

# Zynq UltraScale+ MPSoC: Embedded Design Tutorial

## *A Hands-On Guide to Effective Embedded System Design*

UG1209 (v2019.2) October 30, 2019

## Revision History

The following table shows the revision history for this document.

Section	Revision Summary
<b>10/30/2019 Version 2019.2</b>	
Updated for Vitis™ unified software platform	Migrated the flow to Vitis™ unified software platform.
General updates	Validated with Vitis IDE and PetaLinux 2019.2.
<b>07/03/2019 Version 2019.1</b>	
General updates	Validated with Vivado® Design Suite and PetaLinux 2019.1.

# Table of Contents

Revision History .....	2
<b>Chapter 1: Introduction</b>	
About This Guide .....	5
How Zynq UltraScale+ Devices Offer a Single Chip Solution .....	6
How the Xilinx Design Tools Expedite the Design Process .....	9
What You Need to Set Up Before Starting .....	10
<b>Chapter 2: Zynq UltraScale+ MPSoC Processing System Configuration</b>	
Zynq UltraScale+ System Configuration .....	13
Example Project: Creating a New Embedded Project with Zynq UltraScale+ MPSoC .....	14
<b>Chapter 3: Build Software for PS Subsystems</b>	
Processing Units in Zynq UltraScale+ .....	29
Example Project: Running the “Hello World” Application from Arm Cortex-A53 .....	30
Example Project: Running the “Hello World” Application from Arm Cortex-R5 .....	34
Additional Information .....	37
Example Project: Create a Bare-Metal Application Project in the Vitis IDE .....	38
Reviewing Software Projects in the Platform .....	42
Example Project: Create Linux Images using PetaLinux .....	47
<b>Chapter 4: Debugging with the Vitis Debugger</b>	
Xilinx System Debugger .....	53
Debugging Software Using the Vitis Debugger .....	55
Debugging Using XSCT .....	58
<b>Chapter 5: Boot and Configuration</b>	
System Software .....	69
Linux on APU and Bare-Metal on RPU .....	71
Boot Sequence for SD-Boot .....	71
Boot Sequence for QSPI Boot Mode .....	81
Boot Sequence for QSPI-Boot Mode Using JTAG .....	94
Boot Sequence for USB Boot Mode .....	97

Secure Boot Sequence .....	104
----------------------------	-----

## Chapter 6: System Design Examples

Design Example 1: Using GPIOs, Timers, and Interrupts .....	137
Design Example 2: Example Setup for Graphics and Display Port Based Sub-System .....	158

## Appendix A: Debugging Problems with Secure Boot

Determine if PUF Registration is Running .....	165
Read the Boot Image .....	165

## Appendix B: Additional Resources and Legal Notices

Xilinx Resources .....	166
Solution Centers .....	166
Documentation Navigator and Design Hubs .....	166
Design Files for This Tutorial .....	167
Xilinx Resources .....	167
Training Resources .....	168
Please Read: Important Legal Notices .....	169

# Introduction

---

## About This Guide

This document provides an introduction to using the Xilinx® Vivado® Design Suite flow for using the Zynq® UltraScale+™ MPSoC device. The examples are targeted for the Xilinx ZCU102 Rev 1.0 and Rev 1.1 evaluation boards. The tool used is the Vitis™ unified software platform.

The examples in this document were created using the Xilinx tools running on Windows 10, 64-bit operating system, and PetaLinux on Linux 64-bit operating system. Other versions of the tools running on other Window installs might provide varied results. These examples focus on introducing you to the following aspects of embedded design.

**Note:** The sequence mentioned in the tutorial steps for booting Linux on the hardware is specific to the PetaLinux tools released for 2019.2, which must be installed on the Linux host machine for exercising the Linux portions of this document.

- [Chapter 2, Zynq UltraScale+ MPSoC Processing System Configuration](#) describes the creation of a system with the Zynq UltraScale+ MPSoC Processing System (PS) and the creation of a hardware platform for Zynq UltraScale+ MPSoC. This chapter is an introduction to the hardware and software tools using a simple design as the example.
- [Chapter 3, Build Software for PS Subsystems](#) describes the steps to configure and build software for processing blocks in processing system, including application processing unit (APU), real-time processing unit (RPU). Creation of bare metal applications targeting on application processing unit (APU) and RPU is also included. Review of boot components in hardware platform.
- [Chapter 4, Debugging with the Vitis Debugger](#) provides an introduction to debugging software using the debug features of the Vitis IDE. This chapter uses the previous design and runs the software bare metal (without an OS) to show how to debug. This chapter also lists Debug configurations for Zynq UltraScale+ MPSoC.
- [Chapter 5, Boot and Configuration](#) shows integration of components to configure and create Boot images for a Zynq UltraScale+ system. The purpose of this chapter is to understand how to integrate and load Boot loaders.
- [Chapter 6, System Design Examples](#) highlights how you can use the software blocks you configured in [Chapter 3](#) to create a Zynq UltraScale+ system.

## Document Audience and Scope

The purpose of this guide is to familiarize software application developers, system software designers, and system hardware designers by providing the following:

- Tutorials for creating a Zynq UltraScale+ MPSoc System
- Tutorials on building software for the PS subsystem
- Tutorials on debugging using the Vitis IDE
- System design examples

## Example Project

The best way to learn a tool is to use it. This guide provides opportunities for you to work with the tools under discussion. Specifications for sample projects are given in the example sections, along with an explanation of what is happening behind the scenes. Each chapter and examples are meant to showcase different aspects of embedded design. The example takes you through the entire flow to complete the learning and then moves on to another topic.

## Additional Documentation

Additional documentation is listed in [Appendix B, Additional Resources and Legal Notices](#).

---

# How Zynq UltraScale+ Devices Offer a Single Chip Solution

Zynq UltraScale+ MPSoC, the next generation Zynq device, is designed with the idea of using the right engine for the right task. The Zynq UltraScale+ comes with a versatile Processing System (PS) integrated with a highly flexible and high-performance Programmable Logic (PL) section, all on a single System on Chip (SoC). The Zynq UltraScale+ MPSoC PS block includes engines such as the following:

- Quad-core Arm Cortex-A53 based Application Processing Unit (APU)
- Dual-core Arm Cortex-R5 based Real Time Processing Unit (RPU)
- Arm Mali-400 MP2 based Graphics Processing Unit (GPU)
- Dedicated Platform Management Unit (PMU) and Configuration Security Unit (CSU)
- List of High Speed peripherals, including Display port and SATA

The Programmable Logic Section, in addition to the programmable logic cells, also comes integrated with few high performance peripherals, including the following:

- Integrated Block for PCI Express
- Integrated Block for Interlaken
- Integrated Block for 100G Ethernet
- System Monitor
- Video Codec Unit

The PS and the PL in Zynq UltraScale+ can be tightly or loosely coupled with a variety of high performance and high bandwidth PS-PL interfaces.

