See all versions
of this document

Vitis Unified Software Platform Documentation

Embedded Software Development

UG1400 (v2025.2) November 20, 2025

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 non essential programs.*

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:
    a. At the Select Install Type screen, enter your AMD user account credentials, and select **Download and Install Now**.
    b. Click **Next** to open the Accept License Agreements screen of the installer.
    c. Accept the terms and conditions by clicking each **I Agree** check box.
    d. Click **Next** to open the next screen (only required on the web installer).
6. On the Select Product to Install screen:
    a. To install the full Vitis Software Platform for embedded software and application acceleration development, choose **Vitis** and click **Next**. 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.

Although not required, you can enable **Vitis IP Cache** to install cache files (for example designs found in the release). The Vitis tool installs the files 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 and review the location summary. Ensure there is enough disk space and click **Next** to open the Installation Summary screen.
11. Click **Install** to begin the installation of the software.

After a successful installation of the full Vitis unified software, a confirmation message appears, 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*:** Windows do not require this script.

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. Be sure to install any missing packages manually. For example, if your installation of Linux does not include the zip command-line utility, you need to manually install it. 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. The software platform is designed 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. The platform provides the Vitis unified integrated design environment (IDE) as the front end GUI to support embedded software developers. The Vitis unified IDE works with hardware designs created with AMD Vivado™ Design Suite. The IDE enables developers to work efficiently when developing software applications towards AMD embedded processors.

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:

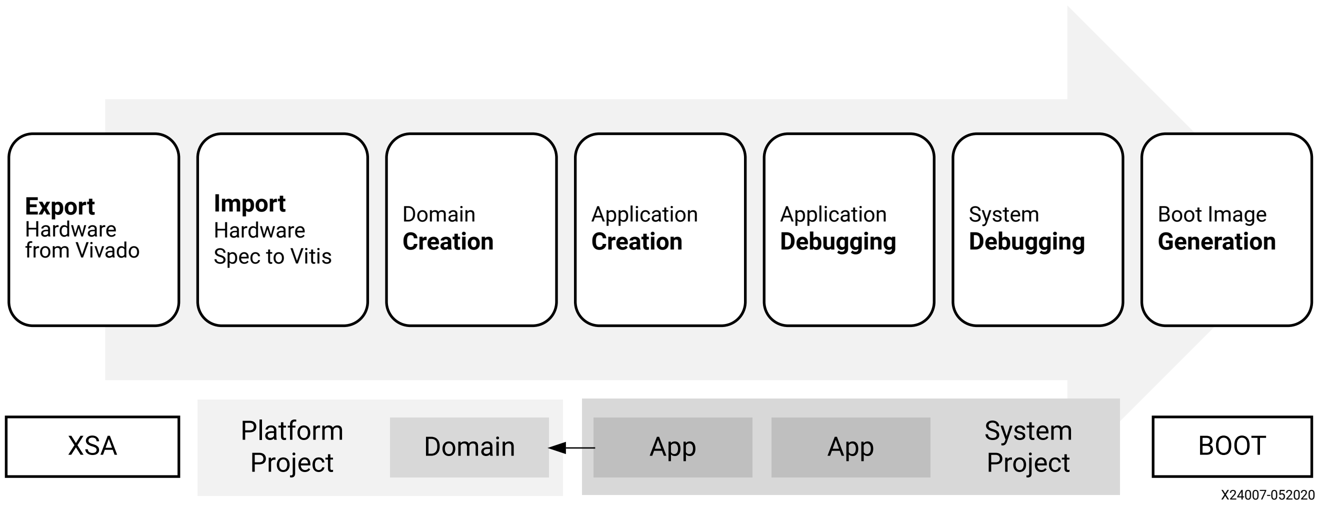
- Creating platforms from AMD Vivado™ generated hardware designs and generating BSP for software development
- Creating applications from example designs or empty template
- Configuring and building the platforms and applications
- Running, debugging, or profiling the applications on hardware
- Managing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

- 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
- The GUI and command line interface (CLI) support the above features

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. Application acceleration projects use the platform heavily. 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.
- Domain is the name for the software environment settings. Domains are also a part of a platform.
- Software developers create applications based on the platform and domains.
- You can debug applications in the Vitis Unified 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 Unified 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. One is called fixed XSA with a fixed platform. The other is called extensible XSA with an extensible platform.

  - A fixed XSA, which includes a complete device image generated within Vivado, establishes a fixed platform for embedded development.
  - An extensible XSA is for acceleration development purposes. During the linking stage in v++, the system creates a comprehensive device image for generating BOOT.bin. The BOOT.bin is for boot operations for acceleration development.

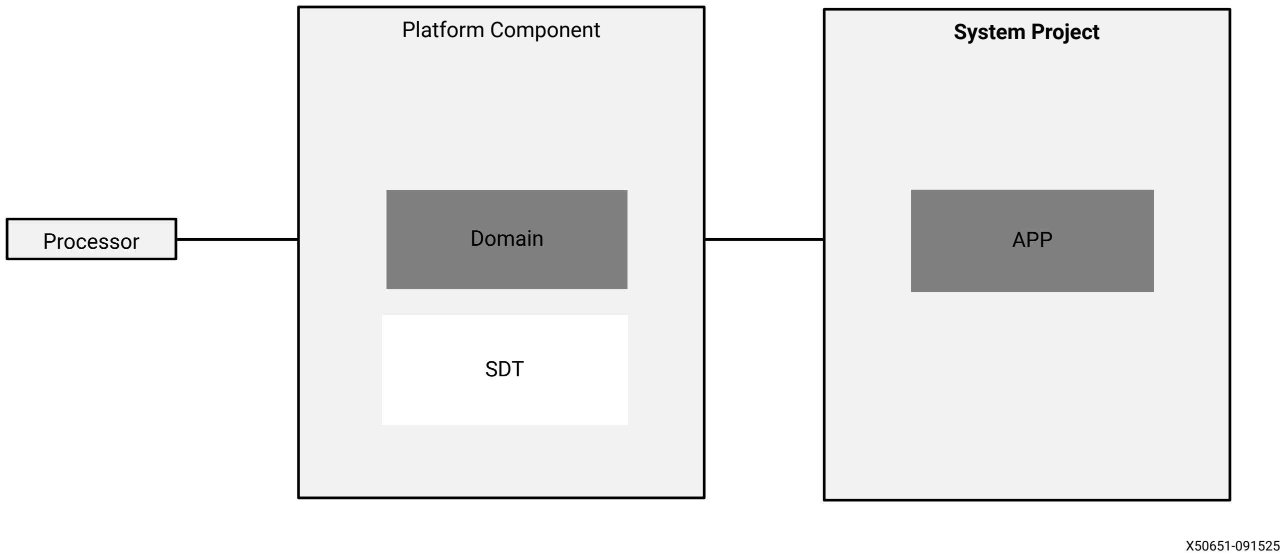
  If you use an extensible platform to create bare-metal applications, you can encounter the issue "cannot find corresponding devices."

  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 need to create a workspace. A workspace is a directory location used by the Vitis unified software platform to store project data and metadata. You must provide an initial workspace location when the Vitis software platform launches.

- **XSA:** The tool exports XSAs from the Vivado Design Suite. The XSA has hardware specifications such as 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 (SDT) is:
  - Generated based on the XSA using the SDTGEN utility upon the Platform Creation.
  - Contains all the processor and respective memory mapped IP for each processor.
  - For capturing and maintaining the HW metadata from the XSA. Does not deploy on the target.

  The Lopper Utility is for extracting HW metadata. The metadata can include the processor list, `xparameters.h` generation, linker script generated, BSP creation, and so on.
- **Platform:** The target 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 operating system (OS) with a collection of software drivers.
  - You can build your application on the domain.
  - 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 ties to a single processor or a cluster of isomorphic processors (such as 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 exist together in a system project.
  - Two Linux applications can exist together in a system project.
  - A workspace can contain multiple system projects.
- **Application (Software Project):** An application contains the source and header files to compile and generate a binary output (ELF) file. A system project can contain multiple applications. Each application 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 Vitis tool cannot modify the hardware design.
- **Embedded Acceleration:** This type of platform supports embedded software development and application acceleration. The platform provides clocks, bus interfaces, and interrupt controllers for the acceleration kernel.
- **Data Center Acceleration:** You can develop acceleration kernels and x86 host applications on this platform. The kernel uses a PCIe® bus.

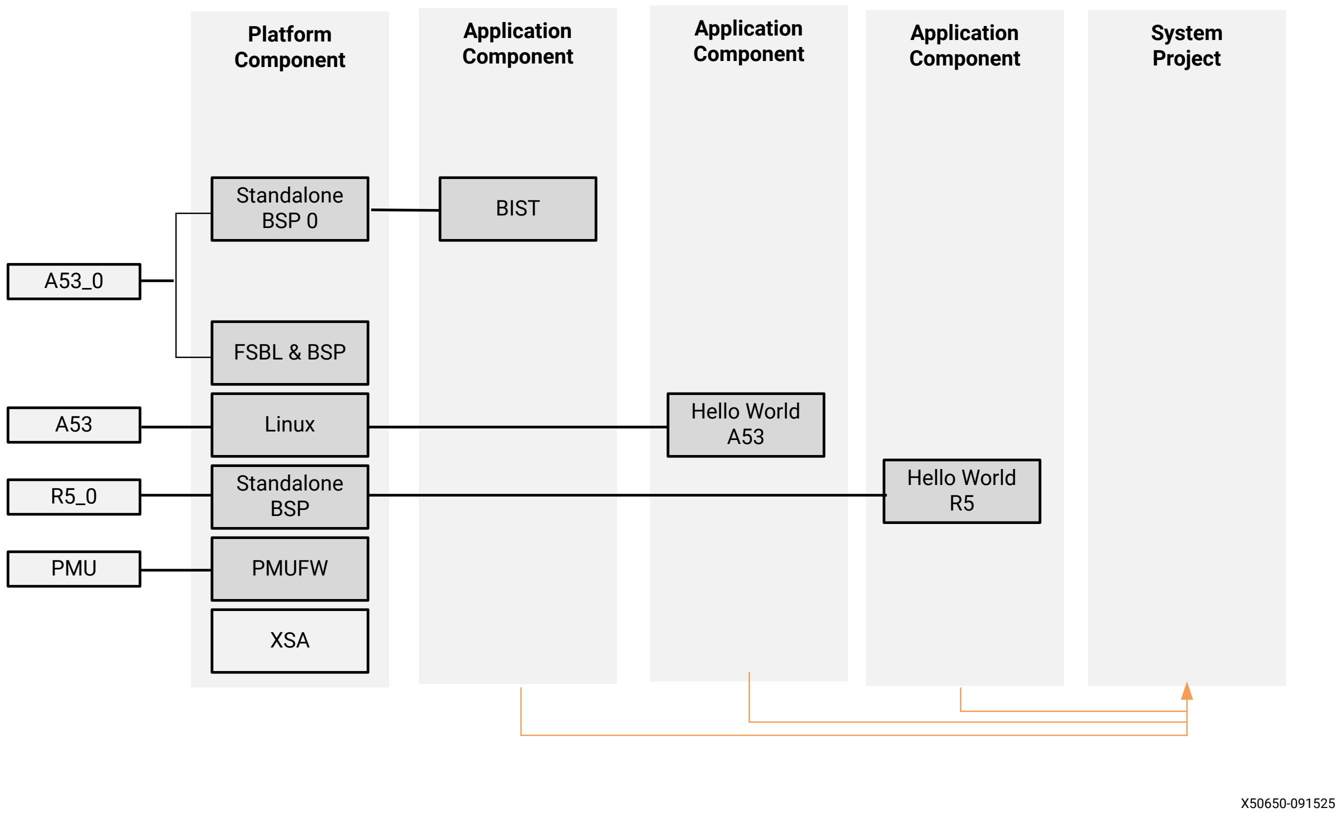
  ***Note*:** The Vitis software platform can:

  - Extend the hardware design of extensible platforms.
  - Add acceleration kernels (PL or AI Engine) to the original hardware design.

  You can use this type of platform 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**



- You can create Linux domains for Arm® Cortex®-A53 SMP clusters. You can compile Linux applications and link them against libraries. The Linux domain's `sysroot` provides these libraries.
- Arm Cortex-A53 core 0 and Arm Cortex®-R5F core 0 can run hello world applications at the same time. You can group these applications 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. The bare metal build-in-self-test application can have its own BSP settings.
- The system project manages multiple application components. Adding multiple application components in one system project means these applications run simutaneously. You do not need a system project if one application runs at one time.
- You can automatically create boot components such as FSBL and PMU firmware in platform projects. Boot components have their own BSP settings.

Migrating from the Classic Vitis IDE to Vitis Unified IDE

The Vitis Unified IDE workspace ensures 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, you can no longer use the Vitis Classic IDE. No migration utility is available in this version. If you did not migrate your workspace in a previous release, you 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

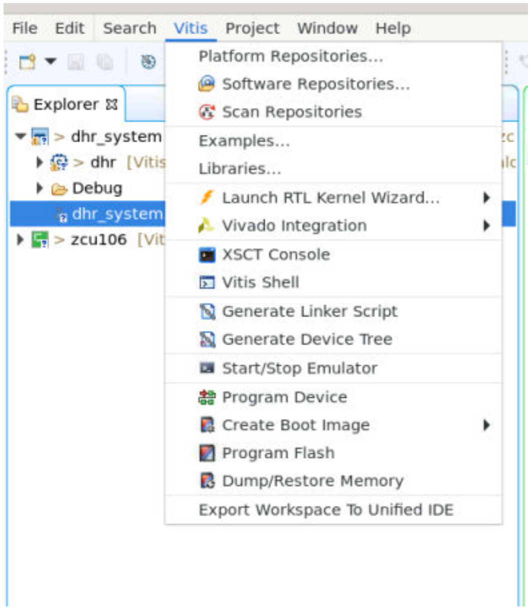
TIP: *For a cleaner and more robust migration process, we recommend manual migration over the migration utility.*

Migrate Projects Using Previous Version Utility

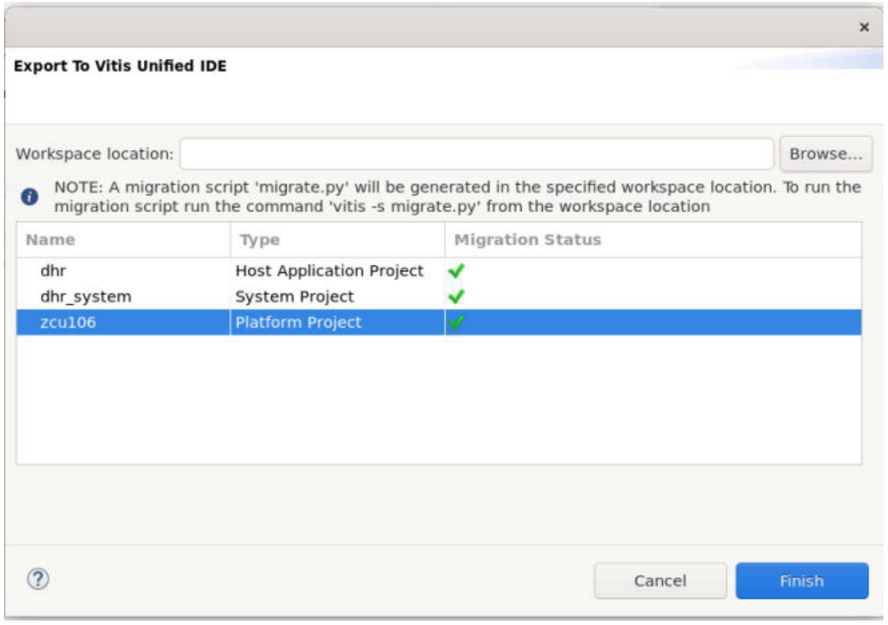
A migration utility is available in the Classic IDE for release versions 2023.2, 2024.1, and 2024.2. After migrating your project in the Classic IDE, you can then update your hardware platform and application to the current release.

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.



NOTE: A migration script 'migrate.py' will be generated in the specified workspace location. To run the migration script run the command 'vitis -s migrate.py' from the workspace location

Name	Type	Migration Status
dhr	Host Application Project	✓
dhr_system	System Project	✓
zcu106	Platform Project	✓

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

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

   The Vitis tool runs the migration script and copies the project resources to the new location. The process takes some time. Once completed, you can see the migrated components and projects in the workspace directory. You can find the project resources in the new location.
6. Open the migrated workspace with the 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 can 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 can 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*:** The Vitis Classic IDE is removed from the 2025.1 release onwards. If you need 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.

Standalone Application Component Migration Details

Driver APIs no longer use the Device ID.

- All driver APIs now use the base address of the peripheral IP due to the industry standard for controlling hardware.
- The system does not generate the DeviceID in `xparameters.h` anymore. Compiling a migrated application directly can result in compilation errors.
- If your application relies on DeviceID for IP driver initialization, refer to the 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

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

Features	Vitis Unified IDE	Vitis Classic IDE
Target Management	Target Connections	Target Connections
Version Control	Source Control	Version Control Git
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.

Section II

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 section also explains options in each view of the IDE, and information about Vitis Utilities.

This section contains the following chapters:

Chapter 4

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 groups together source and data files that make up the design(s). The workspace stores the configurations 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:

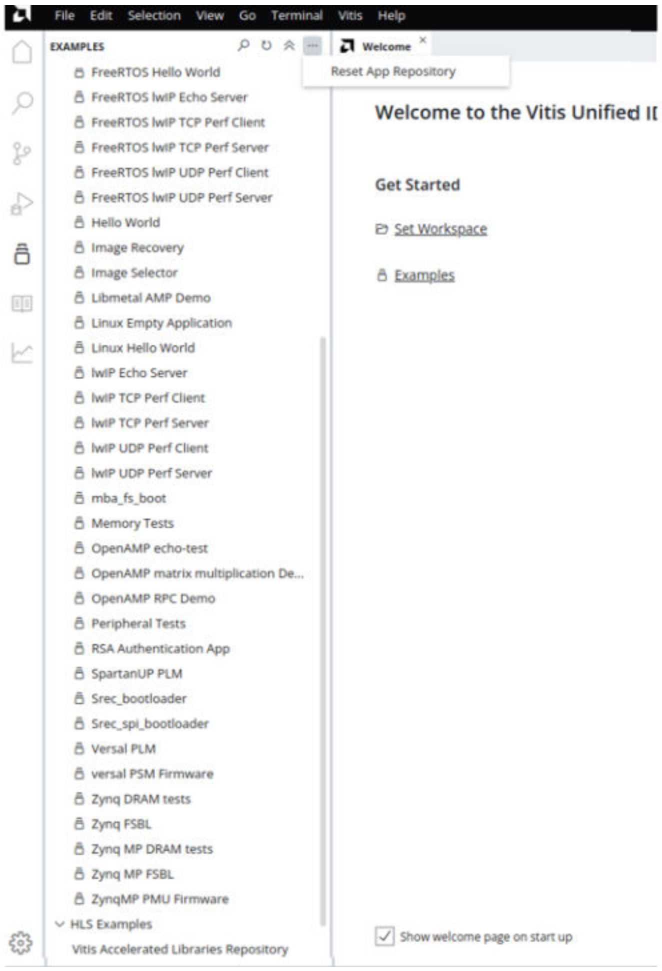
```
source <XRT_Install_Path>/setup.sh
```

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.

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:** This mode lets you enter commands through the interactive Python shell, outside of the GUI. See Vitis Interactive Python Shell in the *Vitis Reference Guide* (UG1702) for details.

  ```
  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 the Jupyter Notebook server with the Vitis environment. Launches the front end UI in your default web browser.

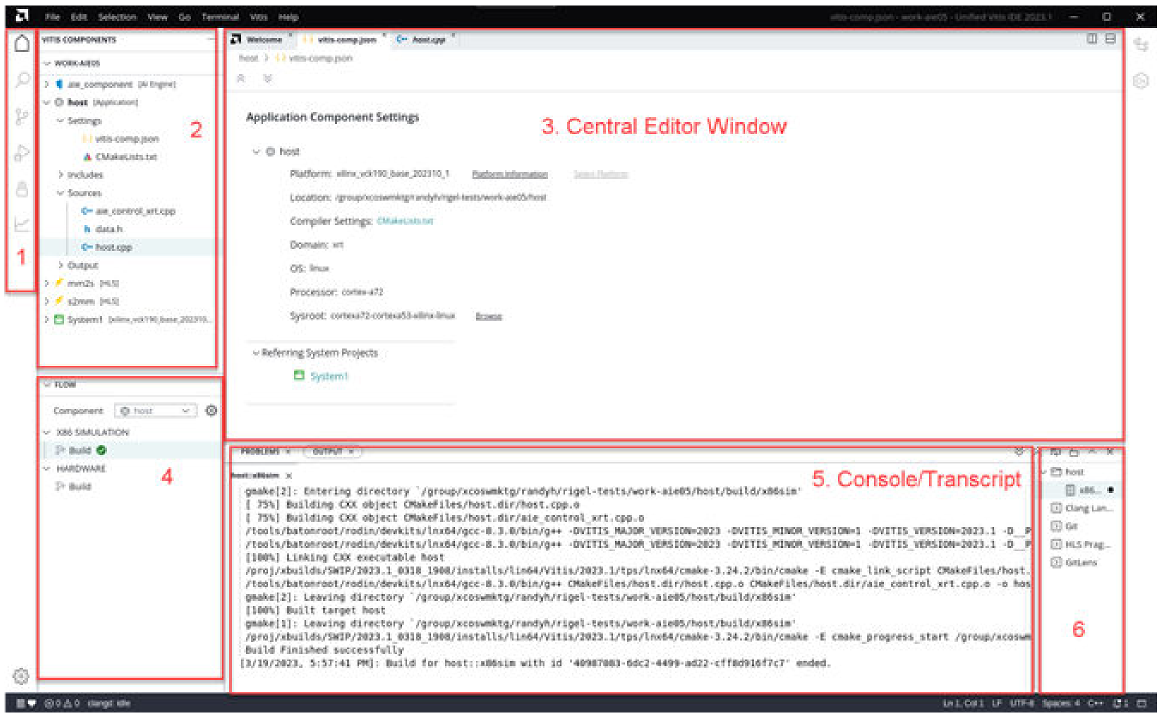
  ```
  vitis -j
  ```

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**



Application Component Settings

host

Platform: xilinx_vck190_base_202310_1

Location: /group/xcoswmktg/randyh/rigel-tests/work-aie05/host

Compiler Settings: CMakeLists.txt

Domain: xrt

OS: linux

Processor: cortex-a72

Sysroot: cortexa72-cortexa53-xilinx-linux

Referring System Projects

System1

```
gmake[2]: Entering directory `/group/xcoswmktg/randyh/rigel-tests/work-aie05/host/build/x86sim'
[ 75%] Building CXX object CMakeFiles/host.dir/host.cpp.o
[ 75%] Building CXX object CMakeFiles/host.dir/aie_control_xrt.cpp.o
/tools/batonroot/rodin/devkits/lnx64/gcc-8.3.0/bin/g++ -DVITIS_MAJOR_VERSION=2023 -DVITIS_MINOR_VERSION=1 -DVITIS_VERSION=2023.1 -D__P
/tools/batonroot/rodin/devkits/lnx64/gcc-8.3.0/bin/g++ -DVITIS_MAJOR_VERSION=2023 -DVITIS_MINOR_VERSION=1 -DVITIS_VERSION=2023.1 -D__P
[100%] Linking CXX executable host
/proj/xbuilds/SWIP/2023.1_0318_1908/installs/lin64/Vitis/2023.1/tps/lnx64/cmake-3.24.2/bin/cmake -E cmake_link_script CMakeFiles/host.
/tools/batonroot/rodin/devkits/lnx64/gcc-8.3.0/bin/g++ CMakeFiles/host.dir/host.cpp.o CMakeFiles/host.dir/aie_control_xrt.cpp.o -o hos
gmake[2]: Leaving directory `/group/xcoswmktg/randyh/rigel-tests/work-aie05/host/build/x86sim'
[100%] Built target host
gmake[1]: Leaving directory `/group/xcoswmktg/randyh/rigel-tests/work-aie05/host/build/x86sim'
/proj/xbuilds/SWIP/2023.1_0318_1908/installs/lin64/Vitis/2023.1/tps/lnx64/cmake-3.24.2/bin/cmake -E cmake_progress_start /group/xcoswm
Build Finished successfully
[3/19/2023, 5:57:41 PM]: Build for host::x86sim with id '40987083-6dc2-4499-ad22-cff8d916f7c7' ended.
```

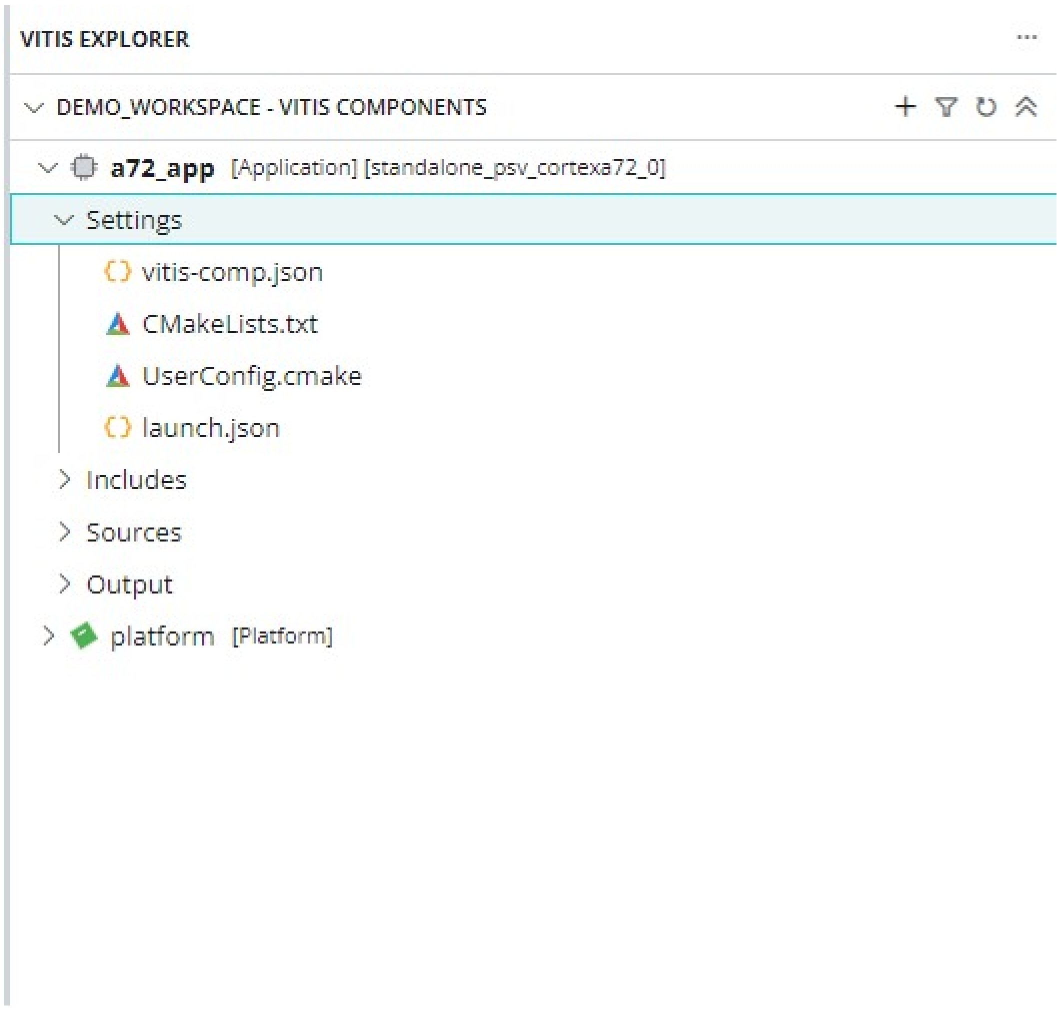
1. The Toolbar menu provides easy access to major features of the tool.
2. You can customize the Toolbar based on your preference using the View dropdown.
3. You can use the Vitis Component Explorer to view a virtual hierarchy of the workspace. The workspace can help you understand the different elements of the component or project.
4. You can use the Central Editor window for editing components, configurations, and source files.
5. The Flow Navigator displays the design flow for the active component:
   - Components have different work flows.
   - The workflow of the active component is in the Flow Navigator.
   - You can specify the active component by selecting it in the Flow Navigator.
6. The Console/Terminal area displays the following:

- The output transcripts of the tool.
- Windows such as the Terminal window and the Pipeline view.
- You can use the folders of the workspace to run scripts on the content.

7. The Index displays a list of transcripts of the build steps during the session. This display allows you to reopen a process transcript you closed.

Vitis Explorer View

Figure 6: **Vitis Explorer View**

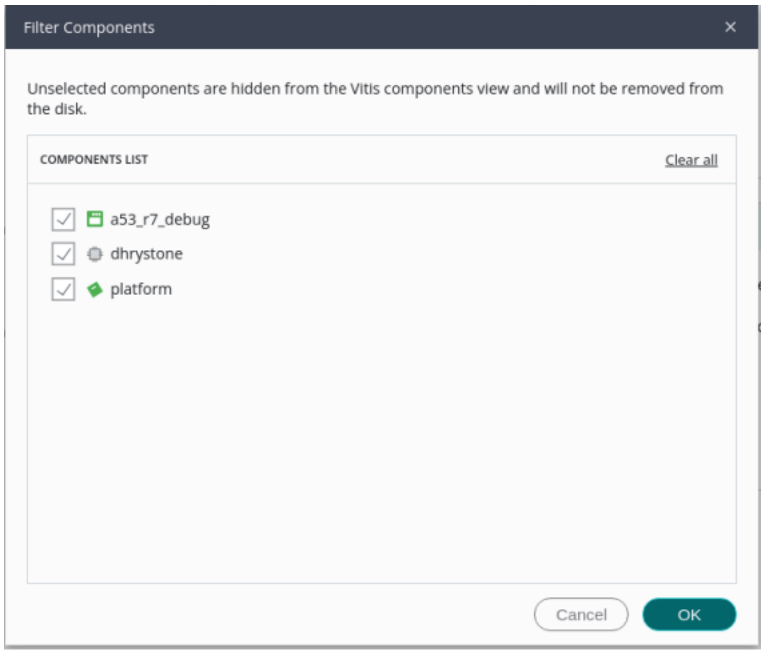


VITIS EXPLORER
DEMO_WORKSPACE - VITIS COMPONENTS
a72_app [Application] [standalone_psv_cortexa72_0]
Settings
vitis-comp.json
CMakeLists.txt
UserConfig.cmake
launch.json
Includes
Sources
Output
platform [Platform]

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

- **Using the toolbar menu, you can do the following::**
  - Create a New Component.

- Select Filter Components to hide or show components.



Unselected components are hidden from the Vitis components view and will not be removed from the disk.

- Refresh the Component Explorer.
- Collapse All to minimize the displayed contents.

- **Using the right-click menu to select a component or project, you can do the following::**
  - Select **Open in Terminal** to open a terminal window and change directory to inside the component.
  - Select **Show in Flow Navigator** and click the component to show it in the Flow Navigator.
  - Select **Delete** to remove the component or project from the workspace.
  - Select **Clone Component** to create a new component from an existing component. This is useful for design exploration where you need to preserve the original component as your baseline. You can then use the cloned component for design exploration. This command does not support System projects.
  - Select **Reset Linker Script** to generate a linker script for a standalone application component.
- **By right-clicking the `Sources` folder of a component or project, you can do the following::**
  - Select **New File** to create a new file in the component or project.
  - Select **New Folder** to create a new folder in the component or project.
  - Select **Open in Terminal** to open a terminal window and change directory to inside the component.
  - Select **Paste** to take a copied item and paste it into the component or project.
  - Select **Import→Files** to import files into the component or project.

- Select **Import→Folders** to import a folder into the component or project.
- Select **Add Source File** or create a New Source File.

- **By right-clicking an object in the component or project hierarchy, you can do the following::**
    - Select **Copy** followed by **Paste** to copy an object from one component or project to another.
    - Select **Copy Path** to copy the absolute path of the object. You can paste the path into the Terminal view of a configuration file.
    - Select **Copy Relative Path** to copy the path of the object relative to the current workspace. You can paste the path into the Terminal view of a configuration file.
    - Select **Delete** to remove the object from the workspace.

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. Actions include 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.

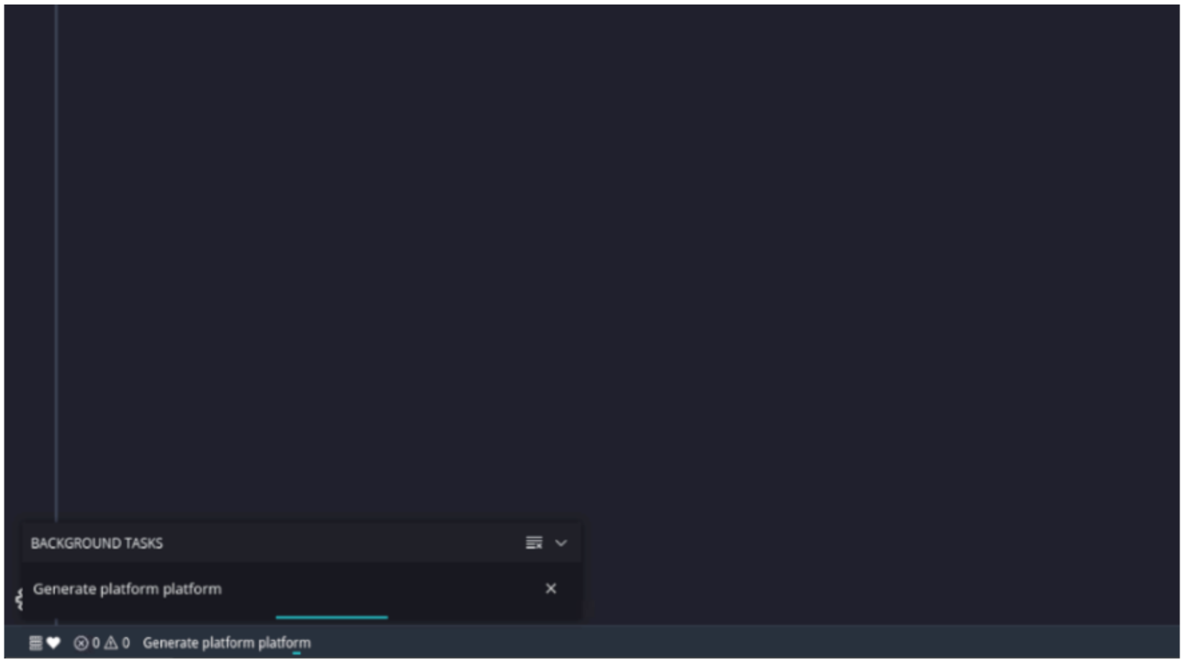
***Note*:** The system displays notifications about unsupported components.

Terminating a Backend Process

In some situations, backend processes can take too long without producing any results or can encounter issues. In these situations, it can be useful to terminate the process that is stuck.

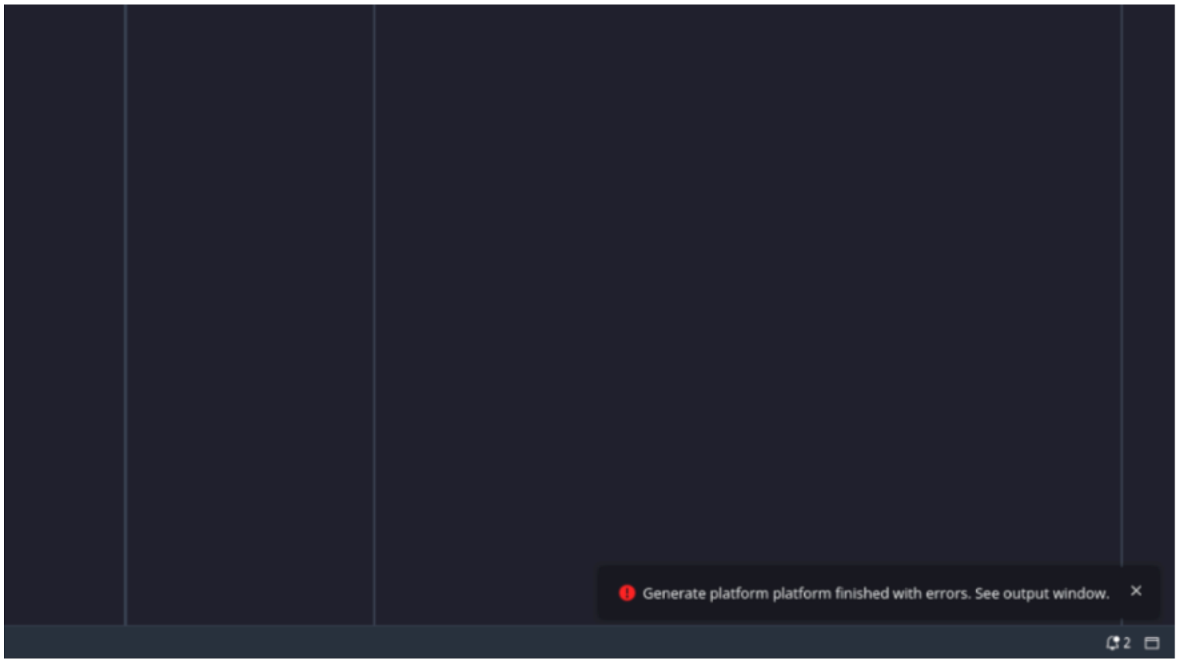
There is a progress bar at the bottom left corner of the Vitis Unified IDE that displays the progress of the current process running. To terminate a backend process, first locate the process that is running on the bottom left section of the Vitis Unified IDE.

Figure 7: **Background Task Process Bar**



Click the 'x' button in the background task bar to terminate the task

Figure 8: **Message Showing Process Cancellation or Failure**



Generate platform platform finished with errors. See output window.

There are a number of processes that can be terminated:

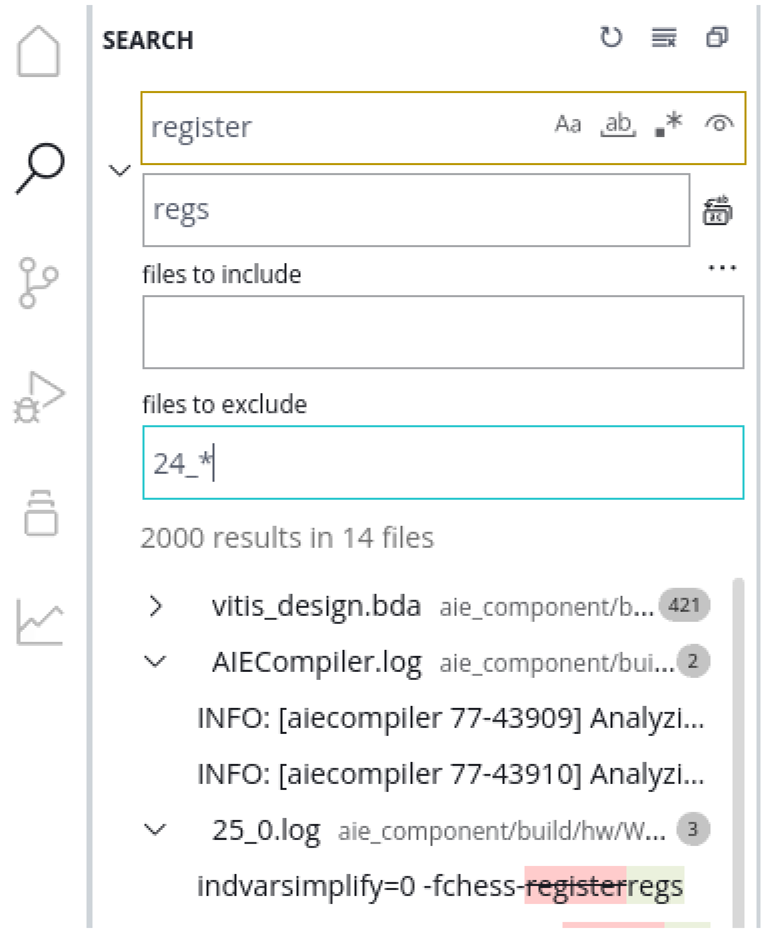
- Building a component

- Creating a component
- Emulation
- Flash programming
- Updating the BSP
- Run, debug and XSDB
- Programming a device

Search View

You can use the search view to find and replace text globally within the current workspace. The search result includes text files in the workspace. These files include source files, configuration files, and log files. The system performs the search inside files. File names are not part of the search.

Figure 9: **Search View**



2000 results in 14 files

```
vitis_design.bda  aie_component/b...  421
AIECompiler.log  aie_component/bui...  2
INFO: [aiecompiler 77-43909] Analyzi...
INFO: [aiecompiler 77-43910] Analyzi...
25_0.log  aie_component/build/hw/W...  3
indvarsimplify=0 -fchess-registerregs
```

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

- **Match Case:** Matches the case of the provided search string.
- **Match Whole Word:** Matches only whole words.
- **Use Regular Expression:** Defines the search using regular expressions.
- **Include Ignored Files:** Restores ignored files to the search list.
- **Replace All:** Replaces all search results.
- **Toggle Search Details:** Expands the Search view to add Files to Include and Files to Exclude fields.
- **Files to include:** Specifies 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:** Specifies 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, because it can introduce errors into your designs.*

Source Control

Software developers widely use source and version control. This chapter describes how to use Git integration in the Vitis Unified IDE.

Cross Operating System Support

The Vitis Unified IDE supports cross OS scenarios were multiple teams can be working on different operating systems. A Vitis workspace can be moved from a Linux OS to a Windows OS and vice versa. You can open projects or components seamlessly across different operating systems and rebuild them without failures.

There are a few recommended ways of distributing a Vitis workspace between operating systems:

1. Version Control: Push the project to a shared GitHub repo and clone it down into the other OS.

2. Export/Import: Export the workspace as a '.zip' file. Open the Vitis Unified IDE in the other OS and import the `.zip` file.

See previous chapters on git usage and export/import functionality for more specific guidelines.

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 committing 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: Ignore the '_ide' folders. When you load a new workspace in the Vitis Unified IDE, the system automatically generates these folders.

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. You can use it to manage compatibility across different IDE versions. Commit this file in 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, do not commit this file to source control.

Note: The `.gitignore` file specifies the files and directories that Git ignores in a project. Commit this file to source control. 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. It is not necessary to commit this file to 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. You can regenerate this file by clicking **Switch/Re-read XSA File** in the Platform Settings page. It is not necessary to commit this file 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
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, use the `-f` (force) option with Git:

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

Note: The `compile_commands.json` file contains the compiler commands used to build the application source files. The system automatically generates this file when the build process starts. You do not need to commit this file to source control.

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

Enabling the Source Control View

To enable the Source Control view:

- Initialize your empty workspace as a Git repository.
- Create a workspace.
- Launch the Vitis Unified IDE to open the workspace.
- Add the workspace to your Git repository and perform the following steps.

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

A message similar to `Initialized empty Git repository in /tests/temp/workVADD/.git/` indicates the end of the process.

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

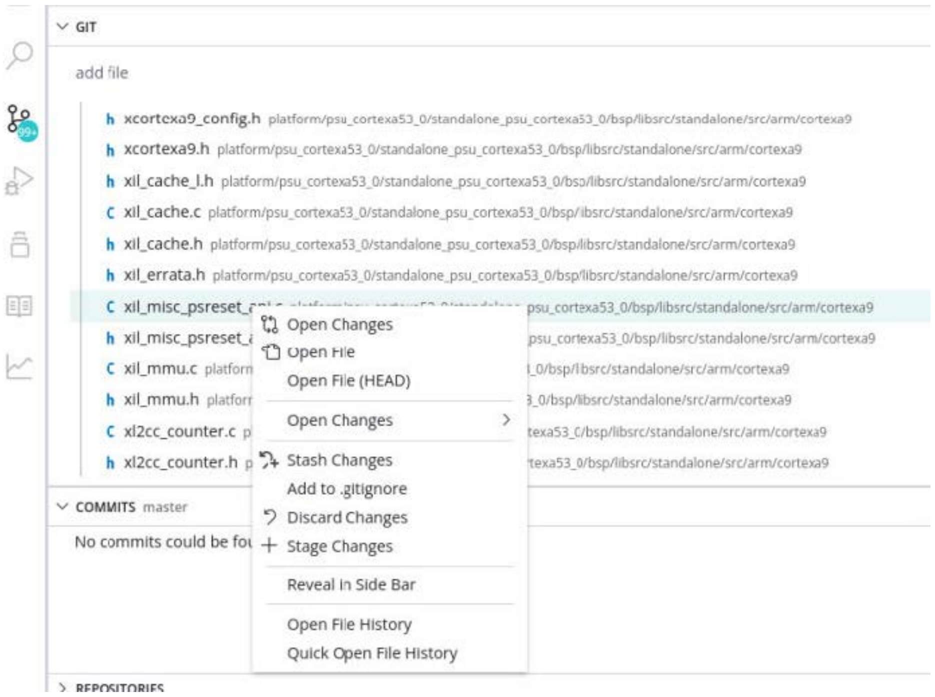
After you create a new component or project in the workspace, Vitis Unified IDE generates a `.gitignore` file. The `.gitignore` file can help you filter out the generated files so that it is easier to pick the files for source control. You can open the `.gitignore` file and edit this file if you have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

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 10: **Add New File for Source Control**

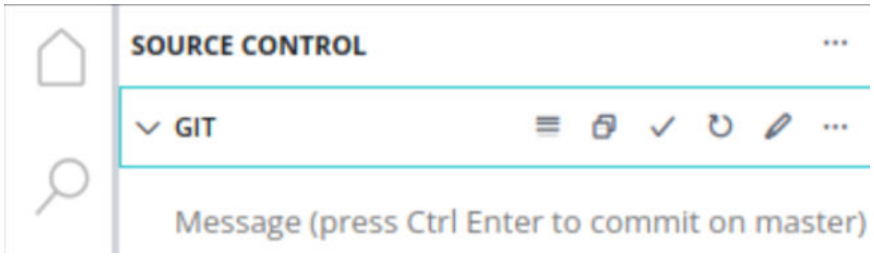

GIT
add file
xcortexa9_config.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xcortexa9.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xil_cache_l.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xil_cache.c platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xil_cache.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xil_errata.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp/libsrc/standalone/src/arm/cortexa9
xil_misc_psreset_
xil_misc_psreset_
xil_mmu.c
xil_mmu.h
xl2cc_counter.c
xl2cc_counter.h
COMMITS master
No commits could be fou
REPOSITORIES
Open Changes
Open File
Open File (HEAD)
Open Changes
Stash Changes
Add to .gitignore
Discard Changes
Stage Changes
Reveal in Side Bar
Open File History
Quick Open File History

This GUI is equivalent to the following git command.

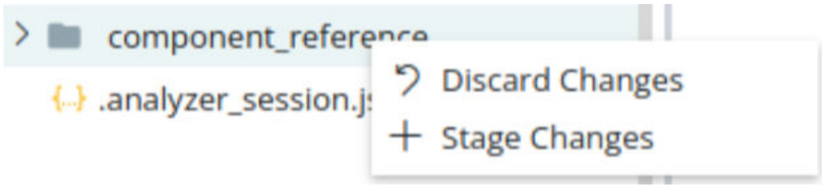
```
git add <file name>
```

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 perform source control.



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

   This GUI functionality is equivalent to the following git command:

```
git add platform/
```

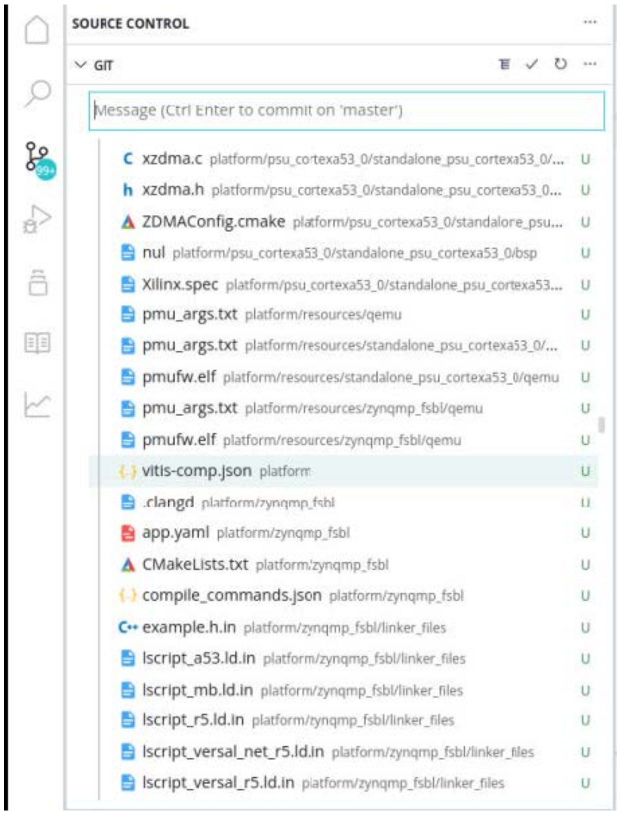
   Note: You need to select the components bound with the system project if you want to add system project for source control.

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 11: **Git View**



Message (Ctrl Enter to commit on 'master')

