - nor
    - emmc

      Note: emmc flash type is applicable for Zynq UltraScale+ MPSoC and Versal 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 the hardware server does not exist, an error message stating `"Could not retrieve Flash Part information. Please check hardware server connection"` is displayed on the page. Based on the device detected, the dialog populates all the flash parts supported for the device.

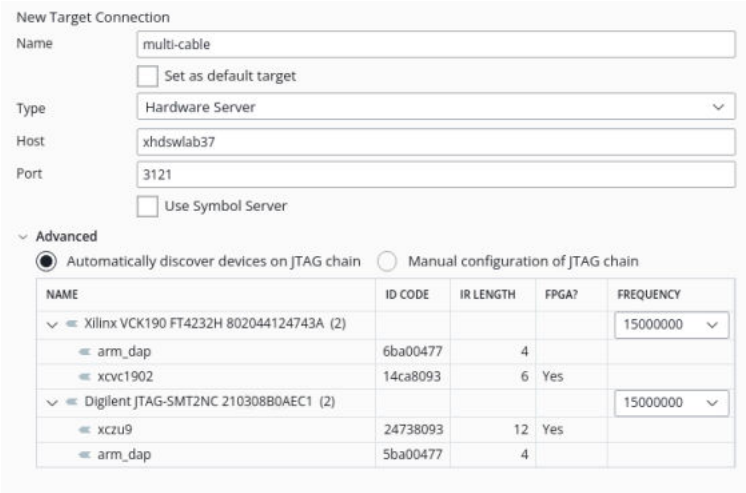
  Note: The appropriate part can be selected based on the design. For AMD boards, the part name can be found from the respective board's user guide.

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

Multi-Cable and Multi-Device Support

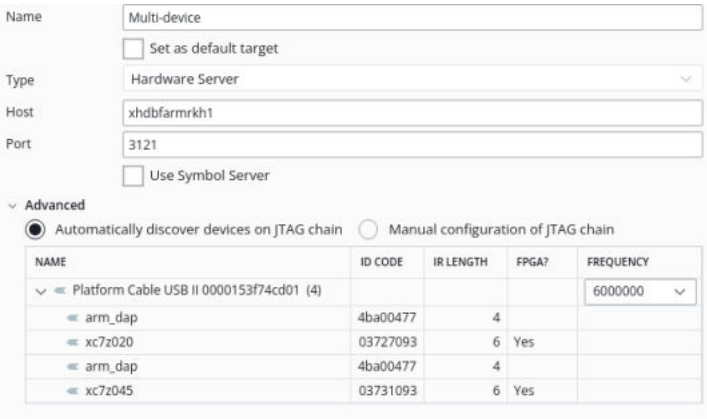
This feature of the AMD Vitis™ environment allows you to run and debug application projects, program the bitstream/PDI, and program the flash using multiple JTAG cables or multiple devices on a single JTAG chain. The main use cases are as follows:

- **Multi-cable:** In this use case, assume you have more than one board connected to the system and you want to work on all of the boards. The following is a multi-cable target connection view. Regarding target connection, please refer to Target Connections



- **Multi-device:** In this use case, assume you have multiple devices in a JTAG chain and you want to work on all of the devices. The following is a multi-device target connection view. Regarding target connection, please refer to Target Connections

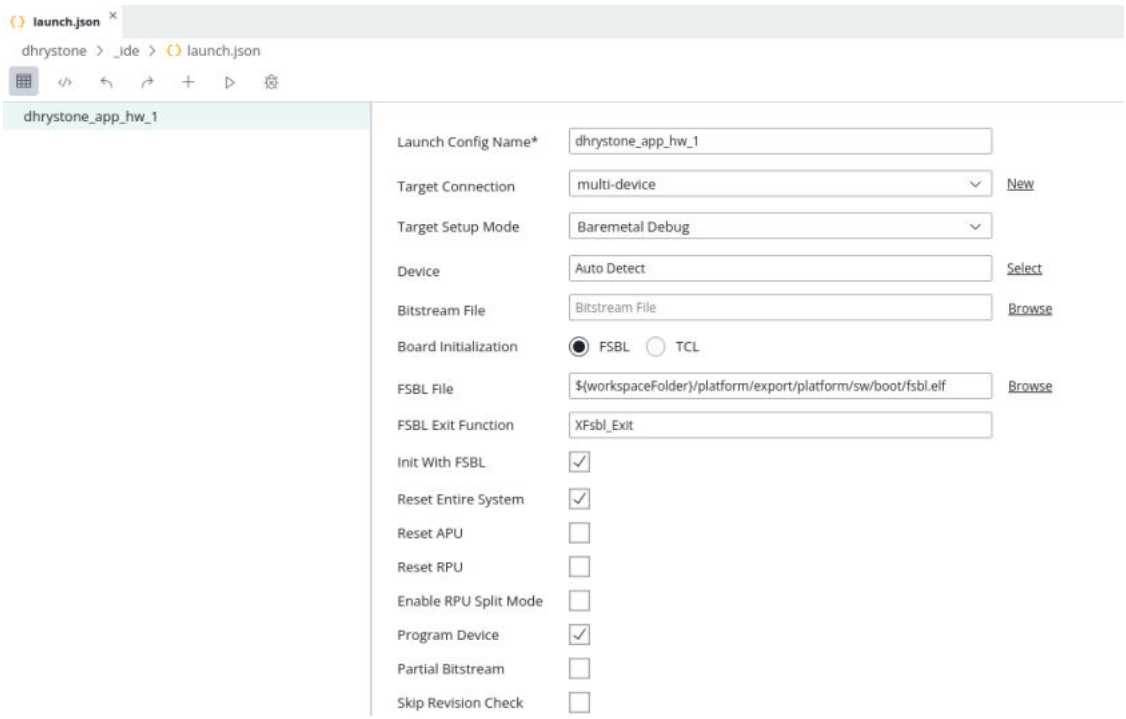
Figure 48: **Multi-device Target Connection View**



Debugging an Application with Multi-cable or Multi-device

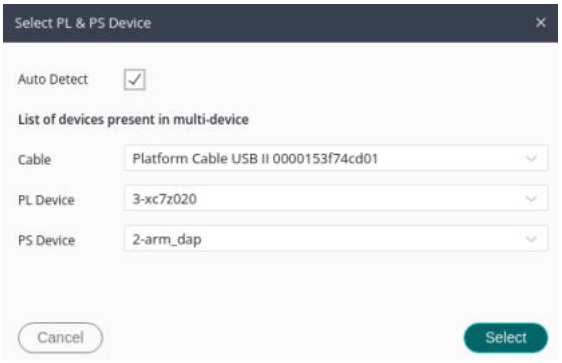
- Navigate to the debug launch configuration file (refer to Launch Configurations for details).
- Click **Select** beside the Device field.

Figure 49: **launch.json Tab**



- Deselect **Auto Detect**

Figure 50: **Select PL and PS Device Screen**



- Select the PS and PL device in this dialog according to your requirement.
- Click **Select** and **Start** to start debugging.

Programming a Device Using a Multi-Device/Multi-Cable Setup

1. Go to **Vitis→ Program Device**
2. To select a device from the multiple cable or multiple device setup, click the **Select** button to the right of the Device field to open the device selection dialog box.

3. Choose the appropriate device or cable from the drop down menu and click Program to program the FPGA/PDI file.

   ***Note*:** If the device status is Auto Detect, the Vitis tool automatically picks up the first device or cable from the list.

   ***Note*:** Use the same method to choose the PS and PL device when programming a flash.

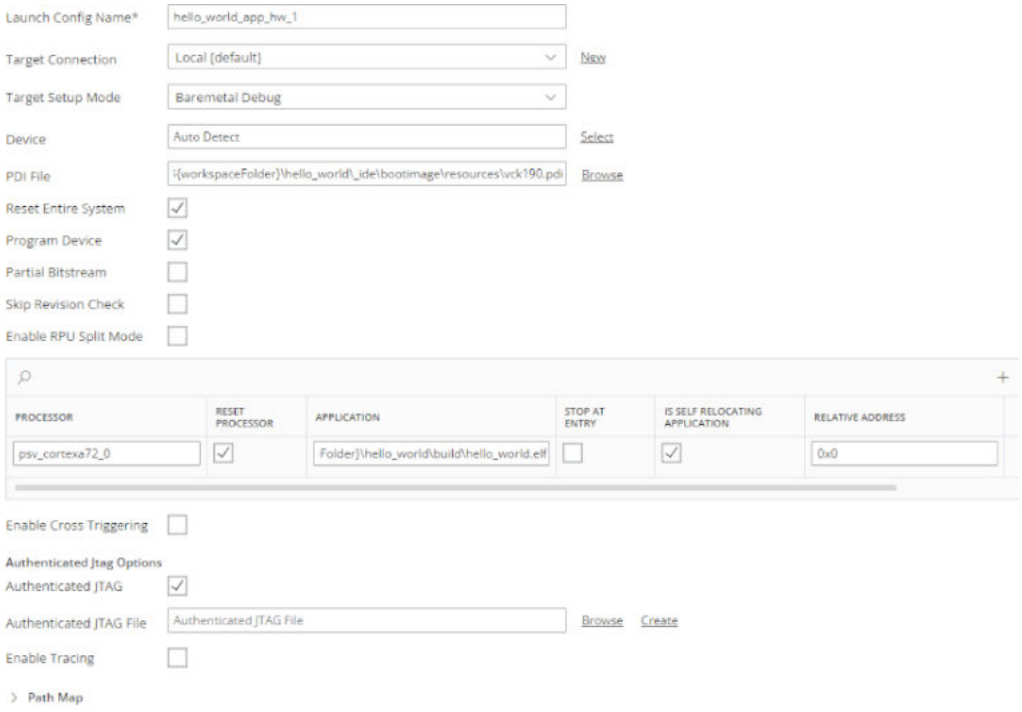
Authenticated Jtag Access

Authenticated JTAG Access provides a secure way to control and protect access to a device's JTAG interface. When this feature is enabled, you must successfully authenticate before performing any JTAG operations such as debug, programming, or device inspection.

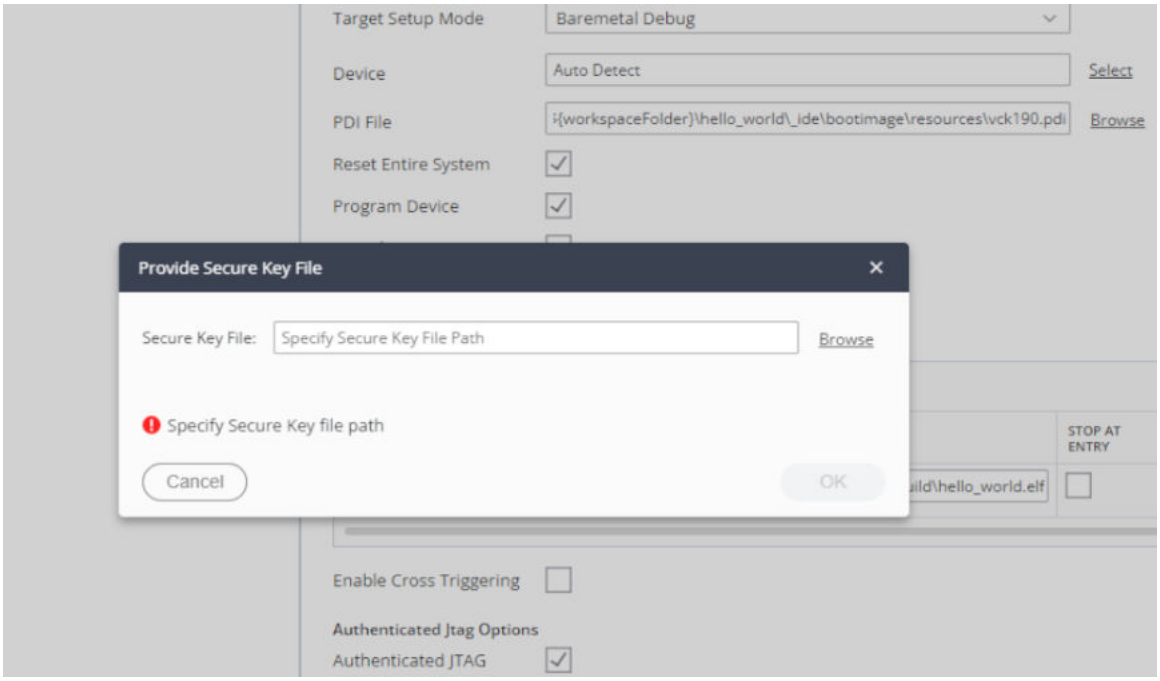
This feature helps prevent unauthorized access to the device through JTAG, even if someone has physical access to the hardware.

To access an authenticated Jtag, follow steps below:

1. Select the option called 'Authenticate JTAG' in the launch configuration.



2. After enabling this option, select the **Create** button and provide a path for the JTAG file to be created. This will generate a decryption file for opening your JTAG gate.



This file is then automatically populated in the Authenticated JTAG File field. The debug session will automatically use this file to open the JTAG gate for enabling secure access.

XEN Aware Debug

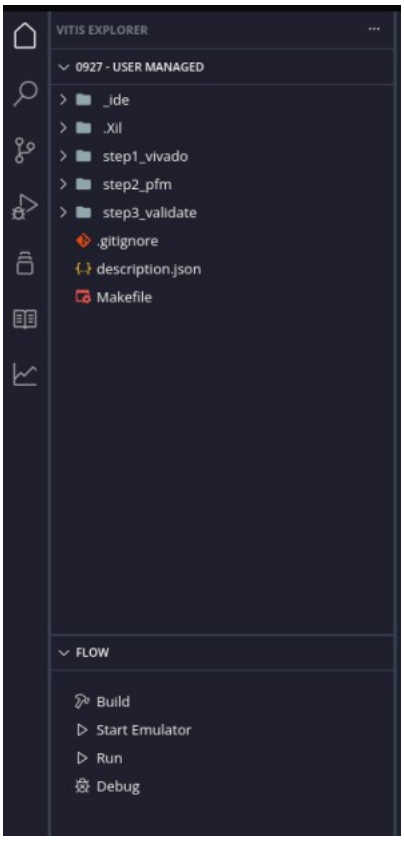
XEN Aware Debug is available as an early access feature. Detailed instructions are provided on the New Feature Preview page, where you can find more information about this feature and enable it if desired.

User Managed Mode

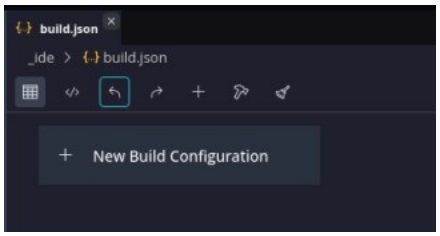
If you prefer to manage your source code hierarchy and build process in a unique way, for instance, by crafting your own Makefile or CMake files, you can use User Managed Mode. This mode empowers you by offering enhanced support and debugging capabilities, especially when you oversee the build flow.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

User Managed Mode is automatically activated when no components are found, but files are detected in the current workspace, allowing you to manage and configure their files directly and build or debug Makefile-based projects.

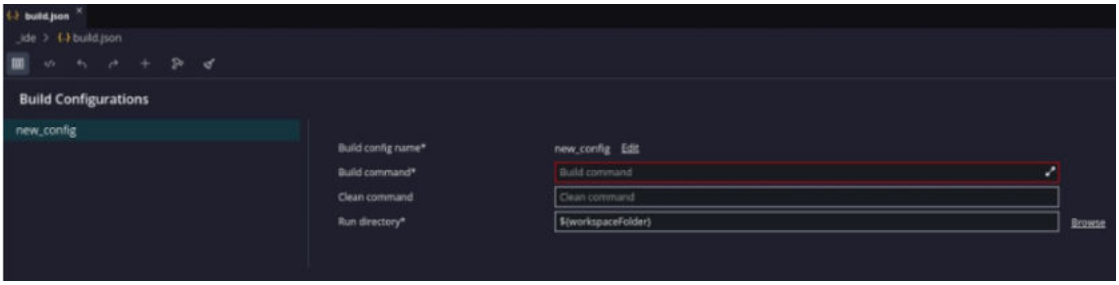
1. Start Vitis Unified IDE.
2. Open a workspace that has no JSON file and contains extra folders. User Managed Mode is automatically activated.



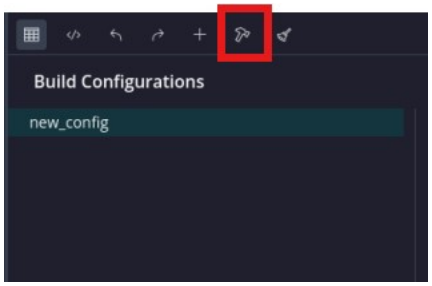
3. Create build configuration and build
   a. Hover the cursor over **Build** to select **Setting** button. Build configuration file Build.json file appears.



   b. Click **+ New Build Configuration** or you can click **+** button to generate a new configuration for a build.
   c. Input the build, clean and run directory.



d. Click **Build** to build the project.