To simplify the design process for such sophisticated devices, Xilinx offers the Vivado Design Suite, Vitis IDE, and PetaLinux Tools for Linux. This set of tools provides you with everything you need to simplify embedded system design for a device that merges an SoC with an FPGA. This combination of tools enables hardware and software application design, code execution and debug, and transfer of the design onto actual boards for verification and validation.

## Vitis Integrated Design Environment (IDE)

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

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

For more information about the Eclipse development environment, refer to <http://www.eclipse.org>.

Other components include:

- Drivers and libraries for embedded software development
- Linaro GCC compiler for C/C++ software development targeting the Arm Cortex-A53 and Arm Cortex-R5F MPCore processors in the Zynq UltraScale+ Processing System.

## The Vivado Design Suite

The Vivado Design Suite offers a broad range of development system tools for FPGA implementation. It can be installed as a standalone tool when software programming is not required. It is also a part of the Vitis IDE installation. Various Vivado Design Suite editions can be used for embedded system development. In this guide the System Edition installed with the Vitis IDE is used. The Vivado Design Suite editions are shown in the following figure.

### Vivado Design Suite - HLx Editions

Vivado Design Suite - HLx Edition Features	Vivado HL Design Edition	Vivado HL System Edition	Vivado Lab Edition	Vivado HL WebPACK Edition (Device Limited)	Free 30-day Evaluation
<b>Accelerating Implementation</b>					
Synthesis and Place and Route	•	•		•	•
Partial Reconfiguration*	•	•		•	•
<b>Accelerating Verification</b>					
Vivado Simulator	•	•		•	•
Vivado Device Programmer	•	•	•	•	•
Vivado Logic Analyzer	•	•	•	•	•
Vivado Serial I/O Analyzer	•	•	•	•	•
Debug IP (ILA/VIO/IBERT)	•	•		•	•
<b>Accelerating High Level Design</b>					
Vivado High-Level Synthesis	•	•		•	•
Vivado IP Integrator	•	•		•	•
System Generator for DSP		•			•

\* Can be purchased as an option.

Figure 1-1: Vivado Design Suite Editions



## Other Vivado Components

Other Vivado components include:

- Embedded/Soft IP for the Xilinx embedded processors
- Documentation
- Sample projects

## PetaLinux Tools

The PetaLinux tools set is an Embedded Linux System Development Kit. It offers a multi-faceted Linux tool flow, which enables complete configuration, build, and deploy environment for Linux OS for the Xilinx Zynq devices, including Zynq UltraScale+.

For more information, see the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 7].

The PetaLinux Tools design hub provides information and links to documentation specific to PetaLinux Tools. For more information, see [Documentation Navigator and Design Hubs](#).

---

## How the Xilinx Design Tools Expedite the Design Process

You can use the Vivado Design Suite tools to add design sources to your hardware. These include the IP integrator, which simplifies the process of adding IP to your existing project and creating connections for ports (such as clock and reset).

You can accomplish all your hardware system development using the Vivado tools along with IP integrator. This includes specification of the Zynq UltraScale+ Processing System, peripherals, and the interconnection of these components, along with their respective detailed configuration.

The Vitis IDE can be used for software development, hardware acceleration, and platform development. It also be used to debug software applications.

The Zynq UltraScale+ Processing System (PS) can be booted and run without programming the FPGA (programmable logic or PL). However, in order to use any soft IP in the fabric, or to bond out PS peripherals using EMIO, programming of the PL is required. You can program the PL using the Vitis IDE or using the Vivado Hardware Manager.

For more information on the embedded design process, refer to the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) [Ref 2].

For more information about the Zynq UltraScale+ Processing System, refer to the *Zynq UltraScale+ Processing System Product Guide* (PG201) [Ref 8].

---

## What You Need to Set Up Before Starting

Before discussing the tools in depth, you should make sure they are installed properly and your environments match the requirements mentioned in the "Example Project" section of this guide.

### Hardware Requirements for this Guide

This tutorial targets the Zynq UltraScale+ ZCU102 evaluation board. The examples in this tutorial were tested using the ZCU102 Rev 1 board. To use this guide, you need the following hardware items, which are included with the evaluation board:

- ZCU102 Rev1 evaluation board
- AC power adapter (12 VDC)
- USB Type-A to USB Micro cable (for UART communications)
- USB Micro cable for programming and debugging via USB-Micro JTAG connection
- SD-MMC flash card for Linux booting
- Ethernet cable to connect target board with host machine
- Monitor with Display Port (DP) capability and at least 1080P resolution.
- DP cable to connect the Display output from ZCU102 Board to a DP monitor.

### Installation Requirements

#### *Vitis Integrated Design Environment and Vivado Design Suite*

Make sure that you have installed the 2019.2 Vitis IDE. Visit <https://www.xilinx.com/support/download.html> to confirm that you have the latest tools version.

Ensure that you have the Vitis 2019.2 software development platform installed. The Vitis IDE is a Xilinx unified tool which comes with all the hardware and software as a package. If you install the Vitis IDE, you will automatically get both the Vivado Design Suite and the Vitis IDE. You do not have to make any extra selections in the installer. The installation and selection window is shown below.

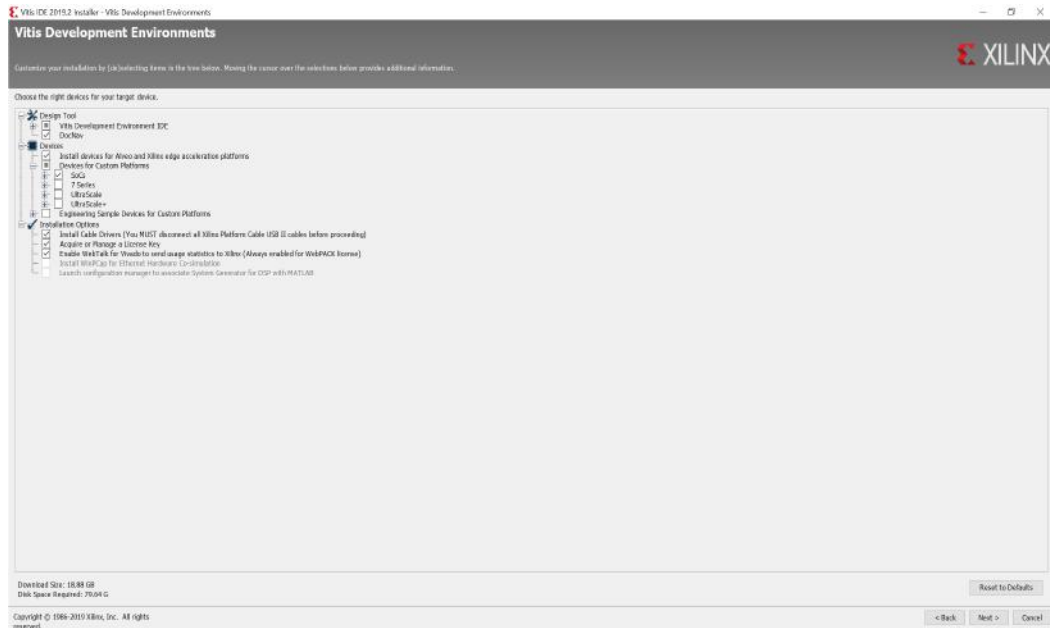


Figure 1-2: Vitis IDE Installer with Vivado

For more information on installing the Vivado Design Suite, refer to the *Vitis Embedded Software Development Flow Documentation* [Ref 13].