```
xzdma.c platform/psu_cortexa53_0/standalone_psu_cortexa53_0/...  U
xzdma.h platform/psu_cortexa53_0/standalone_psu_cortexa53_0...  U
ZDMAConfig.cmake platform/psu_cortexa53_0/standalore_psu...  U
nul platform/psu_cortexa53_0/standalone_psu_cortexa53_0/bsp  U
Xilinx.spec platform/psu_cortexa53_0/standalone_psu_cortexa53...  U
pmu_args.txt platform/resources/qemu  U
pmu_args.txt platform/resources/standalone_psu_cortexa53_0/...  U
pmufw.elf platform/resources/standalone_psu_cortexa53_0/qemu  U
pmu_args.txt platform/resources/zynqmp_fsbl/qemu  U
pmufw.elf platform/resources/zynqmp_fsbl/qemu  U
vitis-comp.json platform  U
.clangd platform/zynqmp_fsbl  U
app.yaml platform/zynqmp_fsbl  U
CMakeLists.txt platform/zynqmp_fsbl  U
compile_commands.json platform/zynqmp_fsbl  U
example.h.in platform/zynqmp_fsbl/linker_files  U
lscript_a53.ld.in platform/zynqmp_fsbl/linker_files  U
lscript_mb.ld.in platform/zynqmp_fsbl/linker_files  U
lscript_r5.ld.in platform/zynqmp_fsbl/linker_files  U
lscript_versal_net_r5.ld.in platform/zynqmp_fsbl/linker_files  U
lscript_versal_r5.ld.in platform/zynqmp_fsbl/linker_files  U
```

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.

Relative Path Support

If the imported files share the same initial node in the path as the workspace, the Vitis tool automatically selects the relative path. The following is an example:

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

The system selects this relative path:

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

Note: When moving the workspace or committing it to source control, remember to keep the relative path relationship.

Source Control for Platform BSP

To customize and version-control the platform BSP, you can keep a local copy of the BSP source repository. Modify the BSP source as needed, then create or regenerate your platform BSP based on the updated local source. Follow the steps below.

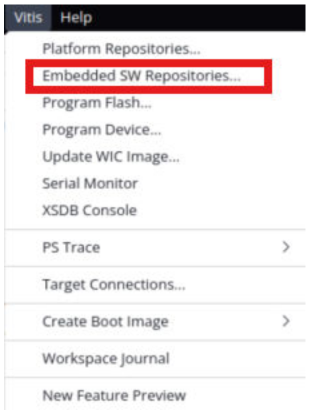
1. Download the embedded BSP repository.

   ```
   git clone https://github.com/Xilinx/embeddedsw.git
   ```

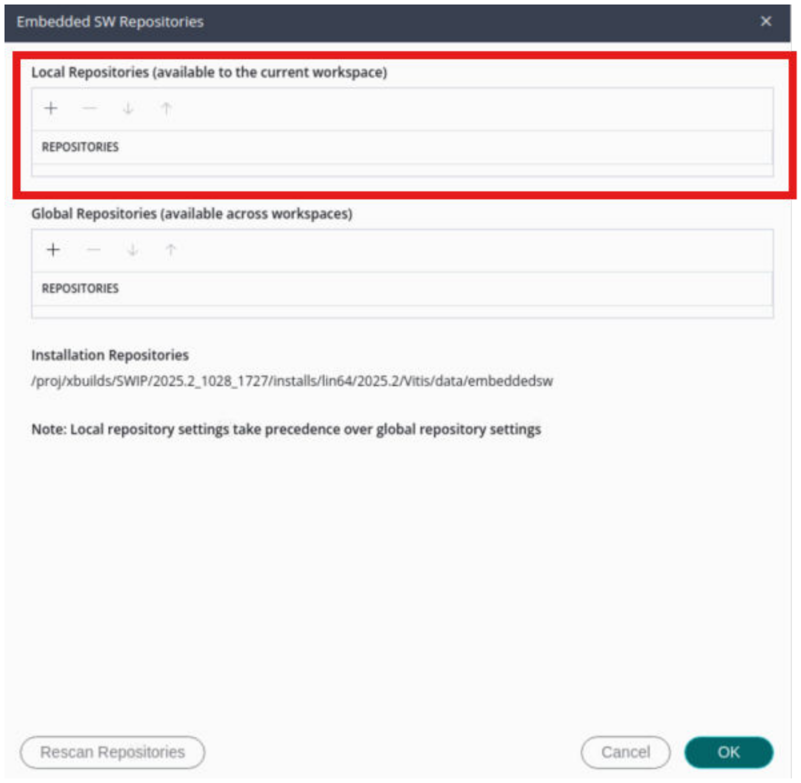
2. Switch the version.

   ```
   cd embeddedsw & git checkout xilinx_v2025.2 # select the version you
   need accordingly.
   ```

3. Launch the Vitis Unified IDE and go to the **Vitis** menu and select **Embedded SW Repositories**.

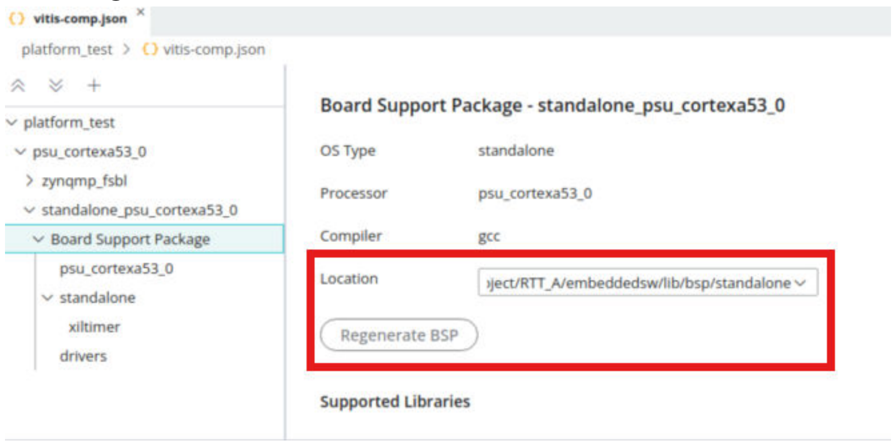
4. Click + to add a local repository, then browse to and select the downloaded embedded BSP repositories.


Embedded SW Repositories
Local Repositories (available to the current workspace)
REPOSITORIES
Global Repositories (available across workspaces)
REPOSITORIES
Rescan Repositories
Cancel
OK

Installation Repositories
/proj/xbuilds/SWIP/2025.2_1028_1727/installs/lin64/2025.2/Vitis/data/embeddedsw

Note: Local repository settings take precedence over global repository settings

5. Click **Rescan Repositories** to ensure the repository is detected by IDE.
6. Click **OK**.
7. Select the BSP you need to use when creating the platform
   a. Open the Platform Component JSON file and select **Board Support Package** in the left panel.
   b. Click the drop-down arrow beside the Location box and select the option which reflect the local BSP repositories.
   c. Click **Regenerate BSP**.



Board Support Package - standalone_psu_cortexa53_0

OS Type standalone
Processor psu_cortexa53_0
Compiler gcc
Location ›ject/RTT_A/embeddedsw/lib/bsp/standalone

In this case, you can create the platform BSP from your local repositories, allowing you to update and manage it under source control easily. There's no need to maintain or version-control the BSP within the platform directory itself.

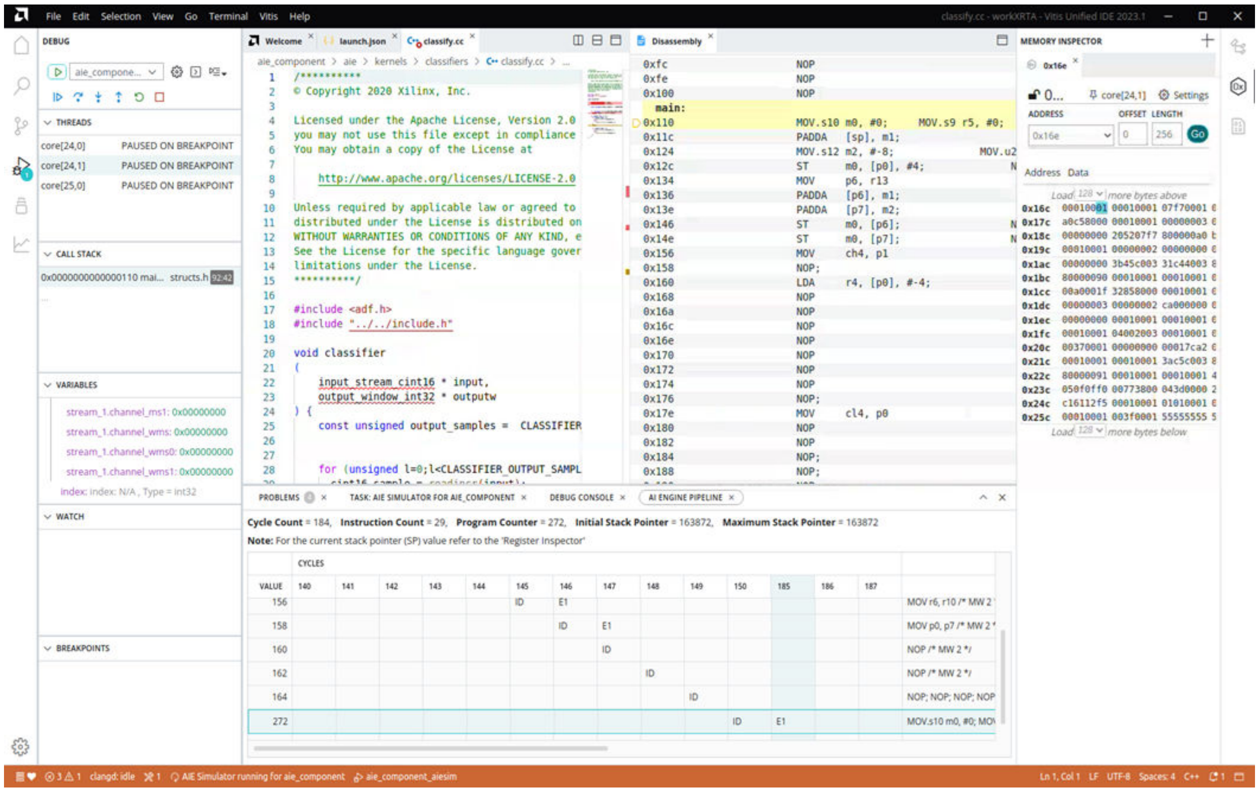
Debug View

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. In the debug environment, you can do the following:

- Add break points to the code.
- Step over or step into specific lines of code, loops, or functions.
- Examine the state of variables and force them to desired values.

The AI Engine Debug view contains several windows or views as shown in the following figure.

Figure 12: **AIE Component Debug View**



```
core[24,0]     PAUSED ON BREAKPOINT
core[24,1]     PAUSED ON BREAKPOINT
core[25,0]     PAUSED ON BREAKPOINT
```

```
0x0000000000000110 mai...  structs.h 92:42
```

```
stream_1.channel_ms1: 0x00000000
stream_1.channel_wms: 0x00000000
stream_1.channel_wms0: 0x00000000
stream_1.channel_wms1: 0x00000000
index: index: N/A , Type = int32
```

```
/**********
© Copyright 2020 Xilinx, Inc.

Licensed under the Apache License, Version 2.0
you may not use this file except in compliance
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to
distributed under the License is distributed on
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, e
See the License for the specific language gover
limitations under the License.
**********/

#include <adf.h>
#include "../../include.h"

void classifier
(
    input_stream_cint16 * input,
    output_window_int32 * outputw
) {
    const unsigned output_samples =  CLASSIFIER


    for (unsigned l=0;l<CLASSIFIER_OUTPUT_SAMPL
```

```
0xfc          NOP
0xfe          NOP
0x100         NOP
 main:
0x110         MOV.s10 m0, #0;     MOV.s9 r5, #0;
0x11c         PADDA   [sp], m1;
0x124         MOV.s12 m2, #-8;                  MOV.u2
0x12c         ST      m0, [p0], #4;                  N
0x134         MOV     p6, r13
0x136         PADDA   [p6], m1;
0x13e         PADDA   [p7], m2;
0x146         ST      m0, [p6];                      N
0x14e         ST      m0, [p7];                      N
0x156         MOV     ch4, p1
0x158         NOP;
0x160         LDA     r4, [p0], #-4;
0x168         NOP
0x16a         NOP
0x16c         NOP
0x16e         NOP
0x170         NOP
0x172         NOP
0x174         NOP
0x176         NOP;
0x17e         MOV     cl4, p0
0x180         NOP
0x182         NOP
0x184         NOP;
0x188         NOP;
```

```
0x16e    0    256
Load 128 more bytes above
0x16c  00010001 00010001 07f70001 0
0x17c  a0c58000 00010001 00000003 0
0x18c  00000000 205207f7 800000a0 b
0x19c  00010001 00000002 00000000 0
0x1ac  00000000 3b45c003 31c44003 8
0x1bc  80000090 00010001 00010001 0
0x1cc  00a0001f 32858000 00010001 0
0x1dc  00000003 00000002 ca000000 0
0x1ec  00000000 00010001 00010001 0
0x1fc  00010001 04002003 00010001 0
0x20c  00370001 00000000 00017ca2 0
0x21c  00010001 00010001 3ac5c003 8
0x22c  80000091 00010001 00010001 4
0x23c  050f0ff0 00773800 043d0000 2
0x24c  c16112f5 00010001 01010001 0
0x25c  00010001 003f0001 55555555 5
Load 128 more bytes below
```

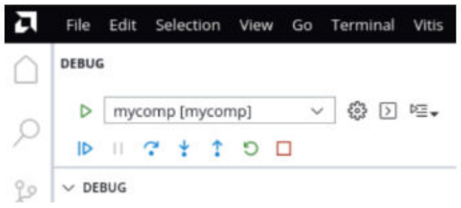
Cycle Count = 184, **Instruction Count** = 29, **Program Counter** = 272, **Initial Stack Pointer** = 163872, **Maximum Stack Pointer** = 163872

Note: For the current stack pointer (SP) value refer to the 'Register Inspector'

	CYCLES													
VALUE	140	141	142	143	144	145	146	147	148	149	150	185	186	187	
156						ID	E1								MOV r6, r10 /* MW 2
158							ID	E1							MOV p0, p7 /* MW 2 *
160								ID							NOP /* MW 2 */
162									ID						NOP /* MW 2 */
164										ID					NOP; NOP; NOP; NOP
272											ID	E1			MOV.s10 m0, #0; MOV

- **Control Panel:** The Debug view's Control Panel is at the upper-left corner of the screen. 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 13: **Debug Control Panel**



- **Threads:** Threads shows the related debugging threads. The debugging process creates and destroys threads. You can switch between multiple threads.
- **Call Stack:** Call Stack shows the function call stack updating when the application is running.
- **Variables:** Variables shows the current value of global and local variables. The variable information updates when switching threads.
- **Watch:** Watch shows variables and expressions you have specified to watch. To add watch points select **Add Expression** (+).
- **Breakpoints:**
  - The system sets break points at the host component's **main** function if it is debuggable.
  - The system sets break points at the top function of the PL kernels if they are debuggable.
  - To add breakpoints, open the source file and click the left side of the line number when a red dot appears.
  - To remove breakpoints, click on a previously added breakpoint.
  - To add conditional breakpoints, right-click when the red dot appears and select **Add Conditional Breakpoint**. To log a message, select **Add Logpoint** when reaching the breakpoint.
- **Source Code Editor:** The Source Code view opens after launching Debug from Flow Navigator or Launch Configuration.
- **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. This view shows the following:
  - The registers of the Cortex-A72 at a triggered breakpoint in the Application Component source code.
  - The registers of the AI Engine at a triggered breakpoint in the AI Engine kernel.
- **Disassembly View:** You can open the Disassembly view from the Source Code window right-click menu.

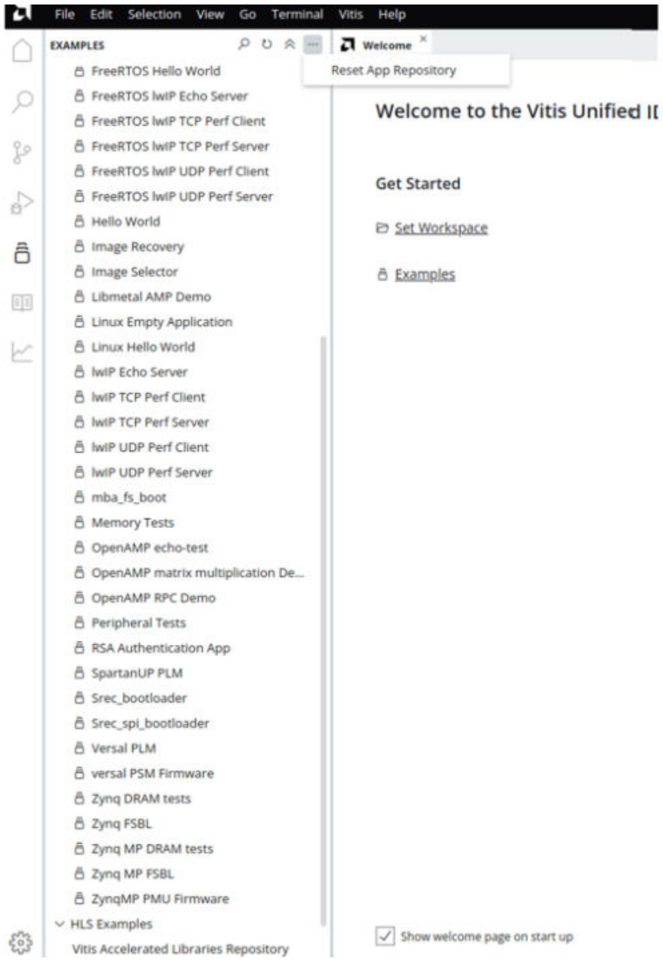
- **Debug Console:** This view displays the transcript of the debug process and messages received from the application.

Note: You can see all the prints from MDM (MicroBlaze/ V Debug Module) UART in the 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 14: **Examples View**



FreeRTOS Hello World
FreeRTOS lwIP Echo Server
FreeRTOS lwIP TCP Perf Client
FreeRTOS lwIP TCP Perf Server
FreeRTOS lwIP UDP Perf Client
FreeRTOS lwIP UDP Perf Server
Hello World
Image Recovery
Image Selector
Libmetal AMP Demo
Linux Empty Application
Linux Hello World
lwIP Echo Server
lwIP TCP Perf Client
lwIP TCP Perf Server
lwIP UDP Perf Client
lwIP UDP Perf Server
mba_fs_boot
Memory Tests
OpenAMP echo-test
OpenAMP matrix multiplication De...
OpenAMP RPC Demo
Peripheral Tests
RSA Authentication App
SpartanUP PLM
Srec_bootloader
Srec_spi_bootloader
Versal PLM
versal PSM Firmware
Zynq DRAM tests
Zynq FSBL
Zynq MP DRAM tests
Zynq MP FSBL
ZynqMP PMU Firmware
HLS Examples
Vitis Accelerated Libraries Repository

Welcome to the Vitis Unified I

Get Started

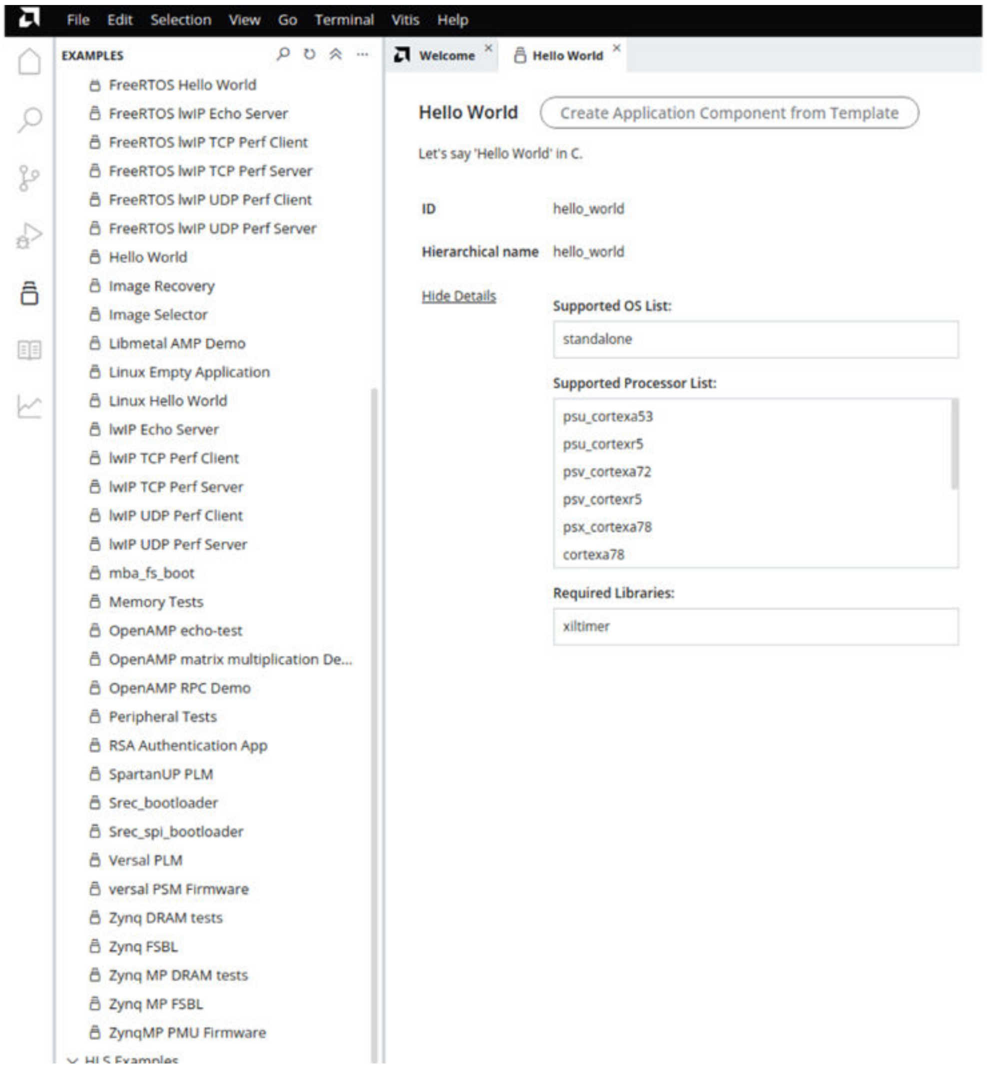
Set Workspace

Examples

✓ Show welcome page on start up

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 15: **Example View Supported Devices**



EXAMPLES

- FreeRTOS Hello World
- FreeRTOS lwIP Echo Server
- FreeRTOS lwIP TCP Perf Client
- FreeRTOS lwIP TCP Perf Server
- FreeRTOS lwIP UDP Perf Client
- FreeRTOS lwIP UDP Perf Server
- Hello World
- Image Recovery
- Image Selector
- Libmetal AMP Demo
- Linux Empty Application
- Linux Hello World
- lwIP Echo Server
- lwIP TCP Perf Client
- lwIP TCP Perf Server
- lwIP UDP Perf Client
- lwIP UDP Perf Server
- mba_fs_boot
- Memory Tests
- OpenAMP echo-test
- OpenAMP matrix multiplication De...
- OpenAMP RPC Demo
- Peripheral Tests
- RSA Authentication App
- SpartanUP PLM
- Srec_bootloader
- Srec_spi_bootloader
- Versal PLM
- versal PSM Firmware
- Zynq DRAM tests
- Zynq FSBL
- Zynq MP DRAM tests
- Zynq MP FSBL
- ZynqMP PMU Firmware

HLS Examples

Hello World

Let's say 'Hello World' in C.

ID hello_world

Hierarchical name hello_world

Supported OS List:
standalone

Supported Processor List:
psu_cortexa53
psu_cortexr5
psv_cortexa72
psv_cortexr5
psx_cortexa78
cortexa78

Required Libraries:
xiltimer

Code View and Smart Editor

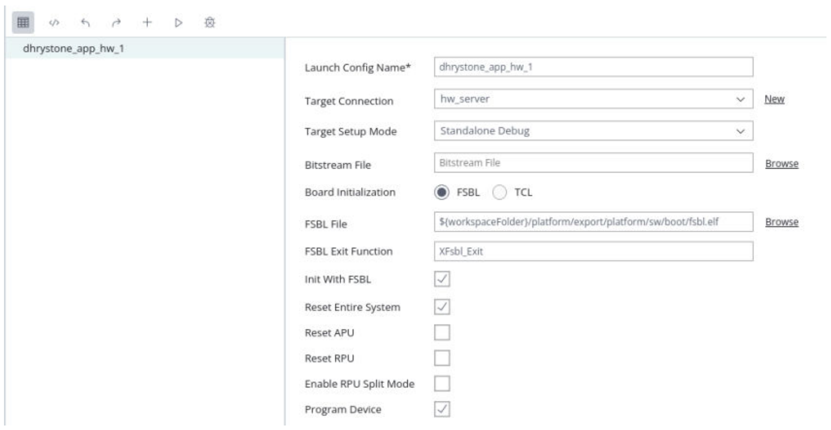
The Source Code editor in the Vitis Unified IDE supports the following features:

- Syntax highlight for C, C++, Python, Makefile, CMakeList.txt file
- Hint for variable names, function names
- Jumping to the definition of variables or functions
- Peeking the definition of variables or functions so you do not to leave the editing file
- Reporting 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 lists 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. Use the table button and code button to switch between views. 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.

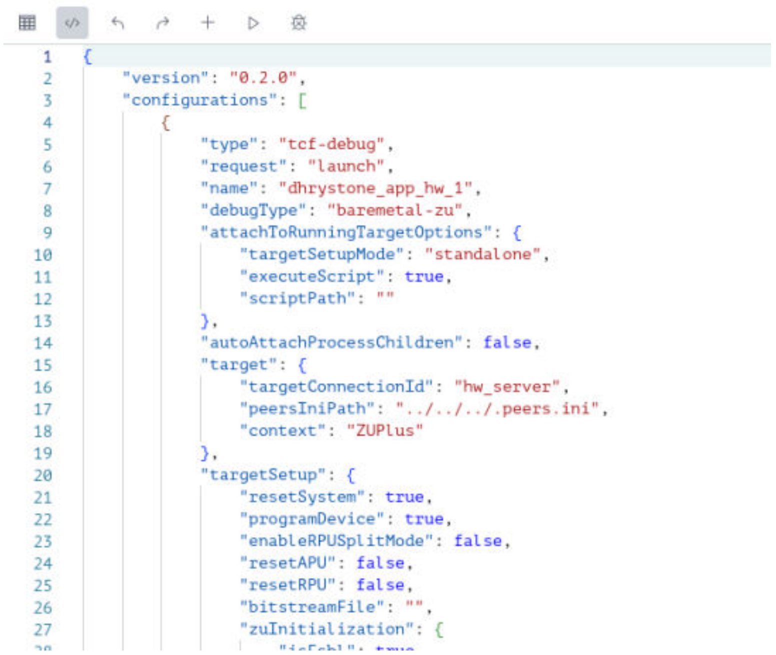
Figure 16: **GUI Format**



dhrystone_app_hw_1

Launch Config Name*	dhrystone_app_hw_1	
Target Connection	hw_server	New
Target Setup Mode	Standalone Debug	
Bitstream File	Bitstream File	Browse
Board Initialization	[✓ FSBL] TCL	
FSBL File	${workspaceFolder}/platform/export/platform/sw/boot/fsbl.elf	Browse
FSBL Exit Function	XFsbl_Exit	
Init With FSBL	✓	
Reset Entire System	✓	
Reset APU	☐	
Reset RPU	☐	
Enable RPU Split Mode	☐	
Program Device	✓	

Figure 17: **Text Format**



```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "tcf-debug",
            "request": "launch",
            "name": "dhrystone_app_hw_1",
            "debugType": "baremetal-zu",
            "attachToRunningTargetOptions": {
                "targetSetupMode": "standalone",
                "executeScript": true,
                "scriptPath": ""
            },
            "autoAttachProcessChildren": false,
            "target": {
                "targetConnectionId": "hw_server",
                "peersIniPath": "../../../.peers.ini",
                "context": "ZUPlus"
            },
            "targetSetup": {
                "resetSystem": true,
                "programDevice": true,
                "enableRPUSplitMode": false,
                "resetAPU": false,
                "resetRPU": false,
                "bitstreamFile": "",
                "zuInitialization": {
                    "isFsbl": true,
```

The system saves the changes automatically when you enable Auto Save. Optionally, you can manually save changes in a file with the keyboard shortcut `Ctrl + S`.

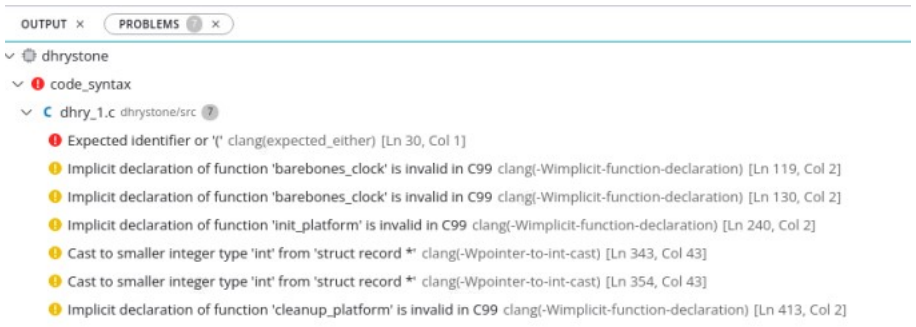
Issue and Source Code Cross-probing

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

Problem View

The output channel displays all warning and error messages. You can quickly navigate to the source of the error or warning in problem pane by clicking the error or warning message.

Figure 18: **Problem View**



```
dhrystone
  code_syntax
    dhry_1.c dhrystone/src 7
      Expected identifier or '(' clang(expected_either) [Ln 30, Col 1]
      Implicit declaration of function 'barebones_clock' is invalid in C99 clang(-Wimplicit-function-declaration) [Ln 119, Col 2]
      Implicit declaration of function 'barebones_clock' is invalid in C99 clang(-Wimplicit-function-declaration) [Ln 130, Col 2]
      Implicit declaration of function 'init_platform' is invalid in C99 clang(-Wimplicit-function-declaration) [Ln 240, Col 2]
      Cast to smaller integer type 'int' from 'struct record *' clang(-Wpointer-to-int-cast) [Ln 343, Col 43]
      Cast to smaller integer type 'int' from 'struct record *' clang(-Wpointer-to-int-cast) [Ln 354, Col 43]
      Implicit declaration of function 'cleanup_platform' is invalid in C99 clang(-Wimplicit-function-declaration) [Ln 413, Col 2]
```

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*:** Expect a slower response if you launch multiple build or emulation jobs at the same time on a slow server.

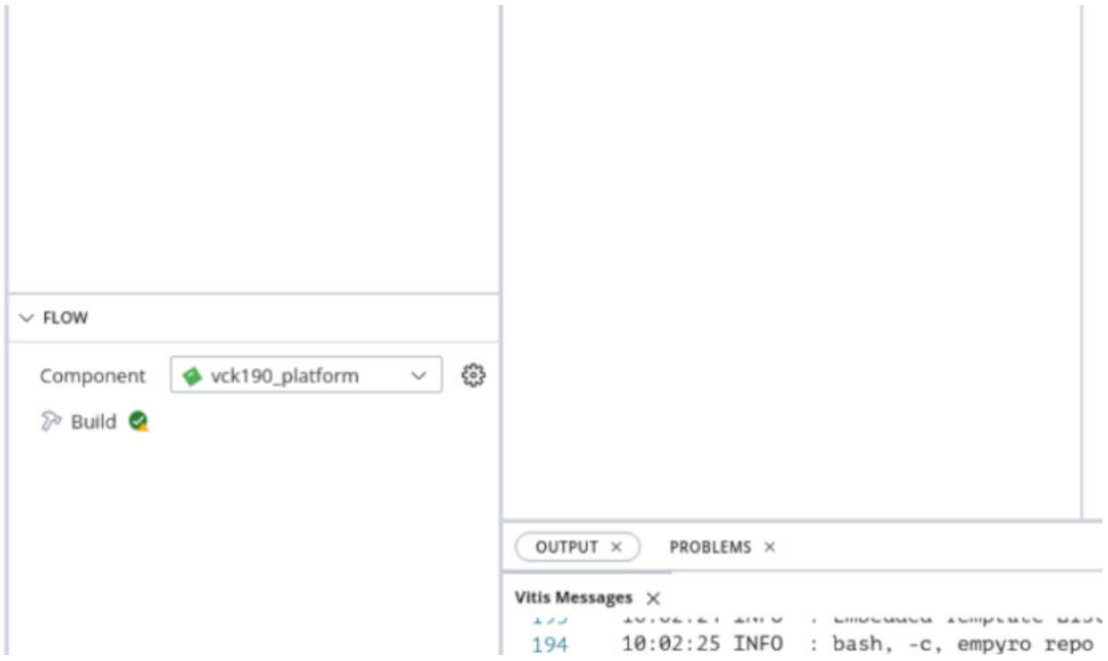
The application building job can launch multiple components in parallel. You can build and launch jobs for the referenced components in parallel.

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

Notification for File Change

When you modify files, the yellow indicator beside the build status icon prompts you to review and rebuild the component if necessary.

Figure 19: **Out-of-date Notification for Components**

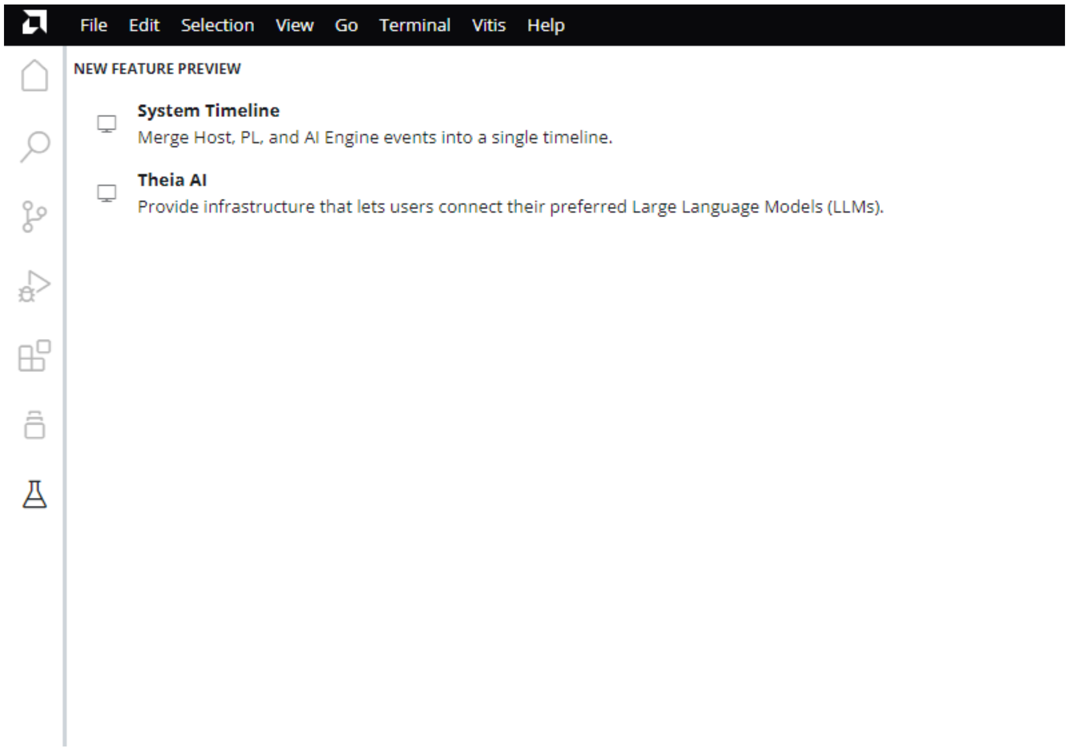


```
194     10:02:25 INFO  : bash, -c, empyro repo
```

New Feature Preview

New features are for you to explore in advance. It is possible these features do not address all use cases and require further development. You are encouraged to share your feedback. You can view the new features by clicking **Vitis → New Feature Preview**.

Figure 20: **New Feature Preview**



NEW FEATURE PREVIEW

System Timeline
Merge Host, PL, and AI Engine events into a single timeline.

Theia AI
Provide infrastructure that lets users connect their preferred Large Language Models (LLMs).

Before using a new feature, you need to enable it.

Disclaimer: preview features are currently in an early access phase. They are currently being developed. There is a chance these features are not be fully functional or stable. By using these features, you acknowledge and accept the following:

- Limited functionality: Early access features can have limited functionality compared to fully released features. Early access features can lack functions, have bugs, or experience performance issues.
- 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.
- 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 is not to 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.*

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 the source and destination paths in `workspace_journal.py` are properly configured and accessible.

External Lopper Support

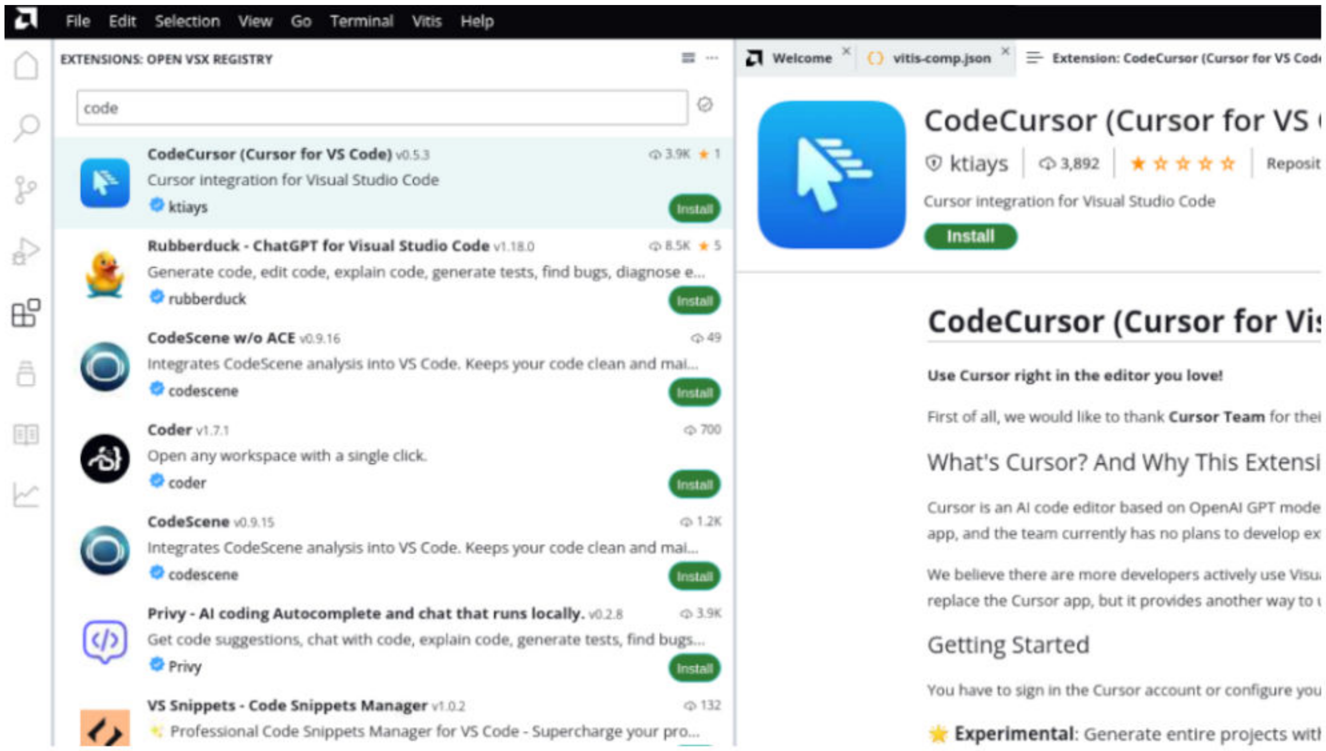
Lopper is a backend Python utility used by AMD Vitis™ to extract hardware metadata from the system device tree. Although included in the Vitis installation, Lopper is also available on GitHub. Vitis provides an environment variable to enable the use of an external Lopper without modifying the Vitis installation. 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.

Figure 21: **Extensions**



CodeCursor (Cursor for VS Code) v0.5.3 — 3.9K ★ 1
Cursor integration for Visual Studio Code
ktiays

Rubberduck - ChatGPT for Visual Studio Code v1.18.0 — 8.5K ★ 5
Generate code, edit code, explain code, generate tests, find bugs, diagnose e...
rubberduck

CodeScene w/o ACE v0.9.16 — 49
Integrates CodeScene analysis into VS Code. Keeps your code clean and mai...
codescene

Coder v1.7.1 — 700
Open any workspace with a single click.
coder

CodeScene v0.9.15 — 1.2K
Integrates CodeScene analysis into VS Code. Keeps your code clean and mai...
codescene

Privy - AI coding Autocomplete and chat that runs locally. v0.2.8 — 3.9K
Get code suggestions, chat with code, explain code, generate tests, find bugs...
Privy

VS Snippets - Code Snippets Manager v1.0.2 — 132
Professional Code Snippets Manager for VS Code - Supercharge your pro...

CodeCursor (Cursor for VS
ktiays | 3,892 | ★★★★★
Cursor integration for Visual Studio Code

CodeCursor (Cursor for Vi

Use Cursor right in the editor you love!

First of all, we would like to thank **Cursor Team** for thei

What's Cursor? And Why This Extensi

Cursor is an AI code editor based on OpenAI GPT mode
app, and the team currently has no plans to develop ex

We believe there are more developers actively use Visu
replace the Cursor app, but it provides another way to

Getting Started

You have to sign in the Cursor account or configure you

Experimental: Generate entire projects wit

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. The Eclipse Foundation offers this extension. AMD does 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 is possible to be unstable. By accessing and using this feature, you acknowledge and accept the following:

- Limited functionality: Early access features can have limited functionality compared to fully released features. Early access features can lack functions, have bugs, or experience performance issues.
- 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.
- 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 is not to 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.*

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**.

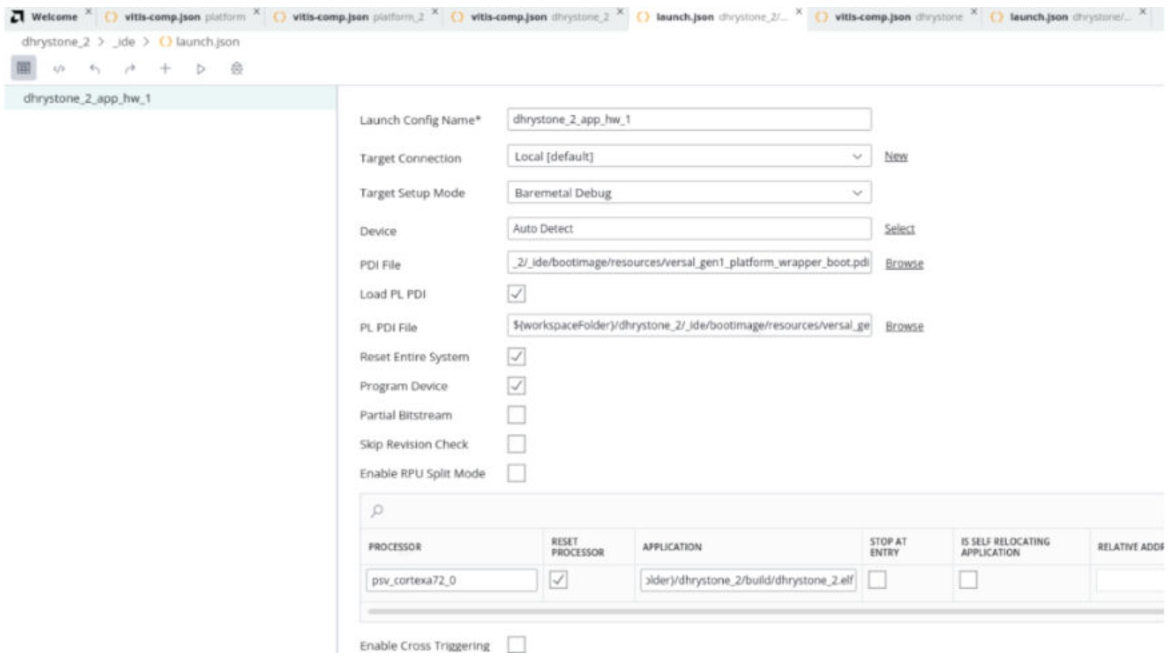
Segmented Configuration

The Segmented Configuration flow is a development process for AMD Versal™ devices 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 the following purposes:

- Booting software quickly
- Minimizing local boot flash requirements
- Loading the PL PDI image by runtime request

This GUI feature allows you to run or debug standalone applications if you configure your design with segmented configuration and select the PL PDI when editing launch configuration.

Figure 22: **Select PL PDI**


Welcome
vitis-comp.json platform
vitis-comp.json platform_2
vitis-comp.json dhrystone_2
launch.json dhrystone_2/...
vitis-comp.json dhrystone
launch.json dhrystone/...
dhrystone_2 > _ide > launch.json
dhrystone_2_app_hw_1

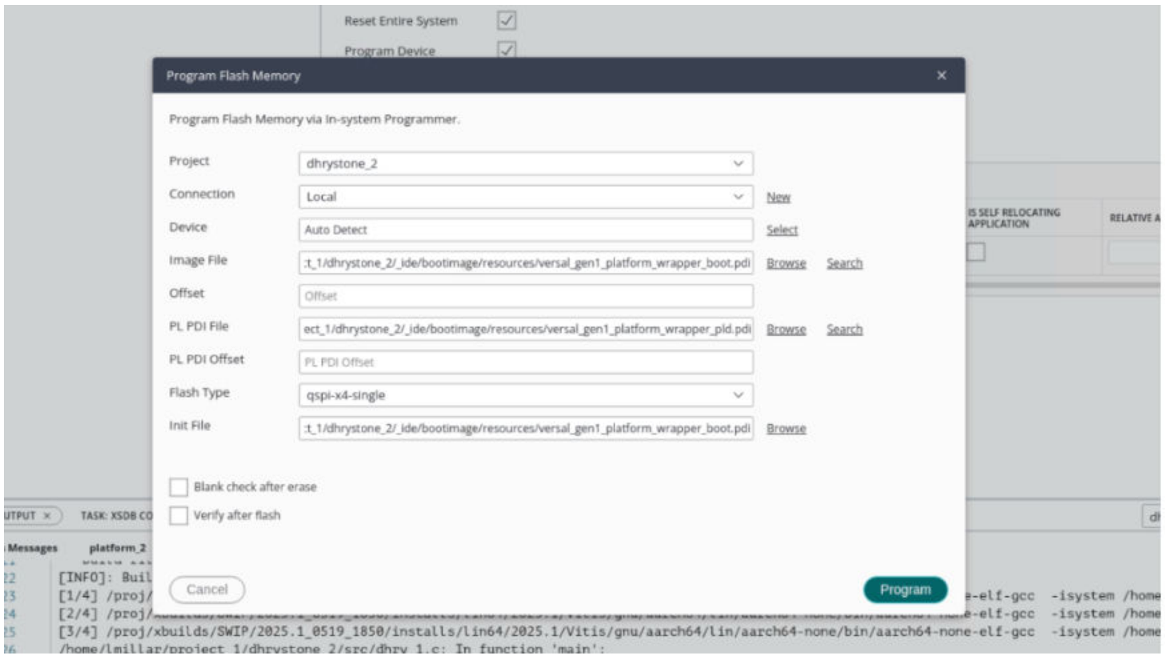
Launch Config Name*	dhrystone_2_app_hw_1	
Target Connection	Local [default]	New
Target Setup Mode	Baremetal Debug	
Device	Auto Detect	Select
PDI File	_2/_ide/bootimage/resources/versal_gen1_platform_wrapper_boot.pdi	Browse
Load PL PDI	✓	
PL PDI File	${workspaceFolder}/dhrystone_2/_ide/bootimage/resources/versal_ge	Browse
Reset Entire System	✓	
Program Device	✓	
Partial Bitstream	☐	
Skip Revision Check	☐	
Enable RPU Split Mode	☐	

PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELATIVE ADDR
psv_cortexa72_0	✓	older}/dhrystone_2/build/dhrystone_2.elf	☐	☐	

Enable Cross Triggering ☐

The Program Flash utility allows you to assign the PL PDI to program all contents to flash memory.

Figure 23: **Program Flash Utility**



Program Flash Memory via In-system Programmer.

Project	dhrystone_2		
Connection	Local	New	
Device	Auto Detect	Select	
Image File	:t_1/dhrystone_2/_ide/bootimage/resources/versal_gen1_platform_wrapper_boot.pdi	Browse	Search
Offset	Offset		
PL PDI File	ect_1/dhrystone_2/_ide/bootimage/resources/versal_gen1_platform_wrapper_pld.pdi	Browse	Search
PL PDI Offset	PL PDI Offset		
Flash Type	qspi-x4-single		
Init File	:t_1/dhrystone_2/_ide/bootimage/resources/versal_gen1_platform_wrapper_boot.pdi	Browse	

☐ Blank check after erase
☐ Verify after flash