4. Create Run/Debug launch configuration after build is finished.
   a. Hover the cursor over Debug and select **Setting**. Launch configuration file `Launch.json` appears.
   b. Click **+ New Launch Configuration** or click **+** button to create a new launch configuration.
   c. It brings up Create Launch Configuration wizard. Two design types are supported for embedded application component.
   d. Embedded Standalone Application has two debug options as following:
      - **Configure the device and start debug:** This is used to debug or run the bare-metal application component. Specify the XSA file and click **Submit** to create the configuration file. Wait until the Launch Configuration file pops up. Refer to Launch Configurations for setting references
      - **Attach to Running Target:** If you desire to debug on the already running target, select this option and click **Submit**. You need to provide the target connection.
   e. Embedded Linux Application has two debug options as following:
      i. Transfer the Elf file and start debugging

         If a Linux application has to be run/debugged, select this option and provide the elf. Launch configuration with the few sections is created for the app. Provide the necessary information post which you can run/debug it.

      ii. Attaching to Running Processor on Linux Target

         To debug a Linux application on an already running Linux target, select this option and click **Submit**. Launch configuration window appears. Provide the target connection and executable file, post which you can run/debug it.

Setting User Tool Chain

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

1. Open terminal in Linux system or open windows command prompt.

2. In the command line, input the following commands.

   For Windows

   ```
   set VITIS_IDE_USER_TOOLCHAIN=<custom_tool_chain_bin_directory_path>
   ```

   For Linux:

   ```
   export VITIS_IDE_USER_TOOLCHAIN=<custom_tool_chain_bin_directory_path>
   ```

You can start Vitis from the terminal or CMD window.

Vitis Utilities

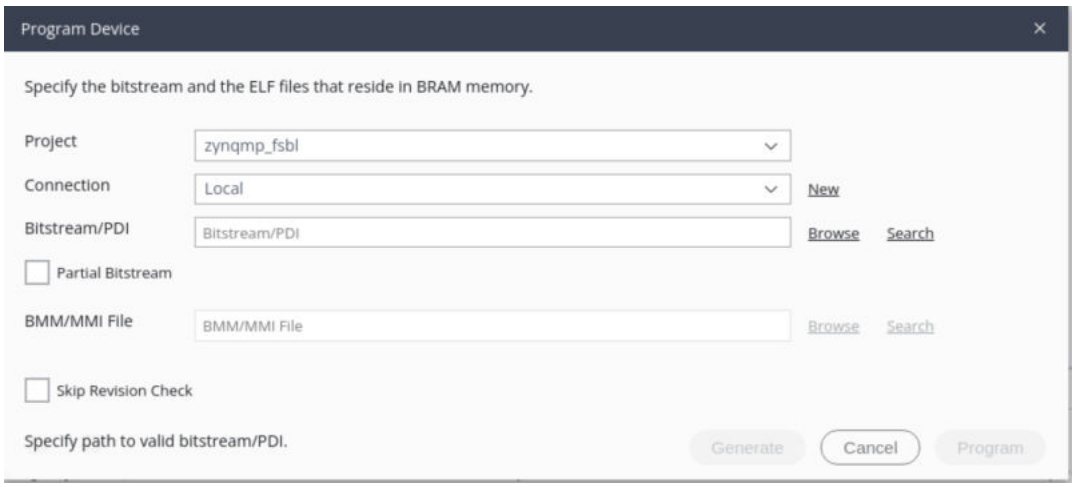
Software Debugger

Launch a console window to interact with XSDB. For more information on XSDB, see the *Software Debugger Reference Guide* (UG1725).

Program Device

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

Figure 51: **Program Device**



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

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

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

Vitis Terminal

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

Project Export and Import

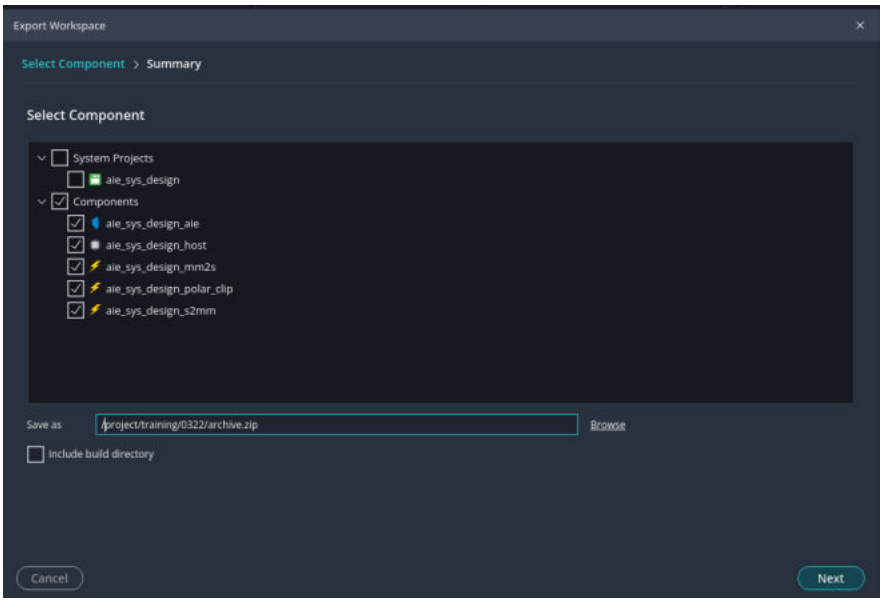
The AMD Vitis™ Unified IDE provides a simplified method for exporting or importing one or more AMD Vitis™ Unified IDE projects or components within your workspace.

Export Vitis Component

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

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

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



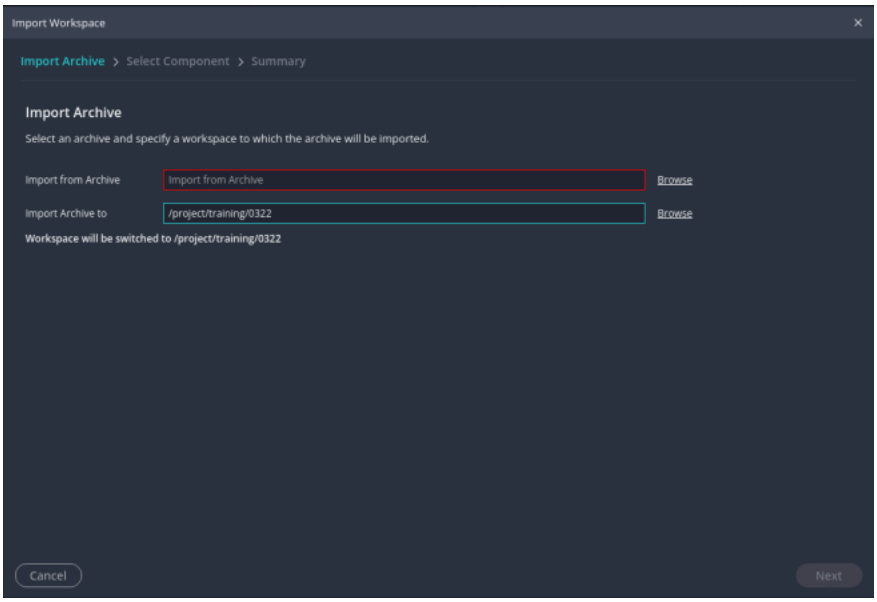
2. Select the component you want to export
3. Click **Browse** to select a location to save the Archive file. Then, click **Next**.
4. Review the summary of the exported components and click **Finish**.

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

Import Vitis Component

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

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

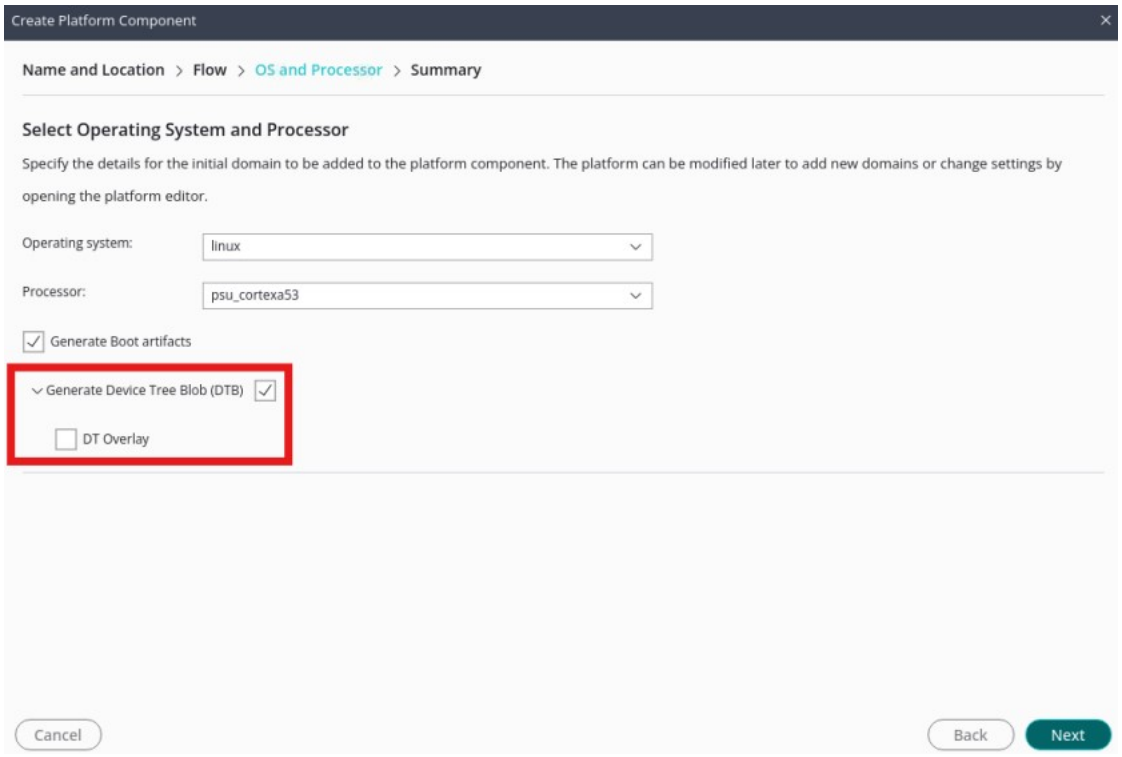


2. Select the archive file to be imported and specify the new workspace and click **Next**.
3. Select the component you want to import to your workspace in this page and click **Next**.
4. Review the summary of the imported components and click **Finish**.

Generating Device Tree

***Note*:** This function is available during the platform creation process, where you can enable the **Generate Device Tree Blob (DTB)** option to generate a DTB or DTB overlay. For more details, refer to Creating a Platform Component from XSA.

Figure 52: **Generating Device Tree**



Vitis Python API

Graphical development environments such as the Vitis IDE are useful for getting up to speed on development for a new processor architecture. It helps to abstract away and group most of the common functions into logical wizards that even a novice can use. However, scriptability of a tool is also essential for providing the flexibility to extend what is done with that tool. It is particularly useful when developing regression tests that be run nightly or when running a set of commands that are frequently used.

The Vitis command line interface (API) is an interactive and scriptable command-line interface to the Vitis IDE. As with other AMD tools, the scripting language for Vitis API is based on the Python. You can run Vitis API commands interactively or script the commands for automation.

Vitis API supports Vitis project management, configuration, building, and debugging, such as:

- Creating platform projects and domains
- Creating system and application projects
- Configuring and building domains/BSPs and applications
- Managing repositories
- Downloading and running applications on hardware targets
- Reading and writing registers
- Setting break points and watch expressions

***Note*:** You can view all the Python APIs from the Vitis menu: go to **Help**→ **Document**→ **Vitis Python API Documentation**.

Python API: A Command-line Tool for Creating and Managing Projects in Vitis

The Vitis IDE provides logical wizards within the development environment to help you develop your design. This is useful for developing a new processor architecture, particularly for novice users. For more flexibility and extended functionality, however, it is essential to become familiar with using scripts in the tool. Scripts are especially useful for developing regression tests that run at regular intervals or when running a frequently used set of commands. The AMD Vitis™ command line interface (CLI) is an interactive and scriptable command-line interface to the AMD Vitis™ IDE. As with other AMD tools, the scripting language for AMD Vitis™ CLI is based on the Python. You can run AMD Vitis™ CLI commands interactively or script the commands for automation. AMD Vitis™ CLI supports AMD Vitis™ project management, configuration, building, and debugging, such as:

- Creating platform projects and domains
- Creating system and application projects
- Configuring and building domains/BSPs and application
- Managing repositories
- Downloading and running applications on hardware targets
- Reading and writing registers
- Setting break points and watch expressions

Vitis CLI Commands

IMPORTANT! *You can find all the Vitis CLI command details in the installation folder:* *<Vitis_install>2025.1/Vitis/cli/api_docs/build/html/vitis.html*

Executing Python APIs

To execute Python APIs, first, you need to establish the connection between the server and the client. Python APIs can be executed in command line mode or as a Python script. The AMD Vitis™ Unified IDE supports two ways to run Python APIs:

1. Run Python API in CLI (Command-Line Interface) mode: Executing the Python API in CLI mode is supported in interactive mode only. For more details, see Vitis Interactive Python Shell in the *Vitis Reference Guide* (UG1702).

```
vitis -i (This opens the Vitis interactive shell)
```

2. Run Python API in a Python Script: The Vitis Unified IDE supports the execution of Python script in batch mode as well as in an interactive mode.

   In batch mode:

   ```
   In batch mode:
   $ vitis -s <.py>
   In Interactive mode:
   $ vitis -i
   (This openes the Python interactive shell)
   vitis [1]: run <.py>
   ```

   Some Python APIs are helpful in managing client life cycles, create Vitis workspaces and manage Vitis project flows. These APIs are described in the following table. Refer the Vitis API CLI documentation for more details.

   Table 12: **Python APIs: Manage Client**

Python API	Description	Example
create_client	Creates a client instance.	client=vitis.create.client()
dispose	Closes all client connections and terminates the connection to the server.	client.dispose()
exit	Closes the session.	exit()

   After creating and building the components and system project, the workspace can be directly opened in the Vitis IDE using the command below:

   ```
   vitis -w <workspace_path>
   ```

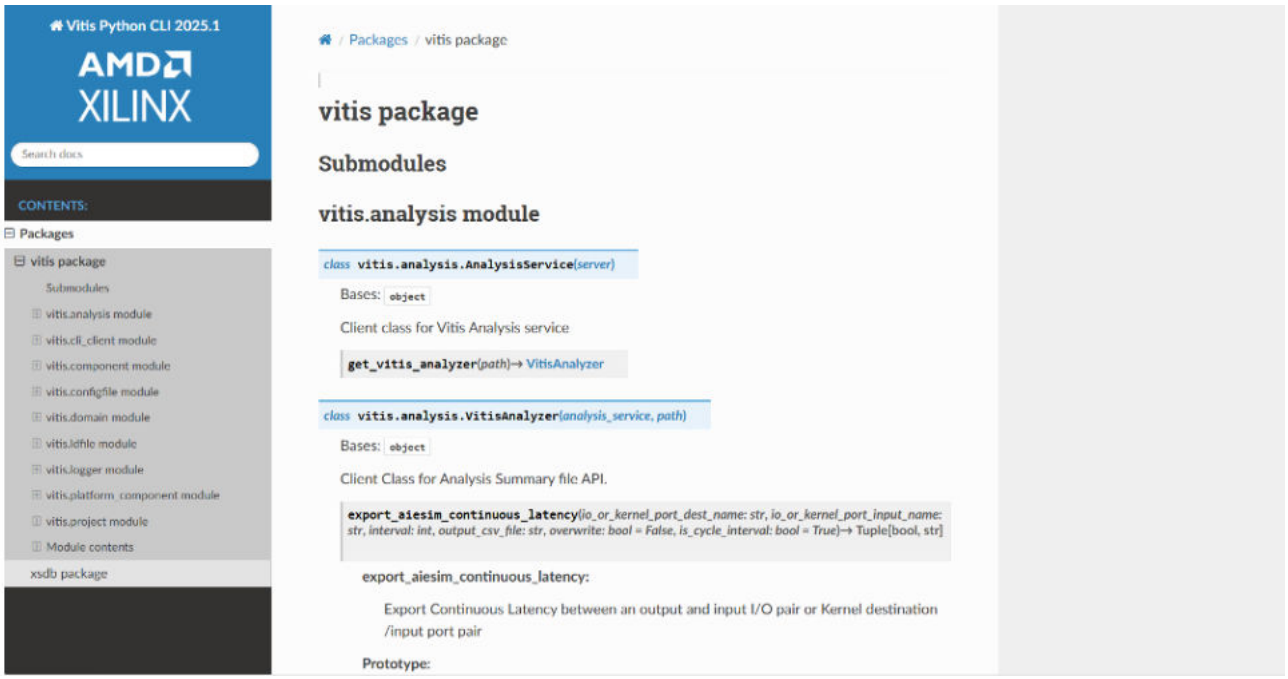
   Components and system projects are opened in the Vitis IDE with build and created status of `done`, if they are created and built through Python APIs.

   The Python APIs used for component creation and build are explained in the following sections.

All Python APIs Supported by Vitis

IMPORTANT! *For more details on all the Python APIs supported by Vitis, refer to* `<vitis_installation_path>/cli/api_docs/build/html/vitis.html`

Figure 53: **Python APIs Supported by Vitis Documentation**



Python examples are provided with the installation for reference:
`<vitis_installation_path>/cli/examples`

Managing Vitis IDE Components through Python APIs

Before running the script, you must set up the environment variables for the Vitis Unified IDE. Refer to "Setting Up the Vitis Tool Environment" in *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393) to set up the Vitis Unified IDE environment. The Vitis IDE supports Python APIs for project flow management, creation and building of the following Vitis components:

- Platform Control
- Application Component
- System Project

System Project

The Vitis Unified IDE supports Python APIs to create and build a system project. After creating platform component and application component, the system project is created. The application components then can be added to the system project and built. Host components can also be built individually. However, when system project build is triggered using Python APIs, it builds all the components associated and the system.