## PetaLinux Tools

Install the PetaLinux Tools to run through the Linux portion of this tutorial. PetaLinux tools run under the Linux host system running one of the following:

- Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6 (64-bit)
- CentOS Workstation/Server 7.4, 7.5, 7.6 (64-bit)
- Ubuntu Linux Workstation/Server 16.04.5, 16.04.6, 18.04.1, 18.04.02 (64-bit)

This can use either a dedicated Linux host system or a virtual machine running one of these Linux operating systems on your Windows development platform.

When you install PetaLinux Tools on your system of choice, you must do the following:

- Download PetaLinux 2019.2 software from the [Xilinx Website](#).
- Download the ZCU102 PetaLinux BSP (ZCU102 BSP (prod-silicon)) from the [2019.2 downloads](#) page.
- Add common system packages and libraries to the workstation or virtual machine. For more information, see the Installation Requirements from the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 7].

### Prerequisites

- 8 GB RAM (recommended minimum for Xilinx tools)
- 2 GHz CPU clock or equivalent (minimum of 8cores)
- 100 GB free HDD space

### Extract the PetaLinux Package

PetaLinux tools installation is straight-forward. Without any options, the PetaLinux tools are installed into the current working directory. Alternatively, an installation path may be specified.

For example: To install PetaLinux tools under `/opt/pkg/petalinux/2019.2`:

```
$ mkdir -p /opt/pkg/petalinux/2019.2
$ ./petalinux-v2019.2-final-installer.run /opt/pkg/petalinux/2019.2
```

For more information, see *PetaLinux Tools Documentation: Reference Guide* [Ref 7].

### Software Licensing

Xilinx software uses FLEXnet licensing. When the software is first run, it performs a license verification process. If the license verification does not find a valid license, the license wizard guides you through the process of obtaining a license and ensuring that the license can be used with the tools installed. If you do not need the full version of the software, you can use an evaluation license. For installation instructions and information, see the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) [Ref 3].

### Tutorial Design Files

See [Design Files for This Tutorial](#) for information about downloading the design files for this tutorial.

# Zynq UltraScale+ MPSoC Processing System Configuration

Now that you have been introduced to the Xilinx® Vivado® Design Suite, you will learn how to use it to develop an embedded system using the Zynq® UltraScale+™ MPSoC Processing System (PS).

The Zynq UltraScale+ device consists of Quad-Core Arm® Cortex®-A53 based APU, Dual-Core Arm Cortex-R5 RPU, Mali 400 MP2 GPU, and many hard Intellectual Property components (IPs), and Programmable Logic (PL). This offering can be used in two ways:

- The Zynq UltraScale+ PS can be used in a standalone mode, without attaching any additional fabric IP.
- IP cores can be instantiated in fabric and attached to the Zynq UltraScale+ PS as a PS+PL combination.

---

## Zynq UltraScale+ System Configuration

Creation of a Zynq UltraScale+ system design involves configuring the PS to select the appropriate boot devices and peripherals. To start with, as long as the PS peripherals and available MIO connections meet the design requirements, no bitstream is required. This chapter guides you through creating a simple PS-based design that does not require a bitstream.

In addition to the basic PS configuration, this chapter will briefly touch upon the concept of Isolation Configuration to create subsystems with protected memory and peripherals. This advanced configuration mode in the PS Block enables you to setup subsystems comprising Masters with dedicated memory and peripherals. The protection is provided by the XMPU and the XPPU in Zynq UltraScale+ PS block. The isolation configuration also allows the TrustZone settings for components to create and configure the systems in Secure and Non-Secure Environments.

## Example Project: Creating a New Embedded Project with Zynq UltraScale+ MPSoC

For this example, you will launch the Vivado Design Suite and create a project with an embedded processor system as the top level.

### Starting Your Design

1. Start the Vivado Design Suite.
2. In the Vivado Quick Start page, click **Create Project** to open the New Project wizard.
3. Use the information in the table below to make selections in each of the wizard screens.

Table 2-1: New Project Wizard Options

Wizard Screen	System Property	Setting or Command to Use
Project Name	Project name	edt_zcu102
	Project Location	C:/edt
	Create Project Subdirectory	Leave this checked
Project Type	Specify the type of sources for your design. You can start with RTL or a synthesized EDIF.	<b>RTL Project</b>
	<b>Do not specify sources at this time</b> check box	Leave this unchecked.
Add Sources	Do not make any changes to this screen.	
Add Constraints	Do not make any changes to this screen.	
Default Part	Select	<b>Boards</b>
	Display Name	<b>Zynq UltraScale+ ZCU102 Evaluation Board</b>
New Project Summary	Project Summary	Review the project summary

4. Click **Finish**. The New Project wizard closes and the project you just created opens in the Vivado design tool.

### Creating a Block Design Project

You will now use the IP Integrator to create a Block Design project.

1. In the Flow Navigator, under **IP Integrator**, click **Create Block Design**.

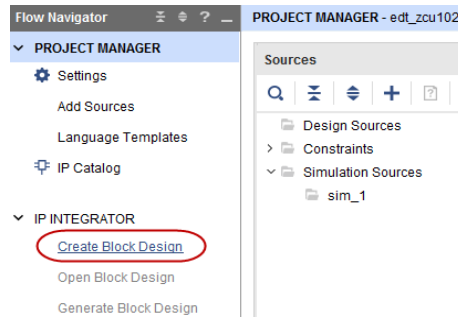


Figure 2-1: Create Block Design Button

The Create Block Design wizard opens.

- Use the following information to make selections in the Create Block Design wizard.

Table 2-2: Setting in Create Block Design Wizard

Wizard Screen	System Property	Setting or Command to Use
Create Block Design	Design Name	edt_zcu102
	Directory	<Local to Project>
	Specify Source Set	<b>Design Sources</b>

- Click **OK**.

The Diagram window view opens with a message that states that this design is empty. To get started, you will next add some IP from the catalog.

- Click the **Add IP** button **+**.
- In the search box, type **zynq** to find the Zynq device IP.
- Double-click the **ZYNQ UltraScale+ MPSoC** IP to add it to the Block Design.

The Zynq UltraScale+ MPSoC processing system IP block appears in the Diagram view, as shown in the following figure.



Figure 2-2: Zynq UltraScale+ MPSoC Processing System IP Block





3. You will now use a preset template created for the ZCU102 board. Click the **Run Block Automation** Link.

The Run Block Automation dialog box opens.