```
[INFO]: Buil
[1/4] /proj/                                                                    e-elf-gcc  -isystem /home
[2/4] /proj/                                                                    e-elf-gcc  -isystem /home
[3/4] /proj/xbuilds/SWIP/2025.1_0519_1850/installs/lin64/2025.1/Vitis/gnu/aarch64/lin/aarch64-none/bin/aarch64-none-elf-gcc  -isystem /home
/home/lmillar/project_1/dhrystone_2/src/dhry_1.c: In function 'main':
```

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.

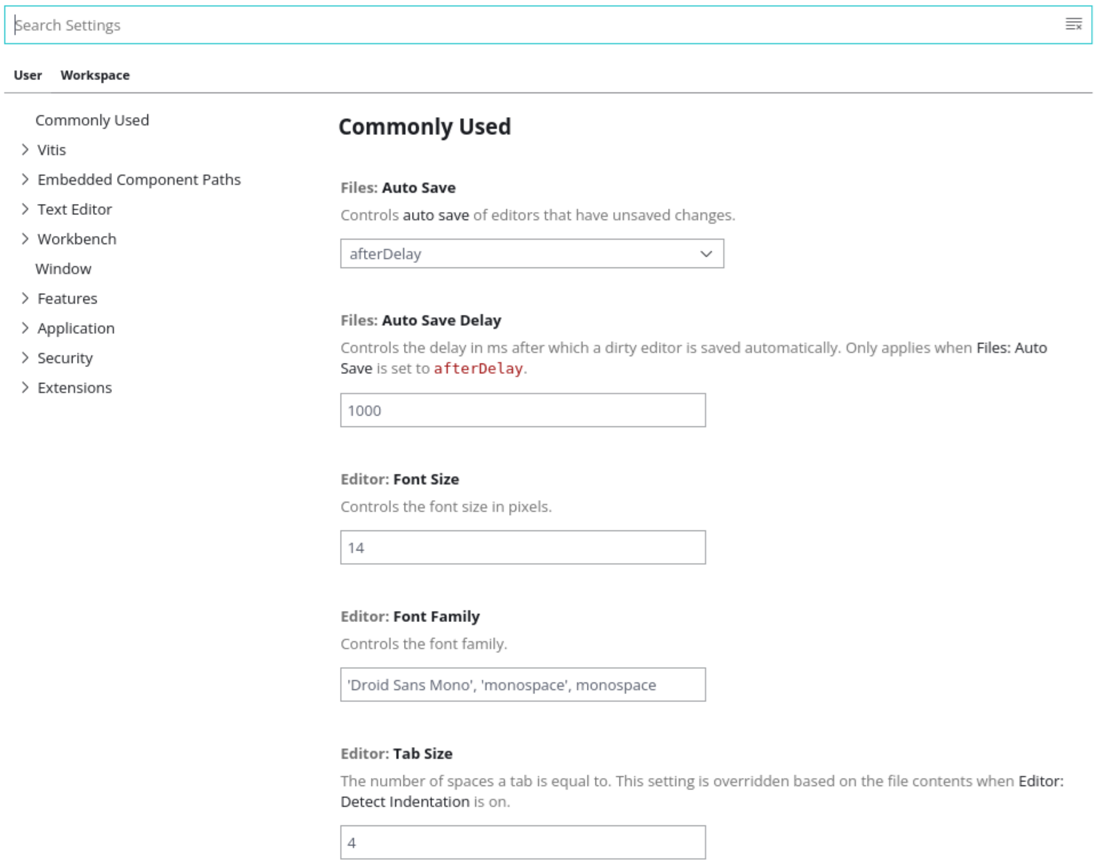
Preferences

The **File→Preferences** menu leads to user preferences that can be set and maintained to configure the Vitis Unified IDE.

Settings

- **Auto Save:** Automatically saves changes to configuration files, source code, `CMakeLists.txt` files, and more. This setting is enabled by default. You can also define Auto Save triggers and Auto Save Delay values.
- **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):** Opens the Preferences view, displaying an array of options to configure your design environment.

Figure 24: **Common Preferences**



Commonly Used

Files: Auto Save
Controls auto save of editors that have unsaved changes.
afterDelay

Files: Auto Save Delay
Controls the delay in ms after which a dirty editor is saved automatically. Only applies when Files: Auto Save is set to `afterDelay`.
1000

Editor: Font Size
Controls the font size in pixels.
14

Editor: Font Family
Controls the font family.
'Droid Sans Mono', 'monospace', monospace

Editor: Tab Size
The number of spaces a tab is equal to. This setting is overridden based on the file contents when Editor: Detect Indentation is on.
4

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

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

Keyboard Shortcuts, Command Palette, and Quick Find

Keyboard Shortcuts

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

1. Use the **Alt + Ctrl + Comma** shortcut to open the Keyboard Shortcuts window.
2. Type `build` in the search bar.
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.

Command Palette

The command palette lets you quickly find and execute commands without remembering 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 the keyword `build`, `run` or `debug`. For example, the Command Palette filters the command names for commands matching your entry.
4. Use the mouse or the up or down arrow keys to scroll through the selection of commands. 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 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, type `@` in the Quick Find box to go to a function within the opened code. You can also 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.

Chapter 6

Develop

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

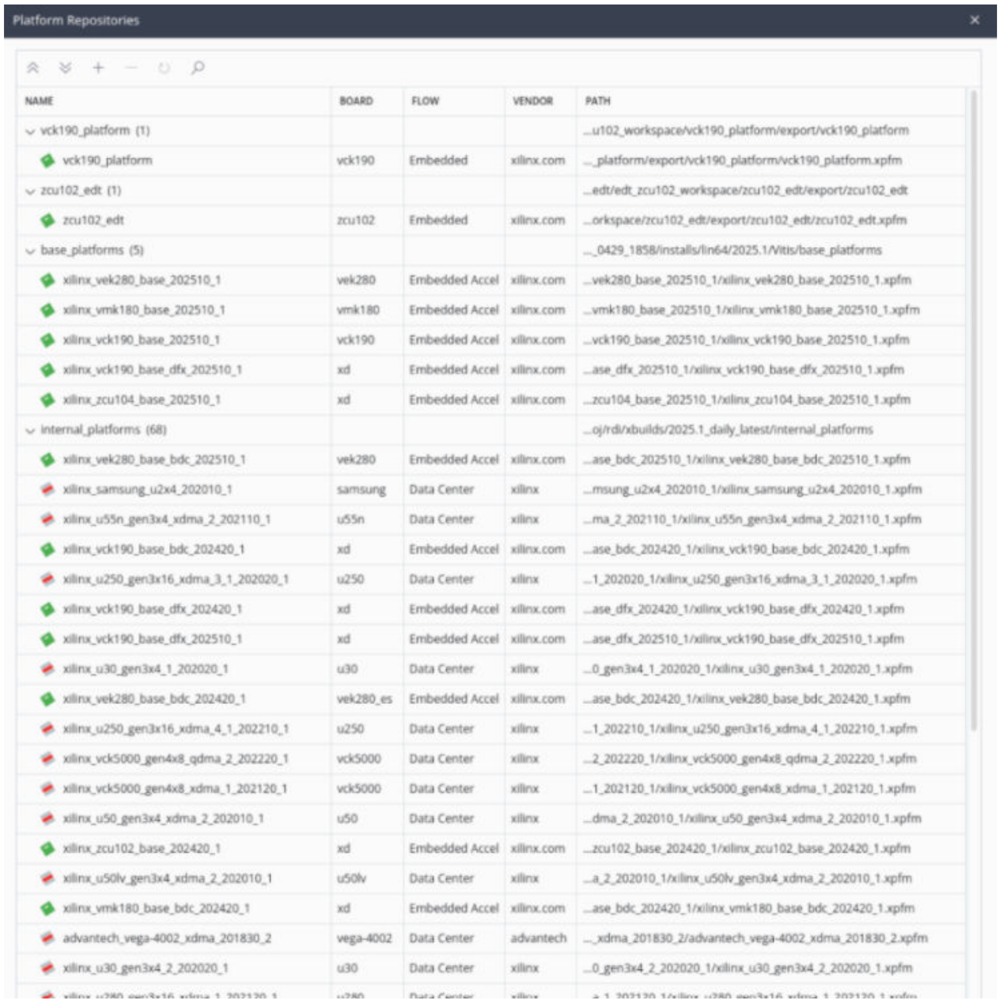
Managing Platforms and Platform Repositories

The Platform Repositories window displays the platforms that are available for use with the Vitis Unified IDE. The upper part of the table displays base platforms in 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. The lower part of the table shows platforms installed in addition to the Vitis tools.

You can manage 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 25: **Platform Repositories**



Platform Repositories

NAME	BOARD	FLOW	VENDOR	PATH
vck190_platform (1)				...u102_workspace/vck190_platform/export/vck190_platform
vck190_platform	vck190	Embedded	xilinx.com	..._platform/export/vck190_platform/vck190_platform.xpfm
zcu102_edt (1)				...edt/edt_zcu102_workspace/zcu102_edt/export/zcu102_edt
zcu102_edt	zcu102	Embedded	xilinx.com	...orkspace/zcu102_edt/export/zcu102_edt/zcu102_edt.xpfm
base_platforms (5)				..._0429_1858/installs/lin64/2025.1/Vitis/base_platforms
xilinx_vek280_base_202510_1	vek280	Embedded Accel	xilinx.com	...vek280_base_202510_1/xilinx_vek280_base_202510_1.xpfm
xilinx_vmk180_base_202510_1	vmk180	Embedded Accel	xilinx.com	...vmk180_base_202510_1/xilinx_vmk180_base_202510_1.xpfm
xilinx_vck190_base_202510_1	vck190	Embedded Accel	xilinx.com	...vck190_base_202510_1/xilinx_vck190_base_202510_1.xpfm
xilinx_vck190_base_dfx_202510_1	xd	Embedded Accel	xilinx.com	...ase_dfx_202510_1/xilinx_vck190_base_dfx_202510_1.xpfm
xilinx_zcu104_base_202510_1	xd	Embedded Accel	xilinx.com	...zcu104_base_202510_1/xilinx_zcu104_base_202510_1.xpfm
internal_platforms (68)				...oj/rdi/xbuilds/2025.1_daily_latest/internal_platforms
xilinx_vek280_base_bdc_202510_1	vek280	Embedded Accel	xilinx.com	...ase_bdc_202510_1/xilinx_vek280_base_bdc_202510_1.xpfm
xilinx_samsung_u2x4_202010_1	samsung	Data Center	xilinx	...msung_u2x4_202010_1/xilinx_samsung_u2x4_202010_1.xpfm
xilinx_u55n_gen3x4_xdma_2_202110_1	u55n	Data Center	xilinx	...ma_2_202110_1/xilinx_u55n_gen3x4_xdma_2_202110_1.xpfm
xilinx_vck190_base_bdc_202420_1	xd	Embedded Accel	xilinx.com	...ase_bdc_202420_1/xilinx_vck190_base_bdc_202420_1.xpfm
xilinx_u250_gen3x16_xdma_3_1_202020_1	u250	Data Center	xilinx	...1_202020_1/xilinx_u250_gen3x16_xdma_3_1_202020_1.xpfm
xilinx_vck190_base_dfx_202420_1	xd	Embedded Accel	xilinx.com	...ase_dfx_202420_1/xilinx_vck190_base_dfx_202420_1.xpfm
xilinx_vck190_base_dfx_202510_1	xd	Embedded Accel	xilinx.com	...ase_dfx_202510_1/xilinx_vck190_base_dfx_202510_1.xpfm
xilinx_u30_gen3x4_1_202020_1	u30	Data Center	xilinx	...0_gen3x4_1_202020_1/xilinx_u30_gen3x4_1_202020_1.xpfm
xilinx_vek280_base_bdc_202420_1	vek280_es	Embedded Accel	xilinx.com	...ase_bdc_202420_1/xilinx_vek280_base_bdc_202420_1.xpfm
xilinx_u250_gen3x16_xdma_4_1_202210_1	u250	Data Center	xilinx	...1_202210_1/xilinx_u250_gen3x16_xdma_4_1_202210_1.xpfm
xilinx_vck5000_gen4x8_qdma_2_202220_1	vck5000	Data Center	xilinx	...2_202220_1/xilinx_vck5000_gen4x8_qdma_2_202220_1.xpfm
xilinx_vck5000_gen4x8_xdma_1_202120_1	vck5000	Data Center	xilinx	...1_202120_1/xilinx_vck5000_gen4x8_xdma_1_202120_1.xpfm
xilinx_u50_gen3x4_xdma_2_202010_1	u50	Data Center	xilinx	...dma_2_202010_1/xilinx_u50_gen3x4_xdma_2_202010_1.xpfm
xilinx_zcu102_base_202420_1	xd	Embedded Accel	xilinx.com	...zcu102_base_202420_1/xilinx_zcu102_base_202420_1.xpfm
xilinx_u50lv_gen3x4_xdma_2_202010_1	u50lv	Data Center	xilinx	...a_2_202010_1/xilinx_u50lv_gen3x4_xdma_2_202010_1.xpfm
xilinx_vmk180_base_bdc_202420_1	xd	Embedded Accel	xilinx.com	...ase_bdc_202420_1/xilinx_vmk180_base_bdc_202420_1.xpfm
advantech_vega-4002_xdma_201830_2	vega-4002	Data Center	advantech	..._xdma_201830_2/advantech_vega-4002_xdma_201830_2.xpfm
xilinx_u30_gen3x4_2_202020_1	u30	Data Center	xilinx	...0_gen3x4_2_202020_1/xilinx_u30_gen3x4_2_202020_1.xpfm

Target Platform

In the Vitis unified software platform, the runtime environment of the application is 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

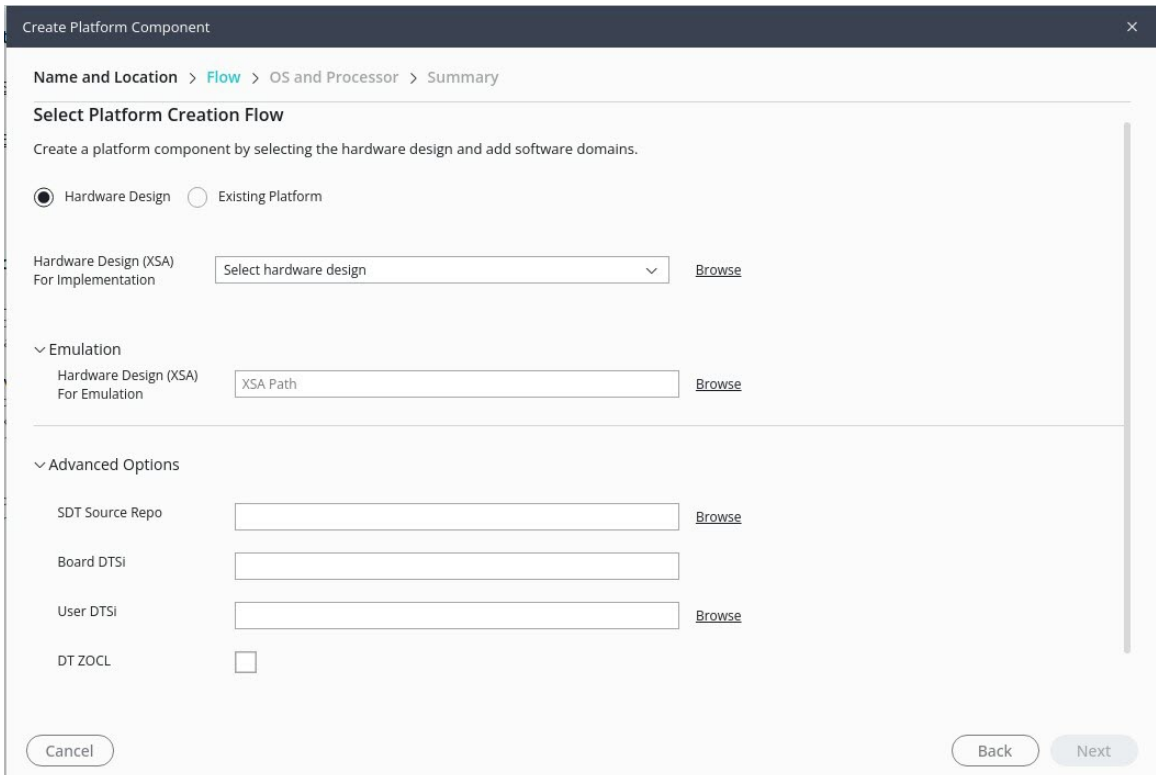
To create a new platform component in the Vitis Unified IDE, execute the following steps.

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. You can also create a platform from an 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 26: **Create Platform Component**


Create Platform Component
Name and Location > Flow > OS and Processor > Summary

Select Platform Creation Flow

Create a platform component by selecting the hardware design and add software domains.

● Hardware Design ○ Existing Platform

Hardware Design (XSA) For Implementation	Select hardware design	Browse
Emulation		
Hardware Design (XSA) For Emulation	XSA Path	Browse
Advanced Options		
SDT Source Repo		Browse
Board DTSi		
User DTSi		Browse
DT ZOCL	☐	

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 input hardware design type determine the platform type you create. If the input hardware design is a fixed XSA, the system creates a platform for an embedded design. If the input hardware design is an extensible XSA, the system creates an extensible platform.

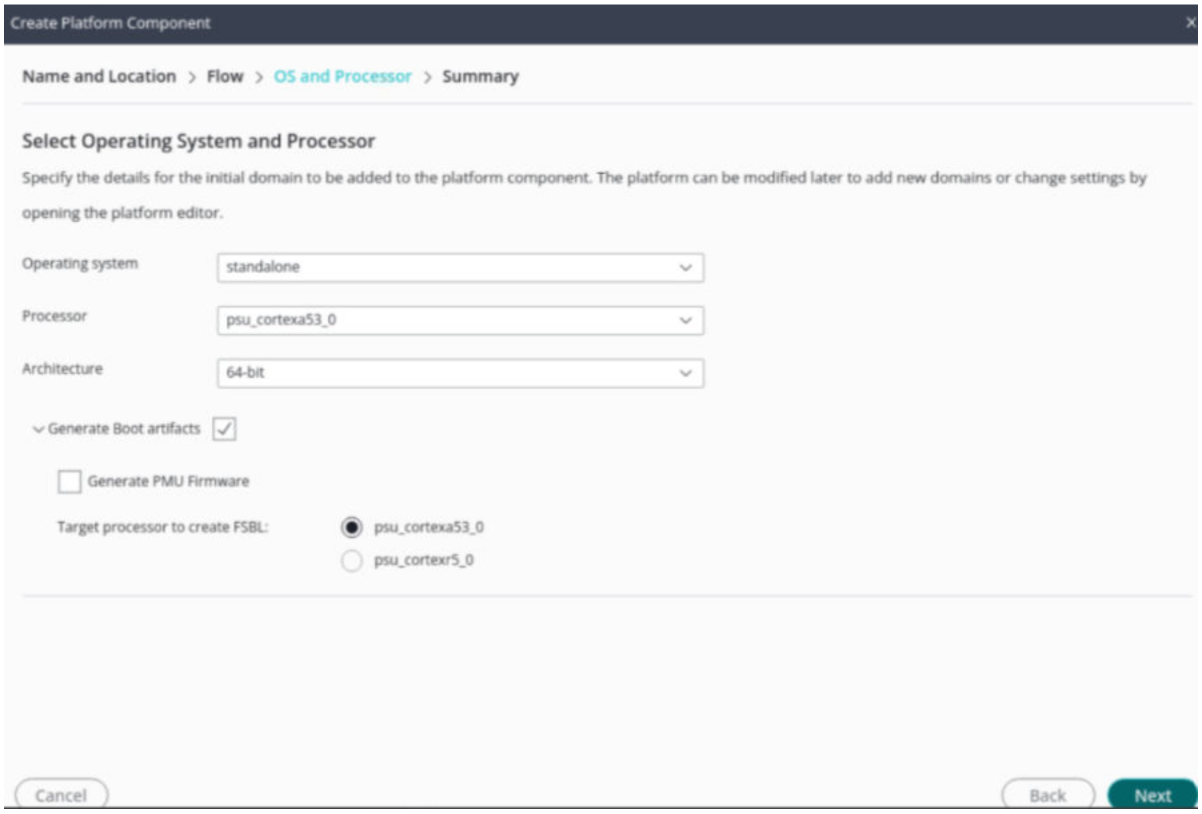
SDTgen is a built-in tool within the Vitis Unified IDE.

- The SDTgen tool 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, parses the SDT and generates the corresponding BSP (Board Support Package) file.
- To facilitate user debugging and take advantage of the SDT flow, advanced options allow you to switch to a custom version SDTgen tool. The uses of the advanced options are as follows:
  - **SDT Source REPO:** Input the customer version SDTgen REPO.

Note: If you encounter issues related to the system device tree or the SDTgen tool, use the updated version of the SDTgen tool for debugging. 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:** Specify the user device tree file (DTSI).

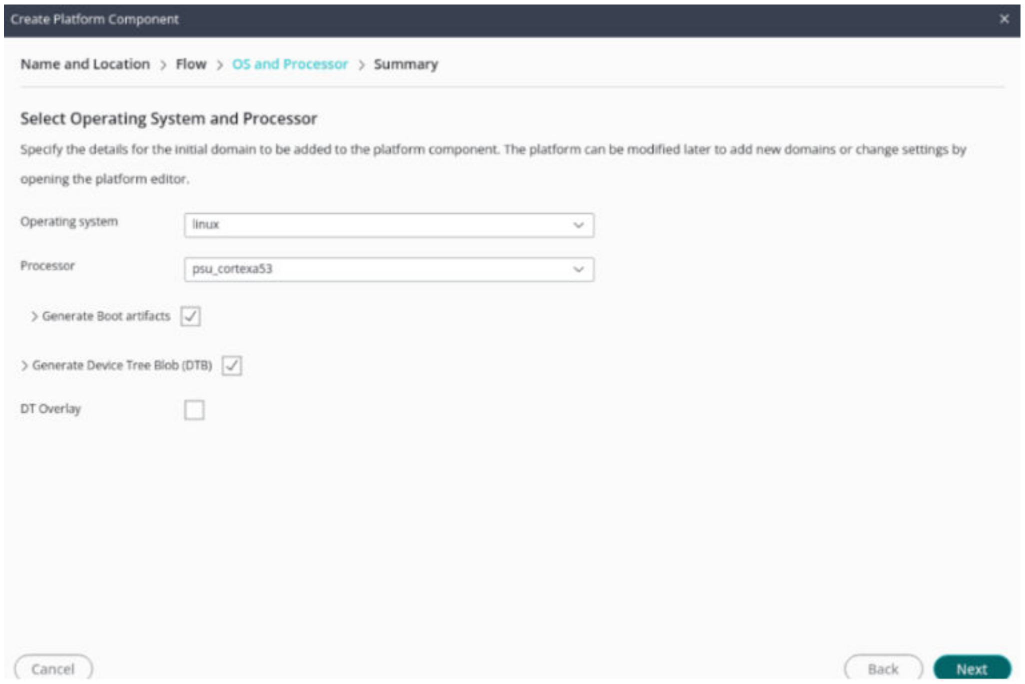
- After selecting the XSA, the tool reads the XSA and identifies the available processors and operating system domains. Specify the Operating system, and the Processor for the platform, and click Next to proceed to the Summary page.


Create Platform Component
Name and Location > Flow > OS and Processor > Summary
Cancel
Back
Next

Select Operating System and Processor

Specify the details for the initial domain to be added to the platform component. The platform can be modified later to add new domains or change settings by opening the platform editor.

Operating system: standalone
Processor: psu_cortexa53_0
Architecture: 64-bit
Generate Boot artifacts ✓
Generate PMU Firmware ☐
Target processor to create FSBL: [✓ psu_cortexa53_0] psu_cortexr5_0


Create Platform Component
Name and Location > Flow > OS and Processor > Summary
Cancel
Back
Next

Select Operating System and Processor

Specify the details for the initial domain to be added to the platform component. The platform can be modified later to add new domains or change settings by opening the platform editor.

Operating system: linux

Processor: psu_cortexa53

Generate Boot artifacts ✓

Generate Device Tree Blob (DTB) ✓

DT Overlay ☐

***Note*:** When you enable the **Generate Boot Artifacts** and **Generate PMU Firmware** options:

- The Vitis tool automatically generates both the FSBL (First Stage Boot Loader) and PMU (Platform Management Unit) firmware components for your platform if the domain is Standalone.
- The Generate Boot Artifacts option is not available for Versal adaptive SoC. Versal adaptive SoC only require the PDI file.

***Note*:** When you enable the **Generate Device Tree Blob** (DTB) option, the system automatically generates the Device Tree Blob for your platform. This only applies 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 Settings for Different Device Families**

Device Family	Generate Boot Artifacts	PMU	DTB	DTB Overlay
Zynq	Enable the FSBL for device initialization.	No option	Only Linux systems require DTB	Only Linux systems require the DTB overlay when loading certain devices after the system is up and running.

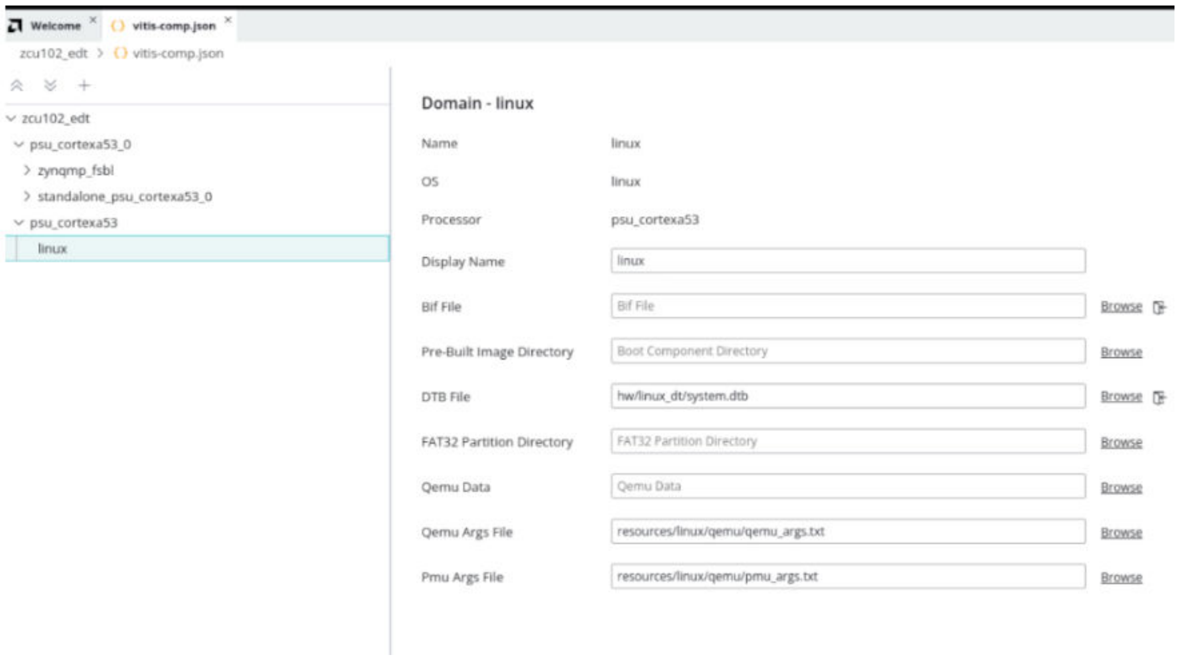
Table 7: **Options Settings for Different Device Families** *(cont'd)*

Device Family	Generate Boot Artifacts	PMU	DTB	DTB Overlay
ZynqMP	Enable the FSBL for device initialization.	Standalone domain does not require the PMU, but it is essential for Linux systems to manage high-security functions.	Only Linux systems require DTB	Only Linux systems require the DTB overlay when loading certain devices after the system is up and running.
Versal	No option	No option	Only Linux systems require DTB	Only Linux systems require the DTB overlay 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.

After you create the Platform component, the `vitis-comp.json` file for the component opens in the central editor window.

Figure 27: **Domain - linux**



Domain - linux

Name	linux	
OS	linux	
Processor	psu_cortexa53	
Display Name	linux	
Bif File	Bif File	Browse
Pre-Built Image Directory	Boot Component Directory	Browse
DTB File	hw/linux_dt/system.dtb	Browse
FAT32 Partition Directory	FAT32 Partition Directory	Browse
Qemu Data	Qemu Data	Browse
Qemu Args File	resources/linux/qemu/qemu_args.txt	Browse
Pmu Args File	resources/linux/qemu/pmu_args.txt	Browse

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

- For the Linux Operating System: specify the Bif file, Boot Component Directory, SD Card Directory and as well as the Qemu data. The Qemu Args file are auto-populated.

- For standalone (baremetal) OS: The system does not require special operations.
- For FreeRTOS: The system does not require special operations.

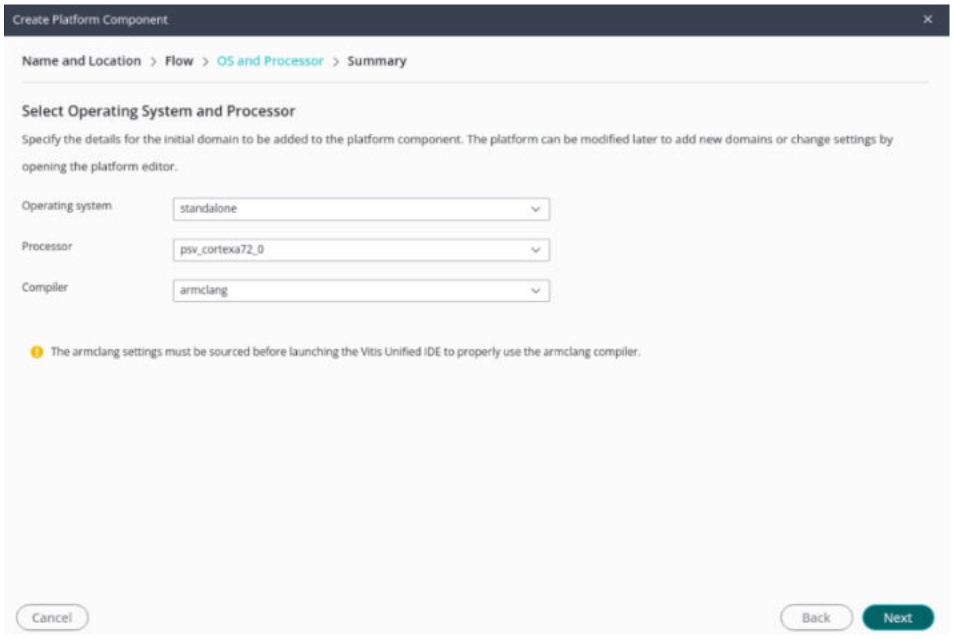
After confirming the settings, you can build the Platform component by selecting it in the Flow Navigator and selecting the **Build** command. After compilation, the Vitis tool populates the platform and its related software and hardware components in the `Output` folder of the Platform component.

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! *The Vitis Unified IDE installation does not include these compilers. Therefore, you 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 28: **OS and Processor Summary**


Create Platform Component
Name and Location > Flow > OS and Processor > Summary
Cancel
Back
Next

Select Operating System and Processor

Specify the details for the initial domain to be added to the platform component. The platform can be modified later to add new domains or change settings by opening the platform editor.

Operating system: standalone

Processor: psv_cortexa72_0

Compiler: armclang

The armclang settings must be sourced before launching the Vitis Unified IDE to properly use the armclang compiler.

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 want 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.

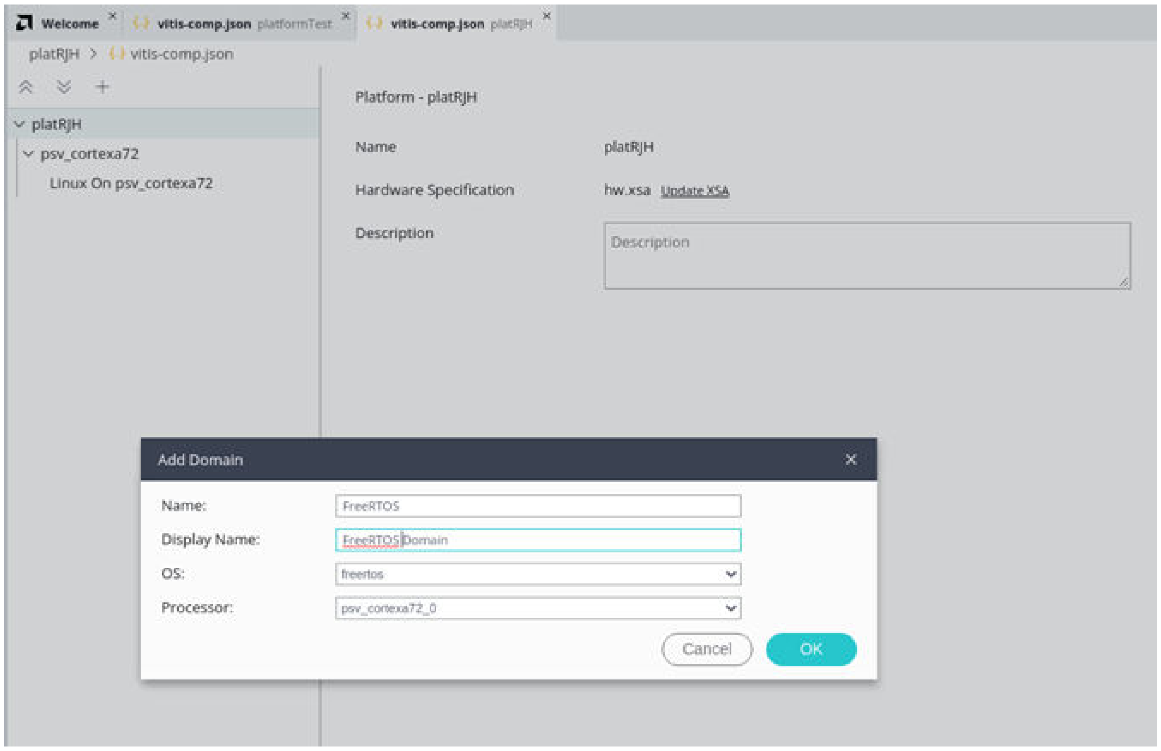
   ***Note*:** Make sure the platform to customize is visible in the platform repository list. If it is not, click **+** to add the platform path to the platform repository.
6. Click **Finish**. You can now modify the new platform in the workspace as any other platform.

Adding a Domain to an Existing Platform

A platform can contain multiple domains, supporting different operating systems and targeting different processors. The Vitis tool supports 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.



Platform - platRJH

Name	platRJH
Hardware Specification	hw.xsa Update XSA
Description	Description

Name:	FreeRTOS
Display Name:	FreeRTOS Domain
OS:	freertos
Processor:	psv_cortexa72_0

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. PetaLinux can generate these components.

TIP: *The components in the Linux domain settings copies 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

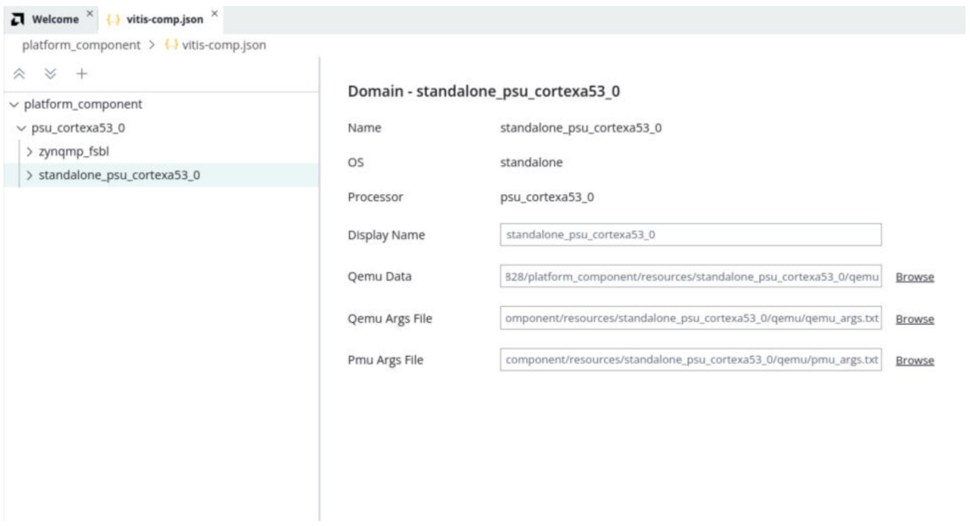
There are different kinds of domains. The standalone domain is the most common. Each domain has an associated BSP which can be configured extensively. Additionally, the domain overview page includes extra settings for the domain.

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 29: **Standalone Domain Overview Page**



Domain - standalone_psu_cortexa53_0

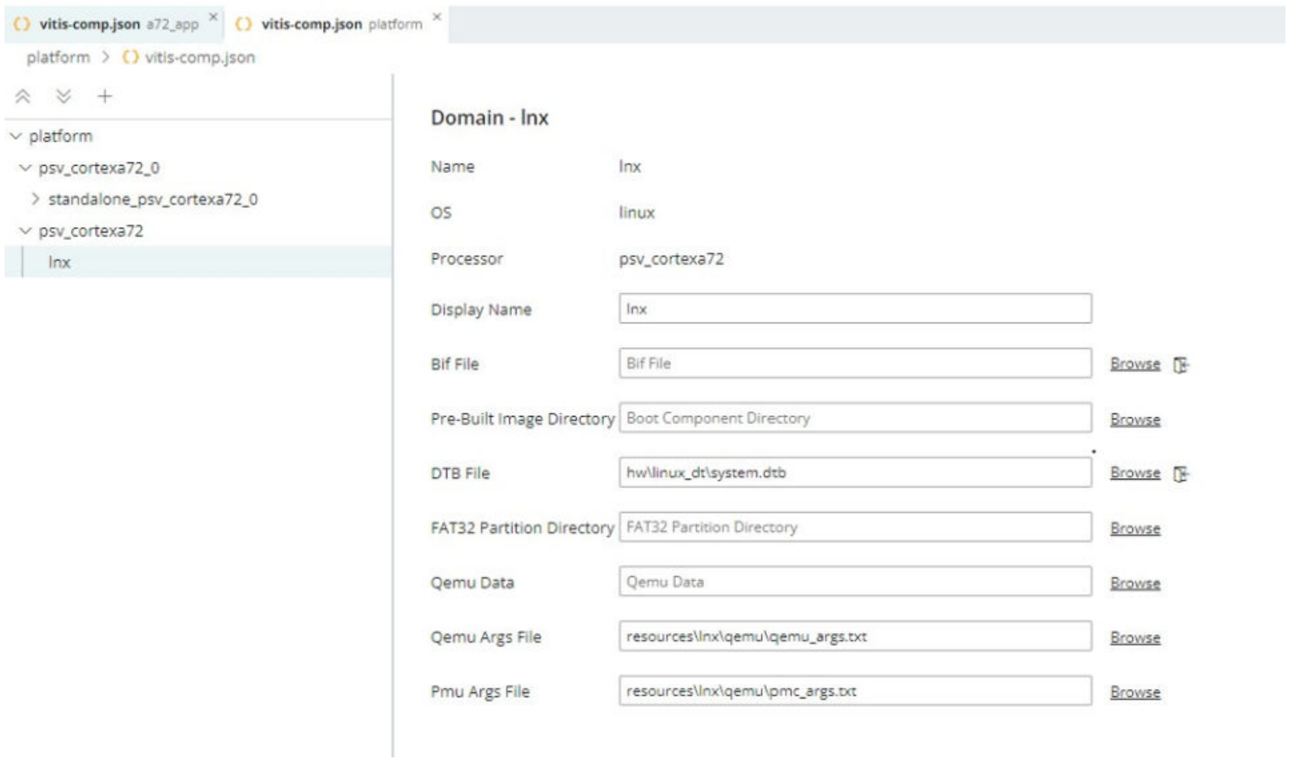
Name	standalone_psu_cortexa53_0
OS	standalone
Processor	psu_cortexa53_0
Display Name	standalone_psu_cortexa53_0
Qemu Data	828/platform_component/resources/standalone_psu_cortexa53_0/qemu
Qemu Args File	omponent/resources/standalone_psu_cortexa53_0/qemu/qemu_args.txt
Pmu Args File	component/resources/standalone_psu_cortexa53_0/qemu/pmu_args.txt

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 30: **Linux Domain Overview Page**



Domain - lnx

Name	lnx
OS	linux
Processor	psv_cortexa72
Display Name	lnx
Bif File	Bif File
Pre-Built Image Directory	Boot Component Directory
DTB File	hw\linux_dt\system.dtb
FAT32 Partition Directory	FAT32 Partition Directory
Qemu Data	Qemu Data
Qemu Args File	resources\lnx\qemu\qemu_args.txt
Pmu Args File	resources\lnx\qemu\pmc_args.txt

- **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, the Vitis tool automatically populates the DTB file in this field.
- **FAT32 Partition Directory:** Used to add additional files to the FAT32 partition.
- **Qemu Data:** Automatically populated when building the platform. This field provides the boot components for hardware emulation.

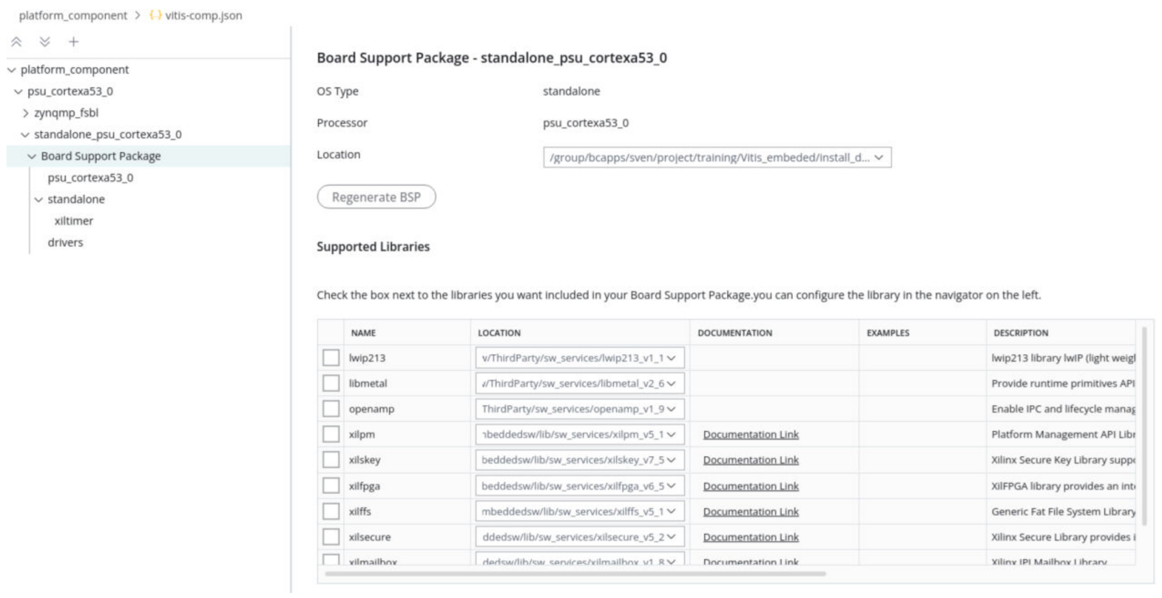
Board Support Package Settings Page

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

***Note*:** You cannot change the OS choice on this page. The software platform creation step determines the OS type.

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**.

Figure 31: **Overview**



Board Support Package - standalone_psu_cortexa53_0

OS Type	standalone
Processor	psu_cortexa53_0
Location	/group/bcapps/sven/project/training/Vitis_embeded/install_d...

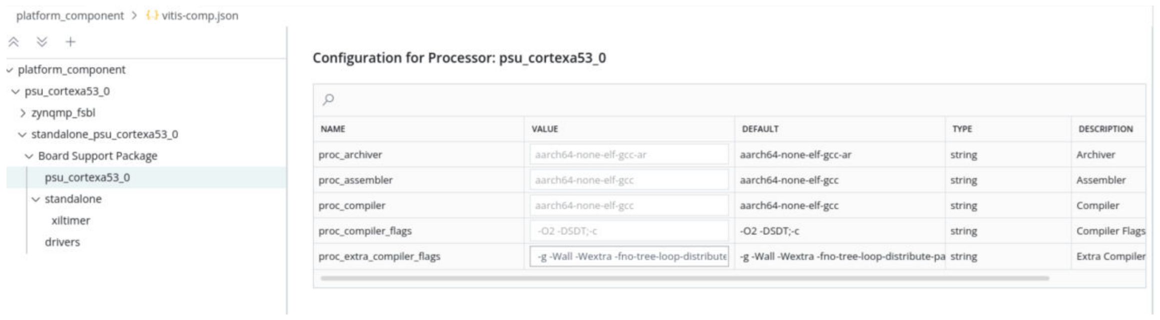
Supported Libraries

Check the box next to the libraries you want included in your Board Support Package.you can configure the library in the navigator on the left.

	NAME	LOCATION	DOCUMENTATION	EXAMPLES	DESCRIPTION
☐	lwip213	v/ThirdParty/sw_services/lwip213_v1_1			lwip213 library lwIP (light weigl
☐	libmetal	ı/ThirdParty/sw_services/libmetal_v2_6			Provide runtime primitives API
☐	openamp	ThirdParty/sw_services/openamp_v1_9			Enable IPC and lifecycle manag
☐	xilpm	ıbeddedsw/lib/sw_services/xilpm_v5_1	Documentation Link		Platform Management API Libr
☐	xilskey	beddedsw/lib/sw_services/xilskey_v7_5	Documentation Link		Xilinx Secure Key Library supp
☐	xilfpga	beddedsw/lib/sw_services/xilfpga_v6_5	Documentation Link		XilFPGA library provides an int
☐	xilffs	mbeddedsw/lib/sw_services/xilffs_v5_1	Documentation Link		Generic Fat File System Library
☐	xilsecure	ddedsw/lib/sw_services/xilsecure_v5_2	Documentation Link		Xilinx Secure Library provides i
☐	xilmailbox	dedsw/lib/sw_services/xilmailbox_v1_8	Documentation Link		Xilinx IPI Mailbox Library

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

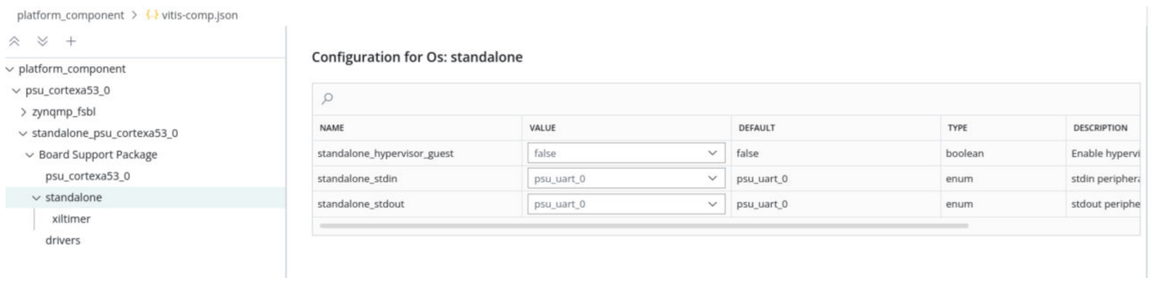
Figure 32: **Processor Parameter Configuration**



Configuration for Processor: psu_cortexa53_0

NAME	VALUE	DEFAULT	TYPE	DESCRIPTION
proc_archiver	aarch64-none-elf-gcc-ar	aarch64-none-elf-gcc-ar	string	Archiver
proc_assembler	aarch64-none-elf-gcc	aarch64-none-elf-gcc	string	Assembler
proc_compiler	aarch64-none-elf-gcc	aarch64-none-elf-gcc	string	Compiler
proc_compiler_flags	-O2 -DSDT;-c	-O2 -DSDT;-c	string	Compiler Flags
proc_extra_compiler_flags	-g -Wall -Wextra -fno-tree-loop-distribute	-g -Wall -Wextra -fno-tree-loop-distribute-pa	string	Extra Compiler

Figure 33: **OS Parameter Configuration**

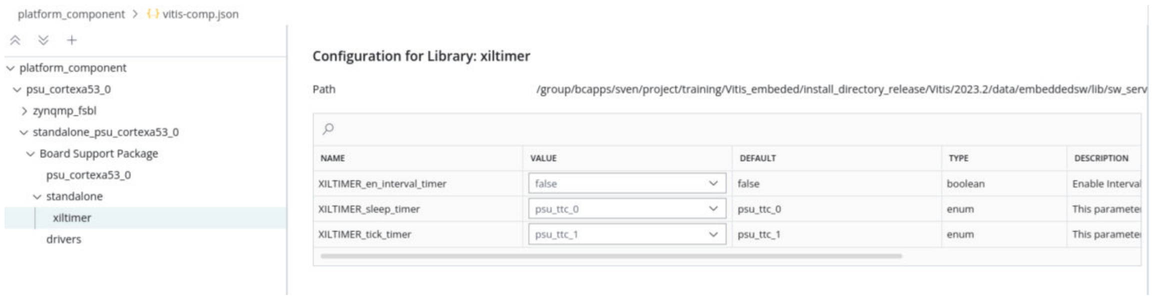


Configuration for Os: standalone

NAME	VALUE	DEFAULT	TYPE	DESCRIPTION
standalone_hypervisor_guest	false	false	boolean	Enable hypervi
standalone_stdin	psu_uart_0	psu_uart_0	enum	stdin periphera
standalone_stdout	psu_uart_0	psu_uart_0	enum	stdout periphe

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

Figure 34: **Library Configuration**



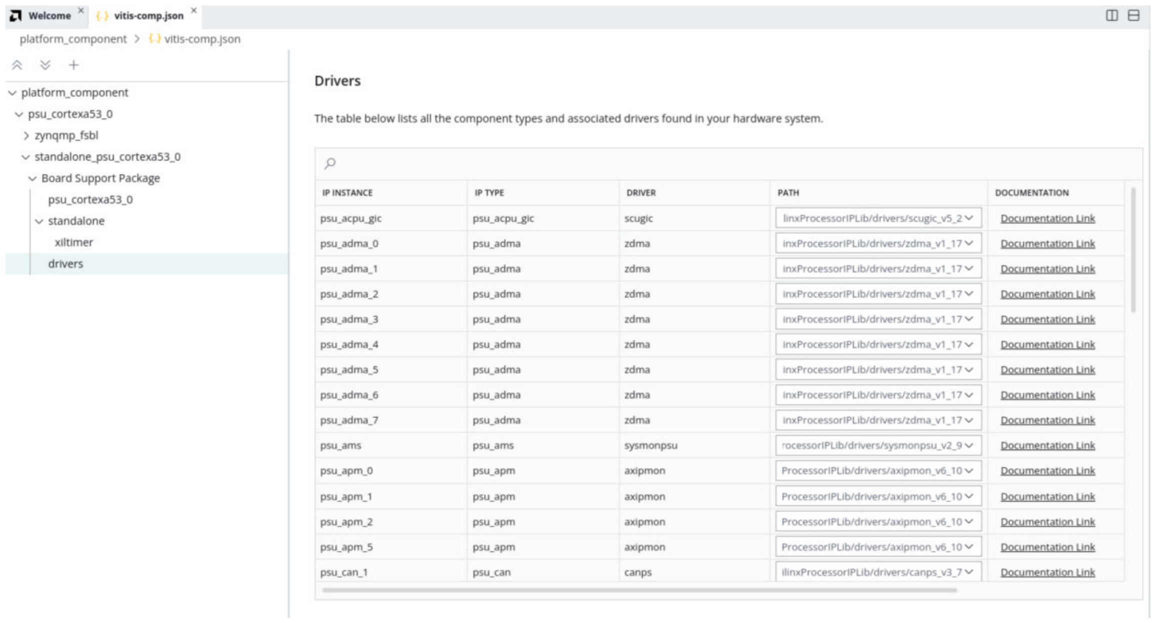
Configuration for Library: xiltimer

Path /group/bcapps/sven/project/training/Vitis_embeded/install_directory_release/Vitis/2023.2/data/embeddedsw/lib/sw_serv

NAME	VALUE	DEFAULT	TYPE	DESCRIPTION
XILTIMER_en_interval_timer	false	false	boolean	Enable Interval
XILTIMER_sleep_timer	psu_ttc_0	psu_ttc_0	enum	This paramete
XILTIMER_tick_timer	psu_ttc_1	psu_ttc_1	enum	This paramete

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 35: **Drivers**



Drivers

The table below lists all the component types and associated drivers found in your hardware system.

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

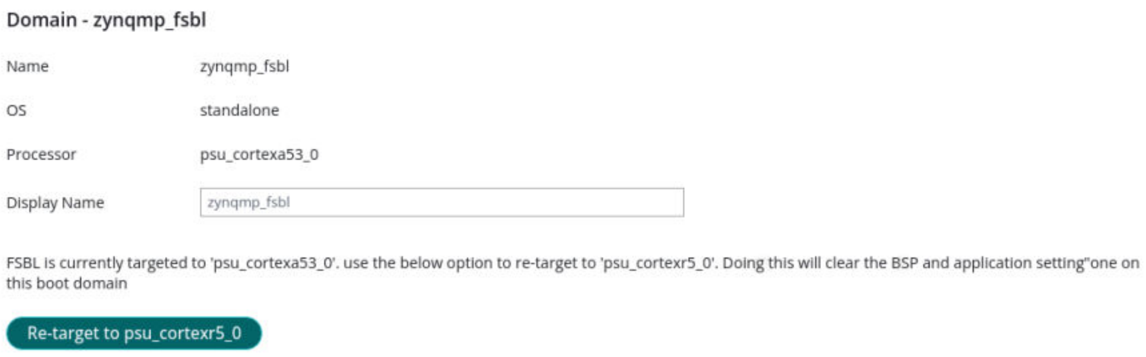
The build settings section lists the toolchain selected to build the BSP as well as some extra configuration settings.

Switching FSBL Targeting Processor

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

1. Click your platform component, expand Settings and double click the **vitis-comp.json** file.
2. Select **psu_cortexa53_0→zynqmp_fsbl**.

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



Domain - zynqmp_fsbl

Name	zynqmp_fsbl
OS	standalone
Processor	psu_cortexa53_0
Display Name	zynqmp_fsbl

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

Modifying Source Code for FSBL

If you select boot component generation in platform generation, the Vitis tool creates FSBL applications 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 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 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.

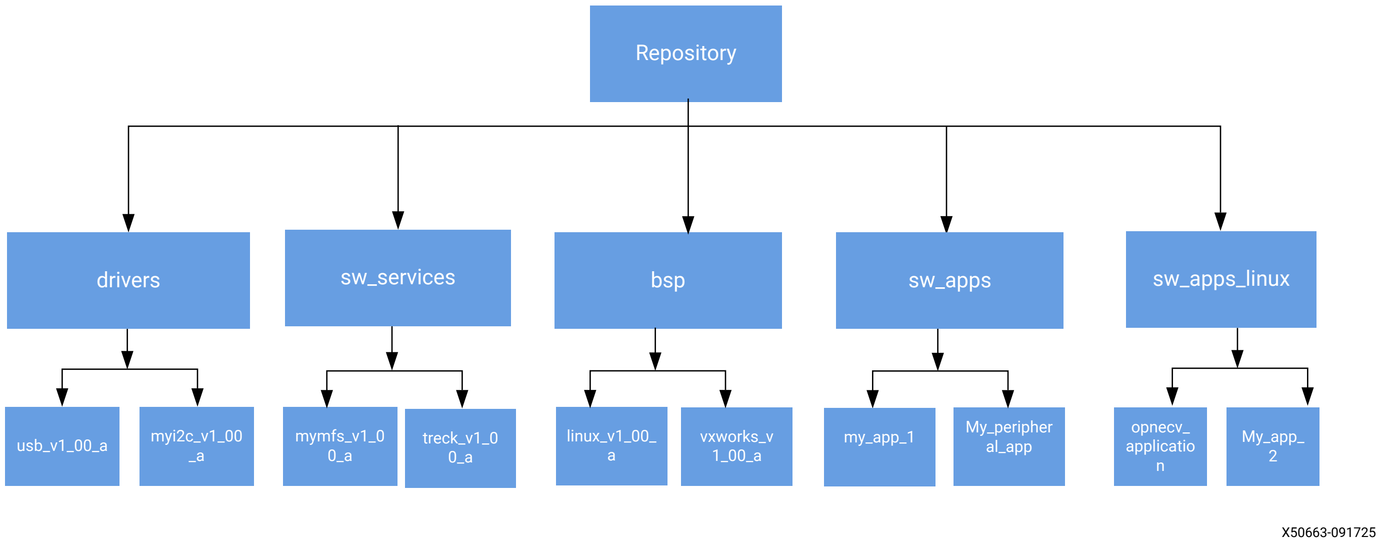
Your Vitis software platform workspace can point to multiple software repositories. The scope of the software repository can be global (available across all workspaces) or local (available only to the current workspace). Components in local software repositories added to the workspace take precedence over identical components in global software repositories. Global software repositories take higher precedence over identical components 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 36: **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 or 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.

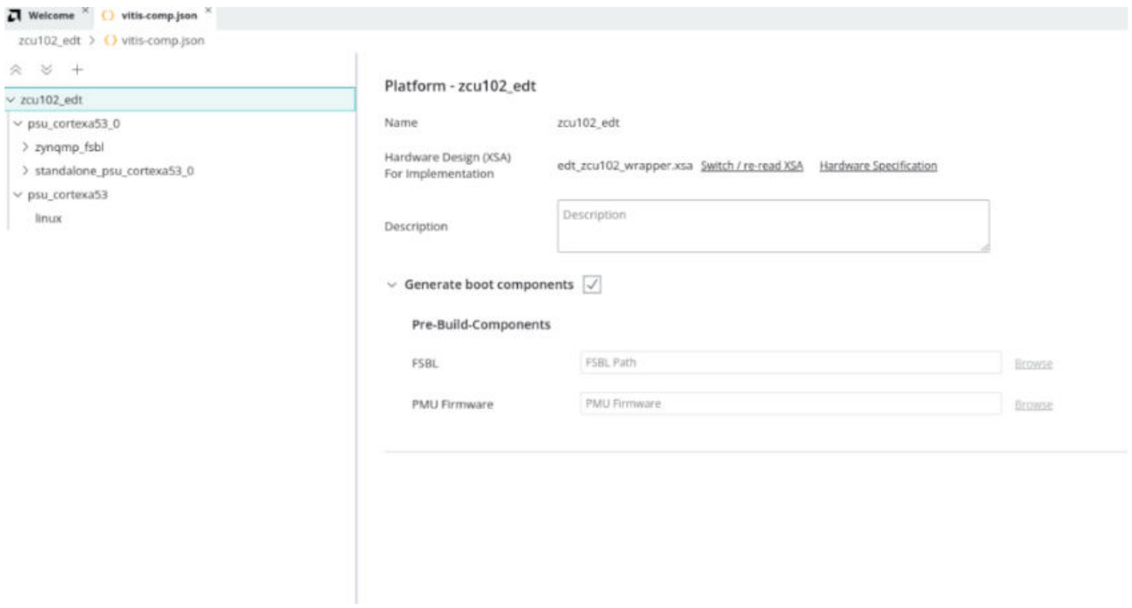
***Note*:** Only source files (not settings) revert to their original state.

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**.



Platform - zcu102_edt

Name zcu102_edt

Hardware Design (XSA) For Implementation edt_zcu102_wrapper.xsa Switch / re-read XSA Hardware Specification

Description

Generate boot components ✓

Pre-Build-Components

FSBL Browse

PMU Firmware Browse

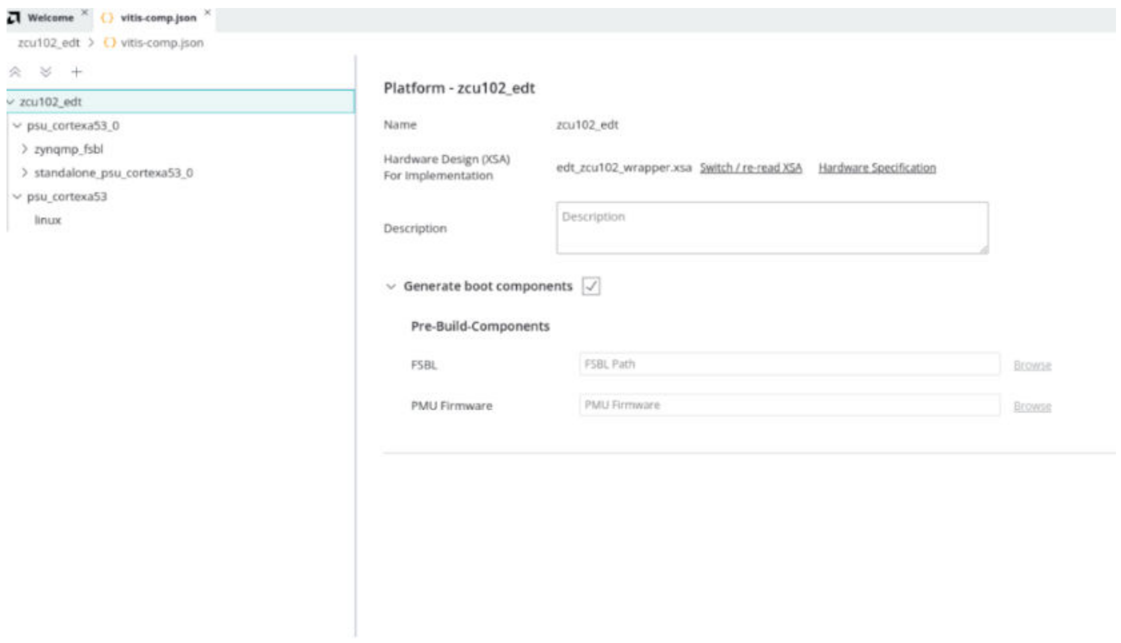
3. Specify the source hardware specification file in pop-up window.
4. Click **open** to select the XSA file.

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.



Platform - zcu102_edt

Name	zcu102_edt
Hardware Design (XSA) For Implementation	edt_zcu102_wrapper.xsa
Description	Description

Generate boot components ✓

Pre-Build-Components

FSBL	FSBL Path
PMU Firmware	PMU Firmware

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

Applications

An embedded application in Vitis tools runs on embedded systems or embedded platforms. Embedded applications have the following traits within the context of Vitis:

- **Targeted for Embedded Systems:** Embedded applications run on embedded hardware systems. Embedded applications 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. These use cases include controlling IoT (Internet of Things) devices, managing sensors and actuators, running real-time control algorithms, performing signal processing, communications, and more.
- **Hardware Integration:** Embedded applications can 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:** Developers often parallel compile embedded applications. It is common for Vitis to compile the application to run on a target embedded system with a different architecture than the host computer.

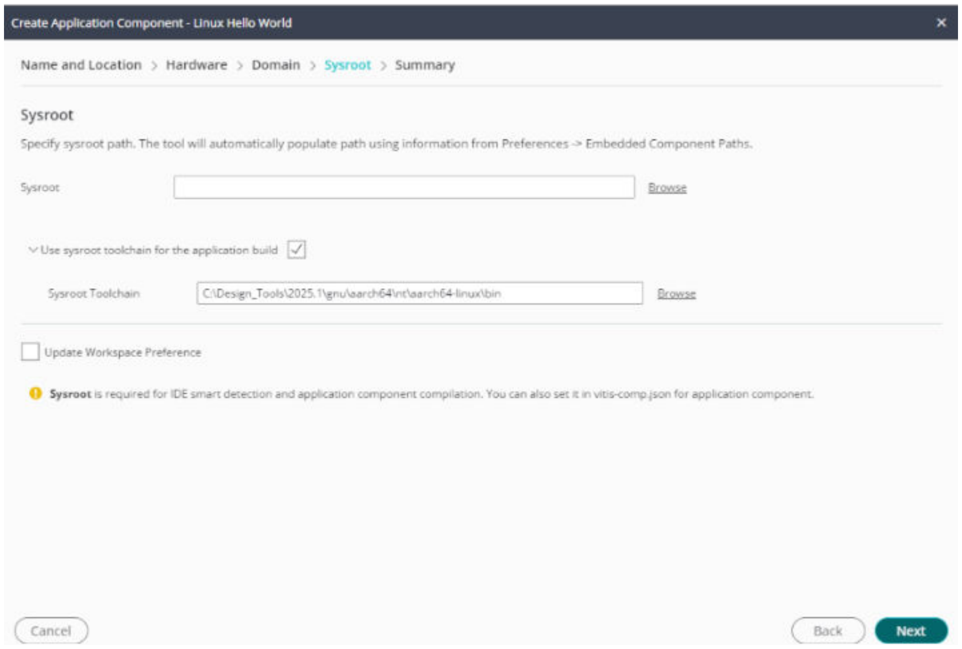
Creating application component

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 a Linux application component, you can select a third party sysroot toolchain by selecting the checkbox and providing the path.


Create Application Component - Linux Hello World

Name and Location > Hardware > Domain > Sysroot > Summary

Sysroot

Specify sysroot path. The tool will automatically populate path using information from Preferences -> Embedded Component Paths.

Sysroot:

Use sysroot toolchain for the application build [✓]

Sysroot Toolchain: C:\Design_Tools\2025.1\gnu\aarch64\nt\aarch64-linux\bin

Update Workspace Preference [ ]

Sysroot is required for IDE smart detection and application component compilation. You can also set it in vitis-comp.json for application component.

If you disable `use sysroot toolchain`, Vitis tools uses the default gcc compiler. If you pass a path that does not have any gcc or g++ compilers, Vitis throws 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

The standalone domain generated BSP can link to applications. Applications can also run on the Linux domain operating system. Creating an application requires a platform.

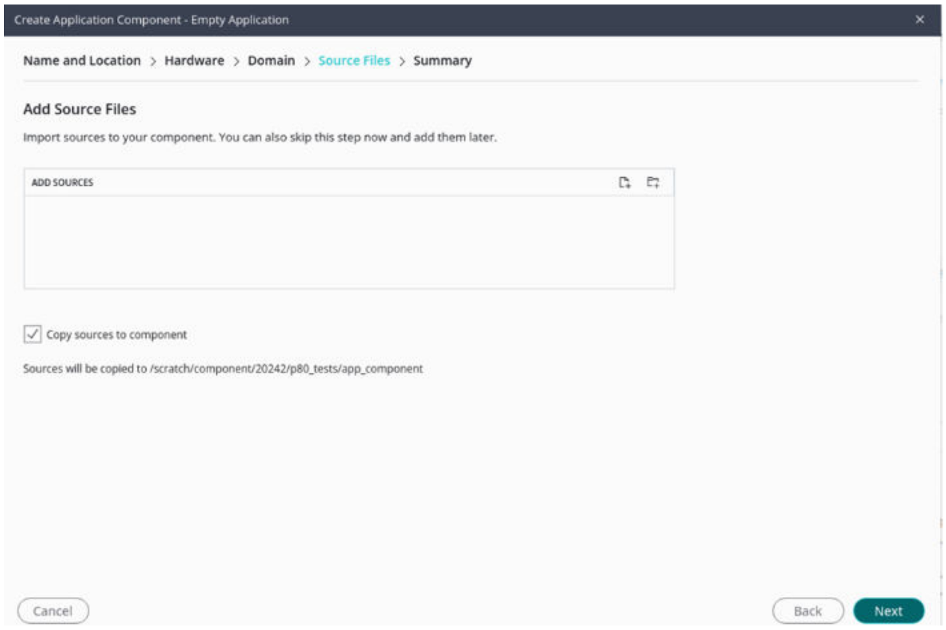
To create an Application component, select **File→New Component→Application**.

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. 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.


Create Application Component - Empty Application
Name and Location > Hardware > Domain > Source Files > Summary
ADD SOURCES
Cancel
Back
Next

Add Source Files

Import sources to your component. You can also skip this step now and add them later.

☑ Copy sources to component

Sources will be copied to /scratch/component/20242/p80_tests/app_component

Note: The Vitis tool selects **Copy Source to Component** by default. Deselect the 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

You can now build your component or configure your building settings.

Creating a System Project

Follow these steps to create a system project.

1. Go to **File→New Component→System Project**.

   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, you don't need to set the Image path when performing embedded development. Click **Next**.

   Note: Acceleration application development uses the Embedded Component Path setting. 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**.

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 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 cannot combine with a Linux application in one system project. This is 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. Next, you can create an application project.

When you build an executable for this application, Vitis automatically performs the following actions.

1. The Vitis software platform builds the board support package, also 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 link together to form the final executable. A linker script specifies where object files should be placed in memory, with object files as inputs.

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

Build Configuration Settings

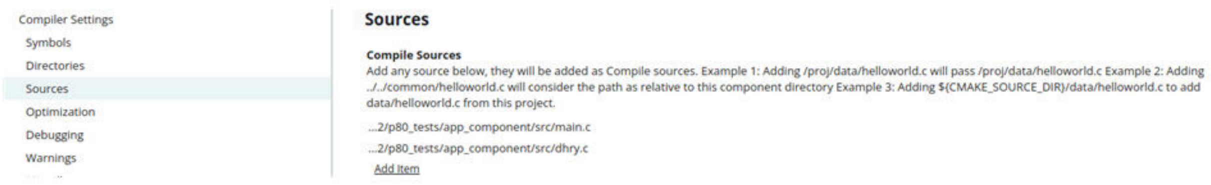
You can modify the build configurations manually:

- 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.



Sources

Compile Sources
Add any source below, they will be added as Compile sources. Example 1: Adding /proj/data/helloworld.c will pass /proj/data/helloworld.c Example 2: Adding ../../common/helloworld.c will consider the path as relative to this component directory Example 3: Adding ${CMAKE_SOURCE_DIR}/data/helloworld.c to add data/helloworld.c from this project.

...2/p80_tests/app_component/src/main.c

...2/p80_tests/app_component/src/dhry.c

Add Item

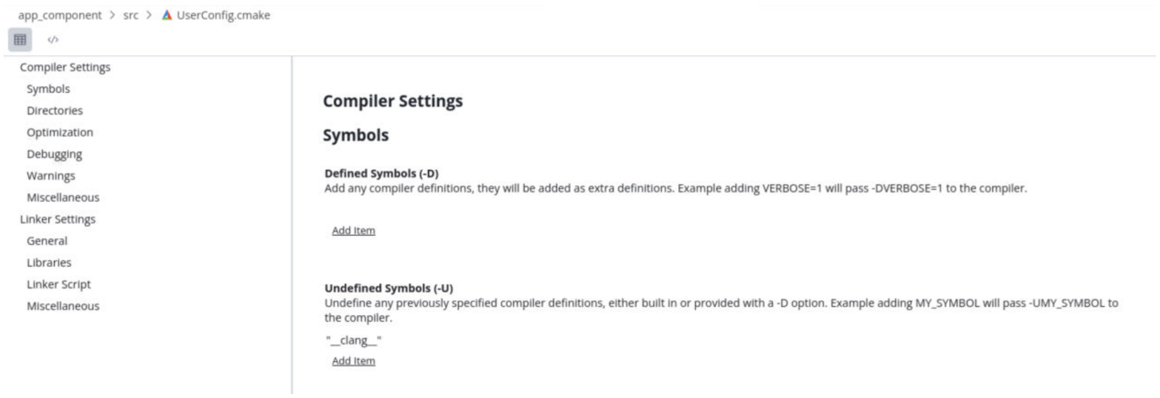
3. Click **Add Item** to add source code for your application.
4. You can also hove your mouse over the name of source file and select to remove it.

   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

The Vitis tool processes and tokenizes definitions and symbols 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.



app_component > src > UserConfig.cmake

Compiler Settings
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
Linker Settings
General
Libraries
Linker Script
Miscellaneous

Compiler Settings

Symbols

Defined Symbols (-D)
Add any compiler definitions, they will be added as extra definitions. Example adding VERBOSE=1 will pass -DVERBOSE=1 to the compiler.

Add Item

Undefined Symbols (-U)
Undefine any previously specified compiler definitions, either built in or provided with a -D option. Example adding MY_SYMBOL will pass -UMY_SYMBOL to the compiler.

"__clang__"
Add Item

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.

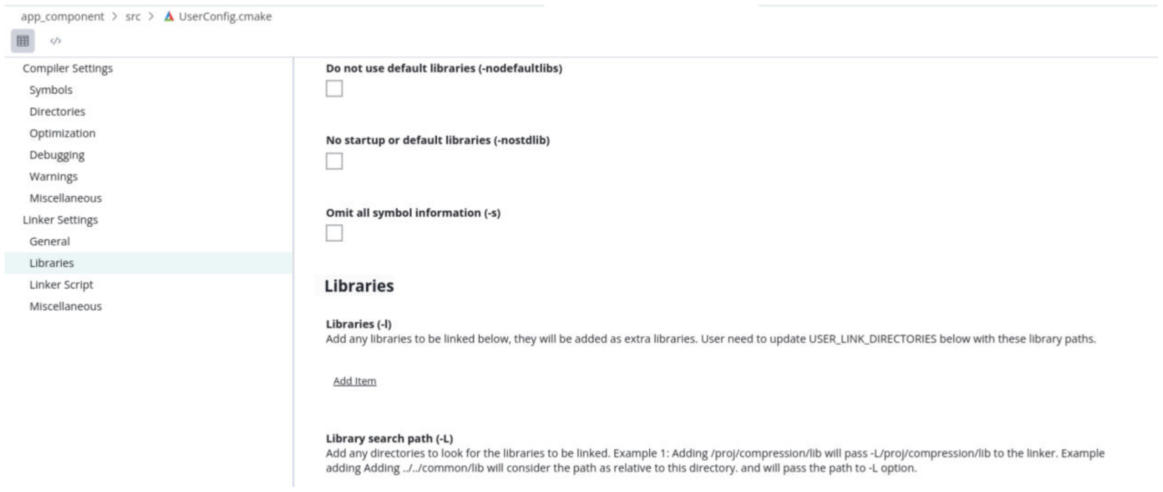
Adding Libraries and Library Paths

You can add libraries and library paths for Application projects. If you have a custom library to link against, you can 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.



Do not use default libraries (-nodefaultlibs)

No startup or default libraries (-nostdlib)

Omit all symbol information (-s)

Libraries

Libraries (-l)
Add any libraries to be linked below, they will be added as extra libraries. User need to update USER_LINK_DIRECTORIES below with these library paths.

Library search path (-L)
Add any directories to look for the libraries to be linked. Example 1: Adding /proj/compression/lib will pass -L/proj/compression/lib to the linker. Example adding Adding ../../common/lib will consider the path as relative to this directory. and will pass the path to -L option.

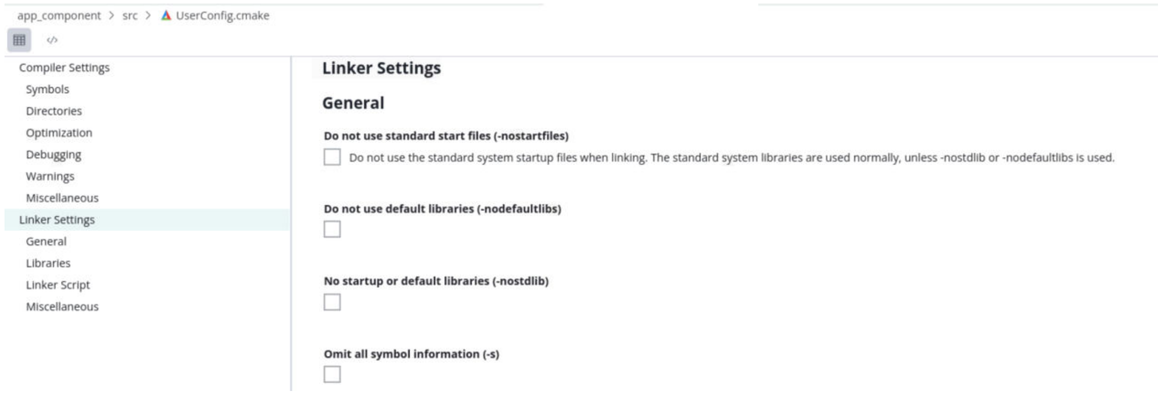
Note: `UserConfig.cmake` supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

Specifying the Linker Options

Specify the linker options for Application components using the steps below. You can also specify any other linker flags not covered in the Tool Settings.

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.



Linker Settings

General

Do not use standard start files (-nostartfiles)
Do not use the standard system startup files when linking. The standard system libraries are used normally, unless -nostdlib or -nodefaultlibs is used.

Do not use default libraries (-nodefaultlibs)

No startup or default libraries (-nostdlib)

Omit all symbol information (-s)

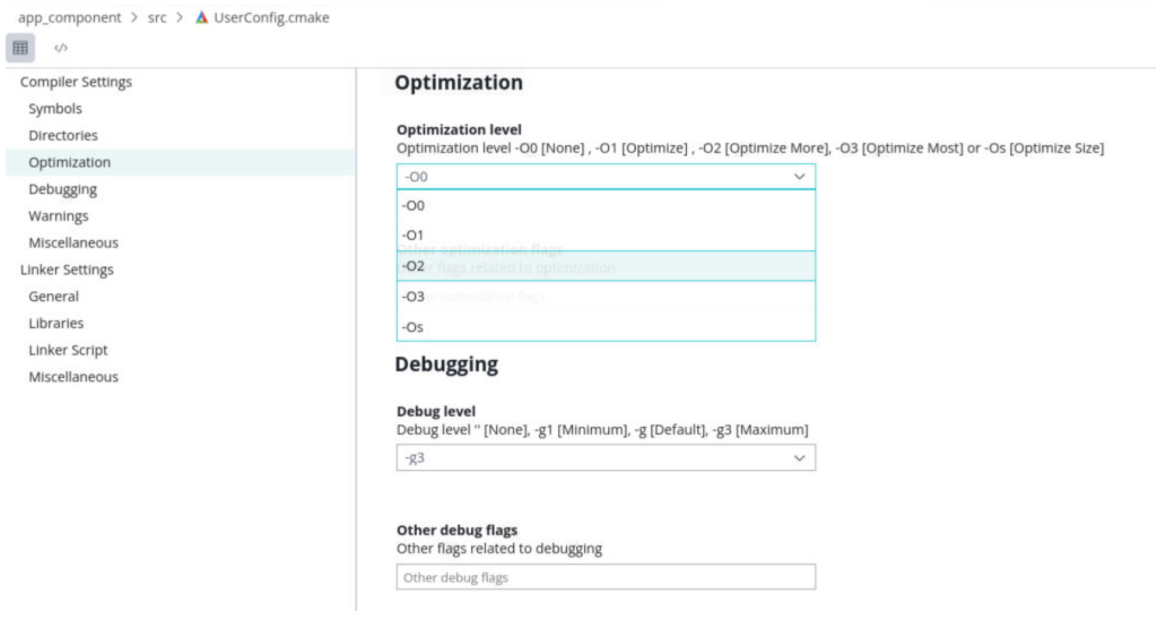
Note: **UserConfig.cmake** supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

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.



Optimization

Optimization level
Optimization level -O0 [None] , -O1 [Optimize] , -O2 [Optimize More], -O3 [Optimize Most] or -Os [Optimize Size]

-O0
-O0
-O1
-O2
-O3
-Os

Debugging

Debug level
Debug level '' [None], -g1 [Minimum], -g [Default], -g3 [Maximum]

-g3

Other debug flags
Other flags related to debugging

Other debug flags

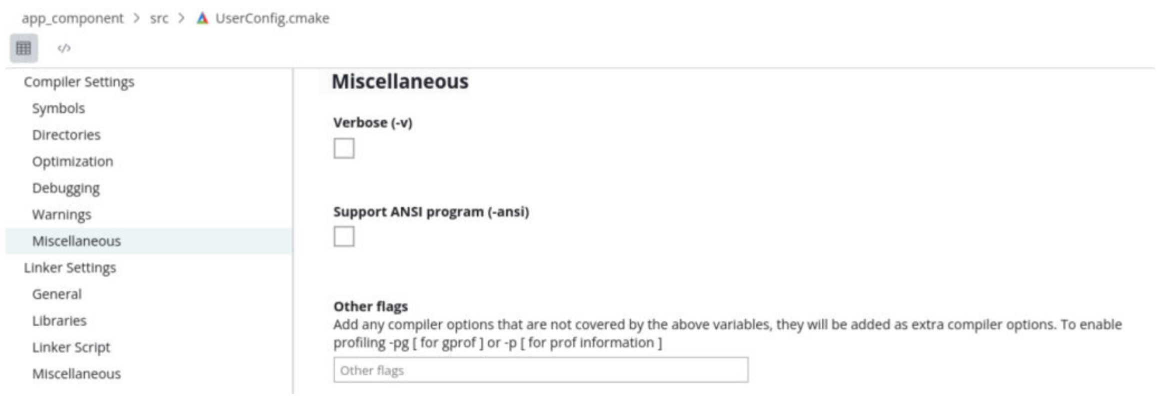
> ***Note:*** `UserConfig.cmake` supports GUI format and text format. You can click the **</>** icons to access the source editor to view the source code and modify the settings in text format.

Specifying 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.



Miscellaneous

Verbose (-v)
☐

Support ANSI program (-ansi)
☐

Other flags
Add any compiler options that are not covered by the above variables, they will be added as extra compiler options. To enable profiling -pg [for gprof] or -p [for prof information]

Other 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.

Linker Scripts

Compiling and linking are the two major steps of the application executable building process. Linker scripts that accepts linker command language files perform the linking. The linker script describes memory layout of the target machine and location of program sections 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*:**

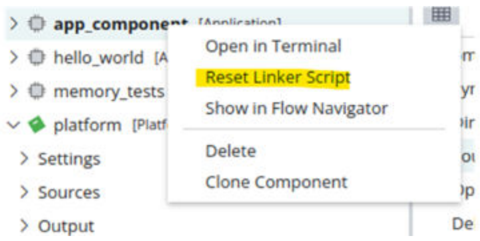
- For multiprocessor systems, each processor runs a different ELF file, and each ELF file requires its own linker script. Ensure that the two ELF files do not overlap in memory.
- The default linker always points to the DDR memory address available in memory. If you are creating an application under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

   Correct any errors before you 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:

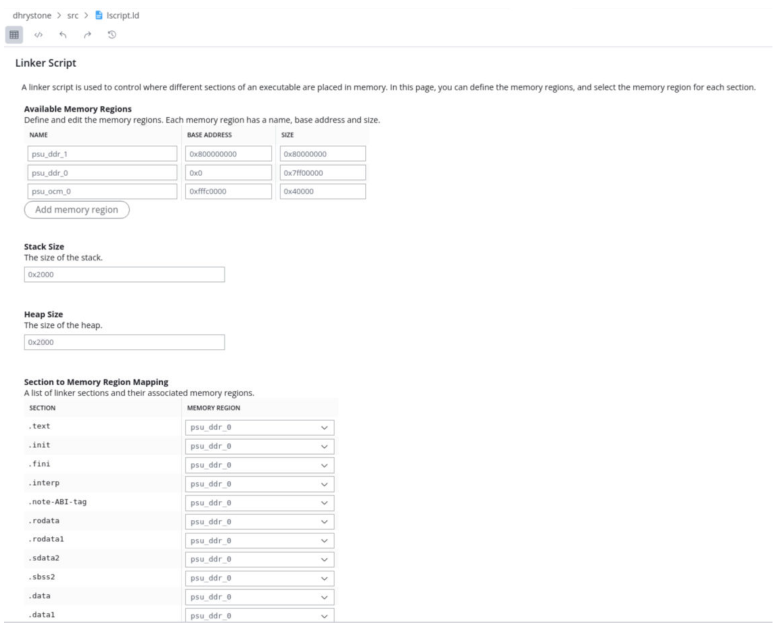
1. Select the application component in the Component View and then expand Settings. Right click **UserConfig.cmake** to open it.
2. Right click **Linker Script** to go to linker script setting section.
3. Click on **Browse** to select your own linker script file.

Modifying a Linker Script

There are multiple ways to update the linker script.

The source code of your application component contain the linker script. Update the script by executing the following steps.

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



dhrystone > src > lscript.ld

Linker Script

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define the memory regions, and select the memory region for each section.

Available Memory Regions
Define and edit the memory regions. Each memory region has a name, base address and size.

NAME	BASE ADDRESS	SIZE
psu_ddr_1	0x800000000	0x80000000
psu_ddr_0	0x0	0x7ff00000
psu_ocm_0	0xfffc0000	0x40000

Stack Size
The size of the stack.
0x2000

Heap Size
The size of the heap.
0x2000

Section to Memory Region Mapping
A list of linker sections and their associated memory regions.

SECTION	MEMORY REGION
.text	psu_ddr_0
.init	psu_ddr_0
.fini	psu_ddr_0
.interp	psu_ddr_0
.note-ABI-tag	psu_ddr_0
.rodata	psu_ddr_0
.rodata1	psu_ddr_0
.sdata2	psu_ddr_0
.sbss2	psu_ddr_0
.data	psu_ddr_0
.data1	psu_ddr_0

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.

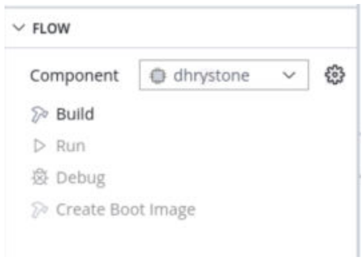
Table 8: **Linker Script Editor Functionality**

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.
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.

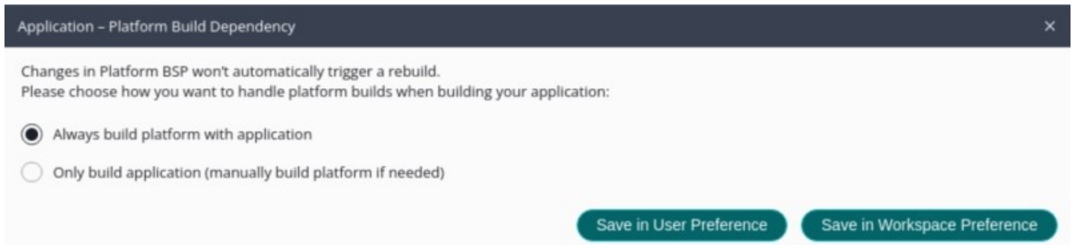
Building the Application Component

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


Application – Platform Build Dependency

Changes in Platform BSP won't automatically trigger a rebuild.
Please choose how you want to handle platform builds when building your application:

[✓ Always build platform with application]
Only build application (manually build platform if needed)

***Note*:** Changes you make to the platform BSP do not automatically trigger a rebuild. When you first build an application, go to workspace or preferences after the popup message displays. Select either "always build the platform before building the application" or "manually build the platform".

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

Reading ELF Disassembly

This task is for you to view the disassembling code.

You can retreive the application ELF file in the output directory after application compilation. You can view the disassemble view of the code by double clicking the **ELF file**.

Creating a Library Project

A collection of object files linking to a program during the compile time is a static library. Creating an application requires a platform.

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 creating a library component by right clicking the SRC folder and import the source.
5. Review the summary and click **Finish**.

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

Note: Vitis does not provide linker support for library component. 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.

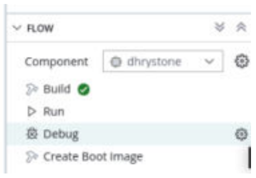
Chapter 7

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 37: **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.

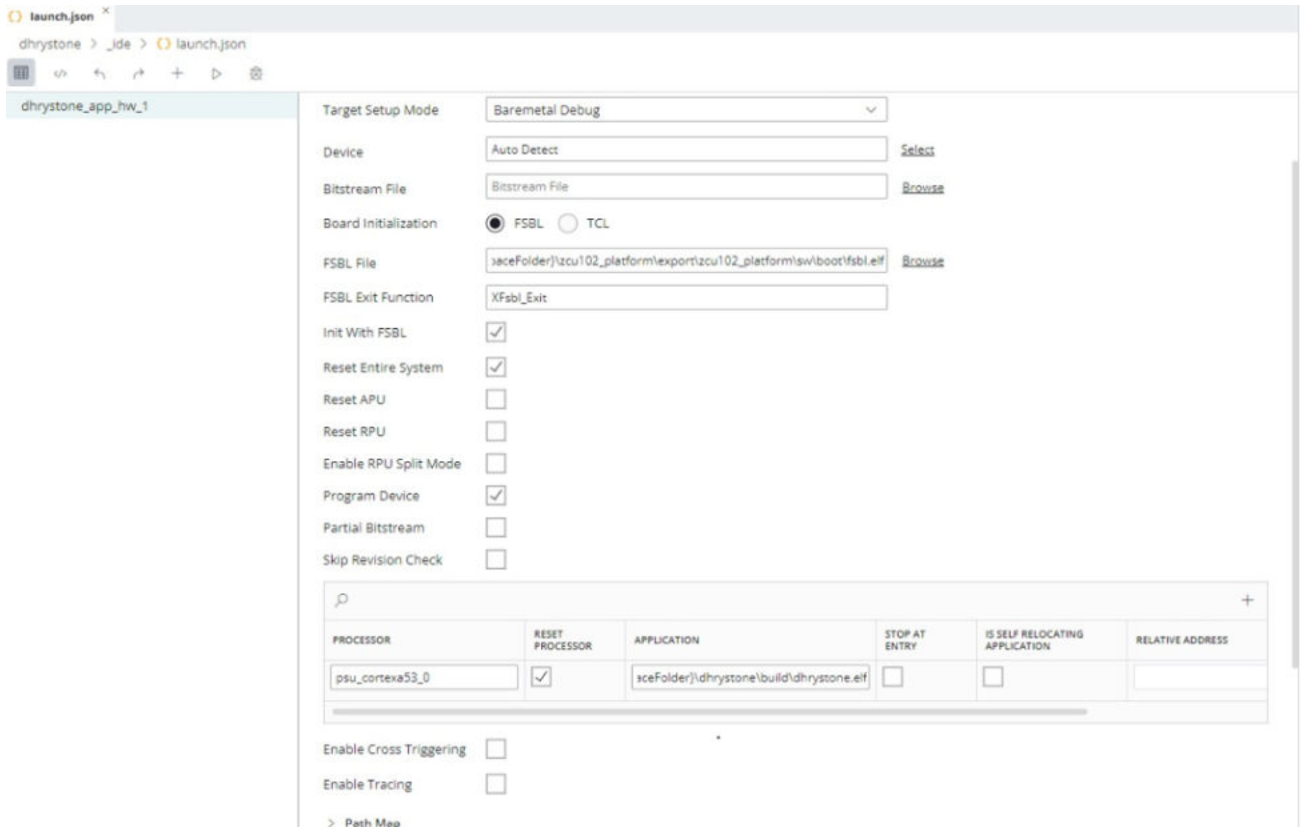
The `launch.json` file allows you to edit the launch configurations. The Launch Configuration editor has a toolbar menu at the top, as shown in the following figure. The specific contents of the launch configuration vary depending on the component or the project selected.

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 38: **Launch Configurations**



Target Setup Mode	Baremetal Debug
Device	Auto Detect
Bitstream File	Bitstream File
Board Initialization	[✓ FSBL] TCL
FSBL File	}aceFolder}\zcu102_platform\export\zcu102_platform\sw\boot\fsbl.elf
FSBL Exit Function	XFsbl_Exit
Init With FSBL	✓
Reset Entire System	✓
Reset APU	
Reset RPU	
Enable RPU Split Mode	
Program Device	✓
Partial Bitstream	
Skip Revision Check	

PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELATIVE ADDRESS
psu_cortexa53_0	✓	}ceFolder}\dhrystone\build\dhrystone.elf			

Enable Cross Triggering	
Enable Tracing	

> Path Map

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. You can also select Run or Debug from the Flow Navigator and specify a launch configuration if more than one is available.

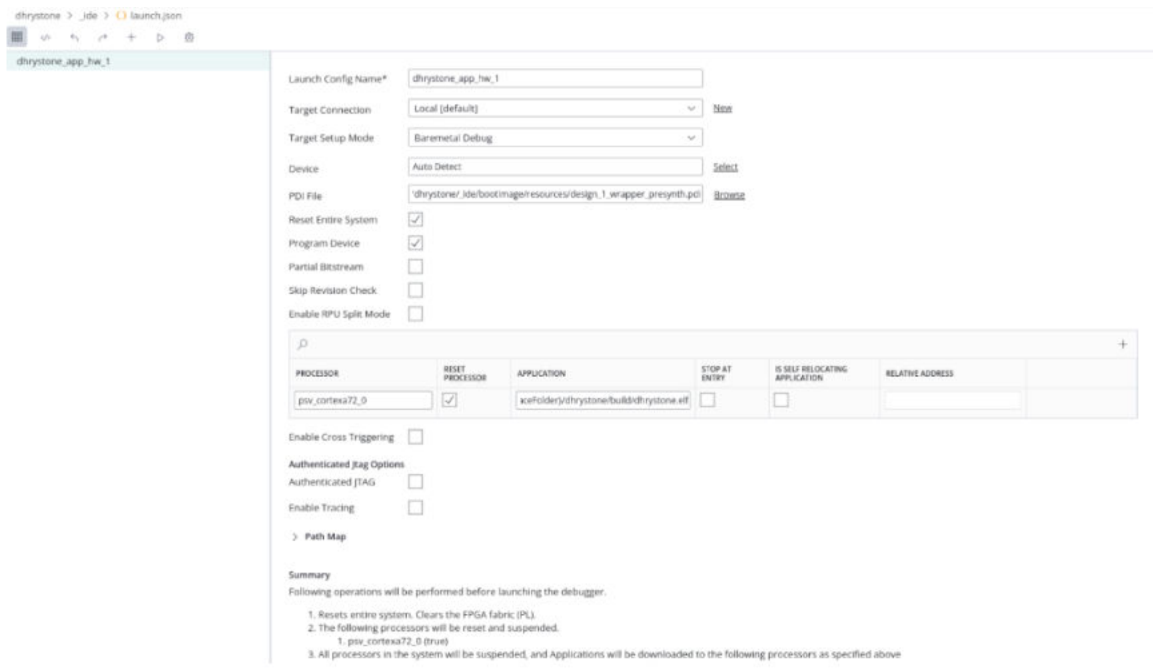
Main Page

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 are 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 39: **Launch Configuration Window**



Launch Config Name*	dhrystone_app_hw_1	
Target Connection	Local (default)	New
Target Setup Mode	Baremetal Debug	
Device	Auto Detect	Select
PDI File	'dhrystone/_ide/bootimage/resources/design_1_wrapper_presynth.pdi	Browse
Reset Entire System	✓	
Program Device	✓	
Partial Bitstream		
Skip Revision Check		
Enable RPU Split Mode		

PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELATIVE ADDRESS
psv_cortexa72_0	✓	xceFolder}/dhrystone/build/dhrystone.elf			

Enable Cross Triggering	

Authenticated Jtag Options

Authenticated JTAG	
Enable Tracing	

> **Path Map**

Summary

Following operations will be performed before launching the debugger.

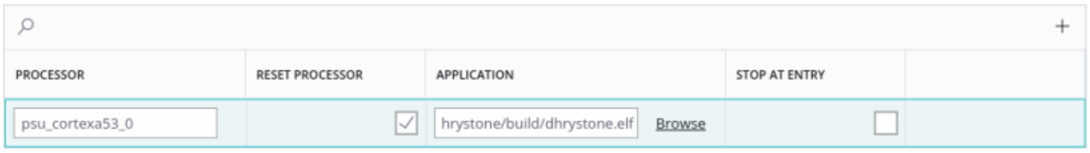
1. Resets entire system. Clears the FPGA fabric (PL).
2. The following processors will be reset and suspended.
    1. psv_cortexa72_0 (true)
3. All processors in the system will be suspended, and Applications will be downloaded to the following processors as specified above

***Note*:** The bottom of the screen displays a summary of the launch configuration.

Application Part

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

Figure 40: **Debug Confrontation Application Part**



PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY
psu_cortexa53_0	✓	hrystone/build/dhrystone.elf Browse	☐

- **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

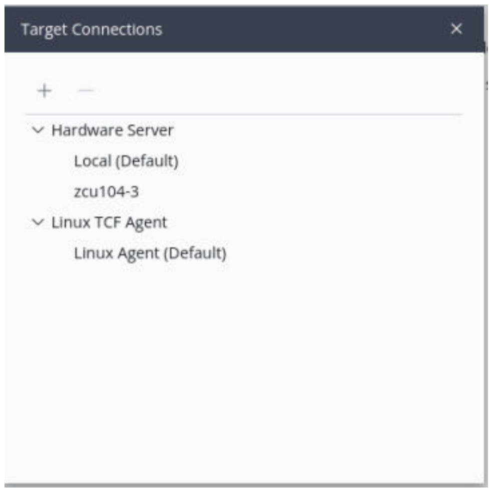
The Target Connections dialog box () allows you to configure multiple remote targets. It shows connected targets and gives you an option to add or delete target connections.

The Vitis software platform establishes target connections through the AMD `hw_server`. The hardware server agent must be running on the remote host connecting to the target hardware.

The target connection extends 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 41: **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 connecting 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 connecting to the target and the machine that `hw_server` is running on.
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 Vitis tool saves the frequency in the workspace and uses the frequency when executing a connect command for the device.
12. ***Note***: If the server connects to only one client, the frequency of the cable resets to the default value whenever the connection closes. Do not 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. You can confirm connections to the local board by checking the local connection.

To use a remote board to establish a target connection:

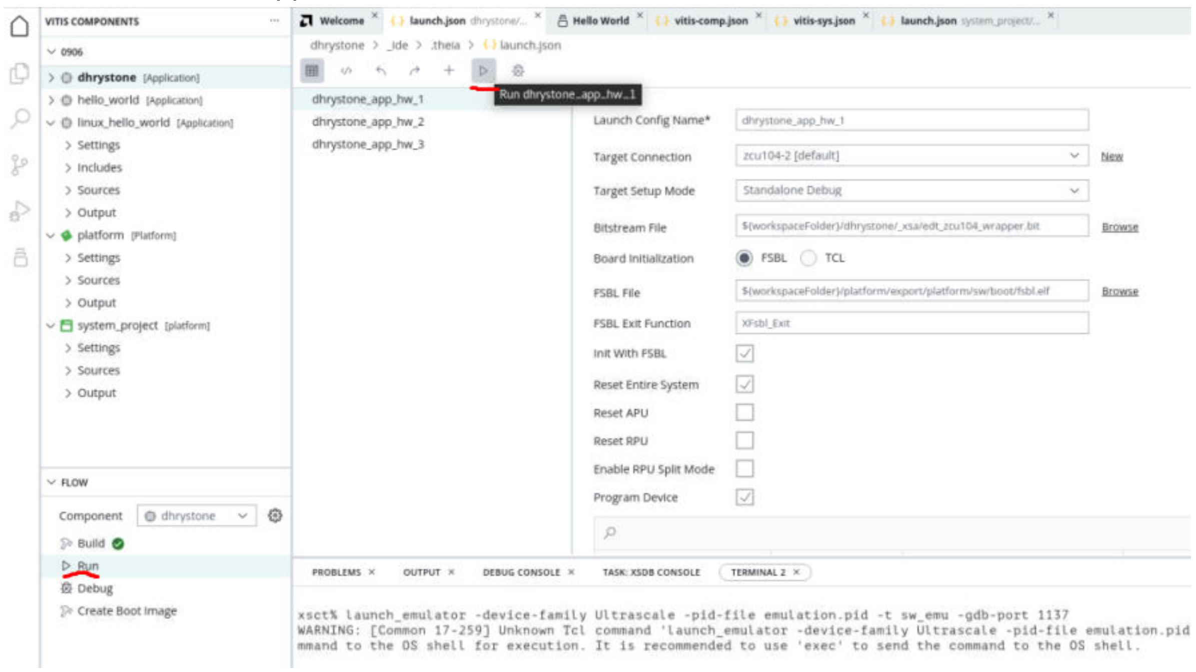
1. Ensure that the remote host connects to the target.
2. Launch the `hw_server` manually on the remote host:
    a. Take a shell on the remote host.
    b. Source the setup scripts by using `C:/Xilinx/<version>Vitis/settings64.bat` (or) `<Vitis_local_install_path>/<version>Vitis/settings64.csh`.
3.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

    Note: Ensure the remote host connects to the target (board).

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.



Launch Config Name*	dhrystone_app_hw_1
Target Connection	zcu104-2 [default]
Target Setup Mode	Standalone Debug
Bitstream File	${workspaceFolder}/dhrystone/_xsa/edt_zcu104_wrapper.bit
Board Initialization	[✓ FSBL] TCL
FSBL File	${workspaceFolder}/platform/export/platform/sw/boot/fsbl.elf
FSBL Exit Function	XFsbl_Exit
Init With FSBL	✓
Reset Entire System	✓
Reset APU	
Reset RPU	
Enable RPU Split Mode	
Program Device	✓