Table 13: **Python APIs: System Project**

Python API	Description	Python API Example
create_sys_project	Creates a system project for the given template.[1]	`proj = client.create_sys_project(name ='system_project', platform=<platform_path>)`
add_component	Adds the specified component to the given system project	`proj.add_component(name = "host_component")`
build	Initiates the build of a system project for the given build target.	`proj.build(target='hw')`

Notes:

1. Accelerated flows are not supported for Embedded installer.

The Python script to create a system project and build for target=Hardware to export Vitis metadata archive can be seen as:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path="./workspace")

# Defining names for platform, application_component and
plat_name="vck190_embd"
comp_name="standalone_embd_app"

# Create and build platform component for vck190 for
standalone_psv_cortexa72
platform_obj=client.create_platform_component(name=plat_name,
hw_design="vck190", cpu="psv_cortexa72_0", os="standalone")
platform_obj.build()

# This returns the platform xpfm path
platform_xpfm=client.find_platform_in_repos(plat_name)

# Create and build application component
comp = client.create_app_component(name=comp_name, platform =
platform_xpfm, domain = "standalone_psv_cortexa72_0", template =
"hello_world")
comp.build()
```

```
# Create system project
sys_proj = client.create_sys_project(name="system_project",
platform=platform_xpfm, template="empty_accelerated_application")
# Add application component to the system project
sys_proj_comp = sys_proj.add_component(name="hello_world")
# Build system project
sys_proj_comp.build()

vitis.dispose()
```

Conclusion

Apart from the discussed examples, a few more Python command examples are provided in `<VITIS_INSTALL_DIR>/2025.1/cli/examples`.

***Note*:** Accelerated flows are not supported in embedded installer.

For more details of all Vitis supported Python APIs, refer to the link:
`<vitis_install_path>cli/api_docs/build/html/vitis.html`

Python Vitis Commands

You can see a list of all the comprehensive Python AMD Vitis™ Commands.

The Vitis CLI can be launched using the following command.

```
vitis -i
```

```
client = vitis.create_client()
help(client)
```

Vitis Python command examples are provided in `<VITIS_INSTALL_DIR>/2025.1/cli/examples`.

All Vitis Python commands to rebuild a Vitis Workspace are provided in `<vitis workspace>/_ide/workspace_journal.py`.

Python XSDB Commands

You can view a list of all the comprehensive Python XSDB API using the following commands in AMD Vitis™ CLI.

Launch the Vitis CLI using the following command.

```
vitis -i
```

```
from xsdb import *
session = start_debug_session()
help("functions")
```

Note: You can find all the XSDB Python CLI command details in the following installation folder:
`<Vitis_install>2025.1/Vitis/cli/api_docs/build/html/xsdb.html`

Python XSDB Usage Examples

Debug Operations on Session Object

In this mode, you can create a session object and run debug operations on different debug targets using the same session object.

```
session=start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
session.targets(3)
# All subsequent commands are run on target 3, until the target is changed
with targets() function
session.dow("test.elf")
session.bpadd(addr='main')
session.con()
session.targets(4)
# All subsequent commands are run on target 4
session.dow("foo.elf")
session.bpadd(addr='foo')
session.con()
```

Debug Operations on Target Object

You can also use the Target object returned by targets() functions and run debug operations can be performed on these objects. The following is an example:

```
session = start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
ta3 = session.targets(3)
ta4 = session.targets(4)
# Run debug commands using target objects
ta3.dow("test.elf")
ta4.dow("foo.elf")
bp1 = ta3.bpadd(addr='main')
bp2 = ta4.bpadd(addr='foo')
ta3.con()
ta4.con()
...
bp1.status()
bp2.status()
```

For interactive usage, it is recommended to use commands and options instead of functions and arguments, as functions require a lot of extra typing. You have defined an interactive() function in xsdb module, which supports the commands. The following is an example:

```
Vitis-ng [1]: import xsdb
Vitis-ng [2]: xsdb.interactive()
% conn -host xhdbfarmrkb9
tcfchan#0
% ta
1 APU
    2 ARM Cortex-A9 MPCore #0 (Running)
    3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020
% ta 2
<xsdpy._target.Target object at 0x7fb2652d3520>
% stop
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffff28 (Suspended)
% q
Vitis-ng [3]:
```

Programming U-Boot over JTAG

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrkg7:3121")
# Disable Security gates to view PMU MB target
s.targets("--set", filter="name =~ PSU")
# By default, JTAG security gates are enabled
# This disables security gates for DAP, PLTAP and PMU.
s.mwr(0xffca0038, words=0x1ff)
time.sleep(0.5)
# Load and run PMU FW
s.targets("--set", filter="name =~ MicroBlaze PMU")
s.dow('pmufw.elf')
s.con()
time.sleep(0.5)
# Reset A53, load and run FSBL
s.targets("--set", filter="name =~ Cortex-A53 #0")
s.targets()
s.rst(type='processor')
s.dow('fsbl.elf')
s.con()
# Give FSBL time to run
time.sleep(5)
s.stop()
# Downloading the other Softwares like u-boot..etc using below command
s.dow('system.dtb', '-d', addr=0x100000)
s.dow('u-boot.elf')
s.dow('bl31.elf')
s.con()
time.sleep(5)
s.stop()
```

Generate SVF Files

```
# Reset values of respective cores
core = 0
apu_reset_a53 = [0x380e, 0x340d, 0x2c0b, 0x1c07]
# Generate SVF file for linking DAP to the JTAG chain
# Next 2 steps are required only for Rev2.0 silicon and above.
s = xsdb.start_debug_session()
s.connect(url="TCP:xhdxxxx41x:3121")
svf = s.svf()
svf.config('--linkdap', scan_chain=[0x14738093, 12, 0x5ba00477, 4],
           device_index=1, out="dapcon.svf")
svf.generate()

svf = s.svf()
# Configure the SVF generation
svf.config(scan_chain=[0x14738093, 12, 0x5ba00477, 4],
           device_index=1, cpu_index=core, delay=10, out="fsbl_hello.svf")
# Record writing of bootloop and release of A53 core from reset
svf.mwr(0xffff0000, 0x14000000)
svf.mwr(0xfd1a0104, apu_reset_a53[core])
# Record stopping the core
svf.stop()
# Record downloading FSBL
svf.dow(file='zynq_fsbl.elf')
# Record executing FSBL
svf.con()
svf.delay(100000)
# Record some delay and then stopping the core
svf.stop()
# Record downloading the application
svf.dow(file='zynq_hello.elf')
# Record executing application
svf.con()
# Generate SVF
svf.generate()
```

Using readjtaguart Function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.readjtaguart()
s.con()
s.readjtaguart('--stop')  # after you are done
```

Using readjtaguart_file Function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.readjtaguart(file='streams.log', mode='w')
s.con()
s.readjtaguart('--stop')  # after you are done
```

Using jtag_terminal Function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.jtagterminal()
s.con()
s.jtagterminal('--stop')  # after you are done
```

Performing_standalone_app_debug:

```
Vitis [2]: import xsdb

Vitis [3]: s = xsdb.start_debug_session()

# connecting to the target
Vitis [4]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[4]: 'tcfchan#0'
```

```
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
Vitis [5]: s.targets()
     1  APU
        2  ARM Cortex-A9 MPCore #0 (Running)
        3  ARM Cortex-A9 MPCore #1 (Running)
     4  xc7z020

Vitis [6]: s.targets("--set", filter="name =~ APU")
Out[6]: <xsdb._target.Target at 0x7f01764c6e20>

Vitis [7]: s.targets()
     1* APU
        2  ARM Cortex-A9 MPCore #0 (Running)
        3  ARM Cortex-A9 MPCore #1 (Running)
     4  xc7z020

Vitis [8]: s.rst()

Vitis [9]: s.fpga(file='zc702.bit')
100%    3.86MB    1.66MB/s    00:02ETA


Vitis [10]: s.targets(2)
Out[10]: <xsdb._target.Target at 0x7f014d1f1610>

Vitis [11]: s.loadhw(hw='zc702.xsa')
INFO: [Hsi 55-2053] elapsed time for repository (/proj/xbuilds/
HEAD_daily_latest/installs/lin64/Vitis/HEAD/data/embeddedsw) loading 1
seconds

# run FSBL for ps_init
Vitis [12]: s.dow('fsbl.elf')
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffff28 (Suspended)
Downloading Program -- fsbl.elf
    section, .text: 0x00000000 - 0x0000fc8b
    section, .handoff: 0x0000fc8c - 0x0000fcd7
    section, .init: 0x0000fcd8 - 0x0000fce3
    section, .fini: 0x0000fce4 - 0x0000fcef
    section, .rodata: 0x0000fcf0 - 0x00010043
    section, .data: 0x00010048 - 0x00013017
    section, .mmu_tbl: 0x00014000 - 0x00017fff
    section, .init_array: 0x00018000 - 0x00018003
    section, .fini_array: 0x00018004 - 0x00018007
    section, .rsa_ac: 0x00018008 - 0x0001903f
    section, .bss: 0x00019040 - 0x0001b3ff
    section, .heap: 0x0001b400 - 0x0001d3ff
    section, .stack: 0xffff0000 - 0xffffd3ff

Successfully downloaded  fsbl.elf
Setting PC to Program Start Address 0x00000000

Vitis [13]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Vitis [14]: s.stop()

Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xfcac (Suspended)
```

```
# Download the application program
Vitis [15]: s.dow('test_jtag_uart.elf')
Downloading Program -- test_jtag_uart.elf
    section, .text: 0x00100000 - 0x00101503
    section, .init: 0x00101504 - 0x0010150f
    section, .fini: 0x00101510 - 0x0010151b
    section, .rodata: 0x0010151c - 0x001016cc
    section, .data: 0x001016d0 - 0x00101b3f
    section, .eh_frame: 0x00101b40 - 0x00101b43
    section, .mmu_tbl: 0x00104000 - 0x00107fff
    section, .init_array: 0x00108000 - 0x00108003
    section, .fini_array: 0x00108004 - 0x00108007
    section, .bss: 0x00108008 - 0x0010802f
    section, .heap: 0x00108030 - 0x0010a02f
    section, .stack: 0x0010a030 - 0x0010d82f

Successfully downloaded  test_jtag_uart.elf
Setting PC to Program Start Address 0x00100000

Vitis [16]: s.bpadd(addr='main')
Out[16]: <xsdb._breakpoint.Breakpoint at 0x7f016fe59af0>

Info: Breakpoint 0 status:
     target 2: {Address: 0x100564 Type: Hardware}
Info: Breakpoint 0 status:
     target 3: {At col 4: Undefined identifier main. Invalid expression}
Vitis [17]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100570 (Breakpoint)
main() at ../src/helloworld.c: 57
57: int l_int_b = 20;
Vitis [18]: s.stp()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100578 (Step)
main() at ../src/helloworld.c: 58
58: int l_int_a = 40;
Vitis [19]: s.stp()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100580 (Step)
main() at ../src/helloworld.c: 60
60: init_platform();
Vitis [20]: s.rrd()
              r0: 00000000              r1: 00000000              r2:
00000001
              r3: 00000028              r4: 00000003              r5:
0000001e
              r6: 0000ffff              r7: f8f00000              r8:
0000767b
              r9: ffffffff             r10: 00000000             r11:
0010c02c
             r12: 00000000              sp: 0010c020              lr:
001007bc
              pc: 00100580            cpsr: 6000005f             usr
             fiq                       irq                       abt
             und                       svc                       mon
             vfp                      cp15                   Jazelle
 gpv_qos301_cpu           gpv_qos301_dmac           gpv_qos301_iou
  gpv_trustzone                   l2cache                    mpcore
```

```
Vitis [21]: s.mrd(0xe000d000)
E000D000 :  800A0000

# Local variable value can be modified
Vitis [22]: s.locals()
l_int_b   : 20
l_int_a   : 40

Vitis [23]: s.locals(name='l_int_b', val=815)

Vitis [24]: s.locals(name='l_int_b')
l_int_b   : 815

# Global variables and be displayed, and its value can be modified
Vitis [25]: s.print(expr='g_int_a')
g_int_a   : 60

Vitis [26]: s.print(expr='g_int_a', val=9919)

Vitis [27]: s.print(expr='g_int_a')
g_int_a   : 9919

# Expressions can be evaluated and its value can be displayed
Vitis [28]: s.print('--add', expr='l_int_a + l_int_b')
l_int_a + l_int_b  : 855

# Step over a line of source code
Vitis [29]: s.nxt()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100584 (Step)
main() at ../src/helloworld.c: 61
61: test_function(l_int_a, l_int_b);

# View stack trace
Vitis [30]: s.bt()
0  0x100584 main()+32:../src/helloworld.c, line 61
1  0x1007bc _start()+88:xil-crt0.S, line 119
2  unknown-pc

# Set a breakpoint at exit and resume execution
Vitis [31]: s.bpadd(addr='&exit')
Out[29]: <xsdb._breakpoint.Breakpoint at 0x7f016fd2c5e0>

Info: Breakpoint 1 status:
     target 2: {Address: 0x1012ac Type: Hardware}
Info: Breakpoint 1 status:
     target 3: {At col 5: Undefined identifier exit. Invalid expression}
Vitis [32]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x1012ac (Breakpoint)

 # View stack trace
Vitis [33]: s.bt()
0  0x1012ac exit()+0
1  0x71034f12
```

Select_target_based_on_target_properties:

```
Vitis [2]: import xsdb

Vitis [3]: s = xsdb.start_debug_session()

# connecting to the target
Vitis [4]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[4]: 'tcfchan#0'

# check the jtag targets connected, the IDs listed with jtag targets are
called node IDs
Vitis [5]: s.jtag_targets()
     1  Digilent JTAG-SMT1 210203344713A ( is_pdi_programmable)
        2  arm_dap (idcode 4ba00477 irlen 4 is_pdi_programmable)
        3  xc7z020 (idcode 23727093 irlen 6 fpga is_pdi_programmable)

# check the targets connected, the IDs listed with targets are called
target IDs
Vitis [6]: s.targets()
     1  APU
        2  ARM Cortex-A9 MPCore #0 (Breakpoint)
        3  ARM Cortex-A9 MPCore #1 (Running)
     4  xc7z020

# check jtag target properties of 2nd device (2nd ARM DAP). Note the
target_ctx here.
Vitis [7]: s.jtag_targets('-t', filter="node_id == 2")
Out[7]:
{'jsn-JTAG-SMT1-210203344713A-4ba00477-0': {'target_ctx': 'jsn-JTAG-
SMT1-210203344713A-4ba00477-0',
  'level': 1,
  'node_id': 2,
  'is_open': 1,
  'is_active': 1,
  'is_current': 0,
  'name': 'arm_dap',
  'jtag_cable_name': 'Digilent JTAG-SMT1 210203344713A',
  'state': '',
  'jtag_cable_manufacturer': 'Digilent',
  'jtag_cable_product': 'JTAG-SMT1',
  'jtag_cable_serial': '210203344713A',
  'idcode': '4ba00477',
  'irlen': '4',
  'is_fpga': 0,
  'is_pdi_programmable': 0}}

# using the target context, select the targets associated with the JTAG
target (2nd ARM DAP - node id = 2)
Vitis [8]: s.targets(filter="jtag_device_ctx == jsn-JTAG-
SMT1-210203344713A-4ba0
      ...: 0477-0")
     1  APU
        2  ARM Cortex-A9 MPCore #0 (Breakpoint)
        3  ARM Cortex-A9 MPCore #1 (Running)
```

Debugging_prog_already_running_on_target:

```
Vitis [1]: import xsdb

Vitis [2]: s = xsdb.start_debug_session()

# connecting to the target
Vitis [3]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[3]: 'tcfchan#0'

# Select the target on which the program is running and specify the symbol
file using the
# memmap command

Vitis [4]: s.target(2)
Out[4]: <xsdb._target.Target at 0x7f3167dd12e0>

Vitis [5]: s.memmap(file='test_jtag_uart.elf')

# When the symbol file is specified, the debugger maps the code on the
target to the symbol
# file. bt command can be used to see the back trace. Further debug is
possible, as shown in
# the first example

Vitis [6]: s.stop()
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x10101c (Suspended)
sleep_A9() at sleep.c: 63
63: } while (tCur < tEnd);

Vitis [7]: s.backtrace()
0  0x10101c sleep_A9()+56:sleep.c, line 63
1  0x100608 test_function()+76:../src/helloworld.c, line 78
2  0x100590 main()+44:../src/helloworld.c, line 61
3  0x1007ac _start()+88:xil-crt0.S, line 119
4  unknown-pc
```

Debug_app_on_zu_plus_mpsoc:

```
Vitis [1]: import xsdb

Vitis [2]: s = xsdb.start_debug_session()

# connect to remote hw_server by specifying its url.
# If the hardware is connected to a local machine,-url option and the <url>
# are not needed. connect command returns the channel ID of the connection
Vitis [3]: s.connect(url="TCP:xhdbfarmrkk5:3121")
tcfchan#0
Out[3]: 'tcfchan#0'

# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
Vitis [4]:  s.targets()
     1  PS TAP
        2  PMU
        3  PL
     4  PSU
        5  RPU (Reset)
           6  Cortex-R5 #0 (RPU Reset)
```

```
           7  Cortex-R5 #1 (RPU Reset)
        8  APU (L2 Cache Reset)
           9  Cortex-A53 #0 (APU Reset)
          10  Cortex-A53 #1 (APU Reset)
          11  Cortex-A53 #2 (APU Reset)
          12  Cortex-A53 #3 (APU Reset)