4. Click **OK** to accept the default processor system options and make default pin connections.

This configuration wizard enables many peripherals in the Processing System with some multiplexed I/O (MIO) pins assigned to them according to the board layout of the ZCU102 board. For example, UART0 and UART1 are enabled. The UART signals are connected to a USB-UART connector through UART to the USB converter chip on the ZCU102 board.

5. To verify, double-click on the **Zynq UltraScale+ Processing System** block in the block diagram window.

Note the check marks that appear next to each peripheral name in the Zynq UltraScale+ device block diagram, signifying the I/O Peripherals that are active.

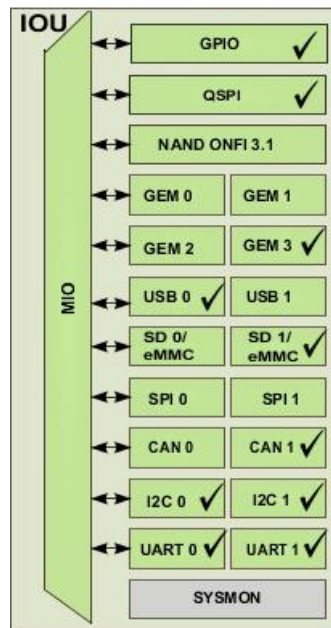


Figure 2-5: I/O Unit with Active Peripherals Identified

- In the block diagram, click one of the green I/O Peripherals, as shown in the previous figure. The I/O Configuration dialog box opens for the selected peripheral.

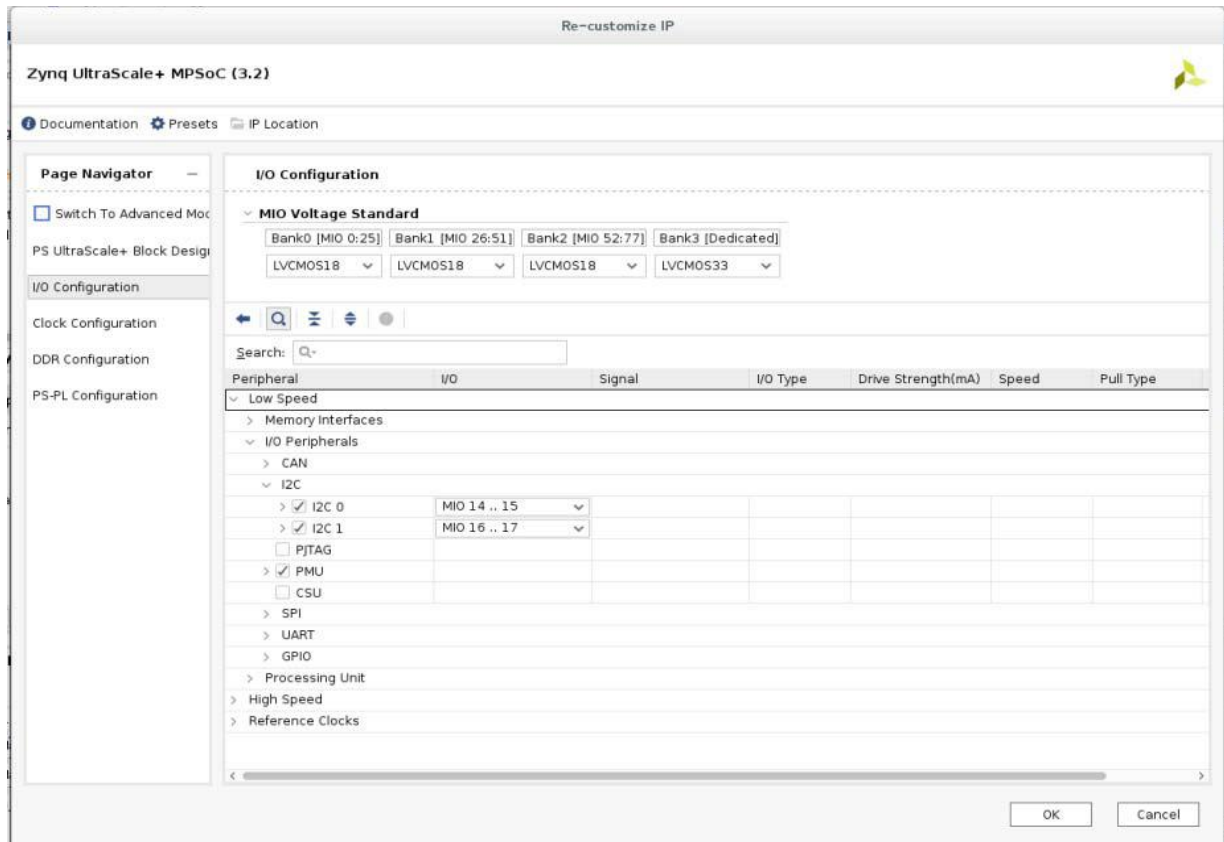


Figure 2-6: I/O Configuration Page of the Re-customize IP Dialog Box

This page enables you to configure low speed and high speed peripherals. For this example, you will continue with the basic connection enabled using Board preset for ZCU102.

- In the Page Navigator, select **PS-PL Configuration**.
- In PS-PL Configuration, expand **PS-PL Interfaces** and expand the **Master Interface**.

For this example, because there is no design in PL, you can disable the PS-PL interface. In this case, AXI HPM0 FPD and AXI HPM1 FPD Master Interfaces can be disabled.

- De-select **AXI HPM0 FPD** and **AXI HPM1 FPD**.

The PS-PL configuration looks like following figure.