```
xsct% launch_emulator -device-family Ultrascale -pid-file emulation.pid -t sw_emu -gdb-port 1137
WARNING: [Common 17-259] Unknown Tcl command 'launch_emulator -device-family Ultrascale -pid-file emulation.pid
mmand to the OS shell for execution. It is recommended to use 'exec' to send the command to the OS shell.
```

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

Debugging Application Component

Debugging Zephyr Applications on Microblaze V

The Vitis Unified IDE enables you to debug Zephyr applications that run on MicroBlaze V processors using the User Managed Mode

For more details on User Managed Mode, see User Managed Mode.

The AMD Zephyr repo is available at: https://github.com/Xilinx/zephyr-amd.git

The following tutorial demonstrates how to debug a Zephyr application using the Vitis Unified IDE in user-managed mode.

Setting up the Zephyr Environment

1. First, follow the official zephyr guide to set up the Zephyr application development environment. See Zephyr Getting Started.
2. Replace the upstream repo you cloned down in the previous step with the AMD clone.

```
cd <local path>/zephyrproject/
mv zephyr zephyr.upstream
git clone https://github.com/Xilinx/zephyr-amd.git  -b xlnx_rel_v2025.1
zephyr
west update
west lopper-install
```

Creating the Hardware Design

Load Vivado and generate the hardware design (XSA) using the MicroBlaze V Design Preset based on Vivado CED, targeting the AC701 board. Alternatively, you can choose to create your own hardware design. For your own design, make sure that the your hardware design is compatible with the current release of the zephyr-amd repository.

Once your design has been finalized and you have exported the XSA file, you can start integrating with Zephyr.

Preset Design Examples

AMD Vivado offers preset design examples for various evaluation boards. Each embedded hardware design offers a subsystem that can be targeted for the real-time and microcontroller configurations. As of now, Zephyr is supported only for the real-time configuration.

The following sections demonstrates how to implement the MicroBlaze V Design Preset CED, targeting the AC701 board.

Generating the System Device Tree

Refer to the AMD Technical Information Portal for the steps to generate a System Device Tree (SDT) using a hardware design XSA. The SDT is mandatory to generate a Zephyr device tree and Kconfig for targeted hardware designs using the west lopper-command.

```
. /tools/AMD/Vivado/2025.2/settings64.sh
sdtgen sdt.tcl
```

Setting up the Zephyr Repository

Because each FPGA based hardware design can have different hardware configurations, you need to update the device tree (`boards/amd/<board_name>/<board_name>.dts`) based on the targeted hardware design. The base address and interrupt IDs can be different from what is configured in the default device tree. You need to either update .dts and kconfigs (`soc/xlnx/<board_name>/Kconfig, soc/xlnx/kconfig.defconfig`) manually based on the hardware design or use west lopper-command to set it up using lopper. Subsequent sections provide steps for setting up the DTS for a targeted hardware design using west lopper-command.

```
LOPPER_DTC_FLAGS="-b 0 -@" west lopper-command -p processor_instance_name
-s output_dir/system-top.dts -w <local path>/zephyrproject/zephyr
```

Example Usage

```
LOPPER_DTC_FLAGS="-b 0 -@" west lopper-command -p microblaze_riscv_0 -s
design_1_wrapper/system-top.dts -w ~/zephyrproject/zephyr
```

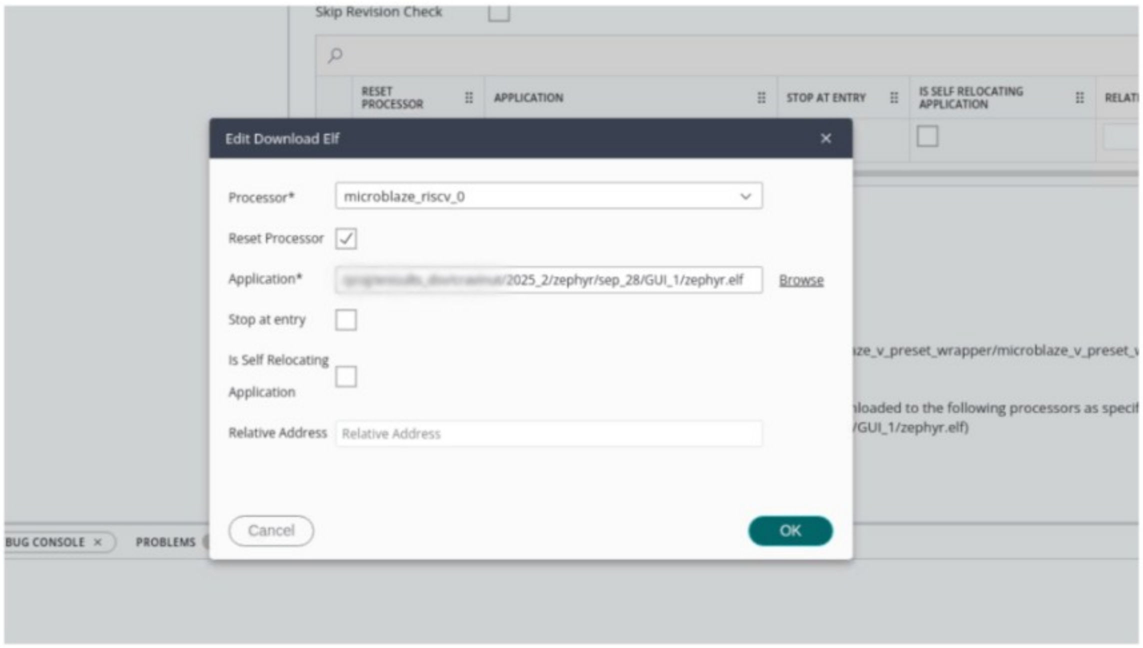
Once the lopper-command has been invoked, the Zephyr environment is ready to build kernel tests and applications for the targeted hardware design. Build a sample test by replacing board with your target using the commands below.