Vitis [5]: s.targets("--set", filter="name =~ PSU")
Out[5]: <xsdb._target.Target at 0x7fe9ca93bac0>

Vitis [6]: s.ta()
     1  PS TAP
        2  PMU
        3  PL
     4* PSU
        5  RPU (Reset)
           6  Cortex-R5 #0 (RPU Reset)
           7  Cortex-R5 #1 (RPU Reset)
        8  APU (L2 Cache Reset)
           9  Cortex-A53 #0 (APU Reset)
          10  Cortex-A53 #1 (APU Reset)
          11  Cortex-A53 #2 (APU Reset)
          12  Cortex-A53 #3 (APU Reset)


Vitis [7]: s.rst(type='system')

# Configure the FPGA. When the active target is not a FPGA device,
# the first FPGA device is configured
Vitis [8]: s.fpga(file='zcu102.bit')
100%    3.86MB    1.66MB/s    00:02ETA

Vitis [9]: s.targets(10)
Out[9]: <xsdb._target.Target at 0x7fe9a99347f0>

Vitis [10]: s.rst(type='cores')

Info: Cortex-A53 #0 (target 9) Stopped at 0xffff0000 (Reset Catch)
Info: Cortex-A53 #1 (target 10) Stopped at 0xffff0000 (Reset Catch)
Info: Cortex-A53 #2 (target 11) Stopped at 0xffff0000 (Reset Catch)
Info: Cortex-A53 #3 (target 12) Stopped at 0xffff0000 (Reset Catch)

# run fsbl to initialize PS
Vitis [11]: s.dow('fsbl_a53.elf')
Downloading Program -- fsbl_a53.elf
    section, .text: 0xfffc0000 - 0xfffd65b7
    section, .note.gnu.build-id: 0xfffd65b8 - 0xfffd65db
    section, .init: 0xfffd6600 - 0xfffd6633
    section, .fini: 0xfffd6640 - 0xfffd6673
    section, .rodata: 0xfffd6680 - 0xfffd6b94
    section, .sys_cfg_data: 0xfffd6bc0 - 0xfffd732b
    section, .mmu_tbl0: 0xfffd8000 - 0xfffd800f
    section, .mmu_tbl1: 0xfffd9000 - 0xfffdafff
    section, .mmu_tbl2: 0xfffdb000 - 0xfffdefff
    section, .data: 0xfffdf000 - 0xfffe02c7
    section, .sbss: 0xfffe02c8 - 0xfffe02ff
    section, .bss: 0xfffe0300 - 0xfffe297f
    section, .heap: 0xfffe2980 - 0xfffe2d7f
    section, .stack: 0xfffe2d80 - 0xfffe4d7f
    section, .dup_data: 0xfffe4d80 - 0xfffe6047
    section, .handoff_params: 0xfffe9e00 - 0xfffe9e87
    section, .bitstream_buffer: 0xffff0040 - 0xffffffc3f
```

```
Successfully downloaded  fsbl_a53.elf
Setting PC to Program Start Address 0xfffc0000

Vitis [12]: s.con()

Info: Cortex-A53 #1 (target 10) Running
Vitis [13]: s.stop()

Info: Cortex-A53 #1 (target 10) Stopped at 0xfffd2c7c (External Debug
Request)

# run the application
Vitis [14]: s.dow('zcu102_hello.elf')
Downloading Program -- zcu102_hello.elf
    section, .text: 0x00000000 - 0x000020b3
    section, .init: 0x000020c0 - 0x000020f3
    section, .fini: 0x00002100 - 0x00002133
    section, .rodata: 0x00002140 - 0x00002360
    section, .rodata1: 0x00002361 - 0x0000237f
    section, .sdata2: 0x00002380 - 0x0000237f
    section, .sbss2: 0x00002380 - 0x0000237f
    section, .data: 0x00002380 - 0x00002bb7
    section, .data1: 0x00002bb8 - 0x00002bbf
    section, .note.gnu.build-id: 0x00002bc0 - 0x00002be3
    section, .ctors: 0x00002be4 - 0x00002bff
    section, .dtors: 0x00002c00 - 0x00002bff
    section, .eh_frame: 0x00002c00 - 0x00002c03
    section, .mmu_tbl0: 0x00003000 - 0x0000300f
    section, .mmu_tbl1: 0x00004000 - 0x00005fff
    section, .mmu_tbl2: 0x00006000 - 0x00009fff
    section, .preinit_array: 0x0000a000 - 0x00009fff
    section, .init_array: 0x0000a000 - 0x0000a007
    section, .fini_array: 0x0000a008 - 0x0000a047
    section, .sdata: 0x0000a048 - 0x0000a07f
    section, .sbss: 0x0000a080 - 0x0000a07f
    section, .tdata: 0x0000a080 - 0x0000a07f
    section, .tbss: 0x0000a080 - 0x0000a07f
    section, .bss: 0x0000a080 - 0x0000a0bf
    section, .heap: 0x0000a0c0 - 0x0000c0bf
    section, .stack: 0x0000c0c0 - 0x0000f0bf

Successfully downloaded  zcu102_hello.elf
Setting PC to Program Start Address 0x00000000

Vitis [15]: s.bpadd(addr='main')
Out[15]: <xsdb._breakpoint.Breakpoint at 0x7fe9cae59cd0>

Info: Breakpoint 0 status:
     target 6: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
     target 7: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
     target 9: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
     target 10: {Address: 0xd00 Type: Hardware}
Info: Breakpoint 0 status:
     target 11: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
     target 12: {At col 4: Undefined identifier main. Invalid expression}
Vitis [16]: s.con()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd08 (Breakpoint)
```

```
main() at ../src/helloworld.c: 29
29: int l_int_b = 20;
Vitis [17]: s.rrd()
      r0: 00000000        r1: 00000000        r2: 00000000        r3:
00000004
      r4: 0000000f        r5: ffffffff        r6: 0000001c        r7:
00000002
      r8: ffffffff        r9: 00000000       r10: fd5c0090       r11:
00000000
     r12: 00000000       r13: 00000000       r14: 00000000       r15:
00000000
     r16: 00000000       r17: 00000000       r18: 00000000       r19:
00000000
     r20: 00000000       r21: 00000000       r22: 00000000       r23:
00000000
     r24: 00000000       r25: 00000000       r26: 00000000       r27:
00000000
     r28: 00000000       r29: 0000e0a0       r30: 00000f34        sp:
0000e0a0
      pc: 00000d08      cpsr: 600002cd       vfp                 sys
     dbg            acpu_gic


Vitis [18]: s.stp()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd10 (Step)
main() at ../src/helloworld.c: 30
30: int l_int_a = 40;
Vitis [19]: s.stp()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd18 (Step)
main() at ../src/helloworld.c: 32
32: init_platform();

# Local variable value can be modified
Vitis [20]: s.locals()
l_int_b   : 20
l_int_a   : 40

Vitis [21]: s.locals(name='l_int_b', val=815)

Vitis [22]: s.locals(name='l_int_b')
l_int_b   : 815

# Global variables and be displayed, and its value can be modified
Vitis [23]: s.print(expr='g_int_a')
g_int_a   : 60

Vitis [24]: s.print(expr='g_int_a', val=9919)

Vitis [25]: s.print(expr='g_int_a')
g_int_a   : 9919

# Expressions can be evaluated and its value can be displayed
Vitis [26]: s.print('--add', expr='l_int_a + l_int_b')
l_int_a + l_int_b  : 855

# Step over a line of source code
Vitis [27]: s.nxt()

Info: Cortex-A53 #1 (target 10) Running
```

```
Info: Cortex-A53 #1 (target 10) Stopped at 0xd1c (Step)
main() at ../src/helloworld.c: 33
33: test_function(l_int_a, l_int_b);

# View stack trace
Vitis [28]: s.bt()
0  0xd1c main()+28:../src/helloworld.c, line 33
1  0xf34 _startup()+124:xil-crt0.S, line 157

# Set a breakpoint at exit and resume execution
Vitis [29]: s.bpadd(addr='&exit')
Info: Breakpoint 1 status:
     target 6: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
     target 7: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
     target 9: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
     target 10: {Address: 0x1a90 Type: Hardware}
Info: Breakpoint 1 status:
     target 11: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
     target 12: {At col 5: Undefined identifier exit. Invalid expression}
Out[29]: <xsdb._breakpoint.Breakpoint at 0x7fe9a35e9ca0>

Vitis [30]: s.con()

Info: Cortex-A53 #1 (target 10) Running

Vitis [31]: s.stop()
Info: Cortex-A53 #1 (target 10) Stopped at 0x192c (External Debug Request)
sleep_A53() at sleep.c: 71
71: } while (tCur < tEnd);