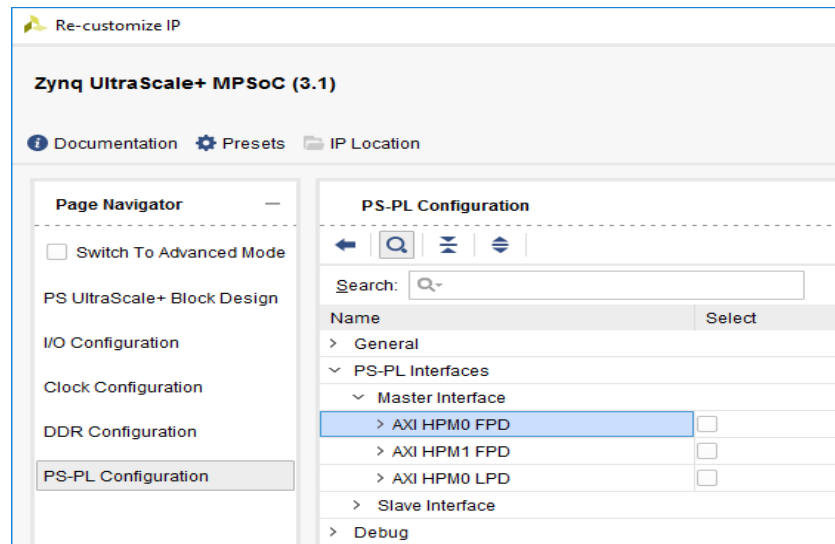


Figure 2-7: PS-PL Configuration

10. Click **OK** to close the Re-customize IP wizard.

## Isolation Configuration

This section is for reference only. It explains the importance of Isolation Configuration settings for different use-cases. Different use-cases may need to establish Isolation Configurations on an as-needed basis. Isolation configuration is optional and you can set it as per your system requirement. Safety/Security critical use cases typically require isolation between safe/non-safe or secure/non-secure portions of the design. This requires a safe/secure region that contains a master (such as the RPU) along with its slaves (memory regions and peripherals) to be isolated from non-safe or non-secure portions of the design. In such cases, the TrustZone attribute can be applied to the dedicated peripherals or memory locations. In this way only a valid and trusted master can access the secure slaves. An other use-case requiring Isolation is for Platform and Power management. In this case, independent subsystems can be created with Masters and slaves. This is used to identify dependencies during run-time power management or warm restart for upgrade or recovery. An example of this use-case can be found on the [Zynq UltraScale+ Restart solution](#) wiki page. The Xilinx Memory Protection Unit (XMPU) and Xilinx Peripheral Protection Unit (XPPU) in Zynq UltraScale+ provide hardware protection for memory and peripherals. These protection units complement the isolation provided by TrustZone (TZ) and the Zynq UltraScale+ MPSoC SMMU.

The XMPU and XPPU in Zynq UltraScale+ allow Isolation of resources at SoC level. Arm MMU and Trustzone enable Isolation within Arm Cortex-A53 Core APU. Hypervisor and SMMU allows setting Isolation between Cortex-A53 cores. From a tools standpoint, these Protection Units can be configured using Isolation Configuration in Zynq UltraScale+ PS IP wizard. The Isolation settings are exported as an initialization file which is loaded as a part

of the bootloader, in this case the First Stage Boot Loader (FSBL). For more details, see the *Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085)* [Ref 5].

1. Double-click the **Zynq UltraScale+ Processing System** in the block diagram window, if it is not open.
2. Select **Switch To Advanced Mode**.

Notice the protection elements indicated by red blocks in the wizard.

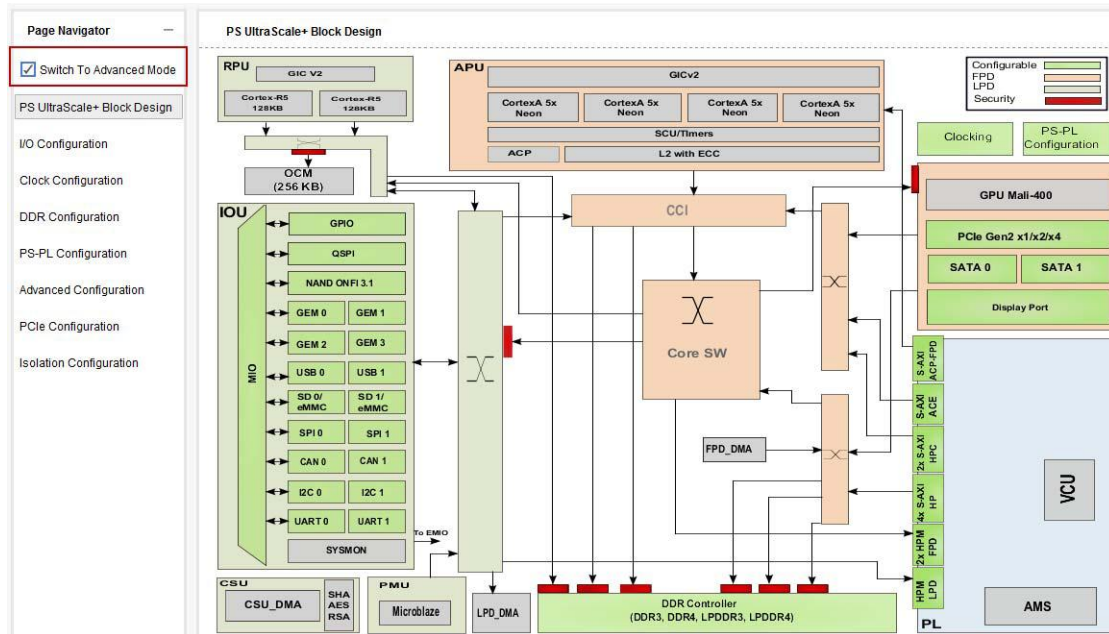


Figure 2-8: PS Configuration Advanced Mode

3. To create an isolation setup, click Isolation Configuration.

This tutorial does not use Isolation Configuration and hence, no Isolation related settings are requested.

4. Click **OK** to close the Re-customize IP wizard.

**Note:** For detailed steps to create isolation configuration, see [XAPP1320](#).

## Validating the Design and Connecting Ports

Use the following steps to validate the design:

1. Right-click in the white space of the Block Diagram view and select **Validate Design**. Alternatively, you can press the **F6** key.
2. A message dialog box opens and states "Validation successful. There are no errors or critical warnings in this design."

3. Click **OK** to close the message.
4. In the Block Design view, click the **Sources** tab.
5. Click **Hierarchy**.
6. Under **Design Sources**, right-click **edt\_zcu102** and select **Create HDL Wrapper**.

The Create HDL Wrapper dialog box opens. You will use this dialog box to create a HDL wrapper file for the processor subsystem.



**TIP:** The HDL wrapper is a top-level entity required by the design tools.

7. Select **Let Vivado manage wrapper and auto-update** and click **OK**.
8. In the Block Diagram, Sources window, under **Design Sources**, expand **edt\_zcu102\_wrapper**.
9. Right-click the top-level block diagram, titled **edt\_zcu102\_i : edt\_zcu102 (edt\_zcu102.bd)** and select **Generate Output Products**.

The Generate Output Products dialog box opens, as shown in the following figure.

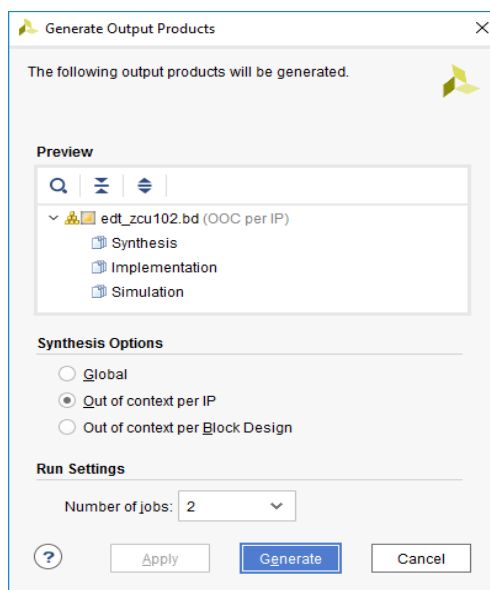


Figure 2-9: Generate Output Products Dialog Box

**Note:** If you are running the Vivado Design Suite on a Linux host machine, you might see additional options under Run Settings. In this case, continue with the default settings.

10. Click **Generate**.

This step builds all required output products for the selected source. For example, constraints do not need to be manually created for the IP processor system. The Vivado tools automatically generate the XDC file for the processor subsystem when **Generate Output Products** is selected.