```
cd zephyr
west build -p -b board tests/misc/test_build/
# The executable would be generated in following location
<zephyr repository path>/build/zephyr/zephyr.elf
west build -p -b mbv32 tests/misc/test_build/
```

Creating The Launch Configuration

Load Vitis Unified Software Platform, 2025.2 version and open a new folder. The folder opens Vitis in User Managed Mode. Create a new launch configuration by right clicking the Explorer view and selecting **Edit Launch Configurations** and then **Add Configuration**.

Set the target processor to `microblaze_riscv_0` and provide the path to the `zephyr.elf` application file.

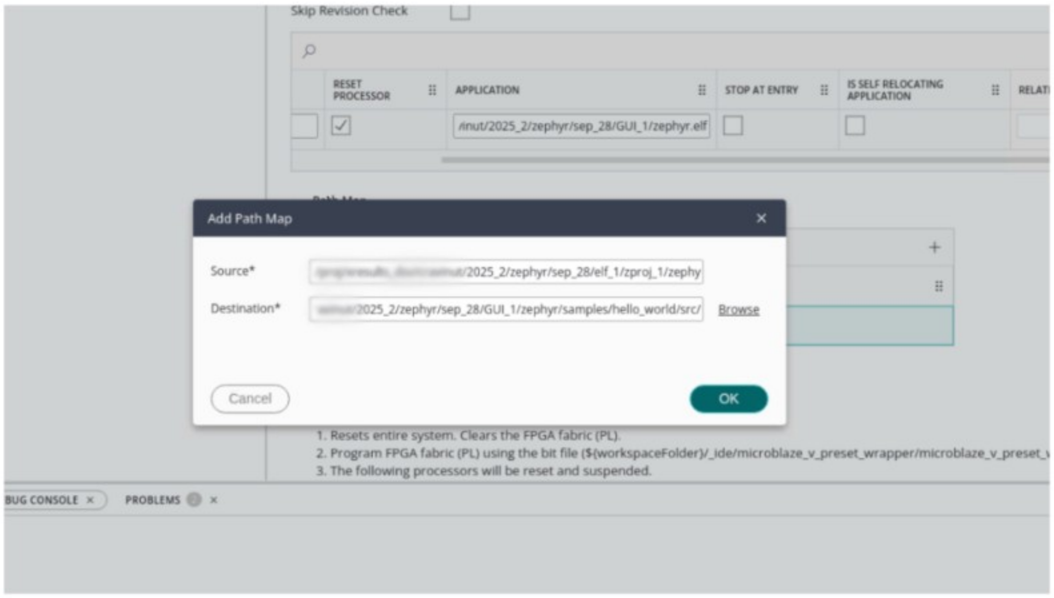


Processor*: microblaze_riscv_0
Reset Processor: ✓
Application*: [illegible]2025_2/zephyr/sep_28/GUI_1/zephyr.elf
Stop at entry: ☐
Is Self Relocating Application: ☐
Relative Address: Relative Address

ize_v_preset_wrapper/microblaze_v_preset_
nloaded to the following processors as specif
/GUI_1/zephyr.elf)

If the application source path has changed or ELF location is not within your workspace, you need to remap the original path of the source file and the destination path of the source code.

Figure 42: **Add Path Map**



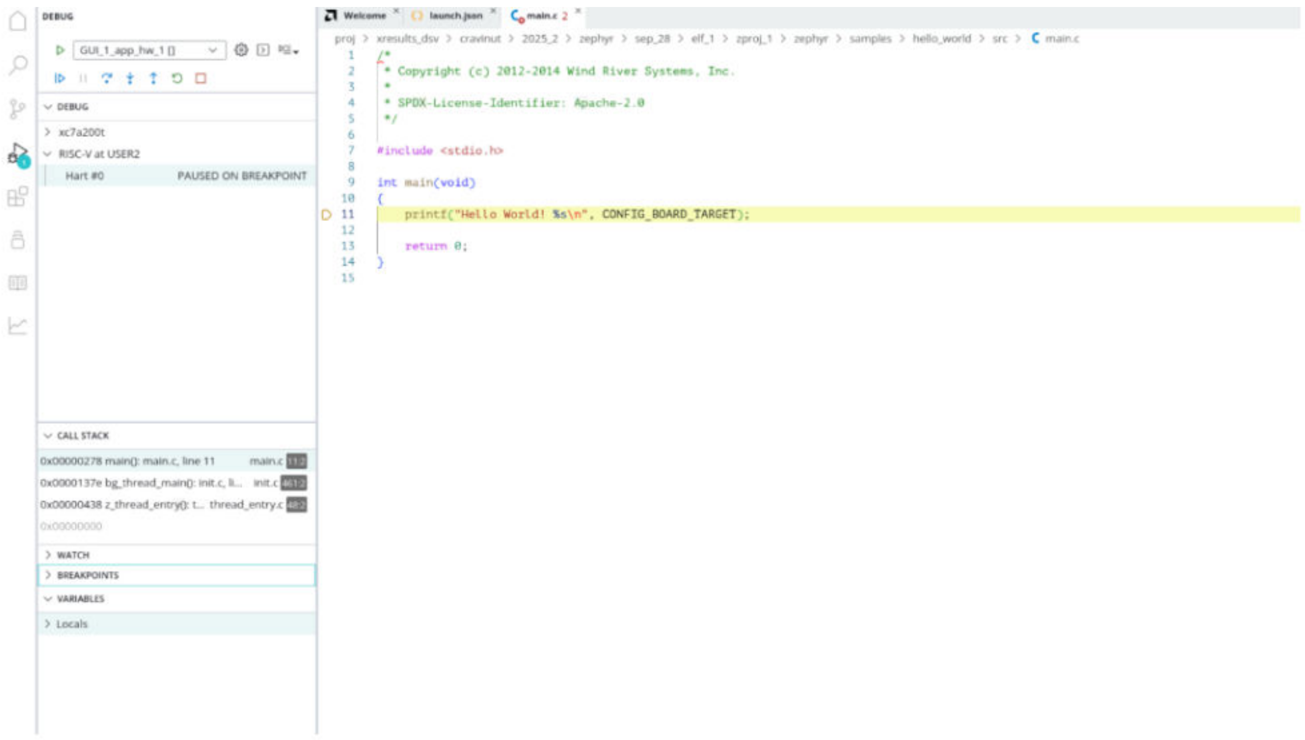
/inut/2025_2/zephyr/sep_28/GUI_1/zephyr.elf

Source*: [illegible]2025_2/zephyr/sep_28/elf_1/zproj_1/zephy
Destination*: [illegible]2025_2/zephyr/sep_28/GUI_1/zephyr/samples/hello_world/src/

1. Resets entire system. Clears the FPGA fabric (PL).
2. Program FPGA fabric (PL) using the bit file (${workspaceFolder}/_ide/microblaze_v_preset_wrapper/microblaze_v_preset_
3. The following processors will be reset and suspended.

Launch the debug session. The `main.c` file loads in the debug perspective.

Figure 43: **main.c Launching in Debug Perspective**



```
0x00000278 main(): main.c, line 11        main.c 11:2
0x0000137e bg_thread_main(): init.c, li...  init.c 461:2
0x00000438 z_thread_entry(): t...  thread_entry.c 48:2
0x00000000
```

```
/*
 * Copyright (c) 2012-2014 Wind River Systems, Inc.
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <stdio.h>

int main(void)
{
	printf("Hello World! %s\n", CONFIG_BOARD_TARGET);

	return 0;
}
```

PS Trace

The PS Trace tool monitors every assembly instruction execution of the CPU (ARMv8). The tool provides insights for online diagnosis, performance debugging and post run data analysis. PS Trace offers granular insight into code execution. During Vitis debug sessions, you can see exactly what the CPU is doing instruction by instruction, not just what the source code says.

You can view and analyze trace data from a number of supported processors:

- Arm® Cortex-A53
- Arm R52
- Arm A78
- Arm R5
- Arm R52
- Arm A72

OpenCSD Setup

OpenCSD is an open source CoreSight Trace Decode library. This library provides an API that facilitates the decode of Arm CoreSight trace streams. The PS Trace functionality uses the OpenCSD library to decode the CoreSight trace data that comes from the Arm processor. To access the trace file, you must first install OpenCSD to employ its library for decoding the trace binary into a human-readable format file. Follow the steps below to install the OpenCSD library.

1. Set the 'INSTALL_PATH' environment variable for specifying the path to install the compiled OpenCSD files.

   ```
   export INSTALL_PATH="/path/to/install/"
   ```

2. Clone the OpenCSD GitHub repository to your local host.

   ```
   git clone https://github.com/Linaro/OpenCSD.git
   ```

3. Install OpenCSD.

   ```
   cd OpenCSD/decoder/build/linux/
   make install PREFIX=$INSTALL_PATH
   ```

4. Lastly, set the OpenCSD environment variable for the Vitis tool to use the OpenCSD library. To do this use the following command before launching Vitis:

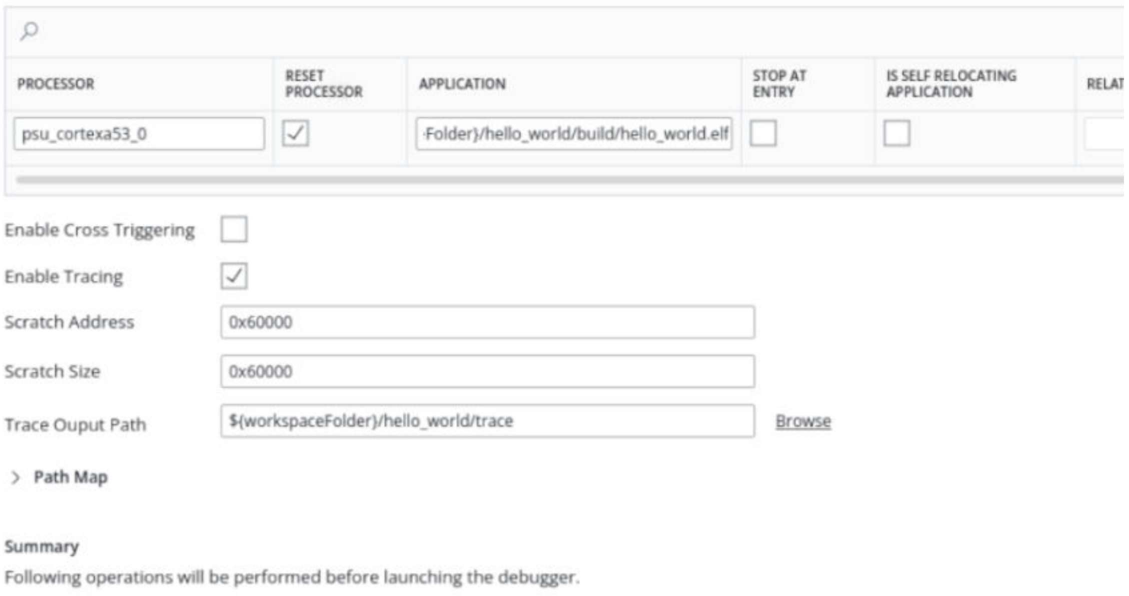
   ```
   export OPENCSD_PATH=$INSTALL_PATH
   ```

Performing PS Trace

Step 1: Launch Configuration Setup

Launch the debug configuration (`launch.json`) for your application. Enable the tracing option and set the trace memory related information.

Figure 44: **Debug Configuration**



PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELAT
psu_cortexa53_0	✓	Folder}/hello_world/build/hello_world.elf	☐	☐	

Enable Cross Triggering	☐
Enable Tracing	✓
Scratch Address	0x60000
Scratch Size	0x60000
Trace Ouput Path	${workspaceFolder}/hello_world/trace

Summary
Following operations will be performed before launching the debugger.

The following are the items on screen:

- Scratch Address: Starting address in memory for storing the trace binary file.
- Trace Decoded File: Memory length for storing the trace binary file.
- Output File path: Location for storing the trace summary file generated from the trace binary file by the Vitis tool.

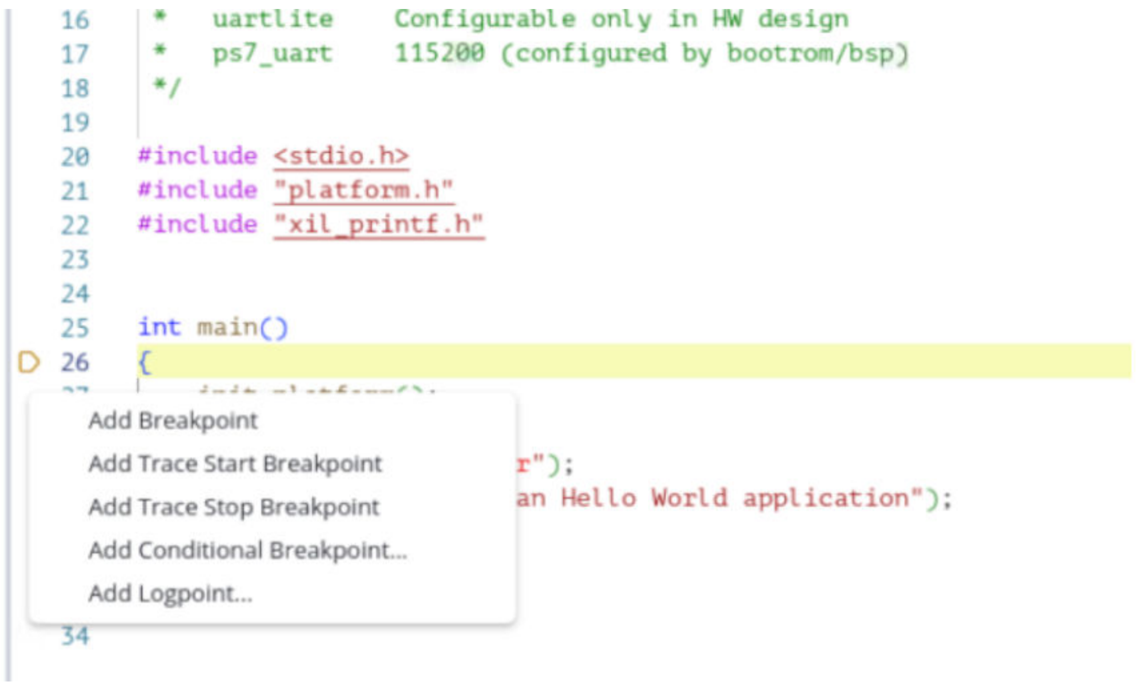
IMPORTANT! *This is a ring buffer. Ensure the buffer size is set appropriately; if the trace data exceeds this size, the new data will overwrite the existing buffer contents.*

Step 2: Adding breakpoints

PS Trace involves the placement of a trace start and trace start breakpoint. The code section you place these breakpoints around is the section of code to be traced. Confirm the lines of code you want to trace before setting the breakpoints.

Start the debug session and set a trace start breakpoint in the Debug view. To add a trace start breakpoint, right-click the line number containing the first instructions you want to trace in the source code view.

Figure 45: **Add Breakpoints**



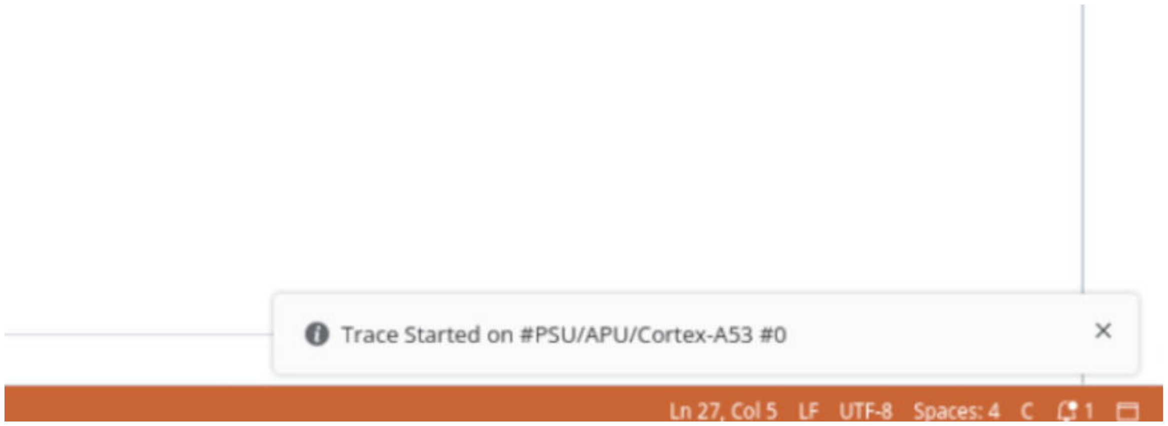
```
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
```

Add Breakpoint
Add Trace Start Breakpoint
Add Trace Stop Breakpoint
Add Conditional Breakpoint...
Add Logpoint...

Similarly, to add a trace stop breakpoint, right-click the line number in the source code view where the instructions you want to stop tracing are located.

Click **Continue** to start the debug session. If the trace start breakpoint is reached, wait for a notification in the bottom-right corner confirming the trace has started. Next, click **Continue** in the Debug view to trigger the trace.

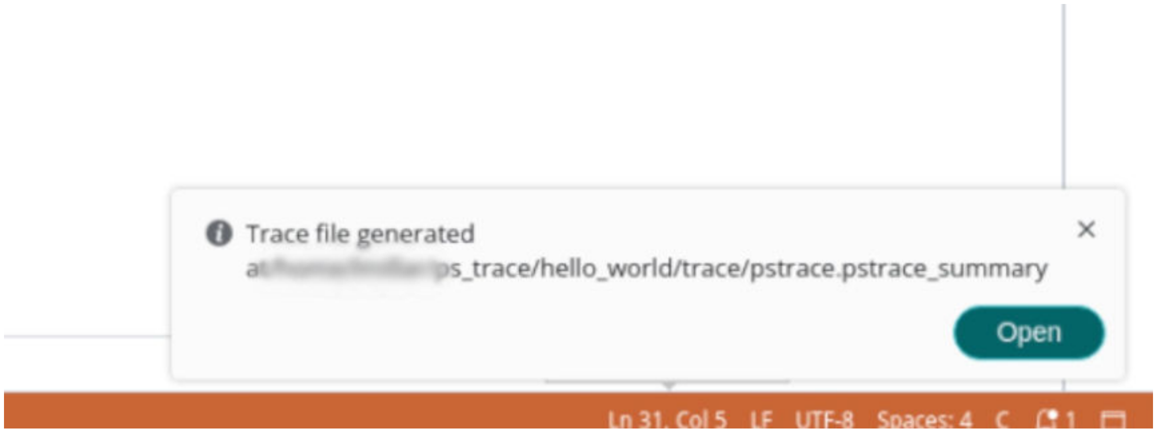
Figure 46: **Trace Start**



Trace Started on #PSU/APU/Cortex-A53 #0

After hitting the trace stop breakpoint, wait for the notification in the bottom-right corner indicating the generation of the trace summary. Click **Open** to view the trace file. You can also go to **Vitis→PS Trace→Open PS Trace** and navigate to the folder you set to store the trace file.

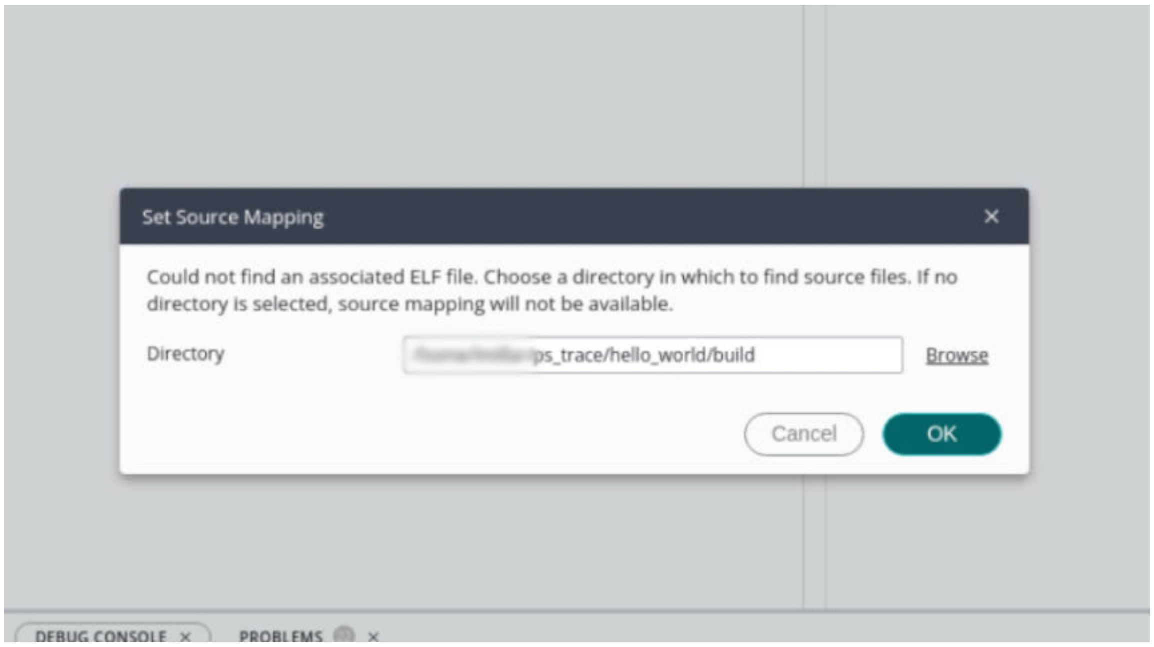
Figure 47: **Trace File Generated**



Trace file generated
at ps_trace/hello_world/trace/pstrace.pstrace_summary

Click Browse in the Set Source Mapping dialog box and navigate to the folder where the ELF file is located.

Figure 48: **Set Source Mapping**



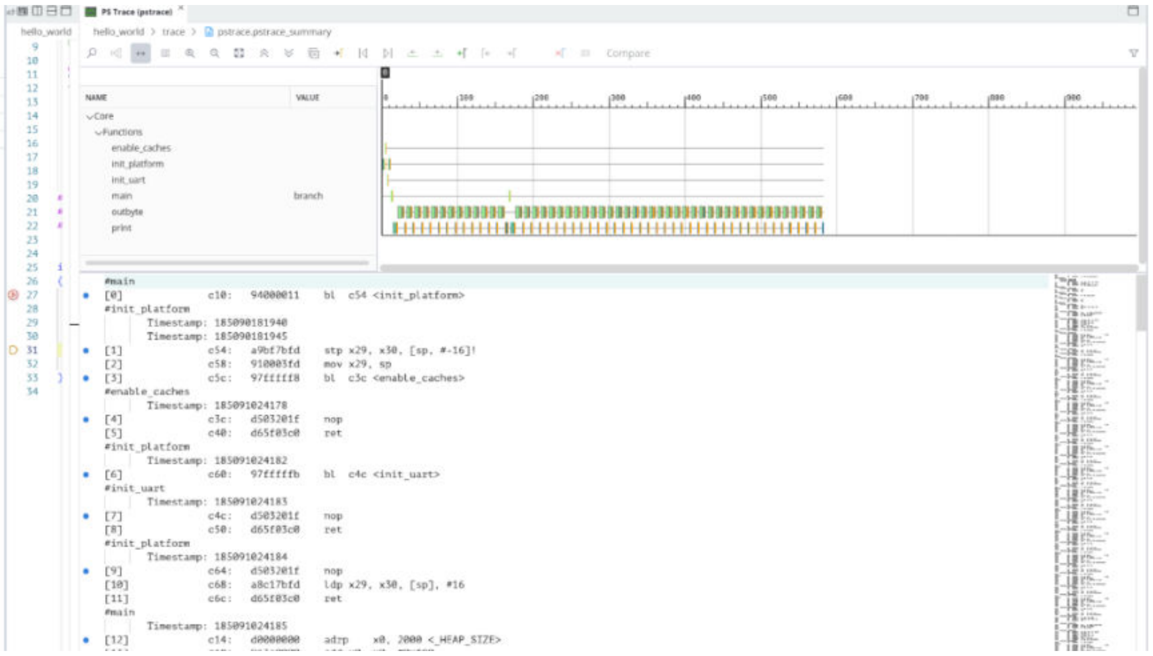
Set Source Mapping

Could not find an associated ELF file. Choose a directory in which to find source files. If no directory is selected, source mapping will not be available.

Directory ps_trace/hello_world/build

Click Ok. The system displays the Trace File screen.

Figure 49: **Trace File**



```
#main
[0]            c10:   94000011    bl  c54 <init_platform>
#init_platform
        Timestamp: 185090181940
        Timestamp: 185090181945
[1]            c54:   a9bf7bfd    stp x29, x30, [sp, #-16]!
[2]            c58:   910003fd    mov x29, sp
[3]            c5c:   97fffff8    bl  c3c <enable_caches>
#enable_caches
        Timestamp: 185091024178
[4]            c3c:   d503201f    nop
[5]            c40:   d65f03c0    ret
#init_platform
        Timestamp: 185091024182
[6]            c60:   97fffffb    bl  c4c <init_uart>
#init_uart
        Timestamp: 185091024183
[7]            c4c:   d503201f    nop
[8]            c50:   d65f03c0    ret
#init_platform
        Timestamp: 185091024184
[9]            c64:   d503201f    nop
[10]           c68:   a8c17bfd    ldp x29, x30, [sp], #16
[11]           c6c:   d65f03c0    ret
#main
        Timestamp: 185091024185
[12]           c14:   d0000000    adrp    x0, 2000 <_HEAP_SIZE>
```

In the Trace File view, you can perform operations to analyze the trace report effectively.

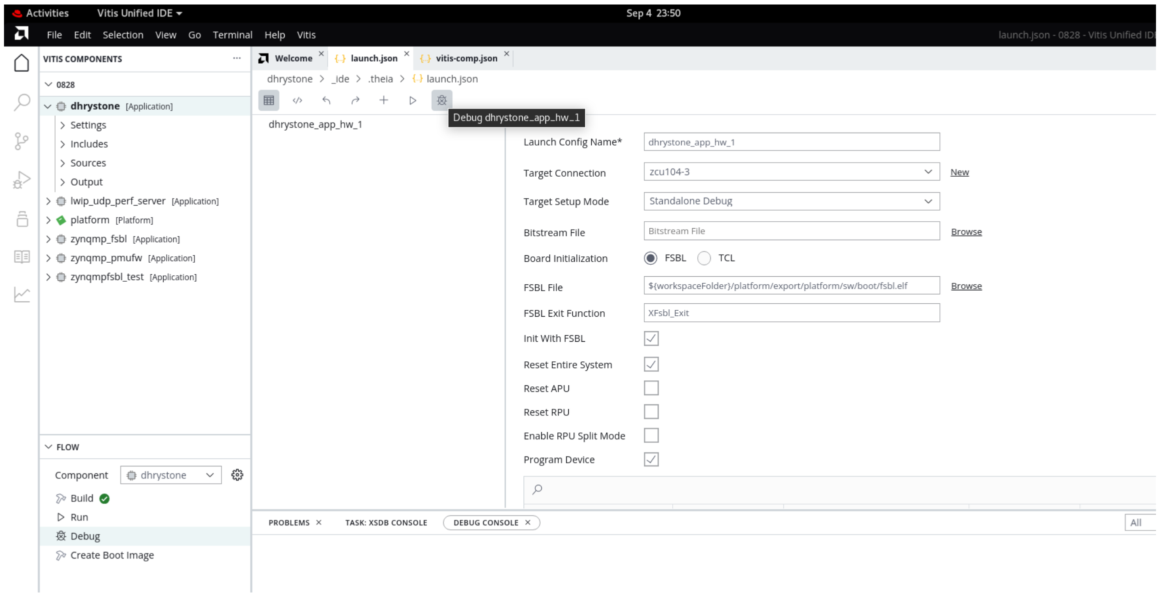
Starting Debug

The following chapters the steps to start the debug process.

Debugging Standalone Application Component

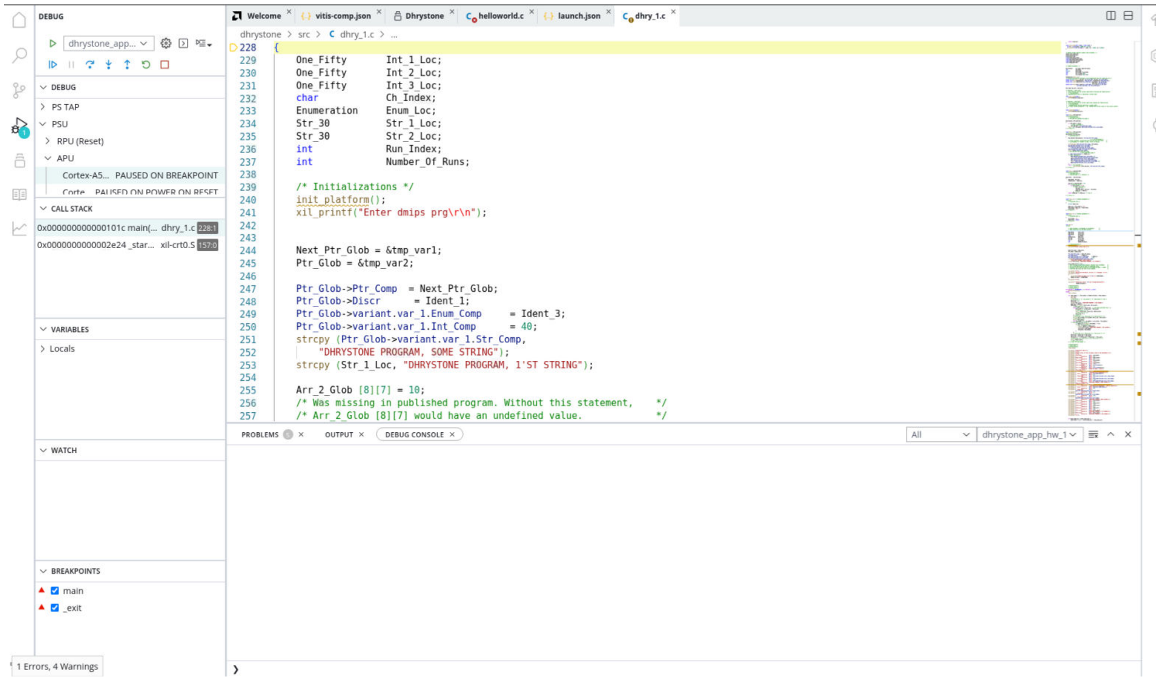
This topic describes how to use the Debugger to debug standalone applications.

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.



Launch Config Name*: dhrystone_app_hw_1
Target Connection: zcu104-3
Target Setup Mode: Standalone Debug
Bitstream File: Bitstream File
Board Initialization: [✓ FSBL] TCL
FSBL File: ${workspaceFolder}/platform/export/platform/sw/boot/fsbl.elf
FSBL Exit Function: XFsbl_Exit
Init With FSBL: ✓
Reset Entire System: ✓
Reset APU: ☐
Reset RPU: ☐
Enable RPU Split Mode: ☐
Program Device: ✓

Figure 50: **Debug View**



```
{
    One_Fifty       Int_1_Loc;
    One_Fifty       Int_2_Loc;
    One_Fifty       Int_3_Loc;
    char            Ch_Index;
    Enumeration     Enum_Loc;
    Str_30          Str_1_Loc;
    Str_30          Str_2_Loc;
    int             Run_Index;
    int             Number_Of_Runs;

    /* Initializations */
    init_platform();
    xil_printf("Enter dmips prg\r\n");


    Next_Ptr_Glob = &tmp_var1;
    Ptr_Glob = &tmp_var2;

    Ptr_Glob->Ptr_Comp  = Next_Ptr_Glob;
    Ptr_Glob->Discr       = Ident_1;
    Ptr_Glob->variant.var_1.Enum_Comp     = Ident_3;
    Ptr_Glob->variant.var_1.Int_Comp      = 40;
    strcpy (Ptr_Glob->variant.var_1.Str_Comp,
        "DHRYSTONE PROGRAM, SOME STRING");
    strcpy (Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");

    Arr_2_Glob [8][7] = 10;
    /* Was missing in published program. Without this statement,    */
    /* Arr_2_Glob [8][7] would have an undefined value.             */
```

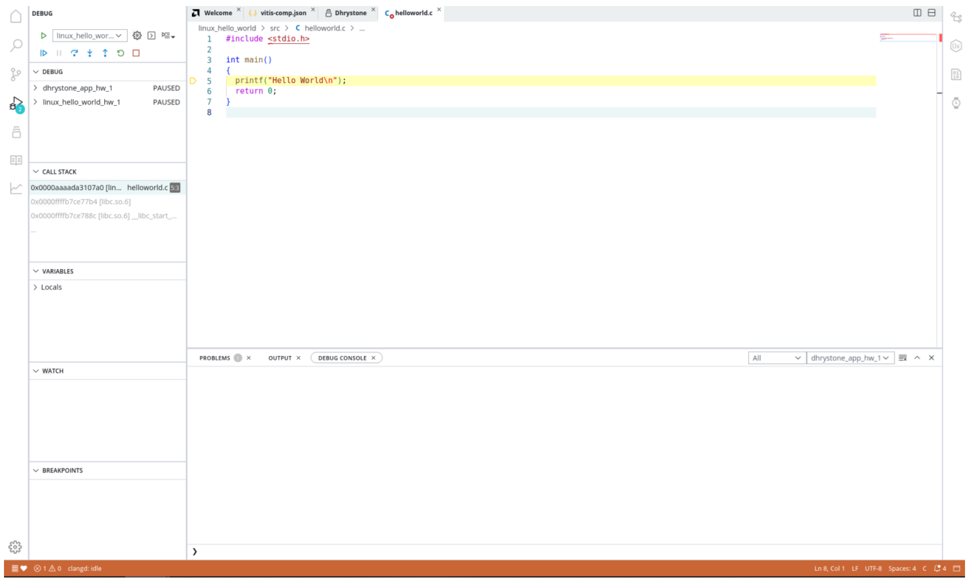
Debugging Linux Application Component

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

1. Go to the Flow Navigator and select the Linux application component you need 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.

Figure 51: **Debug View**


DEBUG
linux_hello_wor...
DEBUG
dhrystone_app_hw_1 PAUSED
linux_hello_world_hw_1 PAUSED
CALL STACK
0x0000aaaada3107a0 [lin... helloworld.c 5:3
0x0000ffffb7ce77b4 [libc.so.6]
0x0000ffffb7ce788c [libc.so.6] __libc_start_...
VARIABLES
Locals
WATCH
BREAKPOINTS
Welcome
vitis-comp.json
Dhrystone
helloworld.c
linux_hello_world > src > helloworld.c > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
All
dhrystone_app_hw_1
clangd: idle
Ln 8, Col 1 LF UTF-8 Spaces: 4 C

```c
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

Debugging Standalone Application Component on QEMU

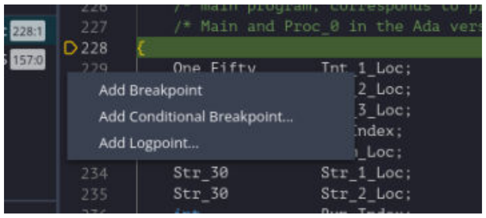
This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

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**.



```
/* Main and Proc_0 in the Ada vers
{
   One_Fifty      Int_1_Loc;
                  2_Loc;
                  3_Loc;
                  ndex;
                  _Loc;
   Str_30         Str_1_Loc;
   Str_30         Str_2_Loc;
```

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

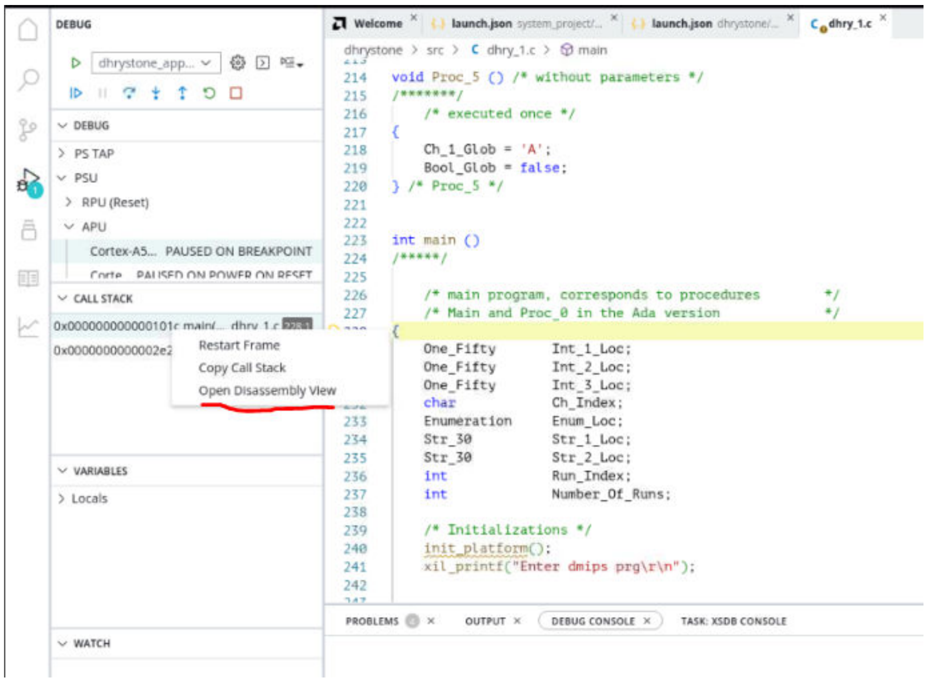
To change a normal breakpoint to conditional breakpoints, follow steps outlined below.

- 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 starts.

Figure 52: **Viewing Disassembly Code**



```
void Proc_5 () /* without parameters */
/*******/
    /* executed once */
{
    Ch_1_Glob = 'A';
    Bool_Glob = false;
} /* Proc_5 */


int main ()
/*****/

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

    /* Initializations */
    init_platform();
    xil_printf("Enter dmips prg\r\n");
```

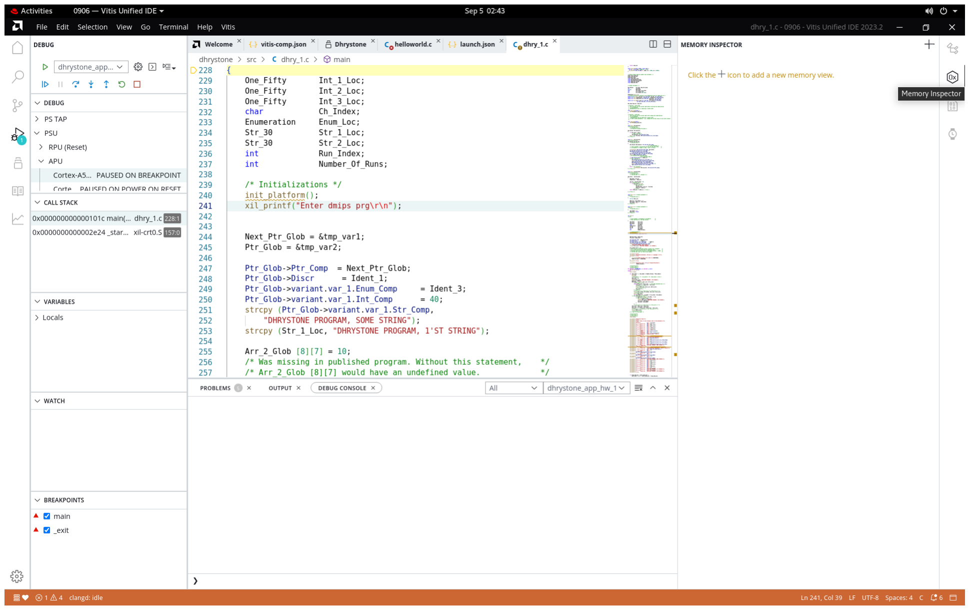
Next, select the Open Disassembly to view disassembly code is displayed.

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.



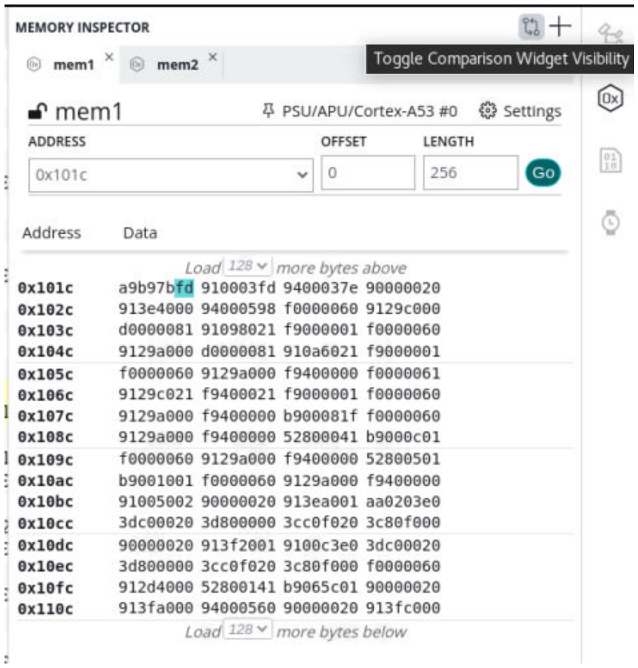
```
{
    One_Fifty       Int_1_Loc;
    One_Fifty       Int_2_Loc;
    One_Fifty       Int_3_Loc;
    char            Ch_Index;
    Enumeration     Enum_Loc;
    Str_30          Str_1_Loc;
    Str_30          Str_2_Loc;
    int             Run_Index;
    int             Number_Of_Runs;

    /* Initializations */
    init_platform();
    xil_printf("Enter dmips prg\r\n");


    Next_Ptr_Glob = &tmp_var1;
    Ptr_Glob = &tmp_var2;

    Ptr_Glob->Ptr_Comp  = Next_Ptr_Glob;
    Ptr_Glob->Discr       = Ident_1;
    Ptr_Glob->variant.var_1.Enum_Comp     = Ident_3;
    Ptr_Glob->variant.var_1.Int_Comp      = 40;
    strcpy (Ptr_Glob->variant.var_1.Str_Comp,
        "DHRYSTONE PROGRAM, SOME STRING");
    strcpy (Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");

    Arr_2_Glob [8][7] = 10;
    /* Was missing in published program. Without this statement,    */
    /* Arr_2_Glob [8][7] would have an undefined value.             */
```

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

Comparing the Memory Value of Two Addresses

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

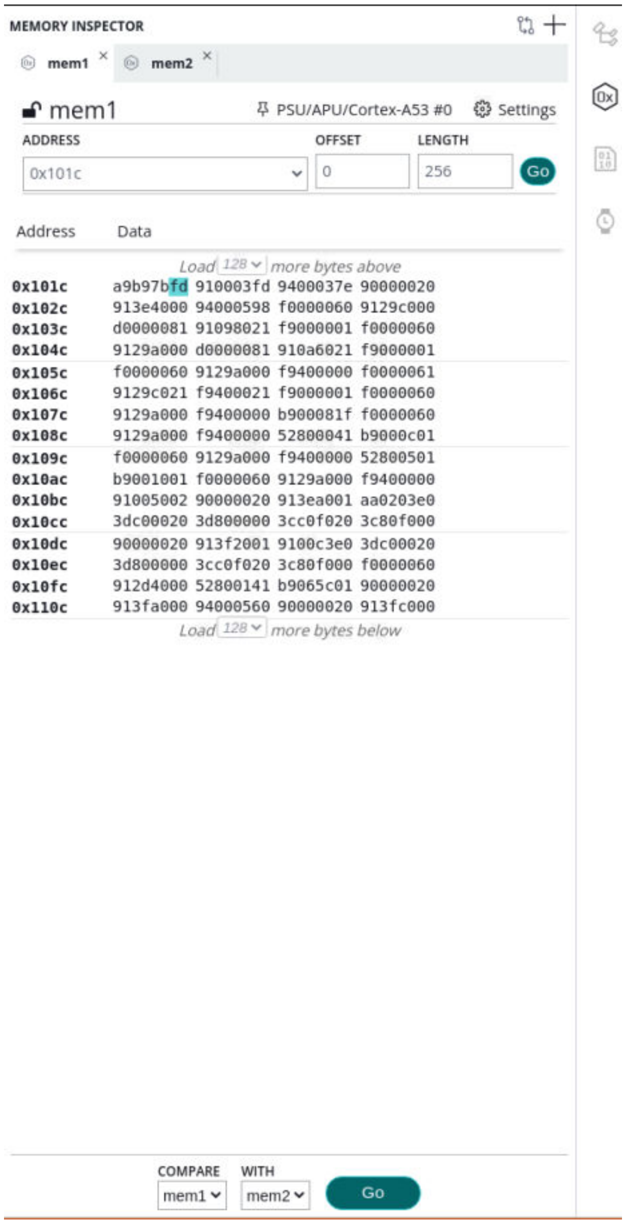
1. Refer to Viewing the Value of a Certain Memory address to create two memory inspectors. Set the difference of the two inspectors.



```
0x101c   a9b97bfd 910003fd 9400037e 90000020
0x102c   913e4000 94000598 f0000060 9129c000
0x103c   d0000081 91098021 f9000001 f0000060
0x104c   9129a000 d0000081 910a6021 f9000001
0x105c   f0000060 9129a000 f9400000 f0000061
0x106c   9129c021 f9400021 f9000001 f0000060
0x107c   9129a000 f9400000 b900081f f0000060
0x108c   9129a000 f9400000 52800041 b9000c01
0x109c   f0000060 9129a000 f9400000 52800501
0x10ac   b9001001 f0000060 9129a000 f9400000
0x10bc   91005002 90000020 913ea001 aa0203e0
0x10cc   3dc00020 3d800000 3cc0f020 3c80f000
0x10dc   90000020 913f2001 9100c3e0 3dc00020
0x10ec   3d800000 3cc0f020 3c80f000 f0000060
0x10fc   912d4000 52800141 b9065c01 90000020
0x110c   913fa000 94000560 90000020 913fc000
```

2. Click **Toggle Comparison Widget Visibility** to compare the difference.

3. Select the memory you want to compare and click **Go**.



ADDRESS: 0x101c OFFSET: 0 LENGTH: 256

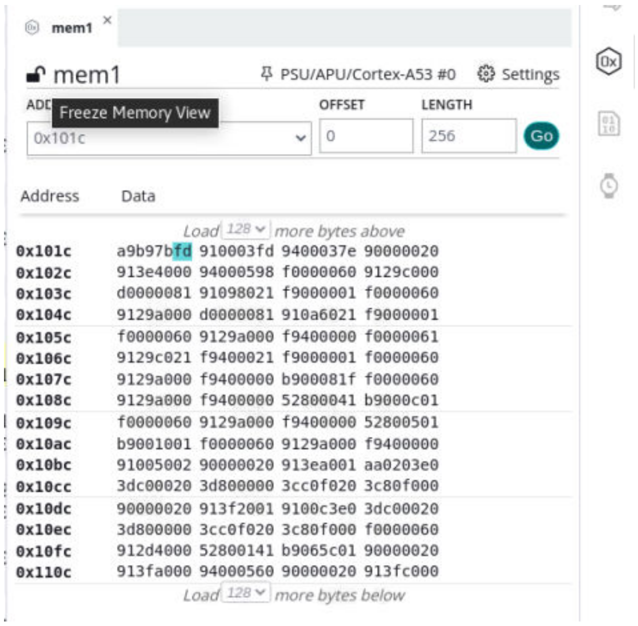
```
Address     Data
                  Load 128 more bytes above
0x101c      a9b97bfd 910003fd 9400037e 90000020
0x102c      913e4000 94000598 f0000060 9129c000
0x103c      d0000081 91098021 f9000001 f0000060
0x104c      9129a000 d0000081 910a6021 f9000001
0x105c      f0000060 9129a000 f9400000 f0000061
0x106c      9129c021 f9400021 f9000001 f0000060
0x107c      9129a000 f9400000 b900081f f0000060
0x108c      9129a000 f9400000 52800041 b9000c01
0x109c      f0000060 9129a000 f9400000 52800501
0x10ac      b9001001 f0000060 9129a000 f9400000
0x10bc      91005002 90000020 913ea001 aa0203e0
0x10cc      3dc00020 3d800000 3cc0f020 3c80f000
0x10dc      90000020 913f2001 9100c3e0 3dc00020
0x10ec      3d800000 3cc0f020 3c80f000 f0000060
0x10fc      912d4000 52800141 b9065c01 90000020
0x110c      913fa000 94000560 90000020 913fc000
                  Load 128 more bytes below
```

COMPARE: mem1 WITH: mem2

Freezing Memory Value

This feature keeps a snapshot of the memory contents until the Vitis tool closes. 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.



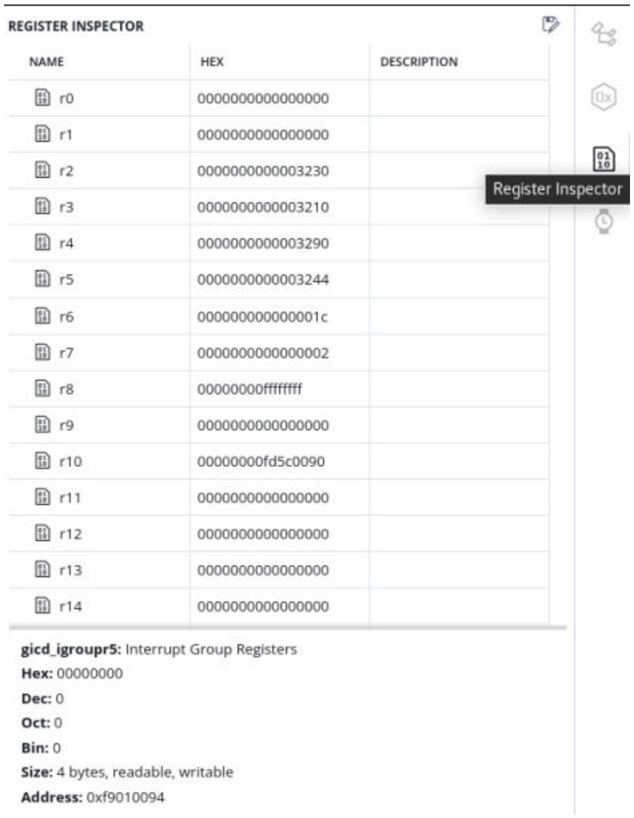
```
0x101c   0
256

Address      Data
                 Load 128 more bytes above
0x101c       a9b97bfd 910003fd 9400037e 90000020
0x102c       913e4000 94000598 f0000060 9129c000
0x103c       d0000081 91098021 f9000001 f0000060
0x104c       9129a000 d0000081 910a6021 f9000001
0x105c       f0000060 9129a000 f9400000 f0000061
0x106c       9129c021 f9400021 f9000001 f0000060
0x107c       9129a000 f9400000 b900081f f0000060
0x108c       9129a000 f9400000 52800041 b9000c01
0x109c       f0000060 9129a000 f9400000 52800501
0x10ac       b9001001 f0000060 9129a000 f9400000
0x10bc       91005002 90000020 913ea001 aa0203e0
0x10cc       3dc00020 3d800000 3cc0f020 3c80f000
0x10dc       90000020 913f2001 9100c3e0 3dc00020
0x10ec       3d800000 3cc0f020 3c80f000 f0000060
0x10fc       912d4000 52800141 b9065c01 90000020
0x110c       913fa000 94000560 90000020 913fc000
                 Load 128 more bytes below
```

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 you select an APU target.

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

Figure 53: **Register Inspector Window**



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

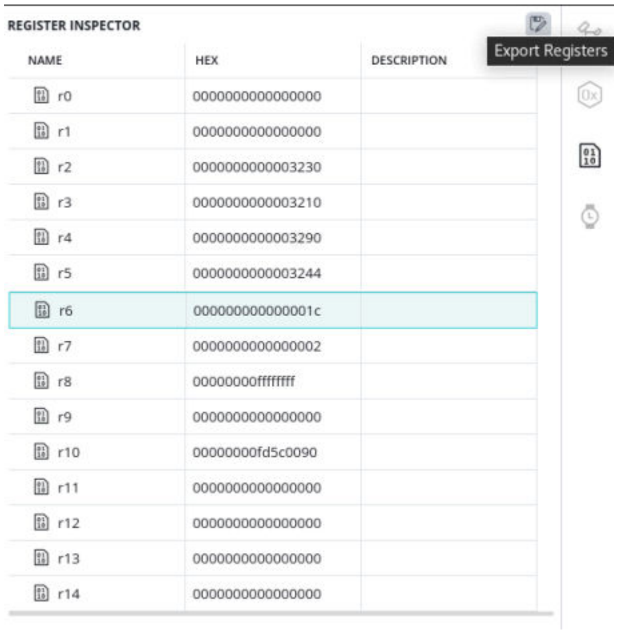
gicd_igroupr5: Interrupt Group Registers
Hex: 00000000
Dec: 0
Oct: 0
Bin: 0
Size: 4 bytes, readable, writable
Address: 0xf9010094

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

Exporting Registers from the Vitis IDE

This feature allows you to export the registers present in a target processor to a text file and read all the register values more easily.