# View stack trace
Vitis [32]: s.bt()
0  0x192c sleep_A53()+44:sleep.c, line 71
1  0x192c sleep_A53()+44:sleep.c, line 70
```

Section IV

XSDB Command Usage

IMPORTANT! *For more information on XSDB command usage, refer to Software Debugger Reference Guide (UG1725).*

XSCT to Python API Migration

In this release:

- XSCT is deprecated
- Debugging can still be carried out using XSDB, or you can convert your scripts to the Python CLI
- The Vits Python API can be used for project management. One useful way to learn how to use the Python API is to review the `workspace_journal.py` file, which can be found by clicking Vitis→**Workplace Journal**

See Section III: Vitis Python API and Vitis Scripting Flows Tutorial for further details.

app

Application project management

Table 14: **app- Application Project Management**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
app create	**-name** <application-name> Name of the application to be created. **-platform** <platform-name> Name of the platform. Use "repo -platforms" to list available pre-defined platforms. **-domain** Name of the domain. Use "platform report <platform-name>" to list the available system configurations in a platform. **-hw** <hw-spec> HW specification file exported from AMD Vivado™ (XSA). **-sysproj** <system-project> Name of the system project. Use "sysproj list" to know available system projects in the workspace. **-proc** <processor> Processor core for which the application should be created	1. client.create_aie_component (For AIE applications) 2. client.create_app_component (For host components)	Required arguments 1. **name** = <comp_name> AIE component name. 2. **platform** = <platform> or **part** = <part> Platform/part for which component is to be created. User needs to specify only one amongst platform or part. Optional arguments 1. **template** = <template>Template for the component to be created.	In XSCT, the system project is mentioned while creating the component (application), on the other hand, in Python CLI, the component is created first and then it is added to the system project. This gives a flexibility to add already created components to be part of various system projects.
app create (cont'd)	**-template** <application template> Name of the template application. Default is 'Hello World'. Use "repo -apps" to list available template applications **-os** <os-name> OS type. Default type is standalone. **-lang** <programming language> Programming language can be c or c++. **-arch** <arch-type> Processor architecture, <arch-type> can be 32 or 64 bits. This option is used to build the project with 32/64 bit toolchain		Required arguments 1. **name** = <comp_name>Application component name. 2. **platform** = <platform> Platform for which component is to be created. In case of baremetal platforms, user can specify the domain along with platform. Optional Arguments 1. **template** = <template> Template for the component to be created. 2. **domain** = <domain> Specify the domain when there is more than one domain on the platform.	
app remove	**app_name**	client.delete_component	**name** = <comp_name> Component name.	

Table 14: **app- Application Project Management** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
switch	**app_name**	client.get_component	**name** = <comp_name> Component name.	Get command returns the app object that can be used to run app commands.
app list	**-dict** List all the applications for the workspace in Tcl dictionary format. Without this option, applications are listed in tabular format. (This argument is not supported in Python CLI)	client.list_components		
app build	**-name** <app-name> Name of the application to be built. **-all** Name of the application to be built.Option to Build all the application projects. (No support in Python CLI for -all)	component.build		To build a component, component object is required. For example, component1 = client.get_component('aie_component1') component1.build()
app clean	**-name** <app-name>	component.clean	omponent.clean	
app report	**app_name**	component.report		

bsp

Configure BSP settings of a baremetal domain

Table 15: **bsp- Configure BSP Settings of a Baremetal Domain**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
bsp config	**<param> <value>**	set_config	Required Arguments: • **Option** = <"lib"/"os"/"proc"> Valid options are "lib"/"os"/"proc". • **param_value** = [key1:value1, key2:value2, **] List of key:value pairs to be set. Use list_param("lib"/"os"/"proc") to get the list of configurable parameters. Optional Arguments: **lib_name**= <lib_name> Library name in case 'lib' option is selected.	
getdrives	**-dict** Return the result as <IP-name driver:version> pairs. (No support for this in Python CLI)	domain.get_drivers	None	
getlibs	**-dict** Return the result as <lib-name version> pairs	domain.get_libs		
getos	**-dict** Return the result as <os-name version> pair	domain.get_os		
listparams	**-lib** <lib-name> Return the configurable parameters of the library in BSP **-os** Return the configurable parameters of the OS in BSP **-proc** Return the configurable parameters of the processor in BSP.	domain.list_params	Required Arguments **option** = <"lib"/"os"/"proc"> Valid options are "lib"/"os"/"proc" Optional Arguments **lib_name** = <lib_name> Library name if 'lib' option is selected	
regenerate		domain.regenerate		
reload				No support in Python CLI
write				No support in Python CLI
removelib		domain.remove_lib		Required Arguments: **lib_name** = <lib_name> Library to be added to the BSP settings

Table 15: **bsp- Configure BSP Settings of a Baremetal Domain** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
setdriver	**-driver** <driver-name> Driver to be assigned to an IP **-ip** <ip-name> IP instance for which the driver has to be added. **-ver** <version> Driver version.	domain.update_path	In the following arguments, option is supposed to be set to 'DRIVER', for setdriver **option** = <'OS'/'DRIVER'/'LIBRARY'> Valid options are 'OS', 'DRIVER' and 'LIB' **name** = <OS/Driver/Library name> Name of the OS/Driver/Library **new_path** = <OS/Driver/Library path> New path to be set for the mentioned OS/Driver/Library	In Python CLI, instead of version, path is to be passed.
setlib	**-name** <lib-name> Library to be added to the BSP settings. This is the default option, so lib-name can be directly specified as an argument without usingthis option. **-ver** <version> Library version.	domain.update_path	In the following arguments, option is supposed to be set to 'LIBRARY', for setlib **option** = <'OS'/'DRIVER'/'LIBRARY'> Valid options are 'OS', 'DRIVER' and 'LIB' **name** = <OS/Driver/Library name> Name of the OS/Driver/Library **new_path** = <OS/Driver/Library path> New path to be set for the mentioned OS/Driver/Library	In Python CLI, instead of version, path is to be passed.
setosversion	**-ver** <version> OS version.	domain.update_path	In the following arguments, option is supposed to be set to 'OS', for setosversion **option** = <'OS'/'DRIVER'/'LIBRARY'> Valid options are 'OS', 'DRIVER' and 'LIB' **name** = <OS/Driver/Library name> Name of the OS/Driver/Library **new_path** = <OS/Driver/Library path> New path to be set for the mentioned OS/Driver/Library	In Python CLI, instead of version, path is to be passed.

createdts

Creates device tree

Table 16: **createdts- Creates Device Tree**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
createdts	**-platform-name** <software-platform name> Name of the software platform to be generated. **-board** <board name> Board name for device tree to be generated. Board names available at <DTG Repo>/device_tree/data/kernel_dtsi. **-hw** <handoff-file> Hardware description file to be used to create the device tree. **-out** <output-directory> The directory where the software platform needs to be created. Workspace is the default directory, if this option is not specified. **-local-repo** <directory location> Location of the directory were bsp for git repo is available. Device tree repo gets cloned from git, if this option is not specified. **-git-url** <Git URL> Git URL of the dtg repo to be cloned. For default repo, click here.	client.create_platform()	Required Arguments: **name** = <platform_name> Name of the platform. **hw** = <handoff_file> Hardware description file to be used to create the platform. **emulation_xsa_path** = <xsa_path> Path to the Emulation xsa on which the component is created. **platform_xpfm_path** =<xpfm_path> Xpfm path of existing platorm. Optional Arguments: **desc** = <description> Description of the platform. **os** = <os> The OS to be used to create the default domain. **cpu** = <processor> The processor to be used to create the default domain. **domain_name** = <domain_name> Name of the domain to be created in the platform.	Create platform creates a DTS and a platform project.

Table 16: **createdts- Creates Device Tree** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
createdts (cont'd)	**-git-branch** <Git Branch> Git branch to be checked out. 'xlnx_rel_v<Vitis-release>' is selected by default. **-zocl** Set zocl flag to enable zocl driver support, default set to False. zocl should only be used when the designs are PL enabled. Only master and xlnx_rel_v2021.2 branch supports zocl property. **-overlay** Set overlay flag to enable device-tree overlay support, default set to False. **-dtsi** <custom-dtsi-file list> Include custom-dtsi file in the device tree, if specified. The filepaths must be in the list format. **-compile** Specify this option to compile the generated dts to create dtb. If this option is not specified, users can manually use dts to compile dtb. For example, dtc -I dts -O dtb -o <file_name>.dtb <file_name>.dts Compile dts(device tree source) or dtsi(device tree source include) files. dtc -I dts -O dtb -f <file_name>.dts -o <file_name>.dtb Convert dts(device tree source) to dtb(device tree blob). dtc -I dtb -O dts -f <file_name>.dtb -o <file_name>.dts Convert dtb(device tree blob) to dts(device tree source). **-update**Set update flag to enable existing device tree platform to update with new xsa.	client.create_platform() (cont'd)	**template** = <template_name> Template for creating domain in case of Baremetal platform. "Empty" (Default) **no_boot_bsp** = <bool> Mark the platform to build without generating boot components. **fsbl_target** = <fsbl_target> Processor-type for which the existing fsbl has to be generated. This option is valid only for ZU+. "psu_cortexa53_0" (Default) **fsbl_path** = <path> Fsbl path for custom fsbl. This option is used when no_boot_bsp is opted. **pmufw_Elf** = <path> Prebuilt fsbl.elf to be used as boot component. This option is used when no_boot_bsp is opted.	

domain

Create, configure, list and report domains

Table 17: **domain- Create, Configure, List and Report Domains**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
domain create	-**name** <domain-name> Name of the domain. **-display-name** <display_name> The name to be displayed in the report for the domain. **-desc** <description> Brief description of the domain. **-proc** <processor> Processor core to be used for creating the domain. For SMP Linux, this can be a Tcl list of processor cores. **-arch** <processor architecture> 32-bit or 64-bit. This option is valid only for A53 processors. **-os** <os> OS type. Default type is standalone.	platform.add_domain	Required Arguments: **cpu** = <cpu_core> Processor core to be used for creating the domain. For SMP Linux, this can be a list of processor cores	
domain create (cont'd)	**-support-app** <app-name> Create a domain with BSP settings needed for application specified by <app-name>. This option is valid only for standalone domains. The "repo -apps" command can be used to list the available application. **-auto-generate-linux** Generate the Linux artifacts automatically. **-sd-dir** <location> For domain with Linux as OS, use pre-built Linux images from this directory, while creating the PetaLinux project. This option is valid only for Linux domains. **-sysroot** <sysroot-dir> The Linux sysroot directory that should be added to the platform. This sysroot is consumed during application build.		Optional Arguments: **os** = <os> OS type. OS is standalone (default). **name** = <domain_name> Name of the domain to be added. **display_name** = <display_name> Display name for the domain **support_app** = ["app1",..]** Create a domain with BSP settings needed for application specified by <app-name>. This option is valid only for standalone domains. **sd_dir** = <location> For domain with Linux as OS, use pre-built Linux images from this directory, while creating the PetaLinux project. This option is valid only for Linux domains.	
domain active	**domain-name**	platform.get_domain	Required Arguments: **name** = <domain_name> Name of the domain	Get command returns the domain object that can be used to run domain commands.

Table 17: **domain- Create, Configure, List and Report Domains** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
domain config	**-display-name** <display name> Display name of the domain. **-desc** <description> Brief description of the domain. **-sd-dir** <location> For domain with Linux as OS, use pre-built Linux images from this directory, while creating the PetaLinux project. This option is valid only for Linux domains. **-generate-bif** Generate a standard bif for the domain. Domain report shows the location of the generated bif. This option is valid only for Linux domains. **-sw-repo** <repositories-list> List of repositories to be used to pick software components like drivers and libraries while generating this domain. Repository list should be a tcl list of software repository paths. **-mss** <mss-file> Use mss from specified by <mss-file>, instead of generating mss file for the domain. **-readme** <file-name> Add a README file for the domain, with boot instructions and so on.	1. update_name 2. add_custom_dtb 3. set_sd_dir 4. add_boot_di 5. add_bif 6. add_qemu_args 7. add_qemu_data	Required Arguments: 1. **name** = <new_name> New display name for the domain 2. **path** = <dtb_file> Path for the custom DTB file. 3. **path** = <path> Path for the pre-built linux image directory 4. **boot_dir** = <boot_dir> Boot directory to be added 5. **path** = <file_path> Bif file to be added 6. **qemu_option** = <"PS"/"PMC"/"PMU" > Valid qemu option are "PS", "PMC" or "PMU". **file_name** = <file_name> File with all pmu/pmc/ps qemu args listed. 7. **data_dir** = <data_dir> Directory containing all the files provided as a part of qemu-args	

Table 17: **domain- Create, Configure, List and Report Domains** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
domain config (cont'd)	**-inc-path** <include-path> Additional include path which should be added while building the application created for this domain. **-lib-path** <library-path> Additional library search path which should be added to the linker settings of the application created for this domain. **-sysroot** <sysroot-dir> The Linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build. **-boot** <boot-dir> Directory to generate components after Linux image build. **-bif** <file-name> Bif file used to create boot image for Linux boot. **-qemu-args** <file-name> File with all PS QEMU args listed. This is used to start PS QEMU. **-pmuqemu-args** <file-name> File with all PMC QEMU args listed. This is used to start PMU QEMU. **-pmcqemu-args** <file-name> File with all pmcqemu args listed. This is used to start pmcqemu. **-qemu-data** <data-dir> Directory which has all the files listed in file-name provided as part of qemu-args and pmuqemu-args options.			
list		platform.list_domain		
report	**domain-name**	domain.report	Required Arguments: **name** = <domain_name> Name of the domain to be deleted	
remove	**domain-name**			

getaddrmap

Get the address ranges of IP connected to processor

getperipherals

Get a list of all peripherals in the HW design

getprocessors

Get a list of all processors in the hardware design.

Table 18: **getprocessors- Get a List of All Processors in the Hardware Design**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
getprocessors	xsa	get_processor_os_list	**xsa** = <xsa_path> or **platform** = <platform_name> XSA or platform from which processor os list is to be extracted.	

getws

Get Vitis workspace

Table 19: **getws- Get Vitis Workspace**

XSCT Methods	Arguments	Corresponding Python API	Arguments
setws	-switch <path>	set_workspace	**path** = <workspace_location>

lscript

Create linker script

Table 20: **lscript- Create Linker Script**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
lscript memory	**-app** <application-name> Name of application from workspace. **-supported-mem** Returns supported memory regions for each section.	lscript.get_memory_regions		***Note*:** In Python CLI, lscript object is to be created using host component. On the other hand, in XSCT you pass application_name to get the memory regions. ***Note*:** In XSCT, there was no provision to create/ update memory regions. In Python CLI, it can be done using the following commands:**lscript.add_memory_regionupdate_memory_region** Required Arguments: **name** = <name> Name for the new memory region. **base_address** = <base_address> Base address for the memory region. **size** = <size> Size of the memory region.
lscript section	**-app** <application-name> Name of application from workspace. **-name** <section-name> Name of the section to be edited. **-mem** <memory-region> Name of the memory region to be used for the section. **-size** <section-size> Size of the section. **-add**Add a new section. **-type** Type of new section to be added. Supported types are CODE, DATA, STACK, HEAP.	1. lscript.get_ld_sections (To list the sections) 2. lscript.update_ld_section (To update the ld sections)	1. None 2. Required Arguments: **section** = <section> An existing memory code section identifier. **region** = <memory_region> The updated memory region for the code section.	

Table 20: **lscript- Create Linker Script** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
lscript def-mem	**-app** <application-name> Name of application from workspace. **-code** Return default code memory. **-data** Return default data memory. **-stack** Return default stack & heap memory.	lscript.get_stack_size (To get stack size) lscript.get_heap_size (To get heap size)		
lscript generate		lscript.regenerate		

platform

Create, configure, list, and report platforms.

Table 21: **platform- Create, Configure, List, and Report Platforms**

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
active	platform-name	client.get_platform_component	Required Arguments: **name** = <platform_name> Name of the platform.	Get command returns the platform object that can be used to run platform commands.
clean		platform.clean		

Table 21: **platform- Create, Configure, List, and Report Platforms** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
config	**-desc** <description> Add a brief description about the platform. **-updatehw** <hw-spec> Update the platform to use a new hardware specification file specified by <hw-spec>. **-samples** <samples-dir> Make the application template specified in <samples-dir> part of the platform. This option can only be used for acceleratable applications. "repo -apps <platform-name>" can be used to list the application templates available for the given platform-name. **-prebuilt-data** <directory-name> For expandable platforms, pre-generated hardware data specified in directory-name is used for building user applications that do not contain accelerators. This reduces the build time. **-make-local** Make the referenced SW components local to the platform. **-fsbl-target** <processor-type> Processor-type for which the existing fsbl has to be re-generated. This option is valid only for ZU+. **-create-boot-bsp** Generate boot components for the platform.	1. update_desc 2. update_hw 3. Not supported 4. Not supported 5. Not supported 6. retarget_fsbl 7. generate_boot_bsp 8. remove_boot_bsp (For 9 and 10, remove_boot_bsp uses fsbl_elf and pmufw_elf options)	Required Arguments: **desc** = <description> Description of the platform Required Arguments **hw** = <hw_spec> or emulation_xsa_path = <xsa_path> Hardware specification file or emulation xsa path. Optional Arguments **target_processor** = <target> Processor for which the existing fsbl has to be re-generated.	

Table 21: **platform- Create, Configure, List, and Report Platforms** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments	Comments
config (cont'd)	**-remove-boot-bsp** Remove all the boot components generated during platform creation. **-fsbl-elf** <fsbl.elf> Prebuilt fsbl.elf to be used as boot component when "remove-boot-bsp" option is specified. **-pmufw-elf** <pmufw.elf> Prebuilt pmufw.elf to be used as boot component when "remove-boot-bsp" option is specified. **-extra-compiler-flags** <param> <value> Set extra compiler flag for the parameter with a provided value. Only FSBL and PMUFW are the supported parameters. If the value is not passed, the existing value returns. **-extra-linker-flags** <param> <value> Set extra linker flag for the parameter with a provided value. Only FSBL and PMUFW are the supported parameters. If the value is not passed, the existing value returns. **-reset-user-defined-flags** <param> Resets the extra compiler and linker flags. Only FSBL and PMUFW are the supported parameters. **-report** <param> Return the list of extra compiler and linker flags set to the given parameter. Only FSBL and PMUFW are the supported parameters.		Optional Arguments **target_processor** = <processor>Target procesor for generating boot bsp. zynqmp_fsbl (default) Optional Arguments **fsbl_path** = <path> Prebuilt fsbl.elf to be used as boot component when boot components are removed. **pmufw_elf** = <pmufw.elf> Prebuilt pmufw.elf to be used as boot component when boot components are removed.	
list		client.list_platforms		
report		platform.report		
remove		platform-name		
read		Not supported in Python CLI		
write		Not supported in Python CLI		

repo

Get, set, or modify software repositories

Table 22: **repo- Get, Set, or Modify Software Repositories**

XSCT Methods	Arguments	Corresponding Python API	Arguments
-set		For example repos add_local_example_repo For sw repos set_sw_repo	Required Arguments: **name** = <repo_name> The local repository name to be added. local_directory = <local_repo_directory_path> The local repository directory path. Optional Arguments: **type** = <repo_type>** The repository type. Valid types are "SYS_PROJ" (system projects), "HLS", and "AIE". "SYS_PROJ" is the default type. **display_name** = <repo_display_name>** The repository display name. **description** = <"Description of the repo">** The description of the local repository. Required Arguments: **level** = <'LOCAL'/'GLOBAL'> Level at which the software repo is to be set. LOCAL - 'available to the current workspace'. GLOBAL - 'available across workspaces'. **path** = <repo_path> Software repo path to be set. Path can be a single path or a list of paths.
-get		1. For example repos list_example_repos 2. For sw repos get_sw_repo	Optional Arguments: **type** = <repo_type>** Valid types are "SYS_PROJ" (system projects) and "HLS", "AIE" and "EMBD_APP" (Embedded applications). "SYS_PROJ" is the default type. Required Arguments: **level** = <'LOCAL'/'GLOBAL'> Level of software repo.
-scan		1. For platform repos rescan_platform_repos 2. For sw repo rescan_sw_repo	Required Arguments: **platform** = <platform_path> User can give single platform path as string or multiple platform paths as strings list.

Table 22: **repo- Get, Set, or Modify Software Repositories** *(cont'd)*

XSCT Methods	Arguments	Corresponding Python API	Arguments
-os		For domain os use get_os()	
-libs		For domain libs use get_libs()	
-drivers		For domain drivers use get_drivers	
-app		For -app option dom.get_applicable_libs	
-add-platforms	<platforms directory>	add_platform_repos	Required Arguments: **platform** = <platform_path> String/list of platform path(s).
-remove-platforms-dir	<platforms directory>	delete_platform_repos	Required Arguments: **platform** = <platform_path> String/list of platform path(s).

setws

Set Vitis workspace

Table 23: **setws- Set Vitis Workspace**

XSCT Methods	Arguments	Corresponding Python API	Arguments
setws	workspace	1. client.set_workspace 2. client.get_workspace	Required Arguments: **path** = <workspace_location>

sysproj

System project management

Table 24: **setws- Set Vitis Workspace**

XSDB Methods	Corresponding Python API	Arguments
build	sys_proj.build	Optional Arguments: **target** = <target name> One of the supported targets (sw_emu/ hw_emu/hw). sw_emu (Default) **comp_name** = <comp_name>** Component to be built. Complete system project (default) **build_comps** = <bool> Build the dependent components if they're not built already. True (Default).
clean	sys_proj.clean	Optional Arguments: **target** = <target name> - One of the supported targets (sw_emu/hw_emu/hw). **comp_name**=<comp_name>** Component to be built. Complete system project (default)
list	client.list_sys_projects	
remove	client.remove_sys_proj	Required Arguments: **name** = <proj_name> The system project to be deleted. (string)
report	sys_proj.report	

importprojects

```
import_projects:
Import components and system_projects from the mentioned zip directory
    to the current workspace.
    (Get the list of existing components and projects from the zip using
    get_project_info(src) API.)

Prototype:
    sys_proj = client.import_projects(src = <source_zip_dir>,
                              components = <component_names>,
                              system_projects = <proj_names>,
                              dest = <destination_dir>)

Required Arguments:
    src = <source_zip_dir>
        Full path of the zipped project.

Optional Arguments:
    components = <component_names>
        Names of components to be imported.
```

```
    system_projects = <proj_names>
        Names of system projects to be imported.
    dest = <destination_dir>
        Path at which the project is to be imported.
        Default location is current workspace.
```

importsources

```
import_files:
    Import files to the platform.

Prototype:
    status = platform.import_files(from_loc = <Location of src or
                                    config file(s) path>,
                                    files = ["file1",..]**,
                                    dest_dir_in_cmp = <Dest location or
                                    config file(s) path>)

Required Arguments:

    from_loc = <Location of src or config file(s) path>
        From The location can be a directory/file, the path can be
        absolute/relative.

Optional Arguments:

    files = ["file1",..]**
        List of files to be imported from the given from_loc path.
        Whole folder is imported if files are not mentioned.

    dest_dir_in_cmp = <Dest location or config file(s) path>
        Destination folder name, created if directory doesn't exist.
        Files are imported directly to the component folder if
        destination folder is not given.
```

library

```
Vitis [12]: cl.create_library_component?
Signature: cl.create_library_component(name, platform, domain=None,
template=None)
Docstring:
create_library_component:
    Create a static library component.
Prototype:
    lib_comp = client.create_library_component(name = <comp_name>,
                                               platform = <platform>,
                                               domain = <domain>)

Required Arguments:
    name = <comp_name>
        Library component name.
```

```
    platform = <platform>
        Platform for which component is to be created.
        In case of baremetal platforms, user can specify the domain
        along with platform.

Optional Argument:
    domain = <domain>
        Specify the domain when there is more than one domain
        on the platform. (Only supported for baremetal domains)

Returns:
    Library component object.