11. Click **OK**, if you see the message: "Out-of-context module run was launched for generating output products".
12. When the Generate Output Products process completes, click **OK**.
13. In the Block Diagram Sources window, click the **IP Sources** tab. Here you may see the output products that you just generated, as shown in the following figure.

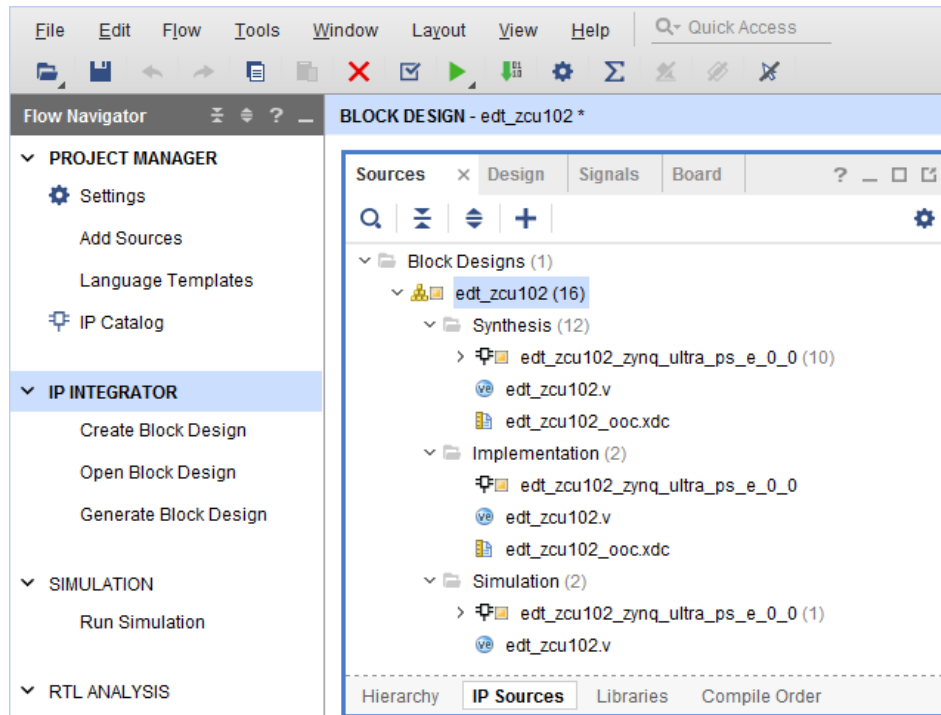


Figure 2-10: Outputs Generated Under IP Sources

## Exporting Hardware

1. Select **File > Export > Export Hardware**.

The Export Hardware dialog box opens. Provide the location for exporting hardware in local path.

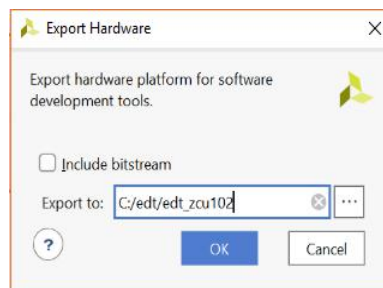


Figure 2-11: Export Hardware

2. Click **OK**.



**TIP:** The hardware is exported in a ZIP file (`<project wrapper>.xsa`). When you create the platform in Vitis IDE using hardware `.xsa`, the ZIP file automatically unzips and files are created in the project platform folder.



### Creating a Hardware Platform using Vitis IDE

1. Launch the Vitis IDE from Windows start menu shortcut or by double-clicking the `C:\xilinx\vitis\2019.2\bin\vitis.bat` file.
2. Select the workspace and continue.