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



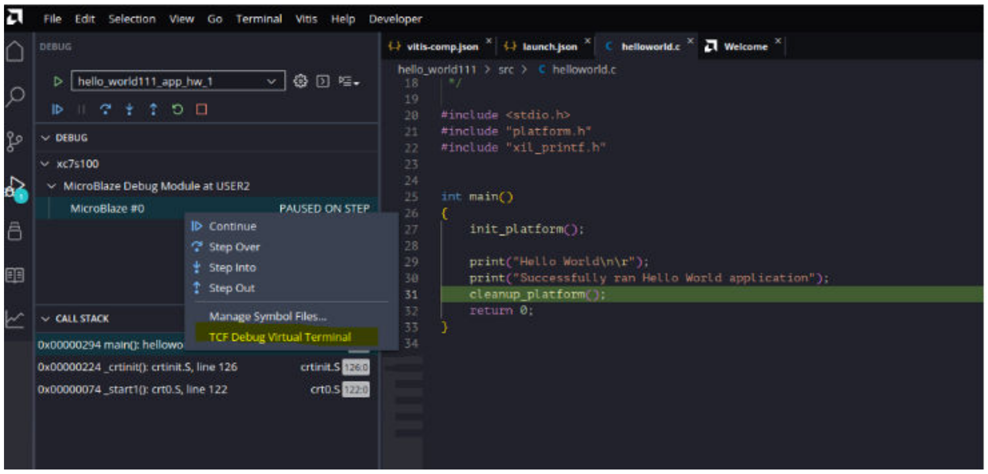
NAME	HEX	DESCRIPTION
r0	0000000000000000	
r1	0000000000000000	
r2	0000000000003230	
r3	0000000000003210	
r4	0000000000003290	
r5	0000000000003244	
r6	000000000000001c	
r7	0000000000000002	
r8	00000000ffffffff	
r9	0000000000000000	
r10	00000000fd5c0090	
r11	0000000000000000	
r12	0000000000000000	
r13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

4. Add the following information on the next screen:
   - **Location:** Provide the location where you want to save the register dump file.
   - **Select registers/groups to export:** Select the list of registers to be dumped. Uncheck any registers you do not need.
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 54: **TCF Debug Virtual Terminal**



```
0x00000294 main(): hellowo
0x00000224 _crtinit(): crtinit.S, line 126    crtinit.S 126:0
0x00000074 _start1(): crt0.S, line 122        crt0.S 122:0
```

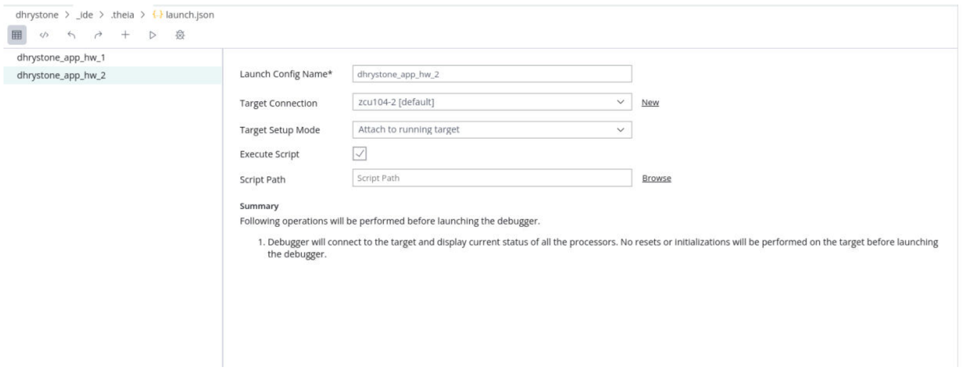
```
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

Debugging an Application Component Already Running On a Target Device

The debugger allows debugging an application that is already running on the target device. An example is a standalone application component or Linux kernel booting from a flash device. 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 that connects to the hardware board. If the hardware board connects to the same machine where the Vitis Unified IDE is running, you can skip this step. You can use the default *Local connection* in the subsequent steps that refer to remote board and remote connection.
   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.



Launch Config Name* dhrystone_app_hw_2
Target Connection zcu104-2 [default]
Target Setup Mode Attach to running target
Execute Script ✓
Script Path Script Path

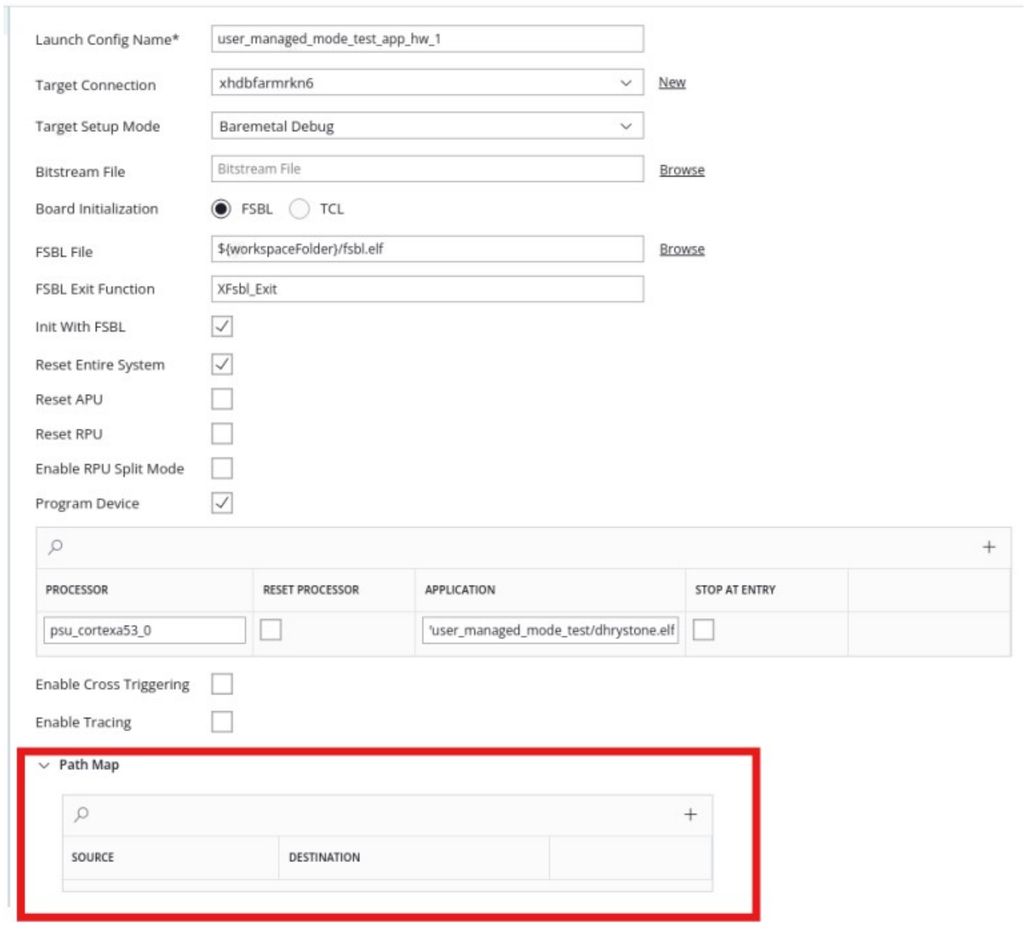
Summary
Following operations will be performed before launching the debugger.
1. Debugger will connect to the target and display current status of all the processors. No resets or initializations will be performed on the target before launching the debugger.

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 or ELF location changed. Follow the steps below:

1. Go to debug launch configuration and expand the Path Map.



Launch Config Name*	user_managed_mode_test_app_hw_1
Target Connection	xhdbfarmrkn6
Target Setup Mode	Baremetal Debug
Bitstream File	Bitstream File
Board Initialization	● FSBL ○ TCL
FSBL File	${workspaceFolder}/fsbl.elf
FSBL Exit Function	XFsbl_Exit
Init With FSBL	✓
Reset Entire System	✓
Reset APU	
Reset RPU	
Enable RPU Split Mode	
Program Device	✓

PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY
psu_cortexa53_0		'user_managed_mode_test/dhrystone.elf	

Enable Cross Triggering	
Enable Tracing	

Path Map

SOURCE	DESTINATION

2. Click **+** to add source map.
   a. SOURCE: 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. Click **OK** and start debug.

Debugging an Application on the Emulator (QEMU)

Note: The Vitis Unified IDE does not support this function. Switch to the classic Vitis IDE (version 2024.2 and older) and refer to the 2024.2 documentation if you need to access this function.

Running and Debugging Application Components Under a System Project Together

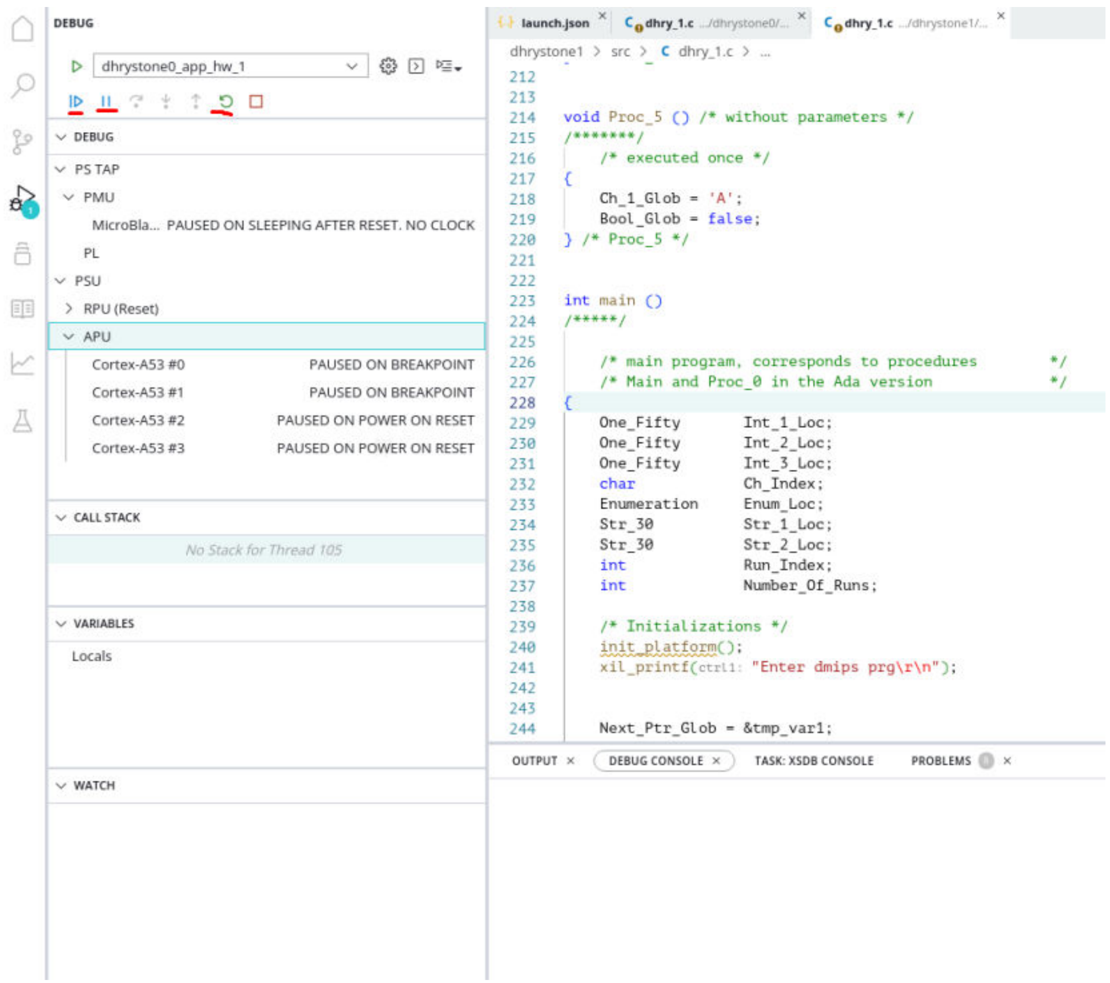
Each application component under a system project can run standalone. Application components under a system project can be launched together as well. The Vitis Unified IDE can download application components one by one and launch them one after another. In debug mode, all applications stop at `main()`.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

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



```
MicroBla...  PAUSED ON SLEEPING AFTER RESET. NO CLOCK
Cortex-A53 #0        PAUSED ON BREAKPOINT
Cortex-A53 #1        PAUSED ON BREAKPOINT
Cortex-A53 #2        PAUSED ON POWER ON RESET
Cortex-A53 #3        PAUSED ON POWER ON RESET
No Stack for Thread 105
```

```

void Proc_5 () /* without parameters */
/*******/
    /* executed once */
{
    Ch_1_Glob = 'A';
    Bool_Glob = false;
} /* Proc_5 */


int main ()
/*****/

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

    /* Initializations */
    init_platform();
    xil_printf(ctrl1: "Enter dmips prg\r\n");


    Next_Ptr_Glob = &tmp_var1;
```

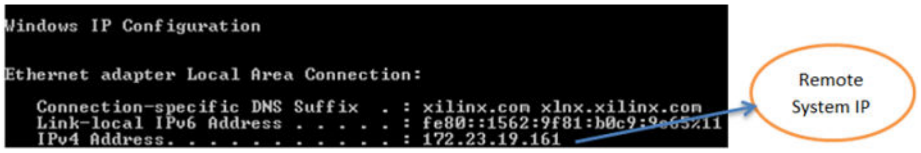
Debugging on a Remote Board

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:

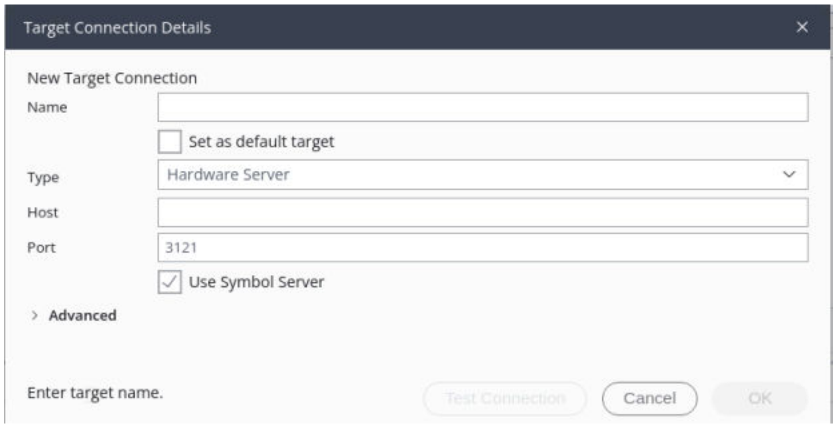
      ```
      the hw_server -s TCP::3122
      ```

   b. Make sure connections to your board are correct.
   c. In a `cmd` window on the host machine, check the IP address:



```
Windows IP Configuration

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : xilinx.com xlnx.xilinx.com
   Link-local IPv6 Address . . . . . : fe80::1562:9f81:b0c9:9e65%11
   IPv4 Address. . . . . . . . . . . : 172.23.19.161
```

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



Target Connection Details

New Target Connection

Name	
	☐ Set as default target
Type	Hardware Server
Host	
Port	3121
	✓ Use Symbol Server

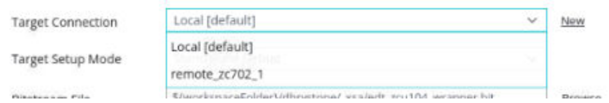
Enter target name.

   e. In the New Target Connection wizard, add the required details for the remote host connecting 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 hardware server's launching Port, 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 Vitis Unified IDE.

j. Click **OK**.

k. In the following screenshot, there are two available connections. `remote_zc702_1` is the remote connection.



l. Click the **debug** button.

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

OS Aware Debugging

The Vitis Unified IDE currently do not support this feature. Use the Classic Vitis IDE if this is a requirement for your project. Refer to the 2023.1 documentation for details on 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.

Follow these steps to enable Xen-aware debug.

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.
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.

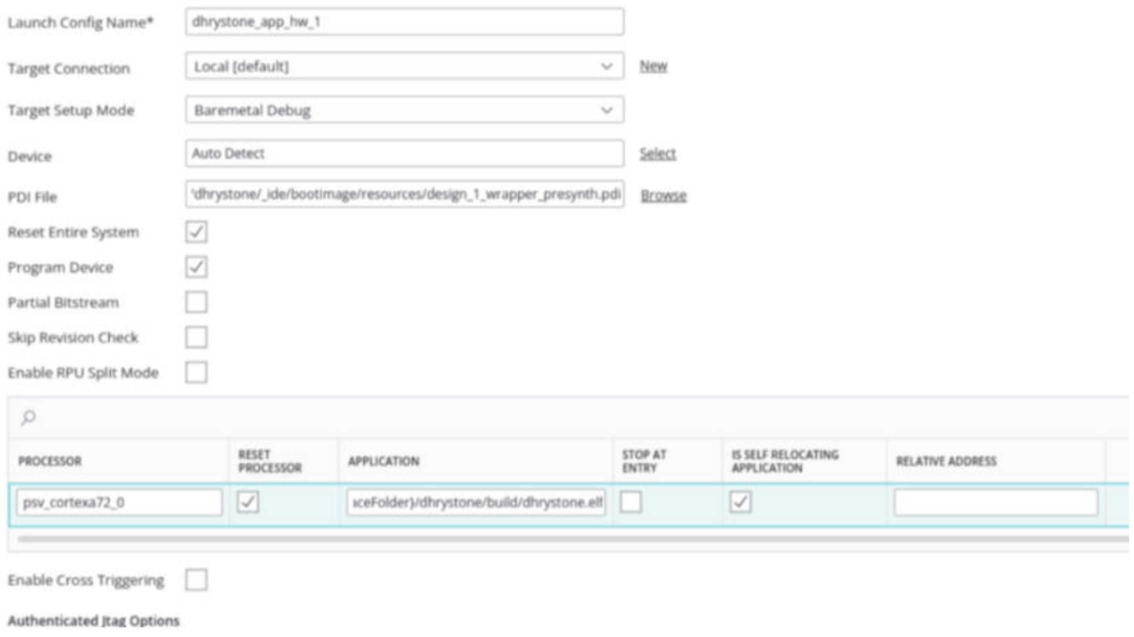
10. Right click VCPU #0 and click **Manage Symbol files**. Add the `vmlinux` symbol file from the PetaLinux project in the first step for DOM0. Do not enable the **OS Awareness** option in this step.
11. Select VCPU #0 and **Pause** in the Debug view.
12. Check the Call Stack view to ensure the mapping of all symbol files. Use the step-in and step-over options for further debugging.

Debugging Self-Relocating Programs

A self-relocating program relocates its own code and data sections during runtime to help you load and relocation applications. The debug information available in the ELF file does not specify where the program sections are to be relocated. Therefore, you must provide the debugger with the relative address so it can relocate the program sections. 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 55: **Debugging Self-Relocating Programs**



Launch Config Name*	dhrystone_app_hw_1	
Target Connection	Local (default)	New
Target Setup Mode	Baremetal Debug	
Device	Auto Detect	Select
PDI File	'dhrystone/_ide/bootimage/resources/design_1_wrapper_presynth.pdi	Browse
Reset Entire System	✓	
Program Device	✓	
Partial Bitstream	☐	
Skip Revision Check	☐	
Enable RPU Split Mode	☐	

PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELATIVE ADDRESS
psv_cortexa72_0	✓	ıceFolder}/dhrystone/build/dhrystone.elf	☐	✓	

Enable Cross Triggering	☐

Authenticated Jtag Options

Use the following command to perform the process for XSDB by providing the address where the program sections reside: `memmap -reloc <addr> -file <path-to-elf>`.

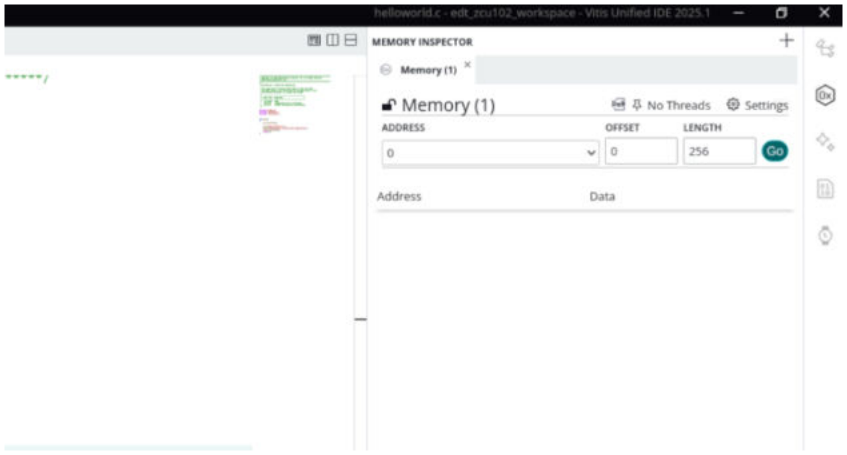
Debugging an Application Project Using the Emulator (Command-Line Flow)

Note: The Vitis Unified IDE does not support this function. Switch to the classic Vitis IDE and refer to 2023.1 documentation if you need to use this function.

Export Memory Function

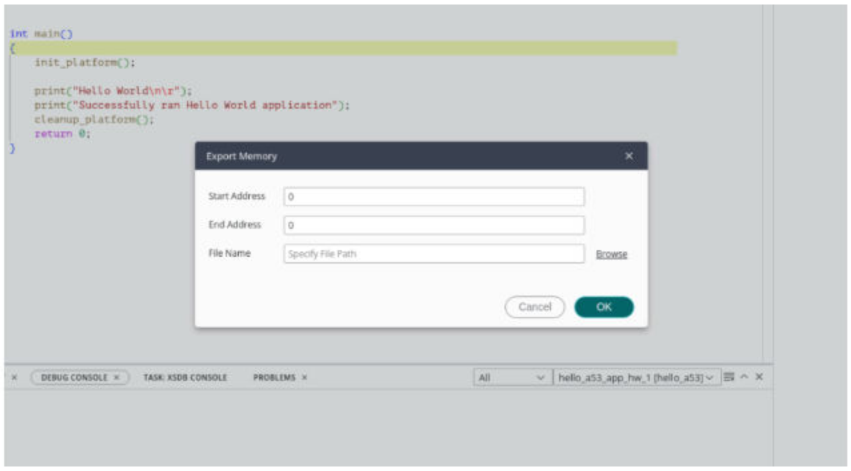
In the Vitis Unified IDE, you can export a memory section to an s record file during a debug session. This process allows you to examine the memory contents during runtime. Follow the below steps for exporting a memory section during an active debug session.

1. When the debug session is open, select the memory inspector view on the right hand side.



ADDRESS: 0
OFFSET: 0
LENGTH: 256

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



```
int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

Start Address: 0
End Address: 0
File Name: Specify File Path

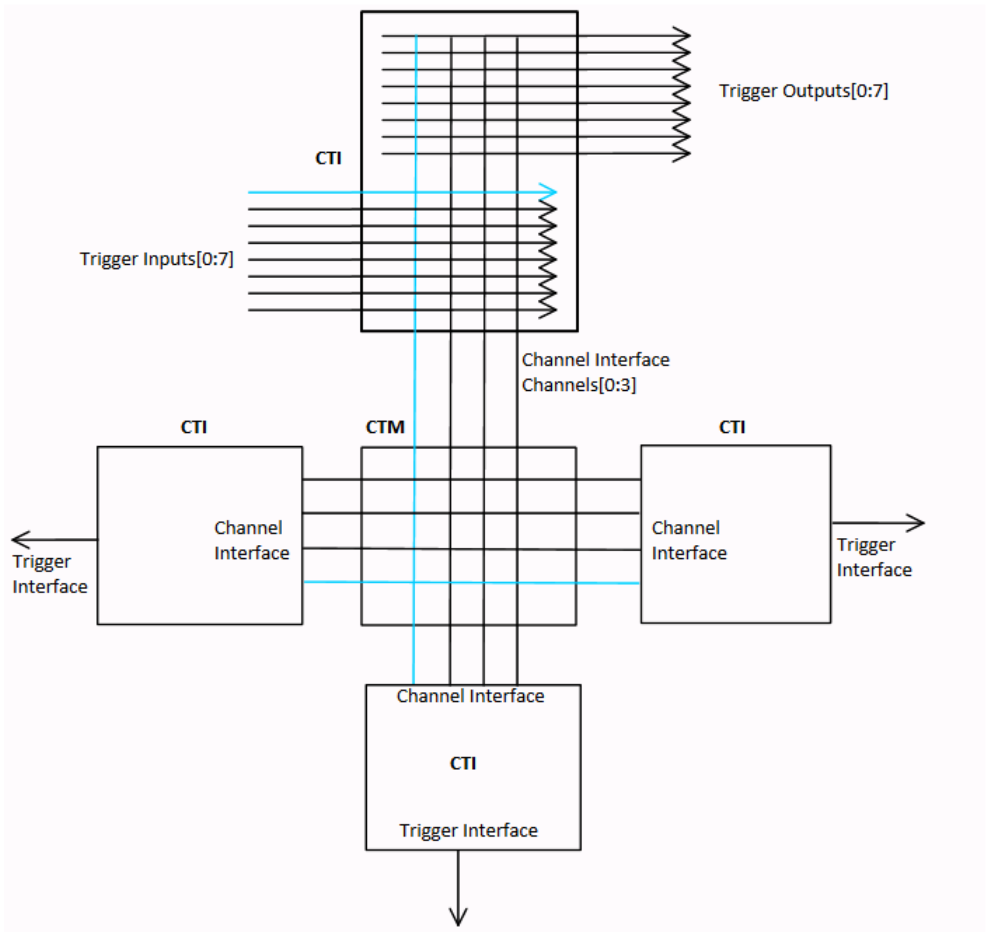
3. Enter your preferred start address, end address and output file path.

Cross-Triggering

The embedded cross-triggering (ECT) module from Arm supports cross-triggering. 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. CTM enables connection of multiple ECTs.

Figure 56: **CTIs and CTM in a Generic Setup**



A CTM forms an event broadcasting network with multiple channels.

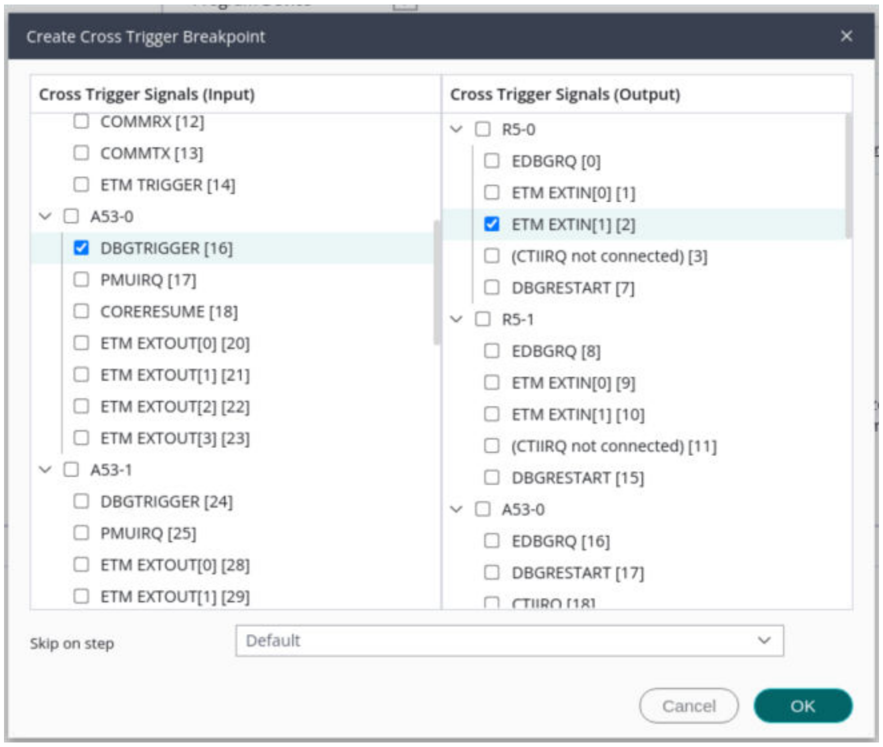
- A CTI listens to one or more channels for an event.
- A CTI maps a received event into a trigger, and sends the trigger to one or more CoreSight components connecting to the CTI.
- A CTI also combines and maps the triggers from connecting CoreSight components and broadcasts them as events on one or more channels.
- You can configure CTI to listen to specific channels for events through its register interface.
- The CTI can broadcast triggers as events to specific channels.

In the above example, there are four channels. The CTI at the top propagates the trigger event on Trigger Input 0 to Channel 0. You can configure other CTIs to listen to this channel for events and broadcast the events through trigger outputs. The trigger outputs debug components connecting to the CTIs. CTIs also support channel gating so that certain channels can be turned off without disabling 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.
6. Click **Add Item** to create new breakpoints.


Create Cross Trigger Breakpoint

Cross Trigger Signals (Input)
- [ ] COMMRX [12]
- [ ] COMMTX [13]
- [ ] ETM TRIGGER [14]
- [ ] A53-0
  - [✓] DBGTRIGGER [16]
  - [ ] PMUIRQ [17]
  - [ ] CORERESUME [18]
  - [ ] ETM EXTOUT[0] [20]
  - [ ] ETM EXTOUT[1] [21]
  - [ ] ETM EXTOUT[2] [22]
  - [ ] ETM EXTOUT[3] [23]
- [ ] A53-1
  - [ ] DBGTRIGGER [24]
  - [ ] PMUIRQ [25]
  - [ ] ETM EXTOUT[0] [28]
  - [ ] ETM EXTOUT[1] [29]

Cross Trigger Signals (Output)
- [ ] R5-0
  - [ ] EDBGRQ [0]
  - [ ] ETM EXTIN[0] [1]
  - [✓] ETM EXTIN[1] [2]
  - [ ] (CTIIRQ not connected) [3]
  - [ ] DBGRESTART [7]
- [ ] R5-1
  - [ ] EDBGRQ [8]
  - [ ] ETM EXTIN[0] [9]
  - [ ] ETM EXTIN[1] [10]
  - [ ] (CTIIRQ not connected) [11]
  - [ ] DBGRESTART [15]
- [ ] A53-0
  - [ ] EDBGRQ [16]
  - [ ] DBGRESTART [17]
  - [ ] CTIIRQ [18]

Skip on step: Default

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 has four broadcast channels, four CTIs, and a CTM. One CTI connects 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 in the following table 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 has 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 has 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

FPGA to CPU triggering is one of the most common use cases of cross-triggering in Zynq.

- There are four trigger inputs on FPGA CTI. You can configure the inputs to halt (EDBGRQ) any of the two CPUs. Similarily, the four FPGA CTI trigger outputs can trigger when a CPU halts (DBGACK).
- The FPGA trigger inputs and outputs can connect to ILA cores. When any of the two CPUs halts, an ILA trigger can halt the CPU(s) and the ILA can trigger to capture the signals it is monitoring.

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 a trigger to halt the CPU(s).

CPU to CPU Triggering

You can use cross-triggering to synchronize the entry and exit from debug state between the CPUs. For example, when CPU0 halts, the event can trigger a CPU1 debug request, which can halt CPU1.

XSDB Cross-Triggering Commands

The XSDB breakpoint add command (bpadd) allows 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 connects to CPU0), which connects to DBGACK (asserts when the core halts). `-ct-output 8` refers to `CTI CPU1 TrigOut0`, which connects 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` halts, the event broadcasts 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

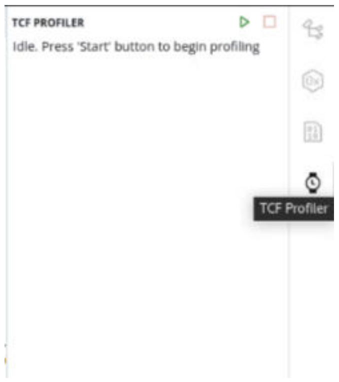
The TCF profiler supports profiling of both standalone and Linux applications. TCF profiling does not require any additional compiler flags to be set while building the application. Profiling standalone applications over JTAG depends on sampling the Program Counter through the debug interface. It does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

1. Start the Vitis Unified IDE and select the application component you need 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.

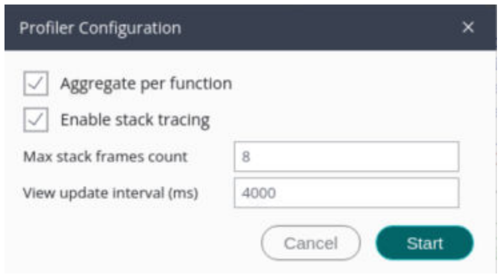
   Note: Regarding the target connection, if the application component is a Linux application, select the TCF agent. If it a standalone application, select HW server.
3. After configuration completes, click **Debug** to start the debug session.

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



Idle. Press 'Start' button to begin profiling

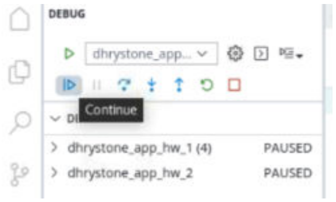
5. Click the **Start** button to begin the profiling. Select the **Enable stack tracing** option to show the stack trace for each address in the sample data. You can also set the frame count and update the interval according to your requirements.
   - Specify the **Max stack frames count** for the maximum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


Profiler Configuration

- [✓] Aggregate per function
- [✓] Enable stack tracing

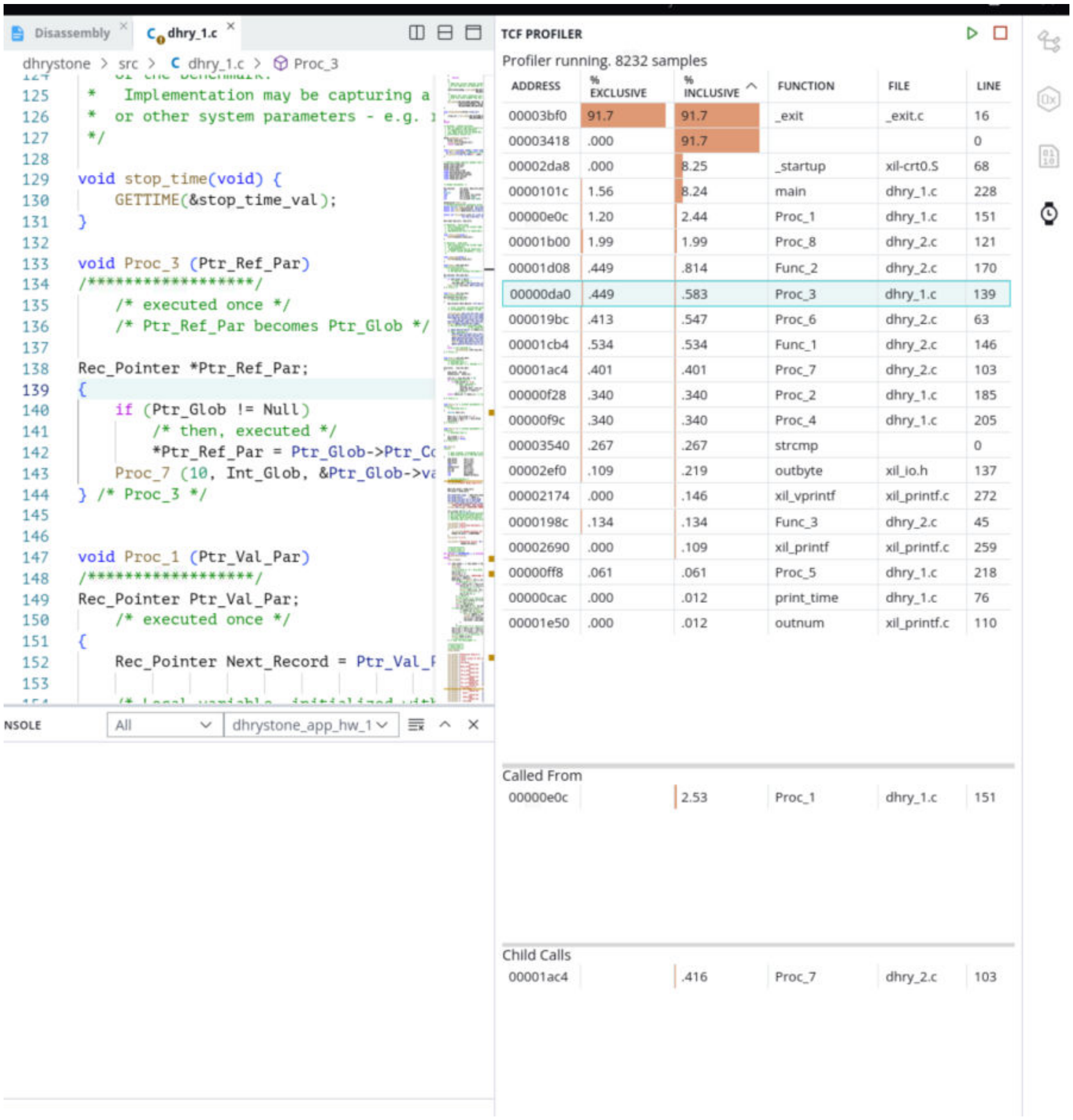
Max stack frames count	8
View update interval (ms)	4000

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



```
dhrystone_app_hw_1 (4)    PAUSED
dhrystone_app_hw_2        PAUSED
```

8. You can now retrieve the profiling information of your application component.



```
 *  Implementation may be capturing a
 *  or other system parameters - e.g. i
 */

void stop_time(void) {
    GETTIME(&stop_time_val);
}

void Proc_3 (Ptr_Ref_Par)
/******************/
    /* executed once */
    /* Ptr_Ref_Par becomes Ptr_Glob */

Rec_Pointer *Ptr_Ref_Par;
{
    if (Ptr_Glob != Null)
        /* then, executed */
        *Ptr_Ref_Par = Ptr_Glob->Ptr_Cc
    Proc_7 (10, Int_Glob, &Ptr_Glob->va
} /* Proc_3 */


void Proc_1 (Ptr_Val_Par)
/******************/
Rec_Pointer Ptr_Val_Par;
    /* executed once */
{
    Rec_Pointer Next_Record = Ptr_Val_F
```

TCF PROFILER

Profiler running. 8232 samples

ADDRESS	% EXCLUSIVE	% INCLUSIVE	FUNCTION	FILE	LINE
00003bf0	91.7	91.7	_exit	_exit.c	16
00003418	.000	91.7			0
00002da8	.000	8.25	_startup	xil-crt0.S	68
0000101c	1.56	8.24	main	dhry_1.c	228
00000e0c	1.20	2.44	Proc_1	dhry_1.c	151
00001b00	1.99	1.99	Proc_8	dhry_2.c	121
00001d08	.449	.814	Func_2	dhry_2.c	170
00000da0	.449	.583	Proc_3	dhry_1.c	139
000019bc	.413	.547	Proc_6	dhry_2.c	63
00001cb4	.534	.534	Func_1	dhry_2.c	146
00001ac4	.401	.401	Proc_7	dhry_2.c	103
00000f28	.340	.340	Proc_2	dhry_1.c	185
00000f9c	.340	.340	Proc_4	dhry_1.c	205
00003540	.267	.267	strcmp		0
00002ef0	.109	.219	outbyte	xil_io.h	137
00002174	.000	.146	xil_vprintf	xil_printf.c	272
0000198c	.134	.134	Func_3	dhry_2.c	45
00002690	.000	.109	xil_printf	xil_printf.c	259
00000ff8	.061	.061	Proc_5	dhry_1.c	218
00000cac	.000	.012	print_time	dhry_1.c	76
00001e50	.000	.012	outnum	xil_printf.c	110

Called From

00000e0c		2.53	Proc_1	dhry_1.c	151

Child Calls

00001ac4		.416	Proc_7	dhry_2.c	103

- 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*:** The Vitis Unified IDE does not support this feature. 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 feature.

Non-Intrusive Profiling for MicroBlaze Processors

The Vitis Unified IDE does not support this function, 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*:** The Vitis Unified IDE does not support this feature. Switch to the classic Vitis IDE (available in version 2024.2 and older) and refer to the 2024.2 documentation if you need to access this function.

Creating a Boot Image

AMD FPGAs and system-on-chip (SoC) devices typically have multiple hardware and software binaries. These binaries boot the FPGA or SoC to their designed functions. The binaries can include FPGA bitstreams, firmware images, bootloaders, operating systems, and user-chosen applications. They can load 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.

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

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

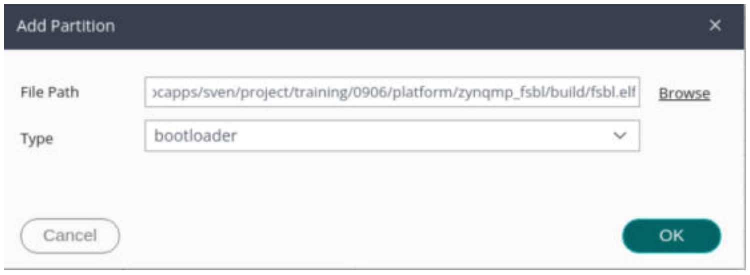
Create a boot image from scratch

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

Select **Vitis → Create Boot Image** and select the device according to your design.

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.



Add Partition

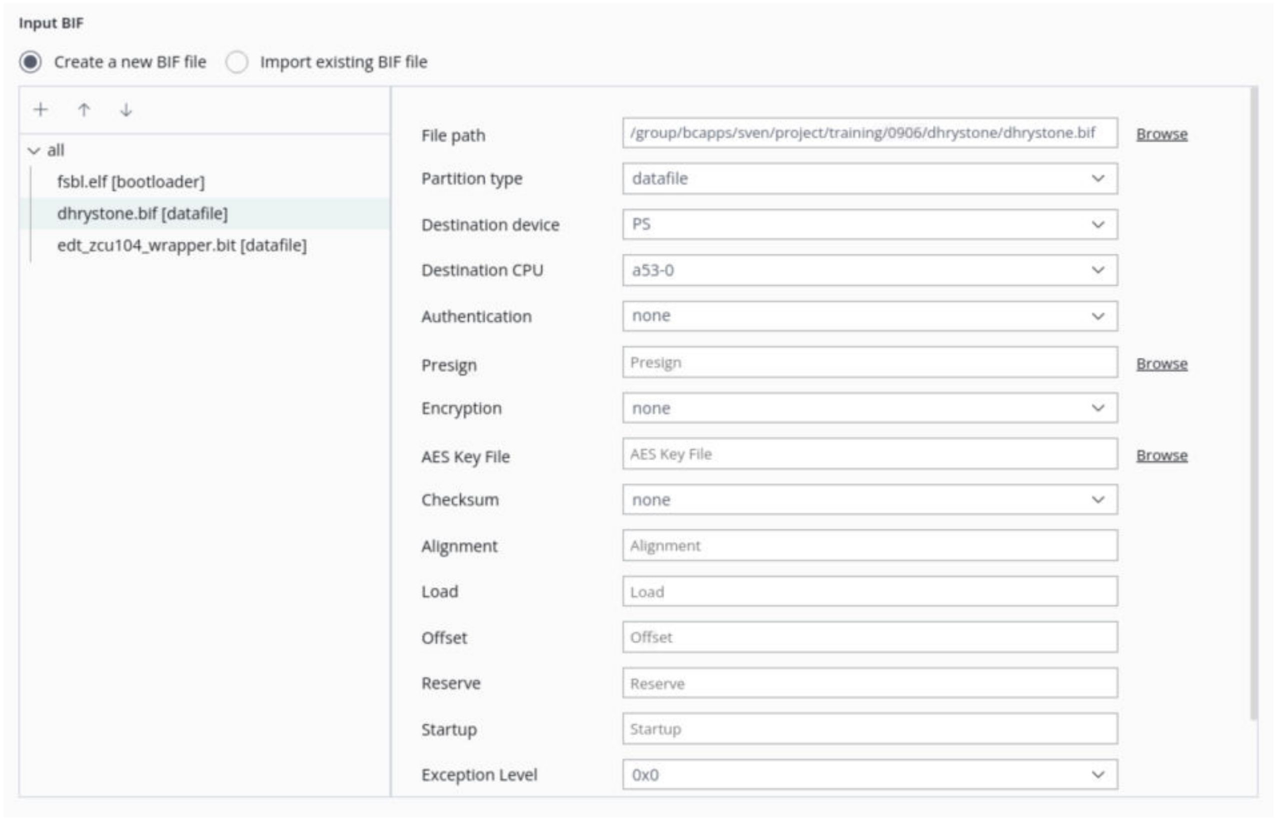
File Path: ɔcapps/sven/project/training/0906/platform/zynqmp_fsbl/build/fsbl.elf

Type: bootloader

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.



Input BIF

(● Create a new BIF file) ○ Import existing BIF file

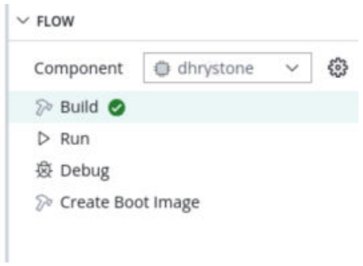
- all
  - fsbl.elf [bootloader]
  - dhrystone.bif [datafile]
  - edt_zcu104_wrapper.bit [datafile]

File path	/group/bcapps/sven/project/training/0906/dhrystone/dhrystone.bif
Partition type	datafile
Destination device	PS
Destination CPU	a53-0
Authentication	none
Presign	Presign
Encryption	none
AES Key File	AES Key File
Checksum	none
Alignment	Alignment
Load	Load
Offset	Offset
Reserve	Reserve
Startup	Startup
Exception Level	0x0

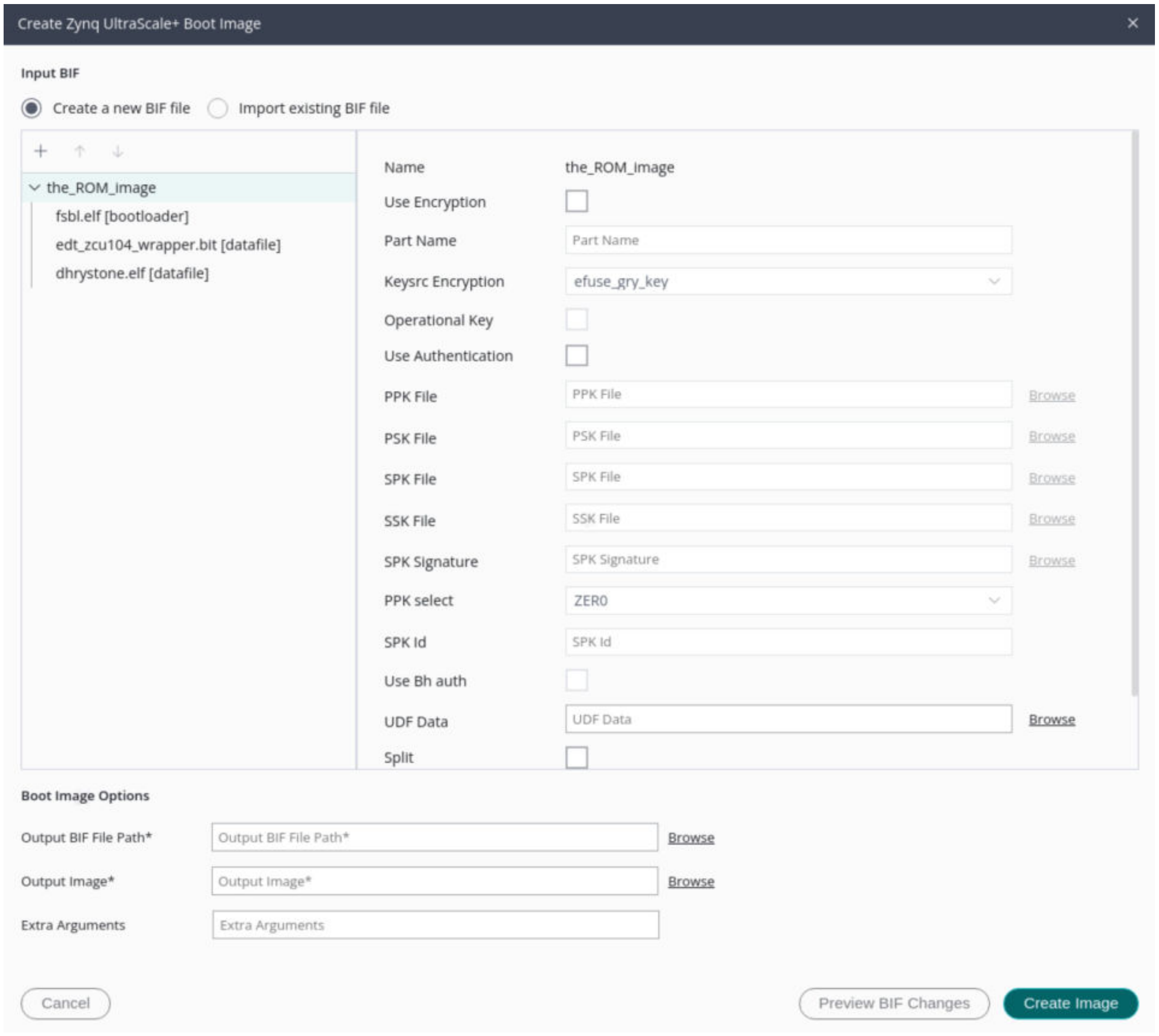
4. Specify the Output Bif File Path and Output Image path.
5. Click **Create Image**. After a few seconds, the log indicates that the image is ready.

Create boot image from Flow Navigator (faster)

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



3. The create boot image setup dialog box shows the required images from your component. Specify the **Output Bif File Path** and **Output Image path**.



Name	the_ROM_image
Use Encryption	
Part Name	Part Name
Keysrc Encryption	efuse_gry_key
Operational Key	
Use Authentication	
PPK File	PPK File
PSK File	PSK File
SPK File	SPK File
SSK File	SSK File
SPK Signature	SPK Signature
PPK select	ZERO
SPK Id	SPK Id
Use Bh auth	
UDF Data	UDF Data
Split	
Output BIF File Path*	Output BIF File Path*
Output Image*	Output Image*
Extra Arguments	Extra Arguments

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

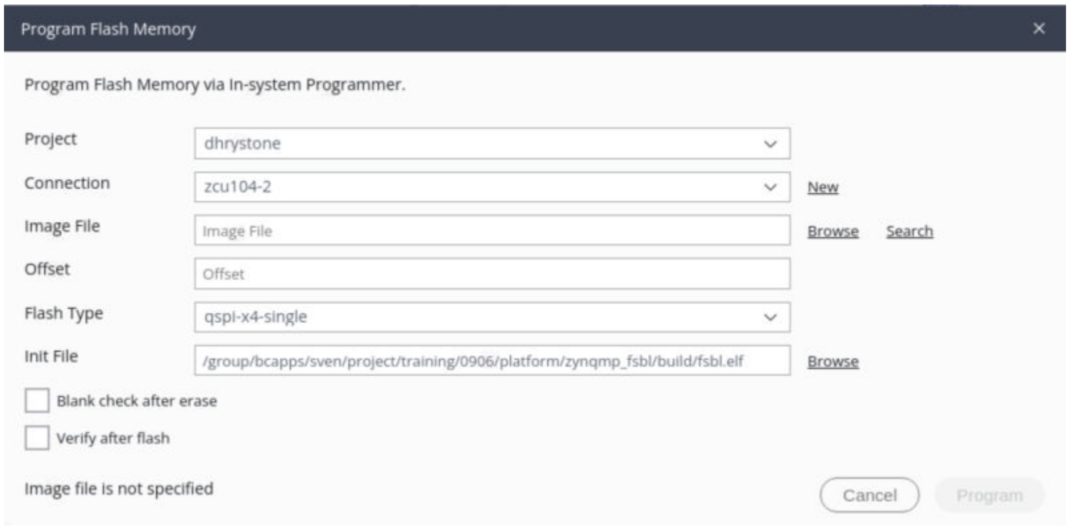
Programming Flash

Program Flash is a tool used to program flash memories in the design. Various types of flash support 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. You can use QSPI in different configurations such as QSPI SINGLE, QSPI DUAL PARALLEL, and QSPI DUAL STACKED.
- For Versal devices – QSPI, emmc, and OSPI. You can use QSPI 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 57: **Program Flash Window**



Program Flash Memory via In-system Programmer.