Examples:
    lib_comp = client.create_library_component(name = "lib_component1",
                                               platform = "/tmp/
vck190.xpfm",
                                               domain = "psv_cortexa72_o")
File:      /proj/xbuilds/SWIP/2025.1_1106_0237/installs/lin64/Vitis/
2025.1/cli/vitis/cli_client.py
Type:      method
```

Section VI

GNU Compiler Tools

This section contains the following chapters:

- Overview
- Compiler Framework
- Common Compiler Usage and Options
- MicroBlaze Compiler Usage and Options
- Arm Compiler Usage and Options
- Other Notes

***Note*:** From 2024.1 release, this user guide supports MicroBlaze™ and MicroBlaze™-V devices.

Overview

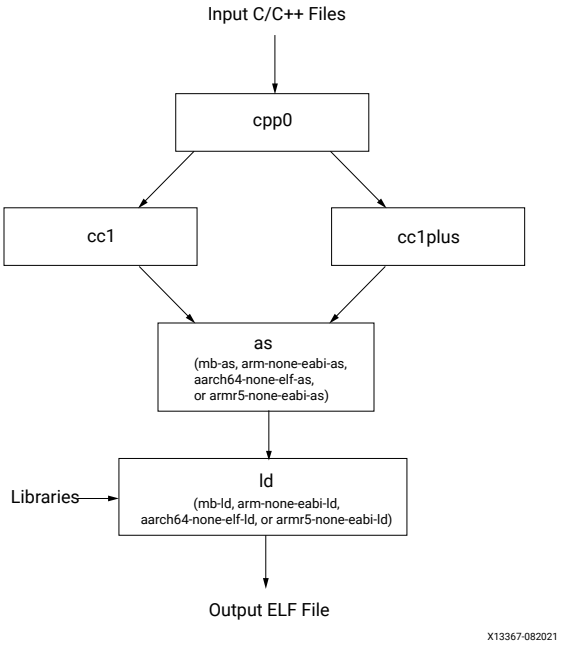
The AMD Vivado™ Design Suite includes the GNU compiler collection (GCC) for the MicroBlaze™ and MicroBlaze™-V processor and the Arm® Cortex® A9, A53, A72, and R5 processors.

- The Vivado GNU tools support both the C and C++ languages.
- The MicroBlaze and MicroBlaze™-V GNU tools include the `mb-gcc` and `mb-g++` compilers, the `mb-as` assembler, and the `mb-ld` linker.
- The Arm processor tools include:
  - The `arm-none-eabi-gcc` and `arm-none-eabi-g++` compilers, `arm-none-eabi-as` assembler, and `arm-none-eabi-ld` linker for Cortex A9 processors
  - `aarch64-none-elf-*` for Cortex-A53 and Cortex-A72 processors
  - `armr5-none-eabi-*` for Cortex-R5F processors
- The toolchains also include the C, math, GCC, and C++ standard libraries.
- The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The MicroBlaze and Arm compiler tools use the GNU binutils based on GNU version 2.31 in 2019.x and 2.32 in 2020.x of the sources.

Compiler Framework

This section discusses the common features of the MicroBlaze™ and MicroBlaze™-V, and Cortex® A9, A53, A72, and R5 processor compilers. The following figure displays the GNU tool flow.

Figure 54: **GNU Tool Flow**



The GNU compilers are named as follows:

- `mb-gcc` for MicroBlaze and MicroBlaze™-V
- `arm-none-eabi-gcc` for Cortex A9 cores
- `aarch64-none-elf-gcc` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-gcc` for Cortex-R5F

The GNU compiler is a wrapper that calls the following executables:

- **Pre-processor (`cpp0`):** This is the first pass invoked by the compiler. The pre-processor replaces all macros with definitions as defined in the source and header files.

- **Machine and language specific compiler:** This compiler works on the pre-processed code, which is the output of the first stage. The language-specific compiler is one of the following:
  - **C Compiler (`cc1`):** The compiler responsible for most of the optimizations done on the input C code and for generating assembly code.
  - **C++ Compiler (`cc1plus`):** The compiler responsible for most of the optimizations done on the input C++ code and for generating assembly code.
- **Assembler:** The assembly code has mnemonics in assembly language. The assembler converts these to machine language. The assembler also resolves some of the labels generated by the compiler. It creates an object file which is passed on to the linker. The assembler executables are named as follows:
  - `mb-as` for MicroBlaze and MicroBlaze™-V
  - `arm-none-eabi-as` for Cortex A9 cores
  - `aarch64-none-elf-as` for Cortex-A53 and Cortex-A72
  - `armr5-none-eabi-as` for Cortex-R5F
- **Linker:** Links all the object files generated by the assembler. If libraries are provided on the command line, the linker resolves some of the undefined references in the code by linking in some of the functions from the assembler. The linker executables named as follows:
  - `mb-ld` for MicroBlaze
  - `arm-none-eabi-ld` for Cortex A9 cores
  - `aarch64-none-elf-ld` for Cortex-A53and Cortex-A72
  - `armr5-none-eabi-ld` for Cortex-R5F

Executable options are described in the following sections:

- Commonly Used Compiler Options: Quick Reference
- Linker Options
- MicroBlaze Compiler Usage and Options
- MicroBlaze Linker Options
- Arm Compiler Usage and Options

***Note*:** From this point forward, the references to GCC in this chapter refer to the MicroBlaze and MicroBlaze™-V compiler, `mb-gcc`, and references to G++ refer to the MicroBlaze C++ compiler, `mb-g++`.

Common Compiler Usage and Options

Usage

To use the GNU compiler, type:

```
<Compiler_Name> options files...
```

Where `<Compiler_Name>` refers to one of the following compilers:

- `mb-gcc` for MicroBlaze and MicroBlaze™-V
- `arm-none-eabi-gcc` for Cortex A9
- `aarch64-none-elf-gcc` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-gcc` for Cortex-R5F

To compile C++ programs, you can use the `mb-g++` or `arm-none-eabi-g++` command.

Input Files

The compilers take one or more of the following files as input:

- C source files
- C++ source files
- Assembly files
- Object files
- Linker scripts

***Note*:** These files are optional. If they are not specified, the default linker script embedded in the linker is used. The default scripts are as follows:

- `mb-ld` for MicroBlaze and MicroBlaze™-V

- `arm-none-eabi-ld` for Cortex A9
- `aarch64-none-elf-ld` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-ld` for Cortex-R5F

The default extensions for each of these types are listed in File Types and Extensions. In addition to the files mentioned above, the compiler implicitly refers to the libraries files `libc.a`, `libgcc.a`, `libm.a`, and `libxil.a`. The default location for these files is the Vivado installation directory. When using the G++ Compiler, the `libsupc++.a` and `libstdc++.a` files are also referenced. These are the C++ language support and C++ platform libraries respectively.

Output Files

The compiler generates the following files as output:

- An ELF file.
- An assembly file, if the `-save-temps` or `-S` option is used.
- An object file, if the `-save-temps` or `-c` option is used.
- Preprocessor output, an `.i` or `.ii` file, if the `-save-temps` option is used.

File Types and Extensions

The GNU compiler determines the type of your file from the file extension. The following table lists the valid extensions and the corresponding file types. The GCC wrapper calls the appropriate lower level tools by recognizing these file types.

Table 25: **File Extensions**

Extension	File Type (Dialect)
.c	C file
.C	C++ file
.cxx	C++ file
.cpp	C++ file
.c++	C++ file
.cc	C++ file
.S	Assembly file, but might have preprocessor directives
.s	Assembly file with no preprocessor directives

Libraries

The following table lists the libraries necessary for the `mb_gcc` and `arm-none-eabi-gcc` compilers.

Table 26: **Libraries Used by the Compilers**

Library	Particular
libxil.a	Contains drivers, software services (such as XilMFS), and initialization files developed for the Vivado tools.
libc.a	Standard C libraries, including functions such as strcmp and strlen.
libgcc.a	GCC low-level library containing emulation routines for floating point and 64-bit arithmetic.
libm.a	Math library containing functions such as cos and sine.
libsupc++.a	C++ support library with routines for exception handling, RTTI, and others.
libstdc++.a	C++ standard platform library. Contains standard language classes, such as those for stream I/O, file I/O, string manipulation, and others.

Libraries are linked in automatically by both compilers. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. The `libxil.a` is modified to add driver and library routines.

Language Dialect

The GCC compiler recognizes both C and C++ dialects and generates code accordingly. By GCC convention, it is possible to use either the GCC or the G++ compilers equivalently on a source file. The compiler that you use and the extension of your source file determines the dialect used on the input and output files.

When using the GCC compiler, the dialect of a program is always determined by the file extension, as listed in File Types and Extensions. If a file extension shows that it is a C++ source file, the language is set to C++. This means that if you have compile C code contained in a CC file, even if you use the GCC compiler, it automatically mangles function names.

The primary difference between GCC and G++ is that G++ automatically sets the default language dialect to C++ (irrespective of the file extension), and if linking, automatically pulls in the C++ support libraries. This means that even if you compile C code in a `.c` file with the G++ compiler, it will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is said to be overloaded if the same function can perform different actions based on the arguments passed in, and can return different return values. To support this, C++ compilers encode the type of the function to be invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful about name mangling if you decide to follow a mixed compilation mode, with some source files containing C code and some others containing C++ code (or using GCC for compiling certain files and G++ for compiling others). To prevent name mangling of a C symbol, you can use the following construct in the symbol declaration.

```
#ifdef __cplusplus
extern “C” {
#endif
int foo();
int morefoo();
#ifdef __cplusplus
}
#endif
```

Make these declarations available in a header file and use them in all source files. This causes the compiler to use the C dialect when compiling definitions or references to these symbols.

***Note*:** All Vivado drivers and libraries follow these conventions in all the header files they provide. You must include the necessary headers, as documented in each driver and library, when you compile with G++. This ensures that the compiler recognizes library symbols as belonging to “C” type.

When compiling with either variant of the compiler, to force a file to a particular dialect, use the -x lang switch. Refer to the GCC documentation for more information on this switch.

- When using the GCC compiler, `libstdc++.a` and `libsupc++.a` are not automatically linked in.
- When compiling C++ programs, use the G++ variant of the compiler to make sure all the required support libraries are linked in automatically.
- Adding `-lstdc++` and `-lsupc++` to the GCC command are also possible options.

For more information about how to invoke the compiler for different languages, refer to the GNU online documentation.

Commonly Used Compiler Options: Quick Reference

The summary below lists compiler options that are common to the compilers for MicroBlaze and MicroBlaze™-V, and Arm processors. The compiler options are case sensitive.

General Options

- `-E`: Preprocess only; do not compile, assemble, and link. The preprocessed output displays on the standard out device.
- `-S`: Compile only; do not assemble, and link. Generates a `.s` file.
- `-c`: Compile and assemble only; do not link. Generates a `.o` file.
- `-g`: This option adds DWARF2-based debugging information to the output file. The debugging information is required by the GNU debugger, mb-gdb or arm-none-eabi-gdb. The debugger provides debugging at the source and the assembly level. This option adds debugging information only when the input is a C/C++ source file.
- `-gstabs`: Use this option for adding STABS-based debugging information on assembly (.S) files and assembly file symbols at the source level. This is an assembler option that is provided directly to the GNU assembler, mb-as or arm-none-eabi-as. If an assembly file is compiled using the compiler mb-gcc or arm-none-eabi-gcc, prefix the option with -Wa.
- `-On`: The GNU compiler provides optimizations at different levels. The optimization levels in the following table apply only to the C and C++ source files.

  Table 27: **Optimizations for Values of n**

n	Optimization
0	No optimization.
1	Medium optimization.
2	Full optimization
3	Full optimization. Attempt automatic inlining of small subprograms.
S	Optimize for size.

  Note: Optimization levels 1 and above cause code re-arrangement. While debugging your code, use of no optimization level is recommended. When an optimized program is debugged through GDB, the displayed results might seem inconsistent.

- `-v`: This option executes the compiler and all the tools underneath the compiler in verbose mode. This option gives complete description of the options passed to all the tools. This description is helpful in discovering the default options for each tool.
- `-save-temps`: The GNU compiler provides a mechanism to save the intermediate files generated during the compilation process. The compiler stores the following files:
  - Preprocessor output -input_file_name.i for C code and input_file_name.ii for C++ code
  - Compiler (cc1) output in assembly format - input_file_name.s
  - Assembler output in ELF format - input_file_name.s

The compiler saves the default output of the entire compilation as a.out.

- **`-o filename`:** The compiler stores the default output of the compilation process in an ELF file named a.out. You can change the default name using -o output_file_name. The output file is created in ELF format.
- **`-Wp,<option>`, `-Wa,<option>`, and `-Wl,<option>`:** The compiler, `mb-gcc` or `arm-none-eabi-gcc`, is a wrapper around other executables such as the preprocessor, compiler (`cc1`), assembler, and the linker. You can run these components of the compiler individually or through the top level compiler.

  There are certain options that are required by tools, but might not be necessary for the top-level compiler. To run these commands, use the options listed in the following table.

  Table 28: **Tool-Specific Options Passed to the Top-Level GCC Compiler**

Option	Tool	Example
-Wp,<option>	Preprocessor	`mb-gcc -Wp,-D -Wp, MYDEFINE ...` Signal the pre-processor to define the symbol MYDEFINE with the -D MYDEFINE option.
-Wa,<option>	Assembler	`mb-as -Wa, ...` Signal the assembler to target the MicroBlaze and MicroBlaze™-V processor.
-Wl,<option>	Linker	`mb-gcc -Wl,-M ...` Signal the linker to produce a map file with the -M option.

- **`-help`:** Use this option with any GNU compiler to get more information about the available options. You can also consult the GCC manual.
- **`-B directory`:** Add directory to the C runtime library search paths.
- **`-L directory`:** Add directory to the library search path.
- **`-I directory`:** Add directory to header search path.
- **`-l library`:** Search library for undefined symbols.

  ***Note*:** The compiler prefixes “lib” to the library name indicated in this command line switch.

Library Search Options

- **`-l <library name>`:** By default, the compiler searches only the standard libraries, such as `libc`, `libm`, and `libxil`. You can also create your own libraries. You can specify the name of the library and where the compiler can find the definition of these functions. The compiler prefixes `lib` to the library name that you provide.

  The compiler is sensitive to the order in which you provide options, particularly the `-l` command line switch. Provide this switch only after all of the sources in the command line.

  For example, if you create your own library called `libproject.a`. you can include functions from this library using the following command:

  ```
  Compiler Source_Files -L${LIBDIR} -l project
  ```

  IMPORTANT! *If you supply the library flag `-l library_name` before the source files, the compiler does not find the functions called from any of the sources. This is because the compiler search is only done in one direction and it does not keep a list of available libraries.*

- **`-L <lib directory>`:** This option indicates the directories in which to search for the libraries. The compiler has a default library search path, where it looks for the standard library. Using the `-L` option, you can include some additional directories in the compiler search path.

Header File Search Option

- **`-I <directory name>`:** This option searches for header files in the `/<dir_name>` directory before searching the header files in the standard path.

Default Search Paths

The compilers (mb-gcc for MicroBlaze and MicroBlaze™-V, `arm-none-eabi-gcc` for Cortex A9, `armr5-none-eabi-gcc` for Cortex-R5F, and `aarch64-none-elf-gcc` for Cortex-A53 Cortex®-A72) search certain paths for libraries and header files. The search paths on the various platforms are described below.

Library Search Procedures

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the `-L <dir_name>` option.

2. The compilers search the following libraries:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/lib
```

Header File Search Procedures

The compilers search header files in the following order:

1. Directories are passed to the compiler with the `-I <dir_name>` option.
2. The compilers search the following header files:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/include
```

Initialization File Search Procedures

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the `-B <dir_name>` option.
2. The compilers search the following files:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/lib
```

Where:

- `<processor>` is `microblaze` for MicroBlaze and MicroBlaze™-V processors, `aarch32` for A9, `aarch64` for A53, and `armr5` for R5 processors.
- `<processor-lib>` is `microblazeeb-xilinx-elf` for MicroBlaze and MicroBlaze™-V, `aarch32-xilinx-eabi` for A9, `aarch64-none/aarch64-xilinx-elf` for A53, and `gcc-arm-none-eabi/armrm-xilinx-eabi` for R5 processors.

Note: `platform` indicates `lin64` for Linux 64-bit and `nt` for Windows Cygwin.

Linker Options

- **`-defsym _STACK_SIZE=value`:** The total memory allocated for the stack can be modified using this linker option. The variable `_STACK_SIZE` is the total space allocated for the stack. The `_STACK_SIZE` variable is given the default value of 100 words, or 400 bytes. If your program is expected to need more than 400 bytes for stack and heap combined, it is recommended that you increase the value of `_STACK_SIZE` using this option. The value is in bytes.

  In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program tries to write in other, incorrect, sections of the program, leading to incorrect execution of the code.

Note: A minimum stack size of 16 bytes (0x0010) is required for programs linked with the AMD-provided C runtime (CRT) files.

- `-defsym _HEAP_SIZE=value`: The total memory allocated for the heap can be controlled by the value given to the variable `_HEAP_SIZE`. The default value of `_HEAP_SIZE` is zero.

  Dynamic memory allocation routines use the heap. If your program uses the heap in this fashion, then you must provide a reasonable value for `_HEAP_SIZE`.

  For advanced users: you can generate linker scripts directly from IP integrator.

Memory Layout

The MicroBlaze and MicroBlaze™-V and Arm Cortex A9 and R5 processors use 32-bit logical addresses and can address any memory in the system in the range 0x0 to 0xFFFFFFFF. This address range can be categorized into reserved memory and I/O memory. The Arm Cortex-A53 and Cortex-A72 processor uses 64-bit logical addresses.

Reserved Memory

Reserved memory has been defined by the hardware and software programming environment for privileged use. This is typically true for memory containing interrupt vector locations and operating system level routines. The following table lists the reserved memory locations for MicroBlaze, MicroBlaze™-V, and Arm processors as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

For information about the Arm Cortex A9 memory map, refer to the *Zynq 7000 SoC Technical Reference Manual* (UG585). For the Cortex-R5F, Cortex-A53, and Cortex-A72 memory map, refer to the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085).

Note: In addition to these memories that are reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform that you are using to find out if any memory locations are deemed reserved.

Table 29: **Hardware Reserved Memory Locations**

Processor Family	Reserved Memories	Reserved Purpose	Default Text Start Address
MicroBlaze and MicroBlaze™-V	0x0 - 0x4F	Reset, Interrupt, Exception, and other reserved vector locations.	0x50

I/O Memory

I/O memory refers to addresses used by your program to communicate with memory-mapped peripherals on the processor buses. These addresses are defined as a part of your hardware platform specification.

User and Program Memory

User and program memory refers to all the memory that is required for your compiled executable to run. By convention, this includes memories for storing instructions, read-only data, read-write data, program stack, and program heap. These sections can be stored in any addressable memory in your system. By default the compiler generates code and data starting from the address listed in the table in Reserved Memory and occupying contiguous memory locations. This is the most common memory layout for programs. You can modify the starting location of your program by defining (in the linker) the symbol `_TEXT_START_ADDR` for MicroBlaze and MicroBlaze™-V, `START_ADDR` for Arm.

In special cases, you might want to partition the various sections of your ELF file across different memories. This is done using the linker command language (refer to Linker Scripts for details). The following are some situations in which you might want to change the memory map of your executable:

- When partitioning large code segments across multiple smaller memories
- Remapping frequently executed sections to fast memories
- Mapping read-only segments to non-volatile flash memories

No restrictions apply to how you can partition your executable. The partitioning can be done at the output section level, or even at the individual function and data level. The resulting ELF can be non-contiguous, that is, there can be “holes” in the memory map. Ensure that you do not use documented reserved locations.

Alternatively, if you are an advanced user and want to modify the default binary data provided by the tools for the reserved memory locations, you can do so. In this case, you must replace the default startup files and the memory mappings provided by the linker.