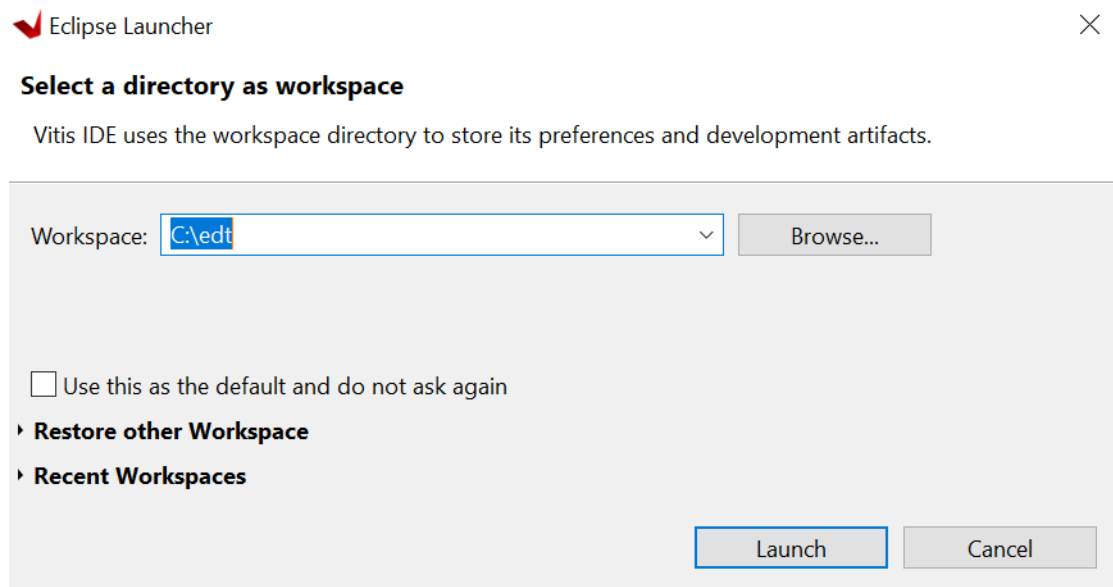


Figure 2-12: Launching the Vitis IDE

3. In the Vitis IDE, go to **File > New > Platform Project**.
4. Enter the Hardware Platform name and click **Next**.

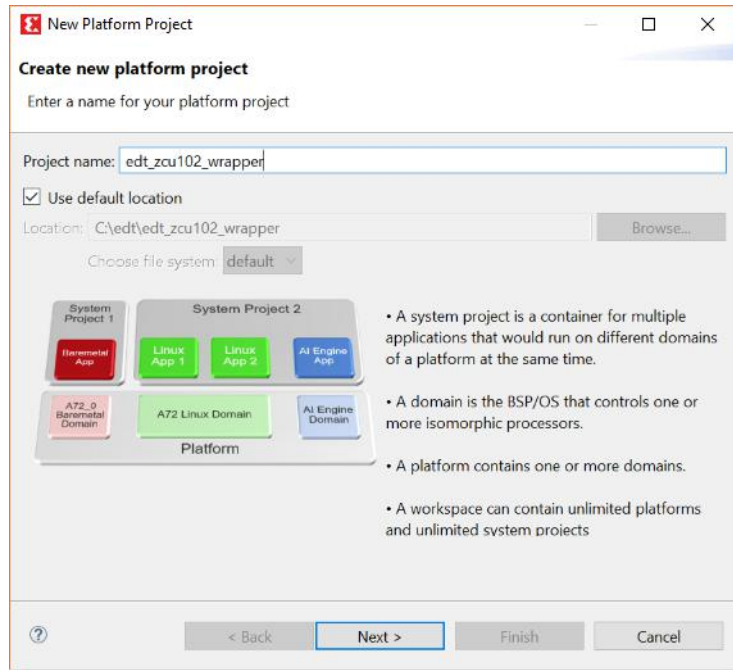


Figure 2-13: New Platform Window

5. Select Create from hardware specification (XSA/DSA), then click **Next**.

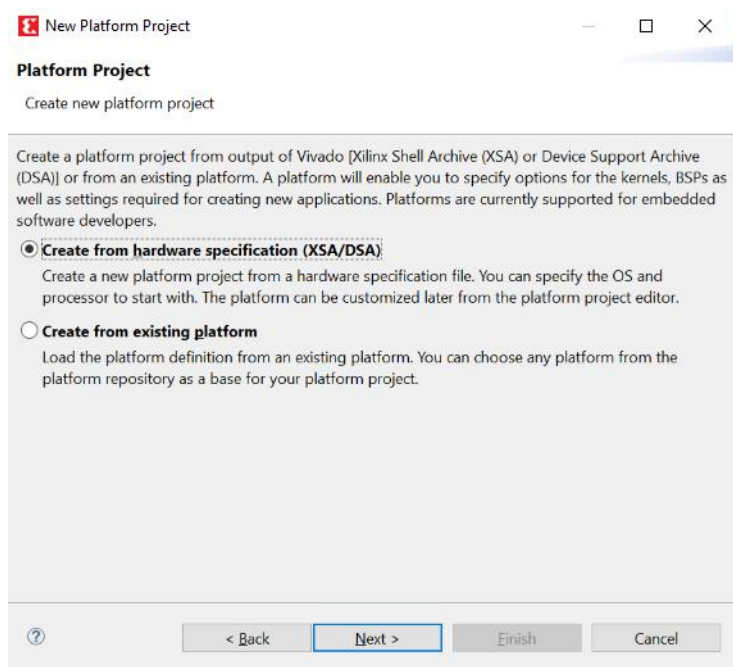


Figure 2-14: Selecting Custom Hardware

6. Browse the `.xsa/` `.dsa` file which is generated from Vivado.



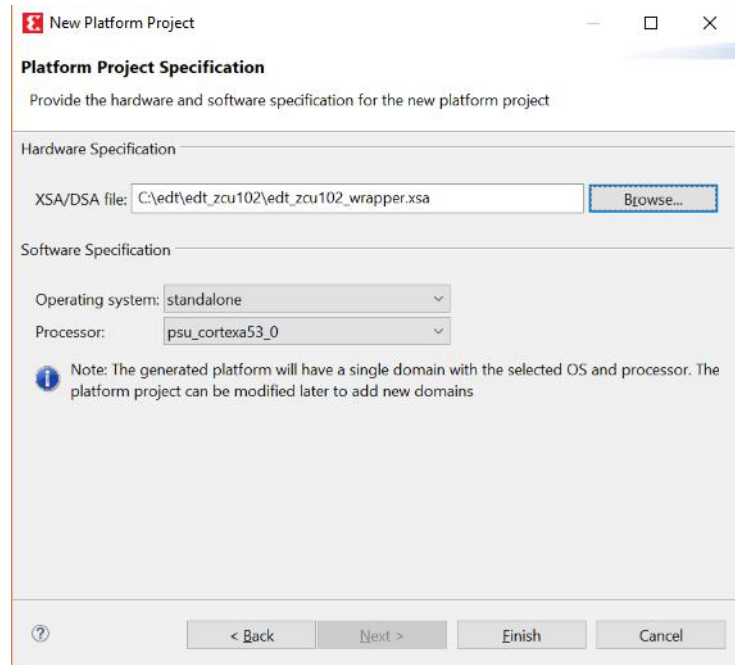


Figure 2-15: Selecting Hardware Specification

7. Click **Finish**.
8. In a few minutes, the Vitis IDE generates the platform. The files that are generated are displayed in the explorer window as shown in the following figure.

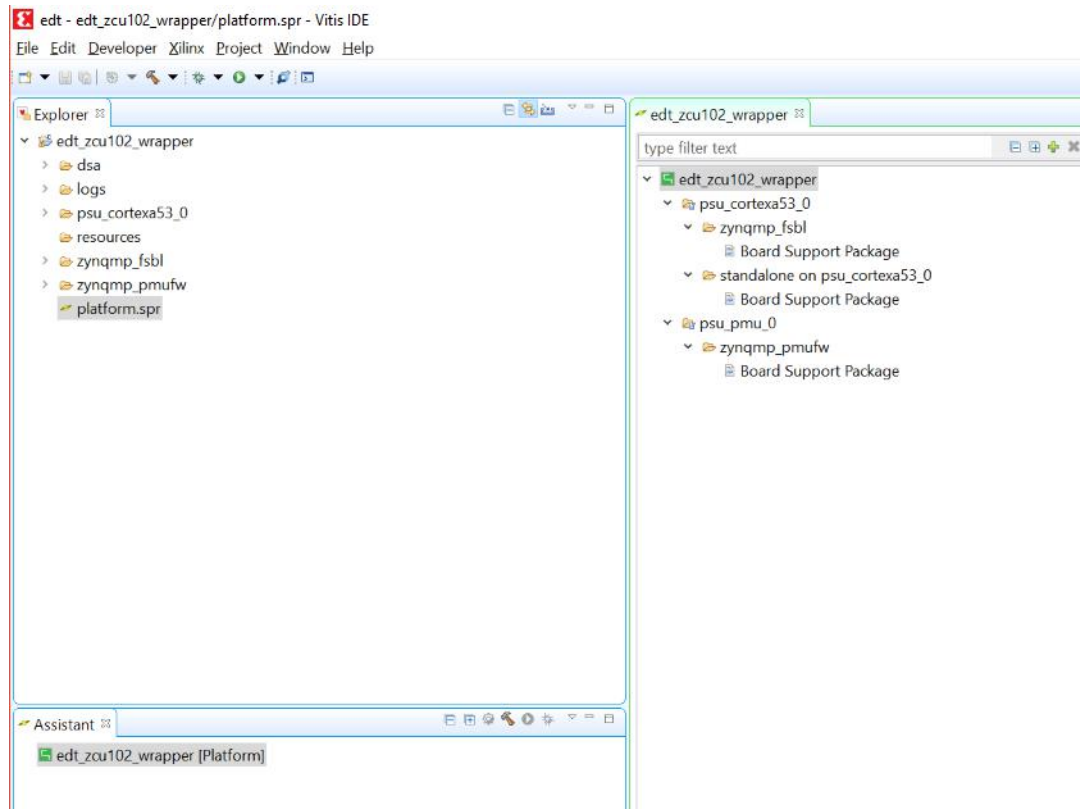


Figure 2-16: Adding a Custom Platform

9. Default FSBL and PMU firmware comes with the platform project and `psu_cortexa53_0` domain also added to the platform. We can add multiple domains to platform and we can also create FSBL like any other application.