Project	dhrystone
Connection	zcu104-2
Image File	Image File
Offset	Offset
Flash Type	qspi-x4-single
Init File	/group/bcapps/sven/project/training/0906/platform/zynqmp_fsbl/build/fsbl.elf

☐ Blank check after erase
☐ Verify after flash

Image file is not specified

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

- **Project:** Select the system project you plan to use. The Vitis tool automatically selects 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. The Vitis tool programs the file in this offset location.

  ***Note*:** You do not need offset 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 populates with information from the connected FPGA. Based on the device detected, the dialog populates all the flash parts supported for the device. If the connection to the hardware server does not exist, the following error message displays on the page.

      ```
      "Could not retrieve Flash Part information. Please check hardware
      server connection"
      ```

  ***Note*:** You can select the appropriate part can 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.

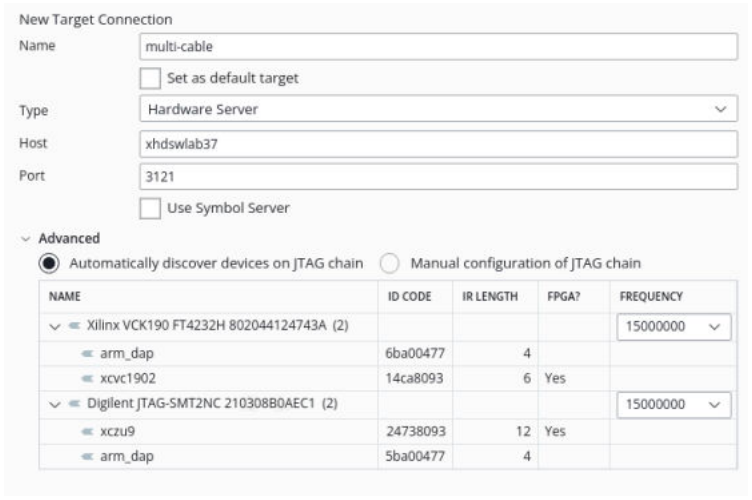
- **Blank check after erase:** The blank check verifies the erase operation and whether the erased region is blank.

- **Verify after Flash:** The Vitis tool reads back and cross-checks the flash contents 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:** This use case assumes you have more than one board connecting 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, refer to Target Connections.



New Target Connection

Name	multi-cable
	☐ Set as default target
Type	Hardware Server
Host	xhdswlab37
Port	3121
	☐ Use Symbol Server

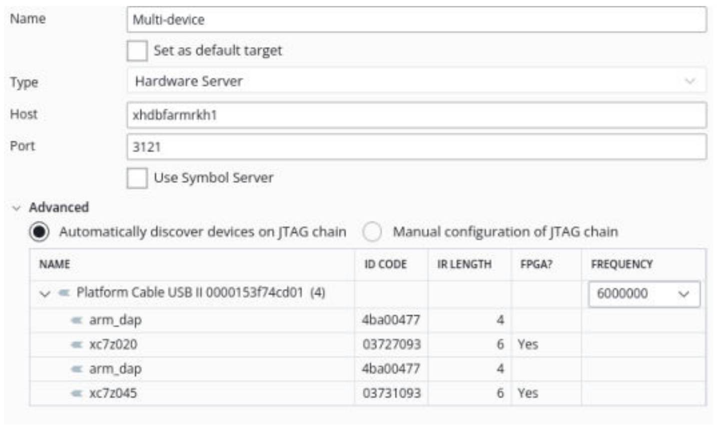
Advanced

● Automatically discover devices on JTAG chain ○ Manual configuration of JTAG chain

NAME	ID CODE	IR LENGTH	FPGA?	FREQUENCY
Xilinx VCK190 FT4232H 802044124743A (2)				15000000
arm_dap	6ba00477	4		
xcvc1902	14ca8093	6	Yes	
Digilent JTAG-SMT2NC 210308B0AEC1 (2)				15000000
xczu9	24738093	12	Yes	
arm_dap	5ba00477	4		

- **Multi-device:** This use case assumes 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, refer to Target Connections.

Figure 58: **Multi-device Target Connection View**



Name	Multi-device
	☐ Set as default target
Type	Hardware Server
Host	xhdbfarmrkh1
Port	3121
	☐ Use Symbol Server

Advanced

● Automatically discover devices on JTAG chain ○ Manual configuration of JTAG chain

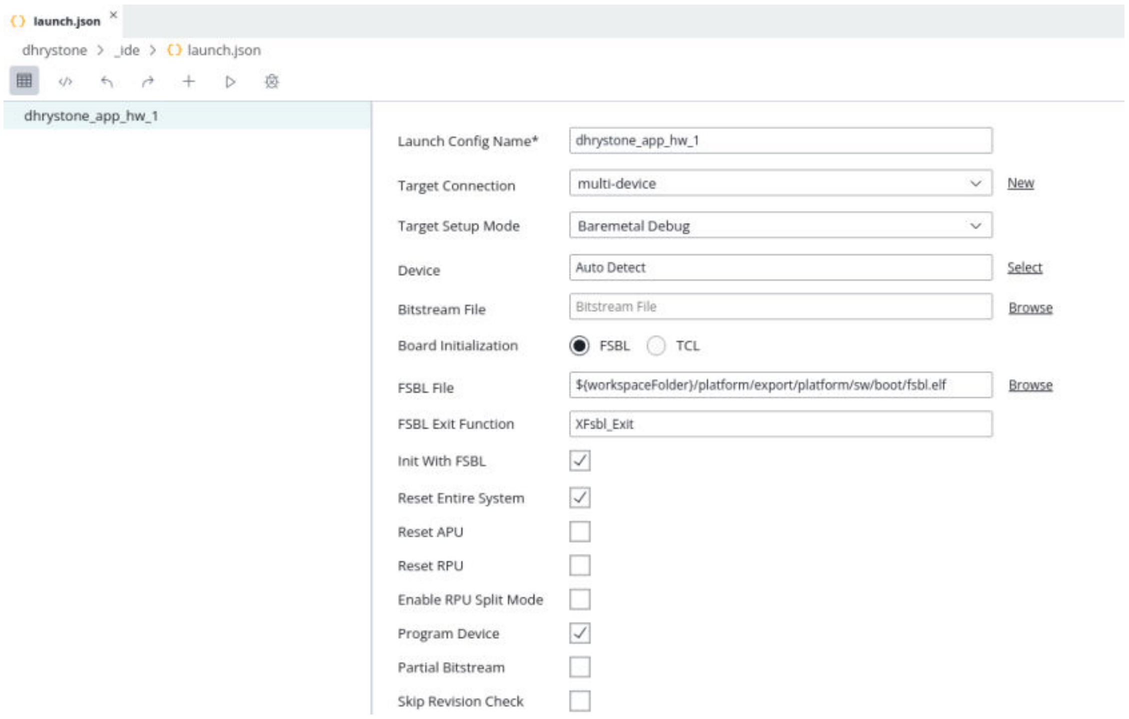
NAME	ID CODE	IR LENGTH	FPGA?	FREQUENCY
Platform Cable USB II 0000153f74cd01 (4)				6000000
arm_dap	4ba00477	4		
xc7z020	03727093	6	Yes	
arm_dap	4ba00477	4		
xc7z045	03731093	6	Yes	

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 59: **launch.json Tab**

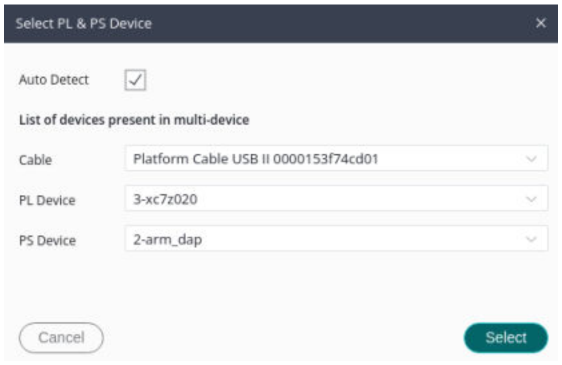


launch.json
dhrystone > _ide > launch.json
dhrystone_app_hw_1

Launch Config Name*	dhrystone_app_hw_1
Target Connection	multi-device
Target Setup Mode	Baremetal Debug
Device	Auto Detect
Bitstream File	Bitstream File
Board Initialization	[✓ FSBL] TCL
FSBL File	${workspaceFolder}/platform/export/platform/sw/boot/fsbl.elf
FSBL Exit Function	XFsbl_Exit
Init With FSBL	✓
Reset Entire System	✓
Reset APU	
Reset RPU	
Enable RPU Split Mode	
Program Device	✓
Partial Bitstream	
Skip Revision Check	

- Deselect **Auto Detect**

Figure 60: **Select PL and PS Device Screen**



Select PL & PS Device

Auto Detect ✓

List of devices present in multi-device

Cable	Platform Cable USB II 0000153f74cd01
PL Device	3-xc7z020
PS Device	2-arm_dap

- 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.
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.

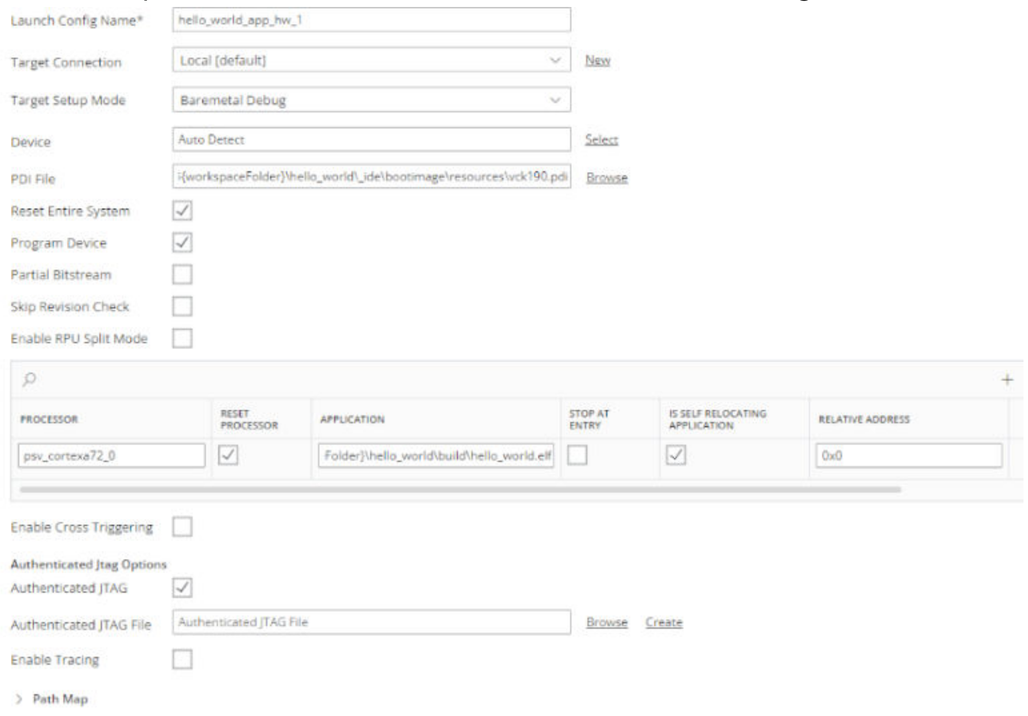
Authenticated Jtag Access

Authenticated JTAG Access provides a secure way to control and protect access to a device's JTAG interface. If you enable this feature, you must successfully authenticate before performing any JTAG operations (i.e. debug, programming, 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.



Launch Config Name*	hello_world_app_hw_1
Target Connection	Local [default]
Target Setup Mode	Baremetal Debug
Device	Auto Detect
PDI File	:{workspaceFolder}\hello_world_ide\bootimage\resources\vck190.pdi
Reset Entire System	✓
Program Device	✓
Partial Bitstream	
Skip Revision Check	
Enable RPU Split Mode	

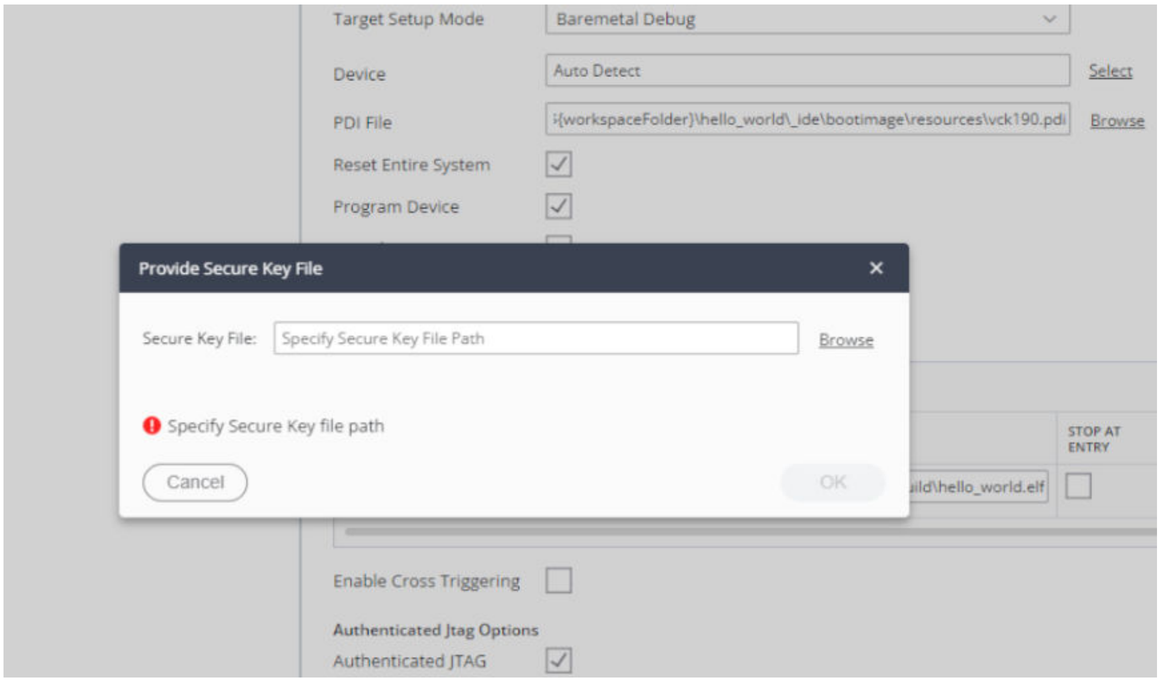
PROCESSOR	RESET PROCESSOR	APPLICATION	STOP AT ENTRY	IS SELF RELOCATING APPLICATION	RELATIVE ADDRESS
psv_cortexa72_0	✓	Folder}\hello_world\build\hello_world.elf		✓	0x0

Enable Cross Triggering	

Authenticated Jtag Options

Authenticated JTAG	✓
Authenticated JTAG File	Authenticated JTAG File
Enable Tracing	

2. After enabling this option, select the **Create** button and provide a path for the JTAG file creation. This generates a decryption file for opening your JTAG gate.


Target Setup Mode: Baremetal Debug
Device: Auto Detect
PDI File: ${workspaceFolder}\hello_world_ide\bootimage\resources\vck190.pdi
Reset Entire System: ✓
Program Device: ✓

Provide Secure Key File
Secure Key File: Specify Secure Key File Path
Specify Secure Key file path

uild\hello_world.elf
Enable Cross Triggering
Authenticated Jtag Options
Authenticated JTAG: ✓

This file then automatically populates in the Authenticated JTAG File field. The debug session automatically uses 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 on the New Feature Preview page.

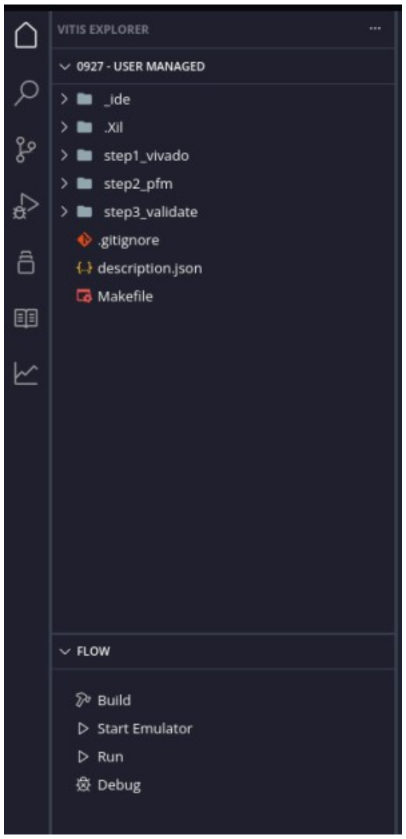
Chapter 8

User Managed Mode

If you prefer to manage your source code hierarchy and build process in a unique way (i.e. 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.

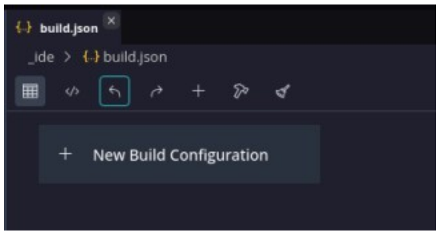
User Managed Mode is automatically activated when no components are found, but files are detected in the current workspace, allowing you to manage and configure their files directly and build or debug Makefile-based projects.

1. Start the Vitis Unified IDE.
2. Open a workspace that has no JSON file and contains extra folders to display the User Managed Mode screen.



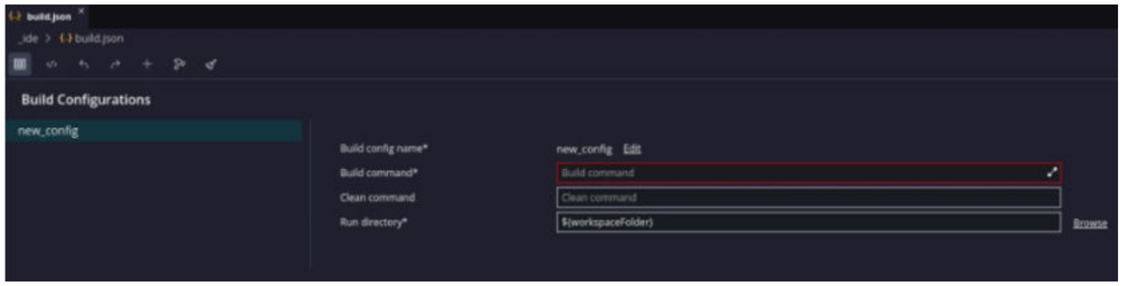
3. Create the build configuration and initiate the build:

   a. Hover the cursor over **Build** to select **Setting** button.



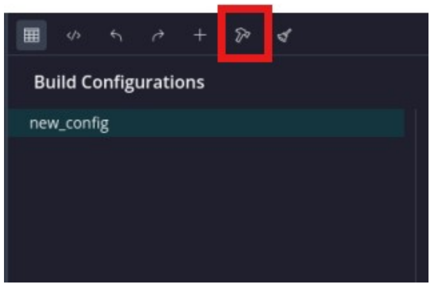
   b. Click **+ New Build Configuration** or you can click **+** button to generate a new configuration for a build.

   c. Input the build, clean and run directory.



Build config name* new_config
Build command* Build command
Clean command Clean command
Run directory* ${workspaceFolder}

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



4. After the build completes, create the Run/Debug launch configuration with the following steps.

   a. Hover the cursor over Debug and select **Setting**.

   b. When the launch configuration file `Launch.json` appears, click **+ New Launch Configuration** or click the **+** button to create a new launch configuration. The Vitis tool supports two design types for embedded application component.

   c. An embedded standalone application has two debug options:

      - **Configure the Device and Start Debug:** Use this option for 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 perform debug on the already running target, select this option and click **Submit**. You need to provide the target connection.

d. An embedded Linux application has two debug options:

   i. Transfer the Elf file and start debugging

      If a Linux application need to be run or debugged, select this option and provide the elf. Next, provide the necessary information for run or debug.

   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**. Provide the target connection and executable file for run or debug.

Setting User Tool Chain

The Vitis tool supports multiple versions of the toolchain in user managed mode.

1. Open a terminal in Linux or open a 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.

Chapter 9

Vitis Utilities

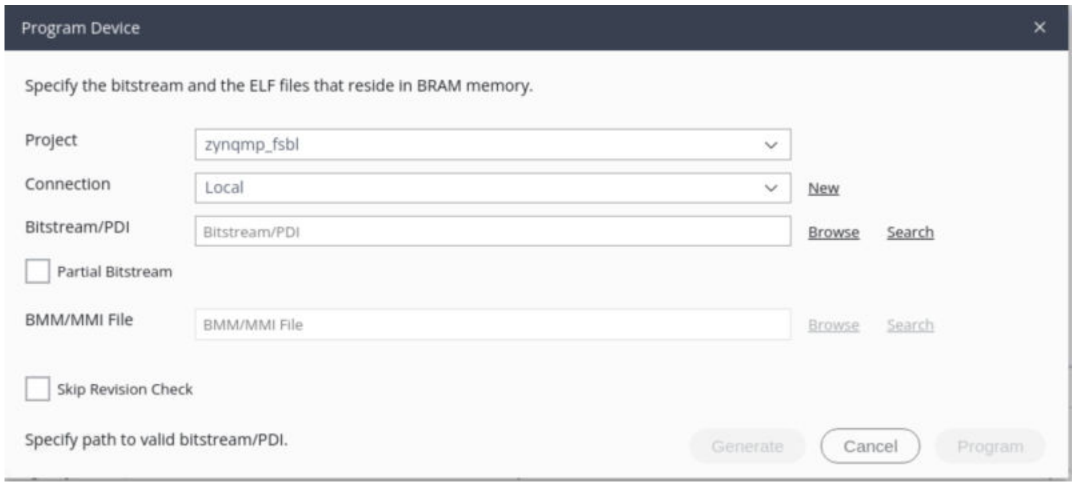
Software Debugger

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

Program Device

You can program the FPGA with the bitstream in the **Program Device** screen.

Figure 61: **Program Device**



Program Device

Specify the bitstream and the ELF files that reside in BRAM memory.

Project: zynqmp_fsbl

Connection: Local

Bitstream/PDI: Bitstream/PDI

Partial Bitstream

BMM/MMI File: BMM/MMI File

Skip Revision Check

Specify path to valid bitstream/PDI.

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

- **Project:** Selects the system project you want to use.
- **Connection:** Selects the hardware or hardware server if the board is on the remote side.
- **Bitstream/PDI File:** Specifies the bitstream/PDI file
- **Partial Bitstream:** Indicates that this is a partial bit stream file.

- **BMM/MMI file:** Only for MicroBlaze design. These files store the Block RAM name and location information of MicroBlaze. Vivado generates these files.
- **Skip Revision Check:** Skips version check.
- **Generate:** Stitches the elf with the bit stream and generate the final bit stream file.
- **Program:** Programs the FPGA.

Vitis Terminal

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

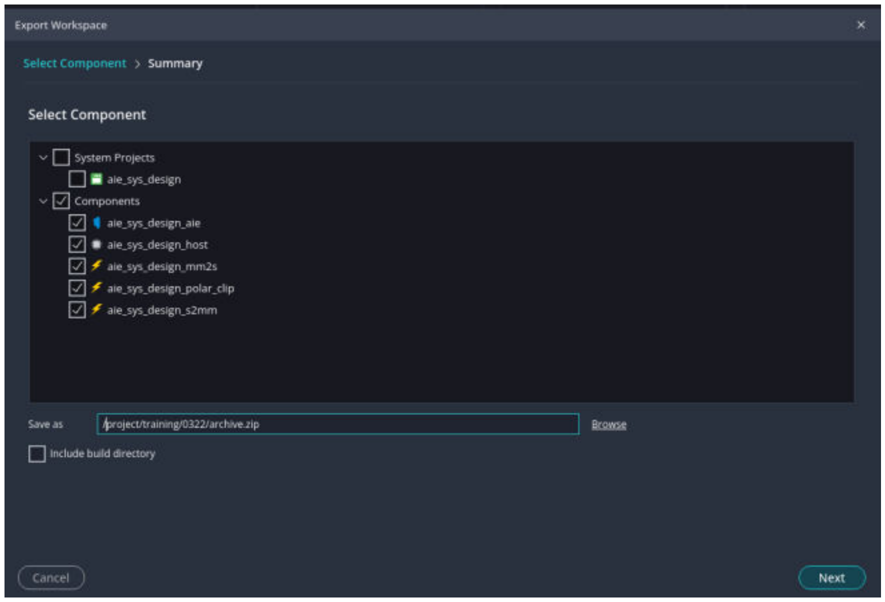
Project Export and Import

The 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, Vitis archives the components in a ZIP file. The ZIP file contains all the relevant files needed to import to another workspace.

1. To export a component, select **File→Export** from the main menu.
2. Select the component or components in the current workspace to export as shown in the following figure.



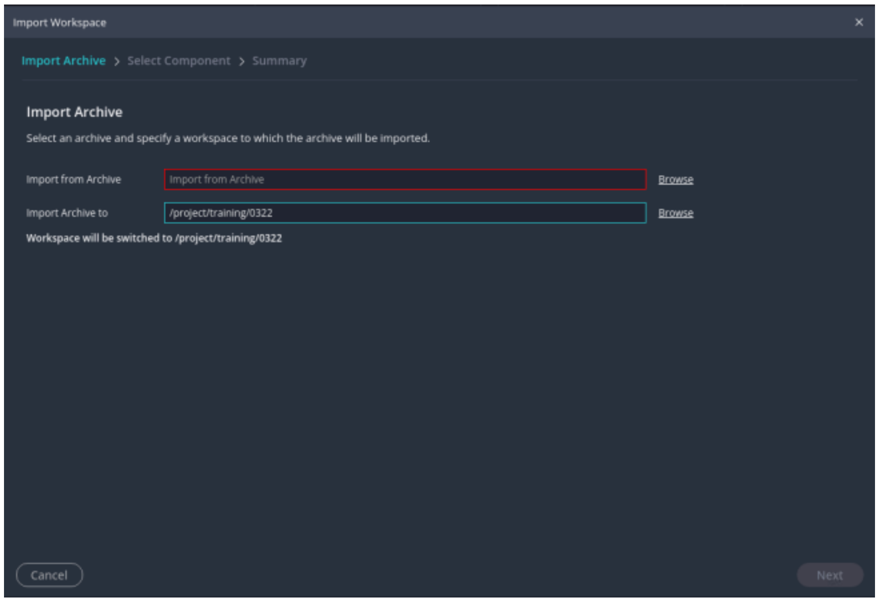
3. Select the component you want to export.
4. Click **Browse** to select a location to save the Archive file. Then, click **Next**.
5. Review the summary of the exported components and click **Finish**.

***Note*:** The Vitis tool archives the IDE components you select in the specified file and location. You can import these archives 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.



Import Archive

Select an archive and specify a workspace to which the archive will be imported.

Import from Archive	Import from Archive	Browse
Import Archive to	/project/training/0322	Browse

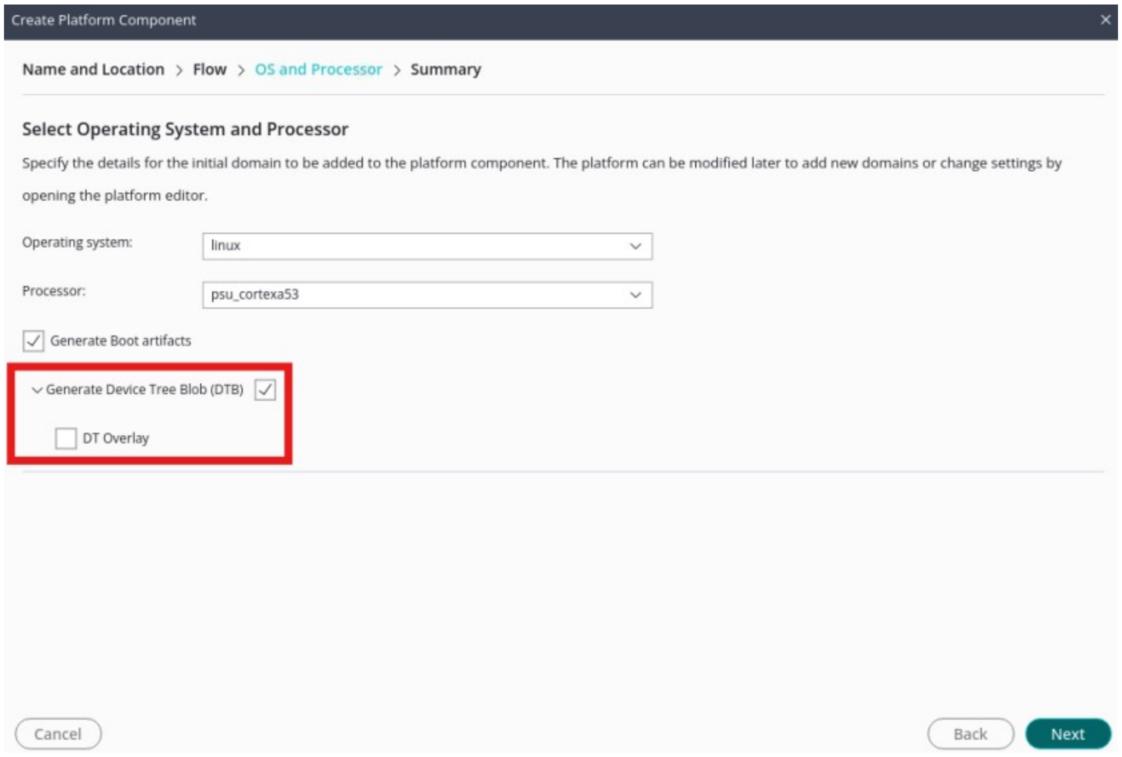
Workspace will be switched to /project/training/0322

2. Select the archive file to be imported, 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. 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 62: **Generating Device Tree**


Create Platform Component
Name and Location > Flow > OS and Processor > Summary

Select Operating System and Processor

Specify the details for the initial domain to be added to the platform component. The platform can be modified later to add new domains or change settings by opening the platform editor.

Operating system: linux

Processor: psu_cortexa53

✓ Generate Boot artifacts

Generate Device Tree Blob (DTB) ✓

☐ DT Overlay

Section III

Vitis Python API

The **AMD Vitis™ Unified 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 Vitis Unified IDE. As with other AMD tools, the scripting language for AMD Vitis™ CLI is based on the Python.

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**.

Enabling the Vitis API to be used in a Python ENV

Enabling the Vitis API in you user Python environment allows you to run a generic Python script within the Vitis tool.

Linux

Open a linux command prompt and execute the following commands:

```
export XILINX_VITIS=<LOCAL_VITIS_INSTALLATION_DIRECTIRY>/2025.2/Vitis
source <LOCAL_VITIS_INSTALLATION_DIRECTIRY>/2025.2/Vitis/cli/examples/
customer_python_utils/setup_vitis_env.sh
```

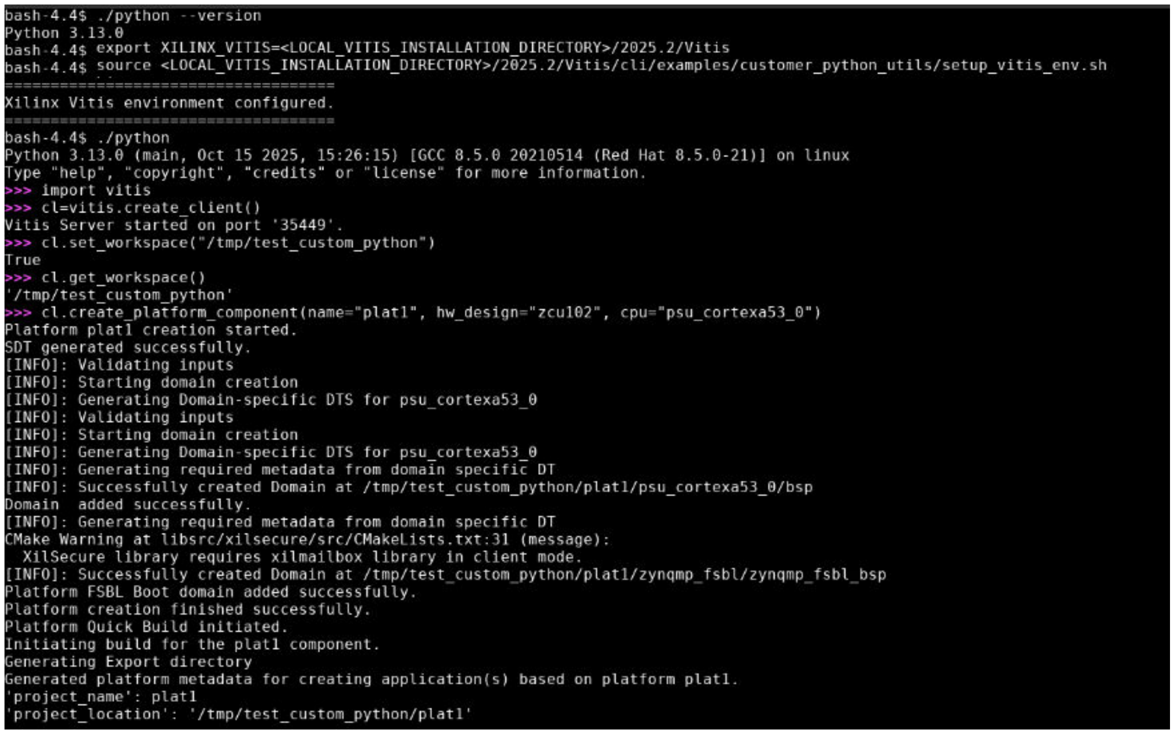
Run Python in the command prompt with the following command:

```
$VITIS_INSTALL_PATH/2025.2/Vitis/tps/lnx64/python-3.13.0/bin/python
or Launch your local python using
./python
```

Execute the following Python commands:

```
import vitis
client = vitis.create_client()
client.get_template()
```

Figure 63: **Python Script**



```
bash-4.4$ ./python --version
Python 3.13.0
bash-4.4$ export XILINX_VITIS=<LOCAL_VITIS_INSTALLATION_DIRECTORY>/2025.2/Vitis
bash-4.4$ source <LOCAL_VITIS_INSTALLATION_DIRECTORY>/2025.2/Vitis/cli/examples/customer_python_utils/setup_vitis_env.sh
====================================
Xilinx Vitis environment configured.
====================================
bash-4.4$ ./python
Python 3.13.0 (main, Oct 15 2025, 15:26:15) [GCC 8.5.0 20210514 (Red Hat 8.5.0-21)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import vitis
>>> cl=vitis.create_client()
Vitis Server started on port '35449'.
>>> cl.set_workspace("/tmp/test_custom_python")
True
>>> cl.get_workspace()
'/tmp/test_custom_python'
>>> cl.create_platform_component(name="plat1", hw_design="zcu102", cpu="psu_cortexa53_0")
Platform plat1 creation started.
SDT generated successfully.
[INFO]: Validating inputs
[INFO]: Starting domain creation
[INFO]: Generating Domain-specific DTS for psu_cortexa53_0
[INFO]: Validating inputs
[INFO]: Starting domain creation
[INFO]: Generating Domain-specific DTS for psu_cortexa53_0
[INFO]: Generating required metadata from domain specific DT
[INFO]: Successfully created Domain at /tmp/test_custom_python/plat1/psu_cortexa53_0/bsp
Domain  added successfully.
[INFO]: Generating required metadata from domain specific DT
CMake Warning at libsrc/xilsecure/src/CMakeLists.txt:31 (message):
  XilSecure library requires xilmailbox library in client mode.
[INFO]: Successfully created Domain at /tmp/test_custom_python/plat1/zynqmp_fsbl/zynqmp_fsbl_bsp
Platform FSBL Boot domain added successfully.
Platform creation finished successfully.
Platform Quick Build initiated.
Initiating build for the plat1 component.
Generating Export directory
Generated platform metadata for creating application(s) based on platform plat1.
'project_name': plat1
'project_location': '/tmp/test_custom_python/plat1'
```

To run a Python script, execute the following command:

```
&VITIS_INSTALL_PATH/2025.2/Vitis/tps/lnx64/python-3.13.0/bin/python -s
sample_testcase.py
```

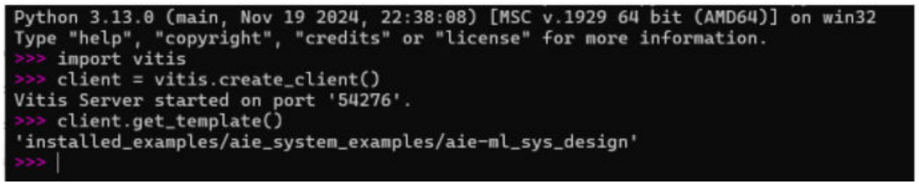
Windows

On Windows, open a windows command prompt and execute the following commands:

```
set XILINX_VITIS=<LOCAL_VITIS_INSTALLATION_DIRECTORY>\2025.2\Vitis

<LOCAL_VITIS_INSTALLATION_DIRECTORY>\2025.2\Vitis\cli\examples\customer_pyth
on_utils>setup_vitis_env.bat
```

To run a Python script in the command prompt, execute the following command:

```
<vitis_install_dir>\2025.2\Vitis\tps\win64\python-3.13.0\python.exe
```



```
Python 3.13.0 (main, Nov 19 2024, 22:38:08) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vitis
>>> client = vitis.create_client()
Vitis Server started on port '54276'.
>>> client.get_template()
'installed_examples/aie_system_examples/aie-ml_sys_design'
>>> |
```

To run a Python script, execute the following command:

```
<vitis_install_dir>\2025.2\Vitis\tps\win64\python-3.13.0\python.exe -s
sample_testcase.py
```

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

The Vitis Unified IDE provides an intuitive graphical interface for assembling and managing your embedded software projects. With the Vitis Unified IDE, you can create and build all Vitis components, including Platforms, Applications, HLS Designs and AI Engines. Alternatively, the Python API offers a scriptable approach, enabling you to create and manage Vitis components programmatically.

Vitis CLI Commands

IMPORTANT! *You can find all the Vitis CLI command details in the installation folder:* *`<Vitis_install>2025.2/Vitis/cli/api_docs/build/html/vitis.html`*

Executing Python APIs

To execute Python APIs, first establish the connection between the server and the client. You can execute Python APIs 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 executing Python script in batch mode and in 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. 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, you can open the workspace using the command below:

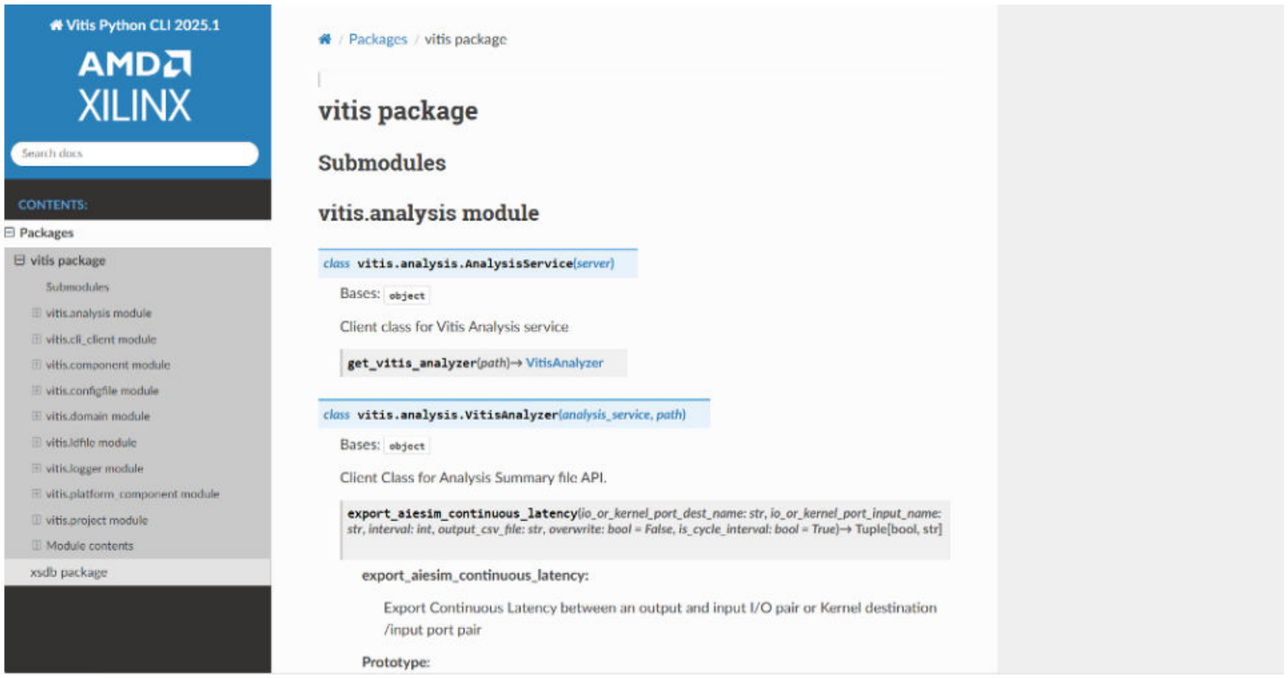
   ```
   vitis -w <workspace_path>
   ```

   If you built or created components through Python APIs, these components exhibit status `done` when you open them.

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 64: **Python APIs Supported by Vitis Documentation**



Vitis Python CLI 2025.1

Search docs

CONTENTS:

- Packages
  - vitis package
    - Submodules
    - vitis.analysis module
    - vitis.cli_client module
    - vitis.component module
    - vitis.configfile module
    - vitis.domain module
    - vitis.ldfile module
    - vitis.logger module
    - vitis.platform_component module
    - vitis.project module
    - Module contents
  - xsdb package

Packages / vitis package

vitis package

Submodules

vitis.analysis module

class `vitis.analysis.AnalysisService`(*server*)

Bases: `object`

Client class for Vitis Analysis service

`get_vitis_analyzer`(*path*)→ VitisAnalyzer

class `vitis.analysis.VitisAnalyzer`(*analysis_service, path*)

Bases: `object`

Client Class for Analysis Summary file API.

`export_aiesim_continuous_latency`(*io_or_kernel_port_dest_name: str, io_or_kernel_port_input_name: str, interval: int, output_csv_file: str, overwrite: bool = False, is_cycle_interval: bool = True*)→ Tuple[bool, str]

export_aiesim_continuous_latency:

Export Continuous Latency between an output and input I/O pair or Kernel destination /input port pair

Prototype:

Python examples are in `<vitis_installation_path>/cli/examples`

Managing Vitis IDE Components through Python APIs

Before running the script, 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). The Vitis Unified IDE supports Python APIs for project flow management, creation and building of the following Vitis components:

- Platform Control
- Application Component
- System Project
- AIE
- HLS

Platform

The platform Python API supports creation, building and modification of a platform.

Table 13: **Python APIs: Platform**

Python API	Description	Python API Example
create_platform_component	Creates a new platform	platform = client.create_platform_component(name = “platform”, hw_design = “zcu102”, cpu = “psu_cortexa53_0”, os = “standalone”, domain_name = “standalone_a53”, architecture = “32-bit”, compiler = “gcc”)
add_domain	Adds the domain to the platform	platform.add_domain(cpu = “psu_cortexa53_0”, os = “standalone”,name = “a53_standalone”, display_name = “a53_standalone”, generate_dtb = True, dt_overlay = True, architecture = “64-bit”, compiler = “gcc”)
build	Generates the platform	platform.build()

The following is an example Python script to create and build a platform:

```
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

platform = client.create_platform_component(name = "platform",hw_design
= "zcu102",os = "standalone",cpu = "psu_cortexa53_0",domain_name
= "standalone_psu_cortexa53_0",generate_dtb = False,architecture = "64-
bit",compiler = "gcc")

platform = client.get_component(name="platform")
status = platform.build()

status = platform.build()
```

Application

After a platform has been created, the Vitis tool's Python API can be used to create an application component. The API also supports creating an application from an example template or an empty application with imported source code.

Python API	Description	Python API Example
create_app_component	Create an application component.	app_comp = client.create_app_component(name = <comp_name>, platform = <platform>, domain = <domain>, template = <template>, cpu = <cpu>, os = <os>, sysroot_toolchain = <sysroot_path>, use_sysroot_toolchain = <boolean>)
import_files	Import files to the app component	status = app_comp.import_files(from_loc = <Location of src or config file(s) path>, files = ["file1",..]**, files_to_exclude = ["file1",..]**, dest_dir_in_cmp = <Dest location or config file(s) path>, is_skip_copy_sources = False)
build	Generate the application.	app_comp.build()

The following is an example of a Python script being used to create an application with the 'Hello World' template.

```
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.create_app_component(name="hello_world",platform
= "/vitis_2025.2_ws/platform/export/platform/platform.xpfm",domain =
"standalone_psu_cortexa53_0",template = "hello_world")

comp = client.get_component(name="hello_world")
comp.build()
```

System Project

The **Vitis Unified IDE** supports Python APIs to create and build a system project. After creating the platform component and application component, the system project is created. You can then add the application components to the system project and build them. You can build host components individually. When Python APIs trigger the system project build, Vitis builds all the components associated with the system.

Table 14: **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. The embedded installer does not support accelerated flows.

The following is the Python script to create a system project and build for target=Hardware to export Vitis metadata archive:

```
# 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()
```

A few more Python command examples are in `<VITIS_INSTALL_DIR>/2025.2/cli/examples`.

***Note*:** The embedded installer does not support accelerated flows.

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

AI Engine Component

The Vitis tool's Python APIs support creating and building AI Engine components either from scratch or with a template.

Python API	Description	Python API Example
create_aie_component	Creates an AIE component	aie_comp = client.create_aie_component(name = <comp_name>, platform = <platform>, part = <part>, template = <template>)
add_cfg_file	Adds configuration file to the component	status = component.add_cfg_file(cfg_file = <file>)
build	Generates the AI Engine component	aie_comp.build()

The following Python code creates and builds an AI Engine component in Vitis tools.

```
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")
```

```
comp = client.create_aie_component(name="simple_aie_application",
platform = "/vitis_2025.2_ws/vck190_platform/export/vck190_platform/
vck190_platform.xpfm", template = "installed_aie_examples/simple")

comp = client.get_component(name="simple_aie_application")
comp.build(target="hw")
```

HLS Component

The Vitis tool's Python APIs support creating and building HLS components either from scratch or with a template.

Python API	Description	Python API Example
create_hls_component	Creates an HLS component	hls_comp = client.create_hls_component(name = <comp_name>, platform = <platform_name>, part = <part>, cfg_file = <cfg_file>, template = <template>)
add_cfg_file	Adds a configuration file to the HLS component	status = component.add_cfg_file(cfg_file = <file>)
run	Runs a specified operation on the HLS component	status = component.run(operation = <operation_type>)

The following is a Python script that creates an empty HLS component and modifies its configuration file:

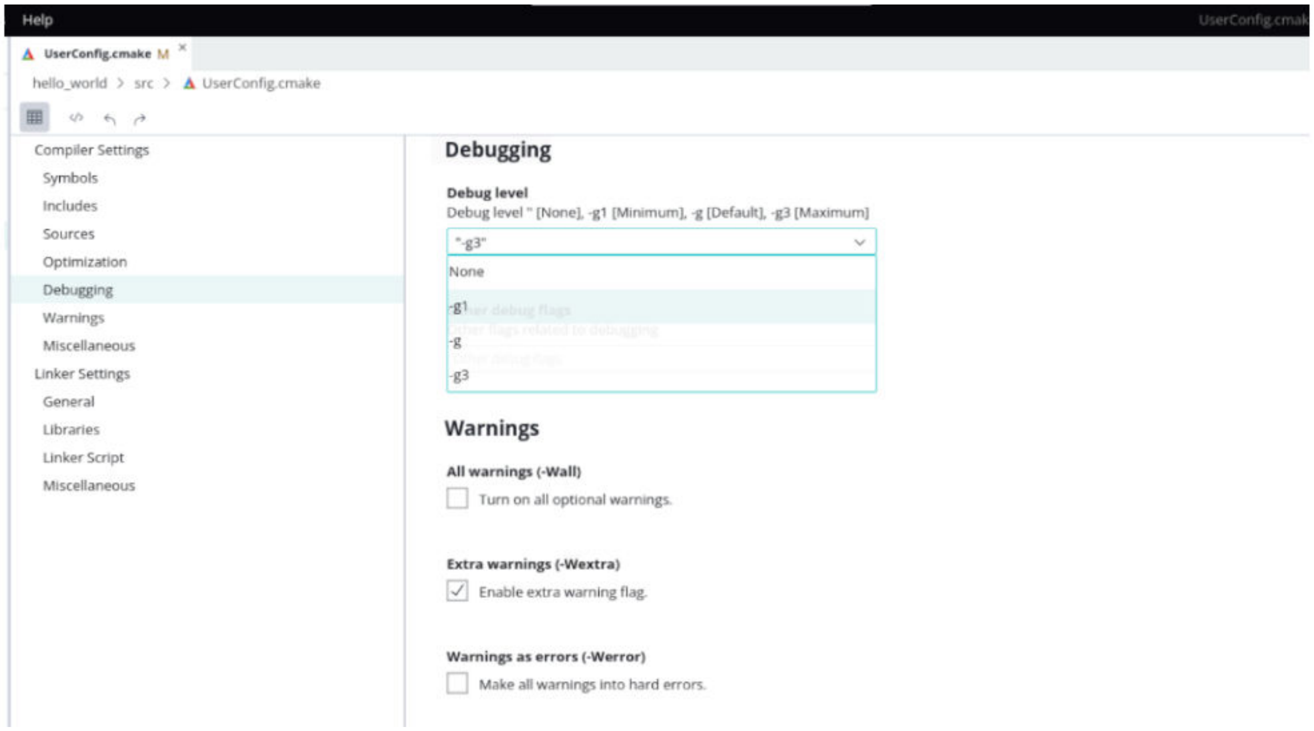
```
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.create_hls_component(name = "mm2s",cfg_file =
["hls_config.cfg"],template = "empty_hls_component")

cfg = client.get_config_file(path="/vitis_2025.2_ws/
simple_aie_application_system_project/hw_link/binary_container_1-link.cfg")

cfg = client.get_config_file(path="/vitis_2025.2_ws/mm2s/hls_config.cfg")

cfg.set_value(section="hls", key="syn.compile.clang_version", value="7")

cfg.set_value(section="hls", key="package.output.format", value="rtl")
```

Modifying Configuration Files

While the Python API can be used for creating and building Vitis components, it can also be used for modifying the configuration files of each component.

First, navigate to the `UserConfig.cmake` file.