Object-File Sections

An executable file is created by concatenating input sections from the object files (`.o` files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following table.

In addition to these sections, you can also create your own custom sections and assign them to memories of your choice.

Table 30: **Sectional Layout of an Object or Executable File**

Section	Description
.text	Text section
.rodata	Read only data section
.sdata2	Small read only data section
.sbss2	Small read only uninitialized data section
.data	Read/write data section
.sdata	Small read/write data section
.sbss	Small uninitialized data section
.bss	Uninitialized data section
.heap	Program heap memory section
.stack	Program stack memory section

The reserved sections that you would not typically modify include: `.init`, `.fini`, `.ctors`, `.dtors`, `.got`, `.got2`, and `.eh_frame`.

- **`.text`:** This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.
- **`.rodata`:** This section contains read-only data. This section has the `r` (read-only) and the `i` (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.
- **`.sdata2`:** This section is similar to the `.rodata` section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. You can change the size of the data going into this section with the `-G` option to the compiler. This section has the `r` (read-only) and the `i` (initialized) flags.
- **`.data`:** This section contains read-write data and has the `w` (read-write) and the `i` (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.
- **`.sdata`:** This section contains small read-write data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the `w` (read-write) and the `i` (initialized) flags and must be mapped to initialized RAM.

- **`.sbss2`:** This section contains small, read-only uninitialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the `r` (read) flag and can be mapped to ROM.
- **`.sbss`:** This section contains small uninitialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the `w` (read-write) flag and must be mapped to RAM.
- **`.bss`:** This section contains uninitialized data. This section has the `w` (read-write) flag and must be mapped to RAM.
- **`.heap`:** This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.
- **`.stack`:** This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections might appear merged together into a section named `.bss_stack`.
- **`.init`:** This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.
- **`.fini`:** This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.
- **`.ctors`:** This section contains a list of functions that must be invoked at program startup and the same flags as `.data` and must be mapped to initialized RAM.
- **`.dtors`:** This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.
- **`.got2/.got`:** This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.
- **`.eh_frame`:** This section contains frame unwind information for exception handling. It contains the same flags as `.rodata`, and can be mapped to initialized ROM.
- **`.tbss`:** This section holds uninitialized thread-local data that contribute to the program memory image. This section has the same flags as `.bss`, and it must be mapped to RAM.
- **`.tdata`:** This section holds initialized thread-local data that contribute to the program memory image. This section must be mapped to initialized RAM.
- **`.gcc_except_table`:** This section holds language specific data. This section must be mapped to initialized RAM.
- **`.jcr`:** This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions. This section must be mapped to initialized RAM.

- `.fixup`: This section contains information necessary for doing fixup, such as the fixup page table and the fixup record table. This section must be mapped to initialized RAM.

Linker Scripts

The linker utility uses commands specified in linker scripts to divide your program on different blocks of memories. It describes the mapping between all of the sections in all of the input object files to output sections in the executable file. The output sections are mapped to memories in the system. You do not need a linker script if you do not want to change the default contiguous assignment of program contents to memory. There is a default linker script provided with the linker that places section contents contiguously.

You can selectively modify only the starting address of your program by defining the linker symbol `_TEXT_START_ADDR` on MicroBlaze and MicroBlaze™-V processors, or START_ADDR on Arm processors, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The choices of the default script that are used by the linker from the `$XILINX_/gnu/<procname>/<platform>/<processor_name>/lib/ldscripts` area are described as follows (where `<procname> =microblaze`, `<processor_name> = microblaze`, and `<platform> = lin` or `nt`):

- `elf32<procname>.x` is used by default when none of the following cases apply.
- `elf32<procname>.xn` is used when the linker is invoked with the `-n` option.
- `elf32<procname>.xbn` is used when the linker is invoked with the `-N` option.
- `elf32<procname>.xr` is used when the linker is invoked with the `-r` option.
- `elf32<procname>.xu` is used when the linker is invoked with the `-Ur` option.

To use a linker script, provide it on the GCC command line. Use the command line option `-T <script>` for the compiler, as described below:

```
compiler -T <linker_script> <Other Options and Input Files>
```

If the linker is executed on its own, include the linker script as follows:

```
linker -T <linker_script> <Other Options and Input Files>
```

This tells GCC to use your linker script in the place of the default built-in linker script. Linker scripts can be generated for your program by right clicking the application component and selecting **Reset Link Script** to generate or reset the linker script..

Linker scripts can be used to assign specific variables or functions to specific memories. This is done through section attributes in the C code. Linker scripts can also be used to assign specific object files to sections in memory. These and other features of GNU linker scripts are explained in the GNU linker documentation, which is a part of the binutils manual. For a specific list of input sections that are assigned by MicroBlaze and MicroBlaze™-V processor linker scripts, see MicroBlaze Linker Script Sections.

MicroBlaze Compiler Usage and Options

The MicroBlaze™ GNU compiler is derived from the standard GNU sources as the AMD port of the compiler. The features and options that are unique to the MicroBlaze compiler are described in the sections that follow. When compiling with the MicroBlaze compiler, the pre-processor provides the definition `__MICROBLAZE__` automatically. You can use this definition in any conditional code.

MicroBlaze Compiler

The `mb-gcc` compiler for the MicroBlaze soft processor introduces new options as well as modifications to certain options supported by the GNU compiler tools. The new and modified options are summarized in this chapter.

Processor Feature Selection Options

- **`-mcpu=vX.YY.Z`:** This option directs the compiler to generate code suited to MicroBlaze hardware version v.X.YY.Z. To get the most optimized and correct code for a given processor, use this switch with the hardware version of the processor.

  The `-mcpu` switch behaves differently for different versions, as described below:

  - **Pr-v3.00.a:** Uses 3-stage processor pipeline mode. Does not inhibit exception causing instructions being moved into delay slots.
  - **v3.00.a and v4.00.a:** Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
  - **v5.00.a and later:** Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots.

- **`-mlittle-endian/-mbig-endian`:** Use these options to select the endianness of the target machine for which code is being compiled. The endianness of the binary object file produced is also set appropriately based on this switch. The GCC driver passes switches to the sub tools (as, cc1, cc1plus, ld) to set the corresponding endianness in the sub tool.

  The default is `-mbig-endian`.

  You cannot link together object files of mixed endianness.

- **`-mno-xl-soft-mul`:** This option permits use of hardware multiply instructions for 32-bit multiplications. The MicroBlaze processor has an option to turn the use of hardware multiplier resources on or off. This option should be used when the hardware multiplier option is enabled on the MicroBlaze processor. Using the hardware multiplier can improve the performance of your application. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.

- **`-mxl-multiply-high`:** The MicroBlaze processor has an option to enable instructions that can compute the higher 32bits of a 32x32-bit multiplication. This option tells the compiler to use these multiply high instructions. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is available or not.

- **`-mno-xl-multiply-high`:** Do not use multiply high instructions. This option is the default.

- **`-mxl-soft-mul`:** This option tells the compiler that there is no hardware multiplier unit on the MicroBlaze processor, so every 32-bit multiply operation is replaced by a call to the software emulation `routine__mulsi3`. This option is the default.

- **`-mno-xl-soft-div`:** You can instantiate a hardware divide unit in MicroBlaze. When the divide unit is present, this option tells the compiler that hardware divide instructions can be used in the program being compiled.

  This option can improve the performance of your program if it has a significant amount of division operations. The compiler automatically defines the C pre-processor definition `HAVE_HW_DIV` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.

- **`-mxl-soft-div`:** This option tells the compiler that there is no hardware divide unit on the target MicroBlaze hardware.

  This option is the default. The compiler replaces all 32-bit divisions with a call to the corresponding software emulation routines (`__divsi3`, `__udivsi3`).

- **`-mxl-barrel-shift`:** The MicroBlaze processor can be configured to be built with a barrel shifter. In order to use the barrel shift feature of the processor, use the option `-mxl-barrel-shift`.

The default option assumes that no barrel shifter is present, and the compiler uses add and multiply operations to shift the operands. Enabling barrel shifts can speed up your application significantly, especially while using a floating point library. The compiler automatically defines the C pre-processor definition `HAVE_HW_BSHIFT` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether or not this feature is specified as available.

- **`-mno-xl-barrel-shift`:** This option tells the compiler not to use hardware barrel shift instructions. This option is the default.
- **`-mxl-pattern-compare`:** This option activates the use of pattern compare instructions in the compiler.

  Using pattern compare instructions can speed up boolean operations in your program. Pattern compare operations also permit operating on word-length data as opposed to byte-length data on string manipulation routines such as `strcpy`, `strlen`, and `strcmp`. On a program heavily dependent on string manipulation routines, the speed increase obtained will be significant. The compiler automatically defines the C pre-processor definition `HAVE_HW_PCMP` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.
- **`-mno-xl-pattern-compare`:** This option tells the compiler not to use pattern compare instructions. This is the default.
- **`-mhard-float`:** This option turns on the usage of single precision floating point instructions (`fadd`, `frsub`, `fmul`, and `fdiv`) in the compiler.

  It also uses `fcmp.p` instructions, where p is a predicate condition such as `le`, `ge`, `lt`, `gt`, `eq`, `ne`. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware. The compiler automatically defines the C pre-processor definition `HAVE_HW_FPU` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.
- **`-msoft-float`:** This option tells the compiler to use software emulation for floating point arithmetic. This option is the default.
- **`-mxl-float-convert`:** This option turns on the usage of single precision floating point conversion instructions (fint and flt) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.
- **`-mxl-float-sqrt`:** This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

General Program Options

- **`-msmall-divides`:** This option generates code optimized for small divides when no hardware divider exists. For signed integer divisions where the numerator and denominator are between 0 and 15 inclusive, this switch provides very fast table-lookup-based divisions. This switch has no effect when the hardware divider is enabled.

- **`-mxl-gp-opt`:** If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load or store operations to that address require two instructions.

  The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 KB of data. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value, needing only one instruction for a load or store to the small data area. This optimization can be turned on with the `-mxl-gp-opt` command line parameter. Variables of size less than a certain threshold value are stored in these areas and can be addressed with fewer instructions. The addresses are calculated during the linking stage.

  > **IMPORTANT!** *If this option is being used, it must be provided to both the compile and the link commands of the build process for your program. Using the switch inconsistently can lead to compile, link, or runtime errors.*

- **`-mno-clearbss`:** This option is useful for compiling programs used in simulation.

  According to the C language standard, uninitialized global variables are allocated in the .bss section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the .bss section with zero when the program starts execution. Optimizing compilers also allocates global variables that are assigned zero in C code to the `.bss` section.

  In a simulation environment, the above two language features can be unwanted overhead. Some simulators automatically zero the entire memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler to not allocate zero-initialized global variables in the .bss and instead move them to the `.data` section. This option might improve startup times for your application. Use this option with care and ensure either that you do not use code that relies on global variables being initialized to zero, or that your simulation platform performs the zeroing of memory.

- **`-mxl-stack-check`:** With this option, you can check whether the stack overflows when the program runs.

The compiler inserts code in the prologue of the every function, comparing the stack pointer value with the available memory. If the stack pointer exceeds the available free memory, the program jumps to a the subroutine `_stack_overflow_exit`. This subroutine sets the value of the `variable _stack_overflow_error` to 1.

You can override the standard stack overflow handler by providing the function `_stack_overflow_exit` in the source code, which acts as the stack overflow handler.

Application Execution Modes

- **`-xl-mode-executable`:** This is the default mode used for compiling programs with mb-gcc. This option need not be provided on the command line for `mb-gcc`. This uses the startup file `crt0.o`.
- **`xl-mode-bootstrap`:** This option is used for applications that are loaded using a bootloader. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If a normal executable is loaded by this bootloader, the application reset vector overwrites the reset vector of the bootloader. In such a scenario, on a processor reset, the bootloader does not execute first (it is typically required to do so) to reload this application and do other initialization as necessary.

  To prevent this, you must compile the bootloaded application with this compiler flag. On a processor reset, control then reaches the bootloader instead of the application.

  Using this switch on an application that is deployed in a scenario different from the one described above will not work. This mode uses `crt2.o` as a startup file.
- **`-xl-mode-novectors`:** This option is used for applications that do not require any of the MicroBlaze vectors. This is typically used in standalone applications that do not use any of the processor's reset, interrupt, or exception features. Using this switch leads to smaller code size due to the elimination of the instructions for the vectors. This mode uses `crt3.o` as a startup file.

> **IMPORTANT!** *Do not use more than one mode of execution on the command line. You will receive link errors due to multiple definition of symbols if you do so.*

Position Independent Code

The GNU compiler for MicroBlaze supports the `-fPIC` and `-fpic` switches. These switches enable position independent code (PIC) generation in the compiler. This feature is used by the Linux operating system only for MicroBlaze to implement shared libraries and relocatable executables. The scheme uses a global offset table (GOT) to relocate all data accesses in the generated code and a procedure linkage table (PLT) for making function calls into shared libraries. This is the standard convention in GNU-based platforms for generating relocatable code and for dynamically linking against shared libraries.

MicroBlaze Application Binary Interface

The GNU compiler for MicroBlaze uses the Application Binary Interface (ABI) defined in the *MicroBlaze Processor Reference Guide* (UG081). Refer to the ABI documentation for register and stack usage conventions and a description of the standard memory model used by the compiler.

MicroBlaze Assembler

The `mb-as` assembler for the MicroBlaze soft processor supports the same set of options supported by the standard GNU compiler tools. It also supports the same set of assembler directives supported by the standard GNU assembler.

The `mb-as` assembler supports all the opcodes in the MicroBlaze machine instruction set, with the exception of the `imm` instruction. The `mb-as` assembler generates `imm` instructions when large immediate values are used. The assembly language programmer is never required to write code with `imm` instructions. For more information on the MicroBlaze instruction set, refer to the *MicroBlaze Processor Reference Guide* (UG081).

The `mb-as` assembler requires all MicroBlaze instructions with an immediate operand to be specified as a constant or a label. If the instruction requires a PC-relative operand, then the `mb-as` assembler computes it and includes an imm instruction if necessary.

For example, the branch immediate if equal (`beqi`) instruction requires a PC-relative operand.

The assembly programmer should use this instruction as follows:

```
beqi r3, mytargetlabel
```

where `mytargetlabel` is the label of the target instruction. The `mb-as` assembler computes the immediate value of the instruction as `mytargetlabel - PC`.

If this immediate value is greater than 16 bits, the `mb-as` assembler automatically inserts an imm instruction. If the value of `mytargetlabel` is not known at the time of compilation, the `mb-as` assembler always inserts an `imm` instruction. Use the relax option of the linker remove any unnecessary imm instructions.

Similarly, if an instruction needs a large constant as an operand, the assembly language programmer should use the operand as is, without using an `imm` instruction. For example, the following code adds the constant 200,000 to the contents of register r3, and stores the results in register r4:

```
addi r4, r3, 200000
```

The `mb-as` assembler recognizes that this operand needs an imm instruction, and inserts one automatically.

In addition to the standard MicroBlaze instruction set, the `mb-as` assembler also supports some pseudo-op codes to ease the task of assembly programming. The following table lists the supported pseudo-opcodes.

Table 31: **Pseudo-Opcodes Supported by the GNU Assembler**

Pseudo Opcodes	Explanation
nop	No operation. Replaced by instruction: or R0, R0, R0
la Rd, Ra, Imm	Replaced by instruction: addik Rd, Ra, imm; = Rd = Ra + Imm;
not Rd, Ra	Replace by instruction: xori Rd, Ra, -1
neg Rd, Ra	Replace by instruction: rsub Rd, Ra, R0
sub Rd, Ra, Rb	Replace by instruction: rsub Rd, Rb, Ra

MicroBlaze Linker Options

The `mb-ld` linker for the MicroBlaze soft processor provides additional options to those supported by the GNU compiler tools. The options are summarized in this section.

- **`-defsym _TEXT_START_ADDR=value`:** By default, the text section of the output code starts with the base address 0x28. This can be overridden by using the `-defsym _TEXT_START_ADDR` option. If this is supplied to `mb-gcc` compiler, the text section of the output code starts from the given value.

  You do not have to use `-defsym _TEXT_START_ADDR` if you want to use the default start address set by the compiler.

  This is a linker option and should be used when you invoke the linker separately. If the linker is being invoked as a part of the `mb-gcc` flow, you must use the following option:

  ```
  -Wl,-defsym -Wl,_TEXT_START_ADDR=value
  ```

- **`-relax`:** This is a linker option that removes all unwanted `imm` instructions generated by the assembler. The assembler generates an `imm` instruction for every instruction where the value of the immediate cannot be calculated during the assembler phase. Most of these instructions do not need an `imm` instruction. These are removed by the linker when the `-relax` command line option is provided.

  This option is required only when linker is invoked on its own. When linker is invoked through the `mb-gcc` compiler, this option is automatically provided to the linker.

- `-N`: This option sets the text and data section as readable and writable. It also does not page-align the data segment. This option is required only for MicroBlaze programs. The top-level GCC compiler automatically includes this option, while invoking the linker, but if you intend to invoke the linker without using GCC, use this option.

  The MicroBlaze linker uses linker scripts to assign sections to memory. These are listed in the following section.

MicroBlaze Linker Script Sections

The following table lists the input sections that are assigned by MicroBlaze linker scripts.