Figure 65: **UserConfig.cmake**



Debugging

Debug level
Debug level " [None], -g1 [Minimum], -g [Default], -g3 [Maximum]

"-g3"
- None
- -g1
- -g
- -g3

Warnings

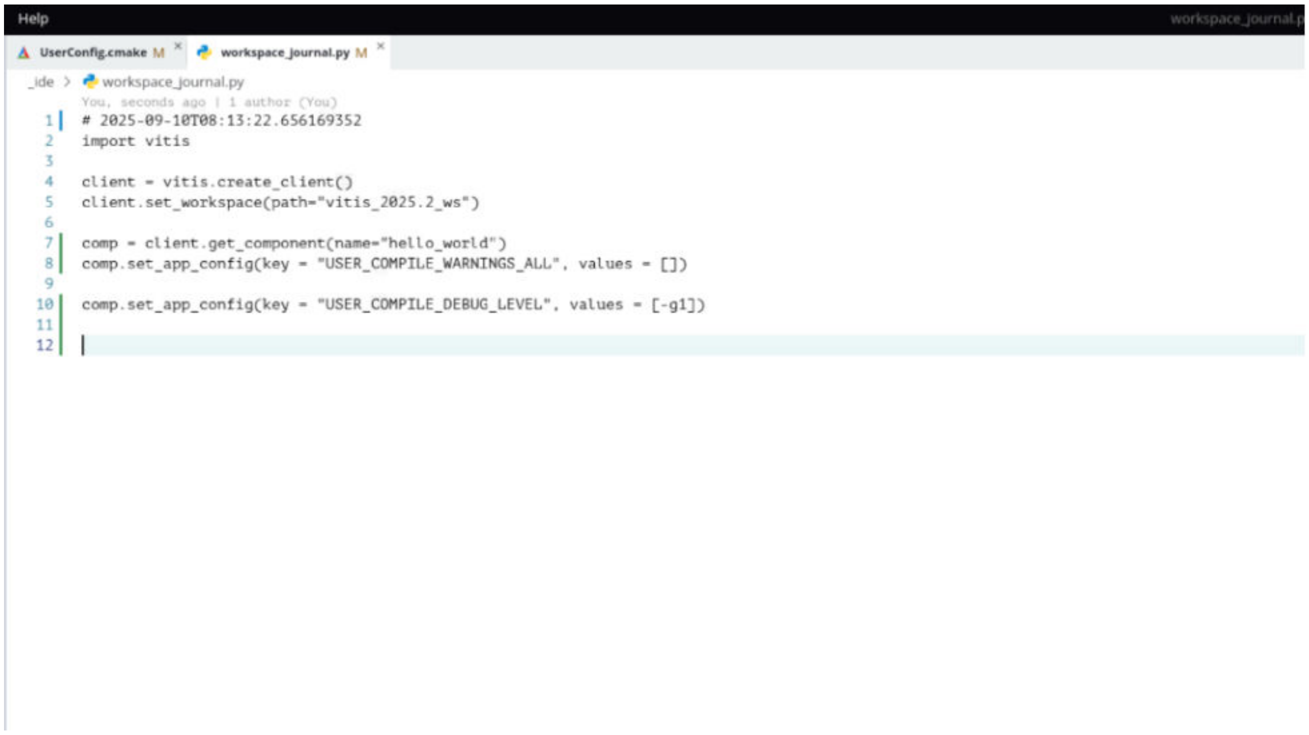
All warnings (-Wall)
- [ ] Turn on all optional warnings.

Extra warnings (-Wextra)
- [✓] Enable extra warning flag.

Warnings as errors (-Werror)
- [ ] Make all warnings into hard errors.

The equivalent Python API commands are visible in the `workspace_journal.py` file.

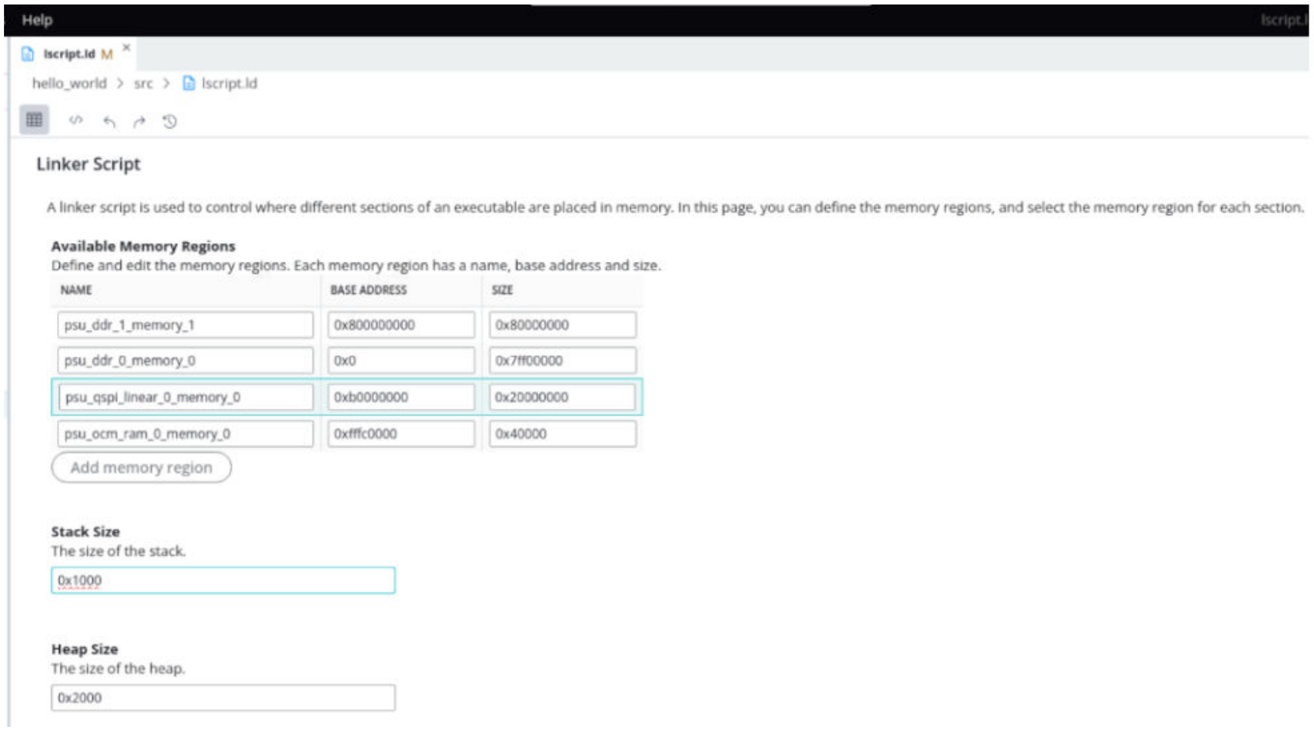
Figure 66: **workspace_journal.py file**



```
# 2025-09-10T08:13:22.656169352
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.get_component(name="hello_world")
comp.set_app_config(key = "USER_COMPILE_WARNINGS_ALL", values = [])

comp.set_app_config(key = "USER_COMPILE_DEBUG_LEVEL", values = [-g1])

```

Next, edit the Linker Script.

Figure 67: **Linker Script**



Linker Script

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define the memory regions, and select the memory region for each section.

Available Memory Regions

Define and edit the memory regions. Each memory region has a name, base address and size.

NAME	BASE ADDRESS	SIZE
psu_ddr_1_memory_1	0x800000000	0x80000000
psu_ddr_0_memory_0	0x0	0x7ff00000
psu_qspi_linear_0_memory_0	0xb0000000	0x20000000
psu_ocm_ram_0_memory_0	0xfffc0000	0x40000

Stack Size

The size of the stack.

0x1000

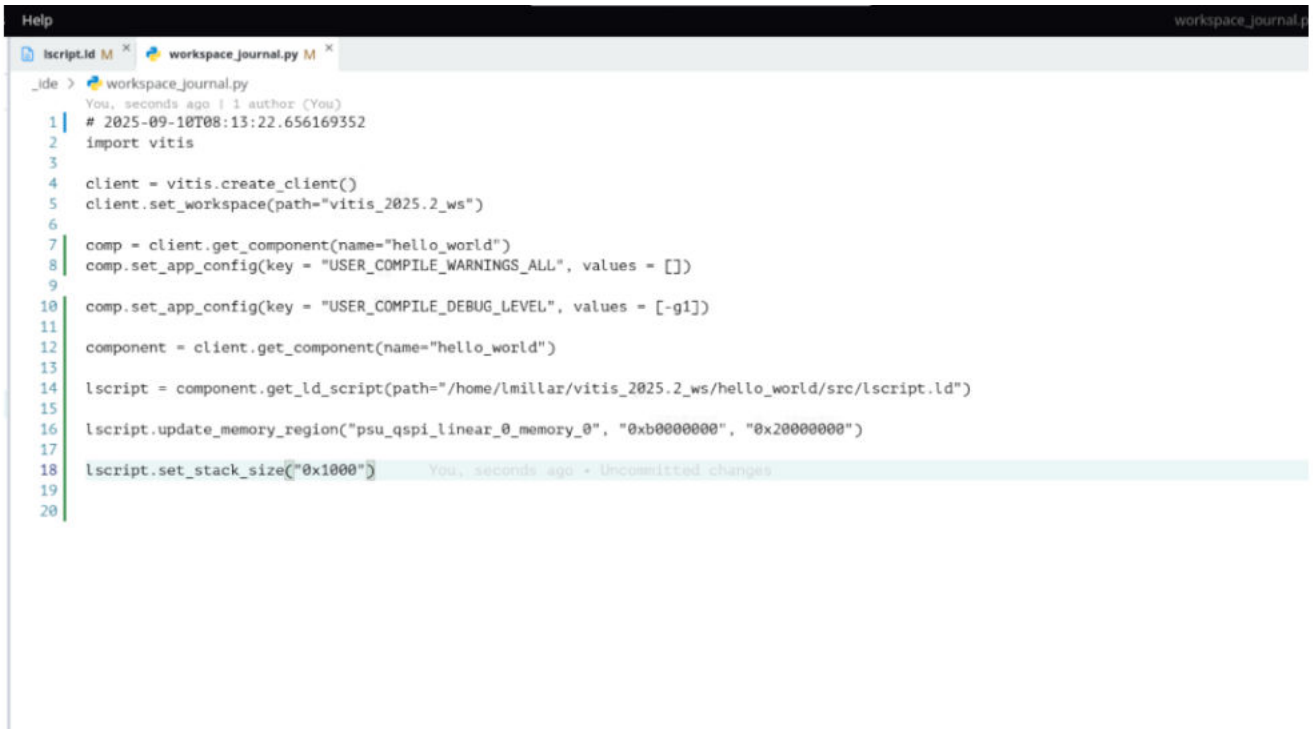
Heap Size

The size of the heap.

0x2000

The workspace journal file is updated with the commands for updating the memory region and setting the stack size.

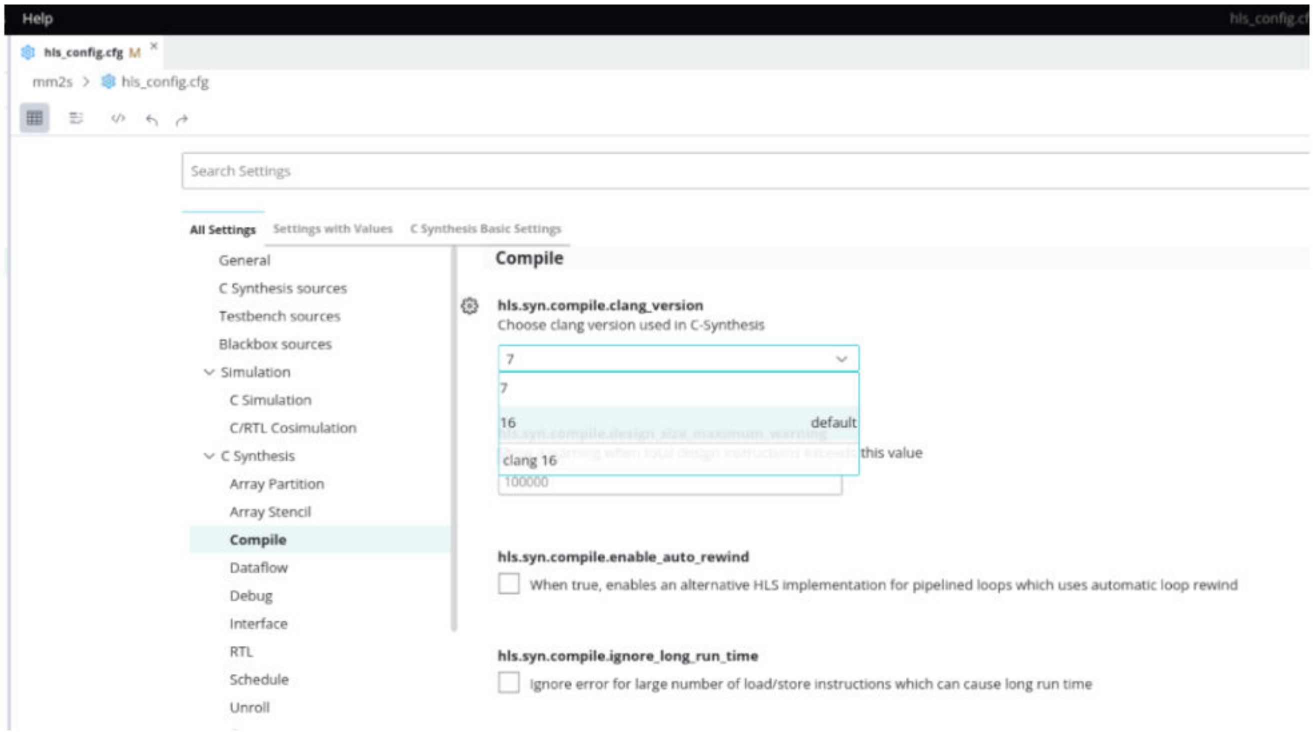
Figure 68: **Workspace Journal**


Help
workspace_journal.p
lscript.ld M
workspace_journal.py M
_ide > workspace_journal.py
You, seconds ago | 1 author (You)
You, seconds ago • Uncommitted changes

```
# 2025-09-10T08:13:22.656169352
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.get_component(name="hello_world")
comp.set_app_config(key = "USER_COMPILE_WARNINGS_ALL", values = [])

comp.set_app_config(key = "USER_COMPILE_DEBUG_LEVEL", values = [-g1])

component = client.get_component(name="hello_world")

lscript = component.get_ld_script(path="/home/lmillar/vitis_2025.2_ws/hello_world/src/lscript.ld")

lscript.update_memory_region("psu_qspi_linear_0_memory_0", "0xb0000000", "0x20000000")

lscript.set_stack_size("0x1000")
```

Confirm the HLS config settings.

Figure 69: **HLS Config**



Compile

hls.syn.compile.clang_version
Choose clang version used in C-Synthesis

7

7
16 default
clang 16

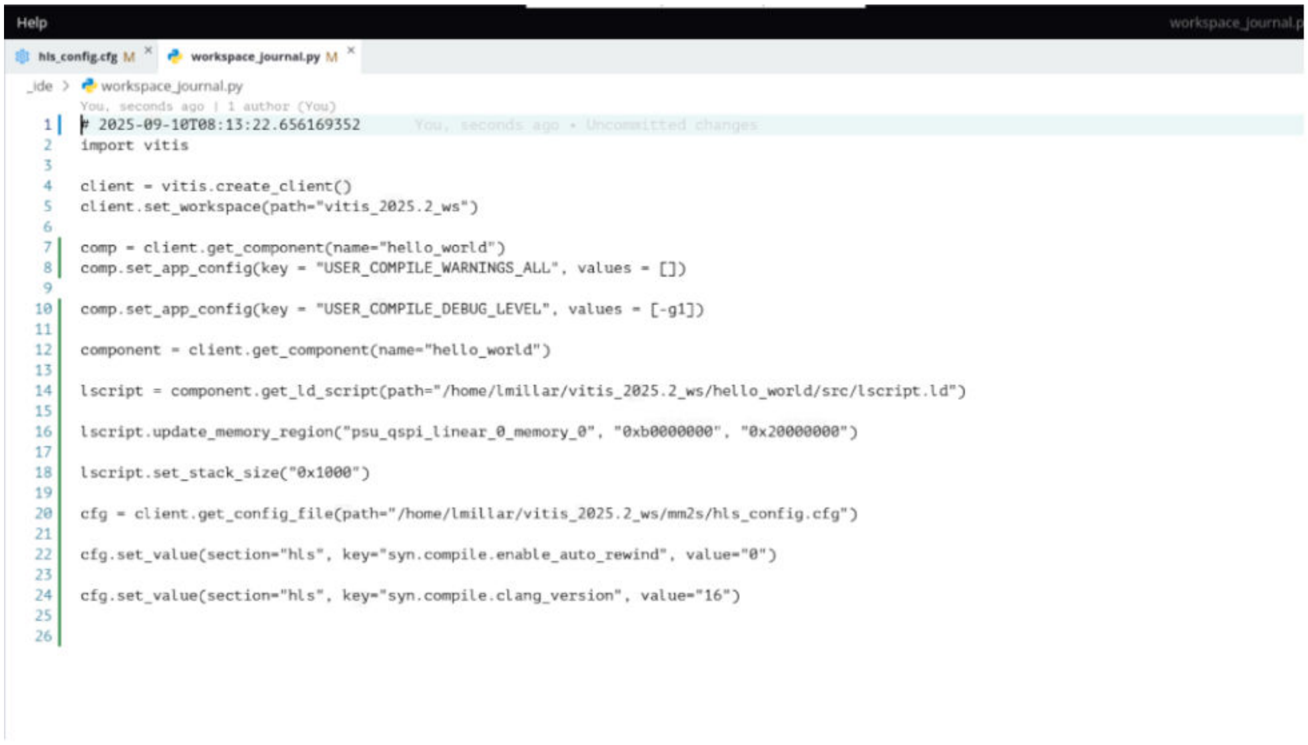
this value
100000

hls.syn.compile.enable_auto_rewind
When true, enables an alternative HLS implementation for pipelined loops which uses automatic loop rewind

hls.syn.compile.ignore_long_run_time
Ignore error for large number of load/store instructions which can cause long run time

The clang version and auto rewind setting are updated in the Python file.

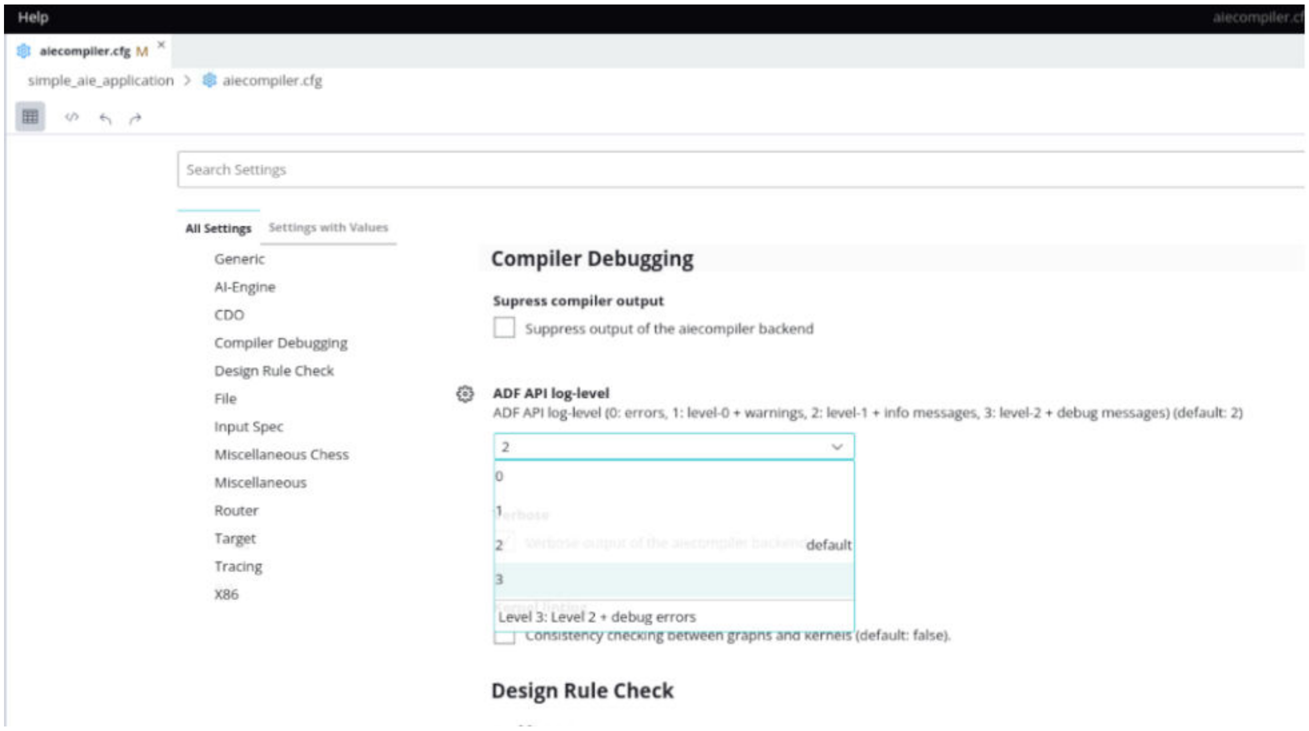
Figure 70: **Python File settings**


Help
workspace_journal.p
hls_config.cfg M
workspace_journal.py M
_ide > workspace_journal.py
You, seconds ago | 1 author (You)
You, seconds ago • Uncommitted changes

```
# 2025-09-10T08:13:22.656169352
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.get_component(name="hello_world")
comp.set_app_config(key = "USER_COMPILE_WARNINGS_ALL", values = [])

comp.set_app_config(key = "USER_COMPILE_DEBUG_LEVEL", values = [-g1])

component = client.get_component(name="hello_world")

lscript = component.get_ld_script(path="/home/lmillar/vitis_2025.2_ws/hello_world/src/lscript.ld")

lscript.update_memory_region("psu_qspi_linear_0_memory_0", "0xb0000000", "0x20000000")

lscript.set_stack_size("0x1000")

cfg = client.get_config_file(path="/home/lmillar/vitis_2025.2_ws/mm2s/hls_config.cfg")

cfg.set_value(section="hls", key="syn.compile.enable_auto_rewind", value="0")

cfg.set_value(section="hls", key="syn.compile.clang_version", value="16")
```

Confirm the AI Engine configuration.

Figure 71: **AI Engine Configuration**



Compiler Debugging

Supress compiler output

Suppress output of the aiecompiler backend

ADF API log-level

ADF API log-level (0: errors, 1: level-0 + warnings, 2: level-1 + info messages, 3: level-2 + debug messages) (default: 2)

2

0
1
2 default
3

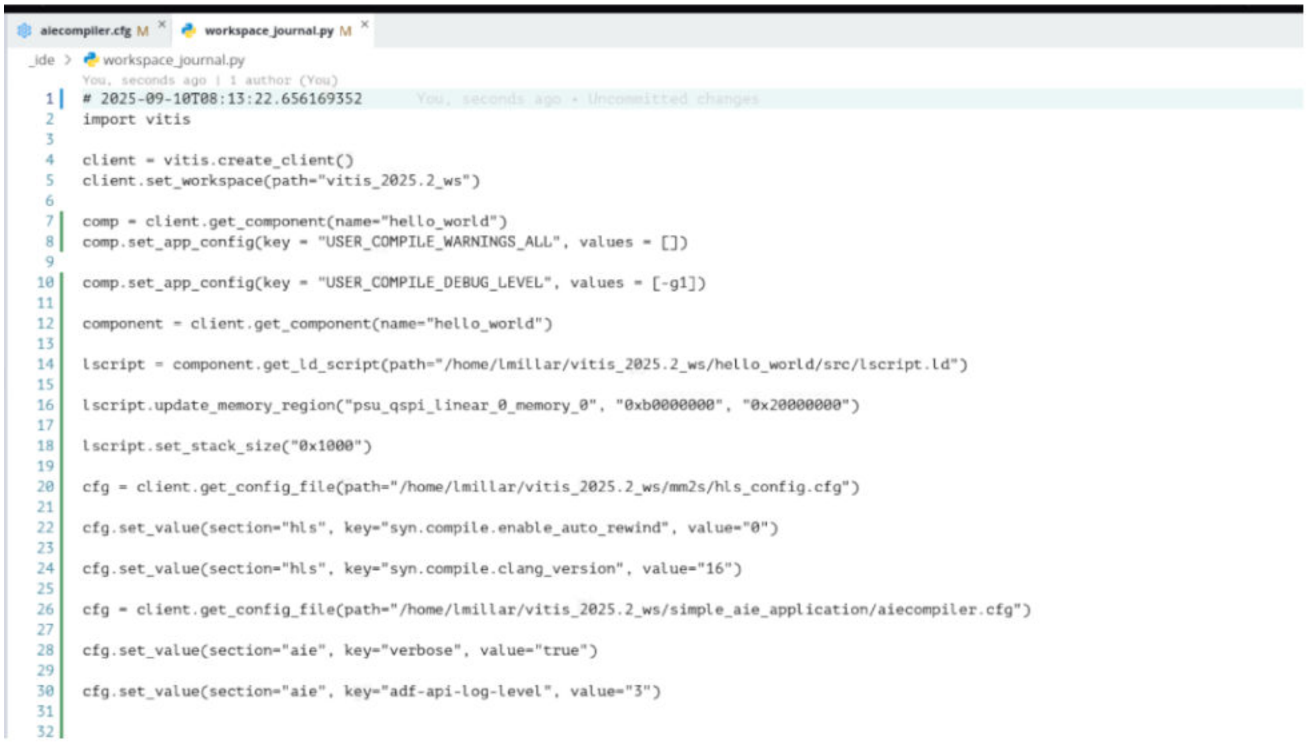
Level 3: Level 2 + debug errors

Consistency checking between graphs and kernels (default: false).

Design Rule Check

Verbose settings and ADF levels are set in the workspace journal.

Figure 72: **Verbose and ADF levels**


aiecompiler.cfg M
workspace_journal.py M
_ide > workspace_journal.py
You, seconds ago | 1 author (You)
You, seconds ago • Uncommitted changes

```
# 2025-09-10T08:13:22.656169352
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

comp = client.get_component(name="hello_world")
comp.set_app_config(key = "USER_COMPILE_WARNINGS_ALL", values = [])

comp.set_app_config(key = "USER_COMPILE_DEBUG_LEVEL", values = [-g1])

component = client.get_component(name="hello_world")

lscript = component.get_ld_script(path="/home/lmillar/vitis_2025.2_ws/hello_world/src/lscript.ld")

lscript.update_memory_region("psu_qspi_linear_0_memory_0", "0xb0000000", "0x20000000")

lscript.set_stack_size("0x1000")

cfg = client.get_config_file(path="/home/lmillar/vitis_2025.2_ws/mm2s/hls_config.cfg")

cfg.set_value(section="hls", key="syn.compile.enable_auto_rewind", value="0")

cfg.set_value(section="hls", key="syn.compile.clang_version", value="16")

cfg = client.get_config_file(path="/home/lmillar/vitis_2025.2_ws/simple_aie_application/aiecompiler.cfg")

cfg.set_value(section="aie", key="verbose", value="true")

cfg.set_value(section="aie", key="adf-api-log-level", value="3")

```

Workspace Journal Coverage

From a workspace management perspective, the workspace journal logs all user actions documented in the previous pages. These include component creation, building and configuration modifications. Specifically, the following actions are logged:

- Component creation
- Component builds
- Modification to configuration files (`UserConfig.cmake`, `lscript.ld`, `hls.config.cfg`, `aie_compiler.cfg`)

The following is an example of a workspace journal file that logs the creation of these components and edits to their configuration files.

```
import vitis

client = vitis.create_client()
client.set_workspace(path="vitis_2025.2_ws")

platform = client.create_platform_component(name =
"vck190_platform",hw_design = "/project_2/ext_platform_wrapper.xsa",os
= "aie_runtime",cpu = "ai_engine",domain_name =
"aie_runtime_ai_engine",generate_dtb = False)

platform = client.get_component(name="vck190_platform")
status = platform.build()

comp = client.create_aie_component(name="simple_aie_application",
platform = "/vitis_2025.2_ws/vck190_platform/export/vck190_platform/
vck190_platform.xpfm", template = "installed_aie_examples/simple")

comp = client.get_component(name="simple_aie_application")
comp.build(target="hw")

comp.build(target="hw")

comp.build(target="hw")

comp = client.create_hls_component(name = "mm2s",cfg_file =
["hls_config.cfg"],template = "empty_hls_component")

comp = client.create_hls_component(name = "s2mm",cfg_file =
["hls_config.cfg"],template = "empty_hls_component")

proj =
client.create_sys_project(name="simple_aie_application_system_project",
platform="/vitis_2025.2_ws/vck190_platform/export/vck190_platform/
vck190_platform.xpfm", template="empty_accelerated_application" ,
build_output_type="xsa")

proj = client.get_sys_project(name="simple_aie_application_system_project")

status = proj.add_container(name="binary_container_1")

proj = proj.add_component(name="mm2s", container_name="binary_container_1")

proj = proj.add_component(name="s2mm", container_name="binary_container_1")

proj = proj.add_component(name="simple_aie_application",
container_name="binary_container_1")

cfg = client.get_config_file(path="/vitis_2025.2_ws/
simple_aie_application_system_project/hw_link/binary_container_1-link.cfg")

cfg.add_lines(values=["param=compiler.addOutputTypes=hw_export"])

cfg = client.get_config_file(path="/vitis_2025.2_ws/simple_aie_application/
aiecompiler.cfg")

cfg.set_value(section="aie", key="event-trace", value="runtime")

cfg.set_value(section="aie", key="xlopt", value="2")

cfg = client.get_config_file(path="/vitis_2025.2_ws/mm2s/hls_config.cfg")

cfg.set_value(section="hls", key="syn.compile.clang_version", value="7")

cfg.set_value(section="hls", key="package.output.format", value="rtl")

vitis.dispose()
```

Note: To view your workspace journal file, in the Vitis Unified IDE navigate to **Vitis→Workspace Journal**.

Python Vitis Commands

Use the following command to launch the Vitis CLI.

```
vitis -i
```

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

You can find Vitis Python command examples in `<VITIS_INSTALL_DIR>/2025.2/cli/examples`.

You can find all the Vitis Python commands to rebuild a Vitis workspace 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>2025.2/Vitis/cli/api_docs/build/html/xsdb.html`

Chapter 12

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 perform debug operations 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, use commands and options instead of functions and arguments because functions require extra typing. The interactive() function in xsdb module 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 - 0xfffffc3f
```

```
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

This release has the following changes:

- XSCT is deprecated
- You can still carry debug using XSDB, or you can convert your scripts to the Python CLI
- You can use the Vitis Python API for project management. You can review the `workspace_journal.py` file to learn how to use the Python API. See Vitis→**Workplace Journal** for details.

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

app

Application project management.

Table 15: **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> Application is created for this core.	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 Vitis tool mentions the system project while creating the component (application). In Python CLI, Vitis tool creates the component first, then adds it to the system project. This gives a flexibility to add 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 15: **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 16: **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 16: **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> Add this library to the BSP settings. This is the default option, so you can specify lib-name directly as an argument without using this 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

Create device tree.

Table 17: **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 where 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 17: **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 18: **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 to 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 18: **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 need to 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 18: **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 that need to be added while building the application created for this domain. **-lib-path** <library-path> Additional library search path that need to be added to the linker settings of the application created for this domain. **-sysroot** <sysroot-dir> The Linux sysroot directory that need to be added to the platform. This sysroot is 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 with all the files listed in the filename 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 a processor.

getperipherals

Get a list of all peripherals in the hardware design.

getprocessors

Get a list of all the processors in a hardware design.

Table 19: **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 20: **getws- Get Vitis Workspace**

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

lscript

Create linker script.

Table 21: **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 21: **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 22: **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 22: **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 22: **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 23: **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 23: **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 24: **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 25: **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/2025.2_1106_0237/installs/lin64/Vitis/
2025.2/cli/vitis/cli_client.py
Type:      method
```

Section VI

GNU Compiler Tools

This section contains the following chapters:

***Note*:** From 2024.1 release, this user guide supports MicroBlaze™ and MicroBlaze™-V devices.

Chapter 13

Overview

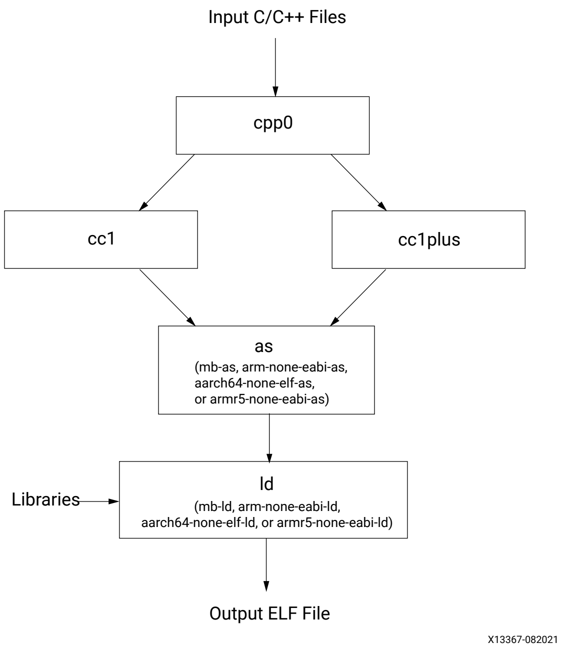
The AMD Vivado™ Design Suite includes the GNU compiler collection (GCC) for the MicroBlaze™ and MicroBlaze™-V processor and the Arm® Cortex® A9, A53, A72, and R5 processors.

- 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 following:
    - The `arm-none-eabi-gcc` and `arm-none-eabi-g++` compilers
    - The `arm-none-eabi-as` assembler
    - The `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 73: **GNU Tool Flow**


Input C/C++ Files
cpp0
cc1
cc1plus
as
(mb-as, arm-none-eabi-as, aarch64-none-elf-as, or armr5-none-eabi-as)
Libraries
ld
(mb-ld, arm-none-eabi-ld, aarch64-none-elf-ld, or armr5-none-eabi-ld)
Output ELF File
X13367-082021

The GNU compilers have the following names:

- **MicroBlaze and MicroBlaze™-V:** `mb-gcc`
- **Cortex A9 cores:** `arm-none-eabi-gcc`
- **Cortex-A53 and Cortex-A72:** `aarch64-none-elf-gcc`
- **Cortex-R5F:** `armr5-none-eabi-gcc`

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 does the following:
  - Converts mnemonics to machine language
  - Resolves some of the labels generated by the compiler
  - Creates an object file that passes on to the linker.

  The assembler executables are named as follows:
  - **MicroBlaze and MicroBlaze™-V:** `mb-as`
  - **Cortex A9 cores:** `arm-none-eabi-as`
  - **Cortex-A53 and Cortex-A72:** `aarch64-none-elf-as`
  - **Cortex-R5F:** `armr5-none-eabi-as`
- **Linker:** Links all object files generated by the assembler. If you provide libraries on the command line, the linker can resolve some undefined references by linking in functions from the assembler. The linker executables are named as follows:
  - **MicroBlaze and MicroBlaze™-V:** `mb-ld`
  - **Cortex A9 cores:** `arm-none-eabi-ld`
  - **Cortex-A53 and Cortex-A72:** `aarch64-none-elf-ld`
  - **Cortex-R5F:** `armr5-none-eabi-ld`

For details of each executable option, see 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++`.

Chapter 15

Common Compiler Usage and Options

Usage

To use the GNU compiler, type the following:

```
<Compiler_Name> options files...
```

Here, `<Compiler_Name>` refers to one of the following compilers:

- **MicroBlaze and MicroBlaze™-V:** `mb-gcc`
- **Cortex A9 cores:** `arm-none-eabi-gcc`
- **Cortex-A53 and Cortex-A72:** `aarch64-none-elf-gcc`
- **Cortex-R5F:** `armr5-none-eabi-gcc`

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:

- **MicroBlaze and MicroBlaze™-V:** `mb-ld`
- **Cortex A9 cores:** `arm-none-eabi-ld`
- **Cortex-A53 and Cortex-A72:** `aarch64-none-elf-ld`
- **Cortex-R5F:** `armr5-none-eabi-ld`

The default extensions for each of these types are 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. This file is generated if you use the `-save-temps` or `-S` options.
- An object file. This file is generated if you use the `-save-temps` or `-c` options.
- Preprocessor output in the form of a `.i` or `.ii` file. This file is generated if you use the `-save-temps` option.

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 26: **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 can have preprocessor directives

Table 26: **File Extensions** *(cont'd)*

Extension	File Type (Dialect)
.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 27: **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.

Both compilers link libraries automatically. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. Vitis modifies the `libxil.a` file 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++. If you have compile C code in a CC file, function names become mangled even if you use the GCC compiler.

The primary difference between GCC and G++ is that G++ automatically sets the default language dialect to C++ irrespective of the file extension. During linking, G++ automatically pulls in the C++ support libraries. This means even if you compile C code in a `.c` file with the G++ compiler, G++ will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is called overloaded if it can perform different actions based on the arguments passed in and return different return values. To support this, C++ compilers encode the type of function invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful on name mangling if you follow a mixed compilation mode. Be careful when some source files contain C code and others contain C++ code. Take note when using GCC for compiling certain files and G++ for compiling others. To prevent name mangling of a C symbol, 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. Using G++ ensures that all required support libraries are linked 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 GNU debugger, mb-gdb or arm-none-eabi-gdb requires the information. 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. Vitis provides this assembler option to the GNU assembler, mb-as or arm-none-eabi-as. If an assembly file compiles 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 28: **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 code, do not use optimization. When an optimized program is debugged through GDB, the results can be 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. Vitis creates the output file 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 options required by tools that are not necessary for the top-level compiler. To run these commands, use the options in the following table.

Table 29: **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 with 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.

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 pass 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 pass 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`: Use this linker option to modify the total memory allocation for the stack. The variable `_STACK_SIZE` is the total space allocation. The `_STACK_SIZE` variable has the default value of 100 words, or 400 bytes. If your program requires more than 400 bytes for stack and heap combined, increase the value of `_STACK_SIZE`. The value is in bytes.

  In certain cases, a program can require a bigger stack. A program requiring a larger stack size than the size available can lead 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`: Allocate the total memory for the heap with 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. Reserved memory and I/O memory make up this address range. The Arm Cortex-A53 and Cortex-A72 processor uses 64-bit logical addresses.

Reserved Memory

The hardware and software programming environment defines reserved memory, especially 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 reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform for more details.

Table 30: **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 are addresses used to communicate with memory-mapped peripherals on the processor buses. These addresses are a part of your hardware platform specification.

User and Program Memory

User and program memory refers to all the memory that the executable requires during execution.

- 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 can partition the various sections of your ELF file across memories using the linker command language. See Linker Scripts for details. The following are situations where changing the memory map of your executable can be necessary.

- 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.

If you are an advanced user, you can modify the default binary data for the reserved memory locations. In this case, you must replace the default startup files and the memory mappings from the linker.

Object-File Sections

You can create an executable file by concatenating input sections from the object files (`.o` files). The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. See the various standard sections of the object file 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.

***Note*:** You do not typically modify the reserved sections, including `.init`, `.fini`, `.ctors`, `.dtors`, `.got`, `.got2`, and `.eh_frame`.

Table 31: **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

- `.text`: This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. You can assign it 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, you can also assign this section to an initialized, read-only memory 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. You must map and initialized it to random access memory (RAM). You cannot map it 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. You must map it 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. You can map it 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. You must map it to RAM.

- `.bss`: This section contains uninitialized data. This section has the `w` (read-write) flag. You must map it 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. You must map this section to RAM.
- `.stack`: This section contains uninitialized data that is used as the program stack. You must map this section to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections can appear merged together into a section named `.bss_stack`.
- `.init`: This section contains language initialization code and has the same flags as `.text`. You must map it to initialized ROM.
- `.fini`: This section contains language cleanup code and has the same flags as `.text`. You must map it to initialized ROM.
- `.ctors`: This section contains a list of functions that must be invoked at program startup and the same flags as `.data`. You must map it to initialized RAM.
- `.dtors`: This section contains a list of functions that must be invoked at program end, the same flags as `.data`. You must map it to initialized RAM.
- `.got2/.got`: This section contains pointers to program data, the same flags as `.data`. You must map it to initialized RAM.
- `.eh_frame`: This section contains frame unwind information for exception handling. It contains the same flags as `.rodata`. You can map it 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`. You must map it to RAM.
- `.tdata`: This section holds initialized thread-local data that contribute to the program memory image. You must map it to initialized RAM.
- `.gcc_except_table`: This section holds language specific data. You must map this section 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. You must map this section to initialized RAM.
- `.fixup`: This section contains information necessary for doing fixup, such as the fixup page table and the fixup record table. You must map this section 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 utility maps sections 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. Define the linker symbol `_TEXT_START_ADDR` on MicroBlaze and MicroBlaze V processors, or START_ADDR on Arm processors. See the following code for an example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The linker uses the following default script from the `$XILINX_/gnu///<processor_name>/lib/ldscripts` area, where`<procname>` = `microblaze`, `<processor_name>` =`microblaze`, and `<platform>` = `lin` or `nt`.

- `elf32<procname>.x` is the default when none of the following cases apply.
- `elf32<procname>.xn` is the default when the linker is invoked with the `-n` option.
- `elf32<procname>.xbn` is the default when the linker is invoked with the `-N` option.
- `elf32<procname>.xr` is the default when the linker is invoked with the `-r` option.
- `elf32<procname>.xu` is the default 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. See the following code for details:

```
compiler -T <linker_script> <Other Options and Input Files>
```

Include the linker script as follows if the linker executes on its own.

```
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. You can generate linker scripts for your program by right clicking the application component and selecting **Reset Link Script**.

Linker scripts assign specific variables or functions to specific memories via section attributes in the C code. Linker scripts can also assign specific object files to sections in memory. To understand these and other features of GNU linker scripts, see the GNU linker section in 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 sections below describe features and options that are unique to the MicroBlaze compiler. 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. This chapter illustrates the new and modified options.

Processor Feature Selection Options

- `-mcpu=vX.YY.Z`: This option directs the compiler to generate code for the MicroBlaze hardware version v.X.YY.Z. To get the 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 a 3-stage processor pipeline mode. Does not inhibit exceptions that move instructions into delay slots.
  - **v3.00.a and v4.00.a:** Uses a 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
  - **v5.00.a and later:** Uses a 5-stage processor pipeline model. Does not inhibit exception causing instructions moving into delay slots.
- `-mlittle-endian`/`-mbig-endian`: Use these options to select the endianness of the target machine for the compiling code.

- The options set the endianness of the binary object file 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.
  - Use this option 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 using this switch. This allows you to write C or assembly code tailored to the hardware.
- `-mxl-multiply-high`: The MicroBlaze processor has an option to enable instructions that can compute the higher 32 bits of a 32x32-bit multiplication.
  - This option tells the compiler to use multiply high instructions.
  - When using this switch, the compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH`. This allows you to write C or assembly code tailored to the hardware
- `-mno-xl-multiply-high`: This option tells the compiler to 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. A call replaces every 32-bit mulitply operation 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 it can use hardware divide instructions in the compiling program.
  - This option can improve the performance of your program if it has a significant amount of division operations.
  - When using this switch, the compiler automatically defines the C pre-processor definition `HAVE_HW_DIV`. This allows you to write C or assembly code tailored to the hardware.
- `-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`: Use this option to configure the barrel shift feature of the MicroBlaze processor.
  - 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 using this switch. This allows you to write C or assembly code tailored to the hardware.
- `-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. Examples are `strcpy`, `strlen`, and `strcmp`.
  - On a program heavily dependent on string manipulation routines, the speed increase can be significant.
  - The compiler automatically defines the C pre-processor definition `HAVE_HW_PCMP` when using this script. This allows you to write C or assembly code tailored to the hardware.
- `-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.
  - This option 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 using this script. This allows you to write C or assembly code tailored to the hardware.
- `-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. MicroBlaze natively decodes and executes these instructions when the FPU is enabled in hardware and these instructions are enabled.
- `-mxl-float-sqrt`: This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. MicroBlaze natively decodes and executes these instructions when the FPU is enabled in hardware and these 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. When you enable the hardware divider, this switch has no effect.
- `-mxl-gp-opt`: If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), load and store operations require two instructions.

  The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 KB of data. You can access any memory location within these areas using the small data area anchors and a 16-bit immediate value. This means they only need 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. You can store variables smaller than a threshold value in these areas. You can address the variables with fewer instructions. Vitis calculates the addresses during the linking stage.

  IMPORTANT! *When using this option, provide it 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 allocate in the .bss section. The variables have the value 0 when the program starts execution. The C startup files run a loop to fill the .bss section with zero when the program starts execution. For global variables that are zero in C code, optimizing compilers also allocates these 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. It is possible for this option to improve startup times for your application. Use this option with care, ensuring that one of the following is true:

  - You do not use code that relies on global variables initialized to zero.
  - 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. Command line for `mb-gcc` does not require it. This uses the startup file `crt0.o`.

- `xl-mode-bootstrap`: This option is used for applications that a bootloader loads. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If this bootloader loads a normal executable, the application reset vector overwrites the reset vectors. In such a scenario, on a processor reset, the bootloader does not execute first to reload this application.

  To prevent this, compile the bootloaded application with this compiler flag. On a processor reset, control reaches the bootloader instead of the application.

  This mode uses `crt2.o` as a startup file. Do not use this switch on an application deployed in a scenario different from the one described above.

- `-xl-mode-novectors`: Use this option for applications that do not require any of the MicroBlaze vectors. This is typically 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.*

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.
- Linux uses this feature 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.
- It also uses 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 using large immediate values. You do not need 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, if the branch immediate if equal (`beqi`) instruction requires a PC-relative operand, use the following instruction.

```
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, you need to 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 32: **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.

- `-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 are using the default start address set by the compiler.

  This is a linker option 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 it cannot determine the value of the immediate one. Most of these instructions do not need an `imm` instruction. These are removed by the linker when the `-relax` command line option is provided.

  The assembler only requires this option when invoking the linker on its own. When the linker is invokes through the `mb-gcc` compiler, the system automatically provides this option.

- `-N`: Required only for MicroBlaze programs, this option sets the text and data section as readable and writable. It does not page-align the data segment. The top-level GCC compiler automatically includes this option while invoking the linker. If you intend to invoke the linker without using GCC, use this option.

  The MicroBlaze linker uses linker scripts to assign sections to memory. See the following for details.

MicroBlaze Linker Script Sections

The following table lists the input sections that MicroBlaze linker scripts assign.

Table 33: **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 assign to the appropriate memories (the MicroBlaze hardware defines this).
- 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 you initialize all uninitialized memory to startup. This is 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 need to 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.

The compiler uses end files to include code that must execute after your program ends. End files typically perform the following:

- Invoke language cleanup functions, such as C++ destructors.
- Deinitialize the hardware subsystem. For example, during program profiling, clean up the profiling sub-system.

The following table lists the register names, values, and descriptions in the C runtime files.

Table 34: **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 for programs executing in standalone mode, without the use of any bootloader or debugging stub. The file 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 for application debugging 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 for executable loading 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. The bootloader can reload and restart the program.
- `crt3.o`: This initialization file is for the circumstance 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. The GNU linking mechanism does not pull in any of the interrupt and exception handling related routines because the other vectors are not populated. This helps save code space.

Second Stage Initialization Files

It is a requirement to initialize all global and static variables to 0 according to C standard specification.

- Initialization all global and static variables is a common requirement of all CRTs above.
- The separate routine `_crtinit` 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 can 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`: The profiling process uses the second stage startup file. This file 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`: When the compiler uses the `-mno-clearbss` switch, it uses this second-stage startup file. This file 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`: During profiling with the `-mno-clearbss` switch, this second-stage startup file 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

Vivado distributes initialization files in both precompiled and source form.

The pre-compiled object files are in the compiler library directory.

Sources for the initialization files for the MicroBlaze GNU compiler are in `<AMD_>/Vitis/<version>/data/embeddedsw/lib/microblaze/src/`.

`<AMD_>` is the Vivado installation path and `<version>` is the release version of the Vitis software platform.

To fulfill a custom startup file requirement, take the files from the source area and include them as a part of your application source. 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 in the source code. They are available in the installation to be used as is. Aim 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, it is recommended to eliminate all sources of overhead. The following describes reducing the overhead of invoking C++ constructor or destructor code in a C program that does not require that code. It is possible 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. If your bootstrapping and profiling is on your application, 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 the files into `.o` files and place them in a directory of your choice. You can also include the files 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. Vivado ships precompiled versions of these libraries. 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 need to use, look in the following folder:

```
$XILINX_/gnu/microblaze/<platform>/microblaze-xilinx-elf/lib
```

Compiler flags and configurations for compiling the library determine filename encoding. 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 35: **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. Double-precision floating-point can fail to fully use the optional single-precision floating-point features on MicroBlaze.

The newlib math libraries have alternate versions that implement these math functions using single-precision arithmetic. It is possible for single-precision libraries to make direct use of the MicroBlaze processor hardware floating point unit (FPU) and can therefore perform better.

If 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 Vivado's C math libraries do not support multi-threaded environments. 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. C runtime routines initialize the command line arguments `argc` and `argv`.

Interrupt Handlers

Interrupt handlers use different compilation methods compared to normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save volatile registers they use. Interrupt handlers 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. The following is the definition of the attribute:

  ```
  void function_name () __attribute__ ((interrupt_handler));
  ```

  Note**:** The attribute for the interrupt handler should only exist in the prototype and *not* in the definition.

  Interrupt handlers can also call other functions, which can 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**:** On-leaf functions are functions that have calls to other sub-routines.

  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 a C function uses the attribute `fast_interrupt`, MicroBlaze saves only minimal registers.

  ```
  void function_name () __attribute__ ((fast_interrupt));
  ```

Table 36: **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, the attribute only saves the volatiles the function use.
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. You can use these flags in the steps above. 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.

Chapter 18

Other Notes

C++ Code Size

A GCC toolchain with the latest open source C++ standard library (`libstdc++-v3`) can generate larger code and data fragments than 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.

Advanced users can use the `-fno-exceptions` and/or the `-fno-rtti` switches to remove the overhead and optimize for size. Refer to the GCC documentation for more specific information on available compiler options and their impact.

C++ programs can 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, the platform supports 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 unsuitable 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 the AMD compiler supports both features, Vivado only provides a standalone platform and do not support the rest of the libraries and tools. Debuggers cannot interpret relocatable code and perform the correct relocations at runtime. AMD libraries, startup files and other tools do not support independent code features. Third-party OS vendors can use these features as a standard in their distribution and tools.

Other Switches and Features

AMD Vivado compilers and platforms do not support all switches and features. An example is `-fprofile-arcs`. Some features are experimental in nature (as defined by open source GCC) and can produce incorrect code if used inappropriately. Refer to the GCC documentation for more information on specific features.

Embedded Design Tutorials

The following hardware specific embedded design tutorials are available for embedded software designers.

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

To learn more about the software features, refer to *Vitis Unified Software Platform Tutorials Landing Page* (UG1605).

Section VIII

Drivers and Libraries

Drivers and libraries are 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
11/20/2025 Version 2025.2	
General Updates	Improved the content, including reducing the length of long sentences, rewriting to be clearer, correcting typos, and generally making the easier to read and understand.
Source Control for Platform BSP	Added new topic.
Terminating a Backend Process	Added content for the Vitis Unified IDE to cancel or terminate backend process.
Python API: A Command-line Tool for Creating and Managing Projects in Vitis	Added content for the Vitis Unified IDE to extend Python CLI Support for Configuration Files and Journal Logging.
Enabling the Vitis API to be used in a Python ENV	Added content for the Vitis Unified IDE to enable Vitis API to be used in a Python ENV.
Cross Operating System Support	Added content for Vitis Unified IDE support for unifying file path mechanisms for Linux and Windows support.
PS Trace	Added content for Vitis Unified IDE feature to enable PS Trace production.
Debugging Zephyr Applications on Microblaze V	Added tutorial to set up AMD's Zephyr fork on MicroBlaze V, generate SDT/DTS with lopper, build the Zephyr ELF, and debug it in Vitis Unified IDE for AC701 or compatible designs.
Modifying Configuration Files	While the Python API can be used for creating and building Vitis components it can also be used for modifying the configuration files of each component.
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

Copyright