Table 32: **Section Names and Descriptions**

Section	Description
.vectors.reset	Reset vector code.
.vectors.sw_exception	Software exception vector code.
.vectors.interrupt	Hardware Interrupt vector code.
.vectors.hw_exception	Hardware exception vector code.
.text	Program instructions from code in functions and global assembly statements.
.rodata	Read-only variables.
.sdata2	Small read-only static and global variables with initial values.
.data	Static and global variables with initial values. Initialized to zero by the boot code.
.sdata	Small static and global variables with initial values.
.sbss2	Small read-only static and global variables without initial values. Initialized to zero by boot code.
.sbss	Small static and global variable without initial values. Initialized to zero by the boot code.
.bss	Static and global variables without initial values. Initialized to zero by the boot code.
.heap	Section of memory defined for the heap.
.stack	Section of memory defined for the stack.

Tips for Writing or Customizing Linker Scripts

Keep the following points in mind when writing or customizing your own linker script:

- Ensure that the different vector sections are assigned to the appropriate memories as defined by the MicroBlaze hardware.

- Allocate space in the `.bss` section for stack and heap. Set the `_stack` variable to the location after `_STACK_SIZE` locations of this area, and the `_heap_start` variable to the next location after the `_STACK_SIZE` location. Because the stack and heap need not be initialized for hardware as well as simulation, define the `_bss_end` variable after the `.bss` and `COMMON` definitions.

  Note: The `.bss` section boundary does not include either stack or heap.

- Ensure that the variables `_SDATA_START__` , `_SDATA_END__`, `SDATA2_START`, `_SDATA2_END__`, `_SBSS2_START__` , `_SBSS2_END__`, `_bss_start`, `_bss_end`, `_sbss_start`, and `_sbss_end` are defined to the beginning and end of the sections `sdata`, `sdata2`, `sbss2`, `bss`, and `sbss` respectively.
- ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap). The standard CRT that is provided assumes a single `.bss` section that is initialized to zero. If there are multiple `.bss` sections, this CRT does not work. You should write your own CRT that initializes all the `.bss` sections.

Startup Files

The compiler includes pre-compiled startup and end files in the final link command when forming an executable. Startup files set up the language and the platform environment before your application code executes. Start up files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to the following table for details.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions, such as C++ constructors.
- Initialize the hardware subsystem. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

Similarly, end files are used to include code that must execute after your program ends. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- Deinitialize the hardware subsystem. For example, if the program is being profiled, clean up the profiling sub-system.

The following table lists the register names, values, and descriptions in the C runtime files.

Table 33: **Register Initialization in C Runtime Files**

Register	Value	Description
r1	_stack-16	The stack pointer register is initialized to point to the bottom of the stack area with an initial negative offset of 16 bytes. The 16 bytes can be used for passing in arguments.
r2	_SDA2_BASE	_SDA2_BASE_ is the read-only small data anchor address.
r13	_SDA_BASE_	_SDA_BASE is the read-write small data anchor address.
Other registers	Undefined	Other registers do not have defined values.

The following subsections describe the initialization files used for various application modes. This information is for advanced users who want to change or understand the startup code of their application.

For MicroBlaze, there are two distinct stages of C runtime initialization. The first stage is primarily responsible for setting up vectors, after which it invokes the second stage initialization. It also provides exit stubs based on the different application modes.

First Stage Initialization Files

- `crt0.o`: This initialization file is used for programs which are to be executed in standalone mode, without the use of any bootloader or debugging stub. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine `_crtinit`. On returning from `_crtinit`, it ends the program by infinitely looping in the `_exit` label.
- `crt1.o`: This initialization file is used when the application is debugged in a software-intrusive manner. It populates all the vectors except the breakpoint and reset vectors and transfers control to the second-stage `_crtinit` startup routine.
- `crt2.o`: This initialization file is used when the executable is loaded using a bootloader. It populates all the vectors except the reset vector and transfers control to the second-stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the reset vector is not populated, on a processor reset, control is transferred to the bootloader, which can reload and restart the program.
- `crt3.o`: This initialization file is employed when the executable does not use any vectors and wishes to reduce code size. It populates only the reset vector and transfers control to the second stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the other vectors are not populated, the GNU linking mechanism does not pull in any of the interrupt and exception handling related routines, thus saving code space.

Second Stage Initialization Files

According to the C standard specification, all global and static variables must be initialized to 0. This is a common functionality required by all the CRTs above. Another routine, `_crtinit`, is invoked. The `_crtinit` routine initializes memory in the `.bss` section of the program. The `_crtinit` routine is also the wrapper that invokes the main procedure. Before invoking the main procedure, it might invoke other initialization functions. The `_crtinit` routine is supplied by the startup files described below.

- **`crtinit.o`:**

  This default, second stage, C startup file performs the following steps:

  1. Clears the `.bss` section to zero.
  2. Invokes `_program_init`.
  3. Invokes "constructor" functions (`_init`).
  4. Sets up the arguments for main and invokes main.
  5. Invokes "destructor" functions (`_fini`).
  6. Invokes `_program_clean` and returns.

- **`pgcrtinit.o`:**

  This second stage startup file is used during profiling, and performs the following steps:

  1. Clears the `.bss` section to zero.
  2. Invokes `_program_init`.
  3. Invokes `_profile_init` to initialize the profiling library.
  4. Invokes "constructor" functions (`_init`).
  5. Sets up the arguments for main and invokes main.
  6. Invokes "destructor" functions (`_fini`).
  7. Invokes `_profile_clean` to cleanup the profiling library.
  8. Invokes `_program_clean`, and then returns.

- **`sim-crtinit.o`:**

  This second-stage startup file is used when the `-mno-clearbss` switch is used in the compiler, and performs the following steps:

  1. Invokes `_program_init`.
  2. Invokes "constructor" functions (`_init`).
  3. Sets up the arguments for main and invokes main.

4. Invokes “destructor” functions (`_fini`).
5. Invokes `_program_clean`, and then returns.

- **`sim-pgcrtinit.o`:**

  This second stage startup file is used during profiling with the `-mno-clearbss` switch, and performs the following steps in order:

  1. Invokes `_program_init`.
  2. Invokes `_profile_init` to initialize the profiling library.
  3. Invokes “constructor” functions (`_init`).
  4. Sets up the arguments for and invokes main.
  5. Invokes “destructor” functions (`_fini`).
  6. Invokes `_profile_clean` to cleanup the profiling library.
  7. Invokes `_program_clean`, and then returns.

Other Files

The compiler also uses certain standard start and end files for C++ language support.

These are `crti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections.

Modifying Startup Files

The initialization files are distributed in both precompiled and source form with Vivado. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the MicroBlaze GNU compiler can be found in the `<XILINX_>/Vitis/<version>/data/embeddedsw/lib/microblaze/src/` directory, where `<XILINX_>` is the Vivado installation path and `<version>` is the release version of the Vitis software platform.

To fulfill a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, you can assemble the files into `.o` files and place them in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory -name` command line option while invoking `mb-gcc`.

To prevent the default startup files from being used, use the `-nostartfiles` on the final compile line.

***Note*:** The miscellaneous compiler standard CRT files, such as `crti.o`, and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You might need to bring them in on your final link command.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you might want to eliminate all sources of overhead. This section describes how to reduce the overhead of invoking the C++ constructor or destructor code in a C program that does not require that code. You might be able to save approximately 220 bytes of code space by making the following modifications:

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, copy over the particular versions of `crtn.s` and `xcrtinit.s` that suit your application. For example, if your application is being bootstrapped and profiled, copy `crt2.s` and `pg-crtinit.s` from the installation area.
2. Modify pg-crtinit.s to remove the following lines:

   ```
   brlid r15, __init
   /* Invoke language initialization functions */
   nop
   ```

   and

   ```
   brlid r15, __fini
   /* Invoke language cleanup functions */
   nop
   ```

   This avoids referencing the extra code usually pulled in for constructor and destructor handling, reducing code size.
3. Compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the -nostartfiles switch to the compiler. Add the -B directory switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

If your application is executing in a different mode, then you must pick the appropriate CRT files based on the description in Startup Files.

Compiler Libraries

The `mb-gcc` compiler requires the GNU C standard library and the GNU math library. Precompiled versions of these libraries are shipped with Vivado. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze. To manually select the library version that you would like to use, look in the following folder:

```
$XILINX_/gnu/microblaze/<platform>/microblaze-xilinx-elf/lib
```

The filenames are encoded based on the compiler flags and configurations used to compile the library. For example, `libc_m_bs.a` is the C library compiled with hardware multiplier and barrel shifter enabled in the compiler.

The following table shows the current encodings used and the configuration of the library specified by the encodings.

Table 34: **Encoded Library Filenames on Compiler Flags**

Encoding	Description
_bs	Configured for barrel shifter.
_m	Configured for hardware multiplier.
_p	Configured for pattern comparator.

Of special interest are the math library files (`libm*.a`). The C standard requires the common math library functions (sin()and cos(), for example) to use double-precision floating point arithmetic. However, double-precision floating point arithmetic might not be able to make full use of the optional, single-precision floating point capabilities in available for MicroBlaze.

The newlib math libraries have alternate versions that implement these math functions using single-precision arithmetic. These single-precision libraries might be able to make direct use of the MicroBlaze processor hardware floating point unit (FPU) and could therefore perform better.

If you are sure that your application does not require standard precision, and you want to implement enhanced performance, you can manually change the version of the linked-in library.

By default, the CPU driver copies the double-precision version (`libm_*_fpd.a`) of the library into your IP integrator project.

To get the single precision version, you can create a custom CPU driver that copies the corresponding `libm_*_fps.a` library instead. Copy the corresponding `libm_*_fps.a` file into your processor library folder (such as `microblaze_0/lib`) as `libm.a`.

When you have copied the library that you want to use, rebuild your application software project.

Thread Safety

The MicroBlaze processor C and math libraries distributed with Vivado are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are not thread-safe and causes unrecoverable errors in the system at runtime. Use appropriate mutual exclusion mechanisms when using the Vivado libraries in a multi-threaded environment.

Command Line Arguments

The MicroBlaze processor programs cannot take command line arguments. The command line arguments `argc` and `argv` are initialized to 0 by the C runtime routines.

Interrupt Handlers

Interrupt handlers must be compiled in a different manner than normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save the volatile registers that are being used. Interrupt handlers should also store the value of the machine status register (RMSR) when an interrupt occurs.

- **`interrupt_handler`:** To distinguish an interrupt handler from a sub-routine, `mb-gcc` looks for an attribute (`interrupt_handler`) in the declaration of the code. This attribute is defined as follows:

  ```
  void function_name () __attribute__ ((interrupt_handler));
  ```

  ***Note*:** The attribute for the interrupt handler is to be given *only* in the prototype and *not* in the definition.

  Interrupt handlers might also call other functions, which might use volatile registers. To maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function.

  ***Note*:** Functions that have calls to other sub-routines are called non-leaf functions.

  Interrupt handlers are defined in the Microprocessor Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions. The interrupt handler uses the instruction `rtid` for returning to the interrupted function.

- **`save_volatiles`:** The MicroBlaze compiler provides the attribute `save_volatiles`, which is similar to the `interrupt_handler` attribute, but returns using `rtsd` instead of `rtid`. This attribute saves all the volatiles for non-leaf functions and only the used volatiles in the case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

- **`fast_interrupt`:** The MicroBlaze compiler provides the attribute `fast_interrupt`, which is similar to the interrupt_handler attribute. On fast interrupt, MicroBlaze jumps to the interrupt routine address instead jumping to the fixed address 0x10.

  Unlike a normal interrupt, when the attribute `fast_interrupt` is used on a C function, MicroBlaze saves only minimal registers.

```
void function_name () __attribute__ ((fast_interrupt));
```

Table 35: **Use of Attributes**

Attributes	Functions
interrupt_handler	This attribute saves the machine status register and all the volatiles, in addition to the non-volatile registers. rtid returns from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved.
save_volatiles	This attribute is similar to interrupt_handler, but it uses rtsd to return to the interrupted function, instead of rtid.
fast_interrupt	This attribute is similar to interrupt_handler, but it jumps directly to the interrupt routine address instead of jumping to the fixed address 0x10.

Arm Compiler Usage and Options

1. Arm® Cortex®-A9 targets can be compiled using the arm-none-eabi toolchain
2. Arm® Cortex-A53 targets can be compiled using the aarch64-none-elf toolchain
3. Arm® Cortex-R5F targets can be compiled using the armr5-none-eabi toolchain

Arm® Cortex® A9 targets can be compiled using the `arm-none-eabi` toolchain. The `arm-none-eabi` toolchain contains the complete GNU toolchain including all of the following components:

- Common startup code sequence
- GNU binary utilities (binutils)
- GNU C compiler (GCC)
- GNU C++ compiler (G++)
- GNU C++ runtime library (libstdc++)
- GNU debugger (GDB)
- Newlib C library

Usage

***Note*:** Cortex®-R5F and A53 toolchains can be used in a similar way.

Compiling

```
arm-none-eabi-gcc -c file1.c -I<include_path> -o file1.o
arm-none-eabi-gcc -c file2.c -I<include_path> -o file2.o
```

Linking

```
arm-none-eabi-gcc -Wl,-T -Wl,lscript.ld -L<libxil.a path> -o
"App.elf"file1.o file2.o -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
```

For descriptions of flags used in the commands above, refer to the compiler help, using any of the following commands:

- `--help`
- `-v --help`
- `--target-help`

Compiler Options

Other GNU compiler options that can be applied using Arm®-related flags can be found on the GNU website. These flags can be used in the steps above, as required. All the Arm® GCC compiler options are listed at the link above. However, actual support depends on the target in use (Arm Cortex-A9, Cortex-A53, Cortex®-A72, and Cortex-R5F in this case) and on the compiler toolchain.

Other Notes

C++ Code Size

The GCC toolchain combined with the latest open source C++ standard library (`libstdc++-v3`) might be found to generate large code and data fragments as compared to an equivalent C program. A significant portion of this overhead comes from code and data for exception handling and runtime type information. Some C++ applications do not require these features.

To remove the overhead and optimize for size, use the `-fno-exceptions` and/or the `-fno-rtti` switches. This is recommended only for advanced users who know the requirements of their application and understand these language features. Refer to the GCC documentation for more specific information on available compiler options and their impact.

C++ programs might have more intensive dynamic memory requirements (stack and heap size) due to more complex language features and library routines. Many of the C++ library routines can request memory to be allocated from the heap. Review your heap and stack size requirements for C++ programs to ensure that they are satisfied.

C++ Standard Library

The C++ standard defines the C++ standard library. A few of these platform features are unavailable on the default AMD Vivado™ software platform. For example, file I/O is supported in only a few well-defined `STDIN`/`STDOUT` streams. Similarly, locale functions, thread-safety, and other such features might not be supported.

***Note*:** The C++ standard library is not built for a multi-threaded environment. Common C++ features such as new and delete are not thread safe. Use caution when using the C++ standard library in an operating system environment.

For more information on the GNU C++ standard library, refer to the documentation available on the GNU website.

Position Independent Code (Relocatable Code)

The MicroBlaze™ processor compilers support the `-fPIC` switch to generate position independent code. While both these features are supported in the AMD compiler, they are not supported by the rest of the libraries and tools, because Vivado only provides a standalone platform. No loader or debugger can interpret relocatable code and perform the correct relocations at runtime. These independent code features are not supported by the AMD libraries, startup files, or other tools. Third-party OS vendors could use these features as a standard in their distribution and tools.

Other Switches and Features

Other switches and features might not be supported by the AMD Vivado compilers and/or platform, such as `-fprofile-arcs`. Some features might also be experimental in nature (as defined by open source GCC) and could produce incorrect code if used inappropriately. Refer to the GCC documentation for more information on specific features.

Section VII

Embedded Design Tutorials

The following hardware specific embedded design tutorials are available for embedded software designers.

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

The software features are demonstrated in the Vitis tutorial under embedded design section. See *Vitis Unified Software Platform Tutorials Landing Page* (UG1605).

Drivers and Libraries

Drivers and libraries are hosted on the AMD wiki. You can access them with the following links:

- Bare-metal Drivers and Libraries
- Linux Drivers

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

1. *Vitis Unified Software Platform Documentation Landing Page* (UG1416)
2. *Vitis Software Platform Release Notes* (UG1742)
3. *Porting Guide for embeddedsw Components System Device Tree Based Build Flow* (UG1647)
4. *Zynq 7000 SoC: Embedded Design Tutorial* (UG1165)
5. *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209)
6. *Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305)
7. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
8. *Versal Adaptive SoC Technical Reference Manual* (AM011)
9. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940)
10. *Bootgen User Guide* (UG1283)
11. *Zynq 7000 SoC Technical Reference Manual* (UG585)
12. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
13. *MicroBlaze Processor Reference Guide* (UG081)
14. *Vitis Unified Software Platform Tutorials Landing Page* (UG1605)
15. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
16. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
17. *Embedded Design Development Using Vitis* (UG1701)
18. *Software Debugger Reference Guide* (UG1725)
19. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
20. *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393)
21. *Vitis Software Platform Release Notes* (UG1742)
22. *Vitis Reference Guide* (UG1702)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/08/2025 Version 2025.1	
Entire document	Added titles to screenshots missing titles.
05/29/2025 Version 2025.1	
Files Required for Source Control	New topic specifying which files are required for source control, and which ones are optional.
Authenticated Jtag Access	Updated content for 2025.1 release.
Multi-Cable and Multi-Device Support	Updated content for 2025.1 release.
app	Updated content for 2025.1 release.
Extensions	New topic: added support for third party extensions
Serial Monitor	New topic: added the serial monitor extension.
XEN Aware Debug	New topic: added support for XEN aware debug.
Segmented Configuration	New topic: added support for segmented configuration.

Please Read: Important Legal Notices

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