**Note:** To add the following libraries by modifying the standalone on `psu_cortexa53_0` domain, follow these steps:

- a. Double-click the **standalone on psu\_cortexa53\_0** BSP.
- b. Click **Modify BSP Settings**.
- c. On the Overview page, select the **xilffs**, **xilpm**, **xilsecure** libraries.

10. Now build the hardware by right-clicking on **Platform** > **Build project**.

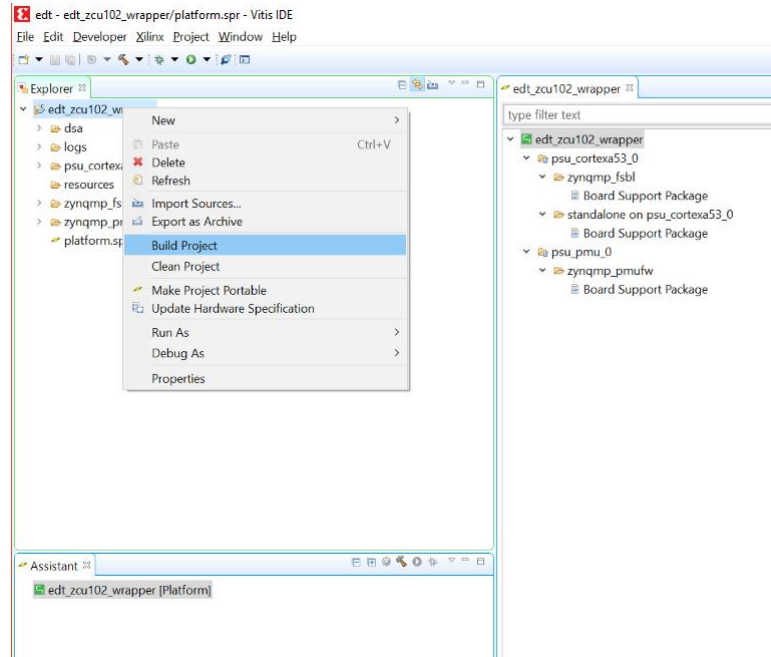


Figure 2-17: Building Hardware Platform

The hardware platform is ready. You can create applications using this platform and test on zcu102 hardware.

### Creating a Vitis Domain for cortexr5\_0

To create a Vitis domain for cortexr5\_0, follow these steps:

1. The **edt\_zcu102\_wrapper** platform is, by default, assigned the default domain for `psu_cortexa53_0`. For applications targeting the RPU, you have to create a domain for `cortexr5_0`
2. Double click **platform.spr**. The platform explorer opens.
3. Click the + in the top right corner to add the domain.
4. Create a domain with the following settings:

Table 2-3: Settings to Create a New Domain

System Properties	Setting or Command to Use
Name	<code>psu_cortexr5_0</code>
Display name	<code>psu_cortexr5_0</code>
OS	Standalone
Version	Standalone (7.1)
Processor	<code>psu_cortexr5_0</code>
Supported Runtime	C/C++

5. The Vitis IDE creates a new domain and **psu\_cortexr5\_0** appears under the **edt\_zcu102\_wrapper** platform.

**Note:** Add the xilffs, xilpm, and xilsecure libraries by modifying psu\_cortexr5\_0 domain. To do so, double-click on **standalone on psu\_cortexr5\_0 bsp**, then click Modify BSP Settings. On the Overview page, select the desired libraries.

# Build Software for PS Subsystems

This chapter lists the steps to configure and build software for PS subsystems. In this chapter, you will use the Zynq® UltraScale™+ hardware platform (hardware definition file) configured in the Vivado® Design Suite.

In [Chapter 2](#), you created and exported the hardware platform from Vivado. This hardware platform contains the hardware handoff file, the processing system initialization files (`psu_init`), and the PL bitstream. In this chapter, you will use the hardware platform in the Vitis IDE and PetaLinux to configure software for the processing system.

This chapter serves two important purposes. One, it helps you build and configure the software components that can be used in future chapters. Second, it describes the build steps for a specific PS subsystem.

---

## Processing Units in Zynq UltraScale+

The main processing units in the processing system in Zynq UltraScale+™ are listed below.

- Application Processing Unit: Quad-core Arm® Cortex®-A53 MPCore Processors
- Real Time Processing Unit: Dual-core Arm Cortex-R5 MPCore Processors
- Graphics Processing Unit: Arm Mali 400 MP2 GPU
- Platform Management Unit (PMU)

This section demonstrates configuring these units using system software. This can be achieved either at the boot level using First Stage Boot Loader (FSBL) or via system firmware, which is applicable to the platform management unit (PMU).

You will use the Zynq UltraScale+ hardware platform in the Vitis IDE to perform the following tasks:

1. Create a First Stage Boot Loader (FSBL) for the Arm Cortex-A53 64-bit quad-core processor unit (APU) and the Cortex-R5 dual-core real-time processor unit (RPU).
2. Create bare-metal applications for APU and RPU.
3. Create platform management unit (PMU) firmware for the platform management unit using the Vitis IDE.

In addition to the bare-metal applications, this chapter also describes building U-Boot and Linux Images for the APU. The Linux images and U-Boot can be configured and built using the PetaLinux build system.

---

## Example Project: Running the “Hello World” Application from Arm Cortex-A53

In this example, you will learn how to manage the board settings, make cable connections, connect to the board through your PC, and run a simple hello world software application from Arm Cortex-A53 in JTAG mode using System Debugger in the Vitis IDE.

1. Connect the power cable to the board.
2. Connect a USB Micro cable between the Windows Host machine and J2 USB JTAG connector on the Target board.
3. Connect a USB micro cable to connector J83 on the target board with the Windows Host machine. This is used for USB to serial transfer.



**IMPORTANT:** Ensure that SW6 Switch, on the bottom right, is set to JTAG boot mode as shown in the following figure.

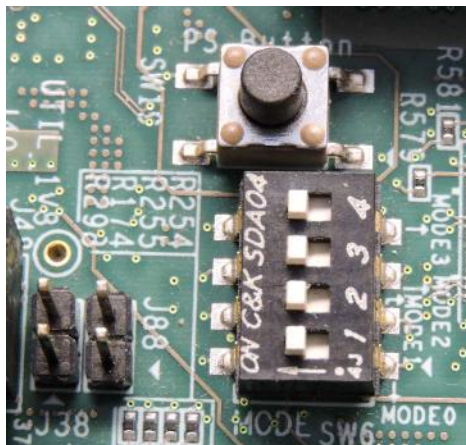


Figure 3-1: SW6 Switch Settings for JTAG Boot Mode

4. Power on the ZCU102 board using the switch indicated in the figure below.

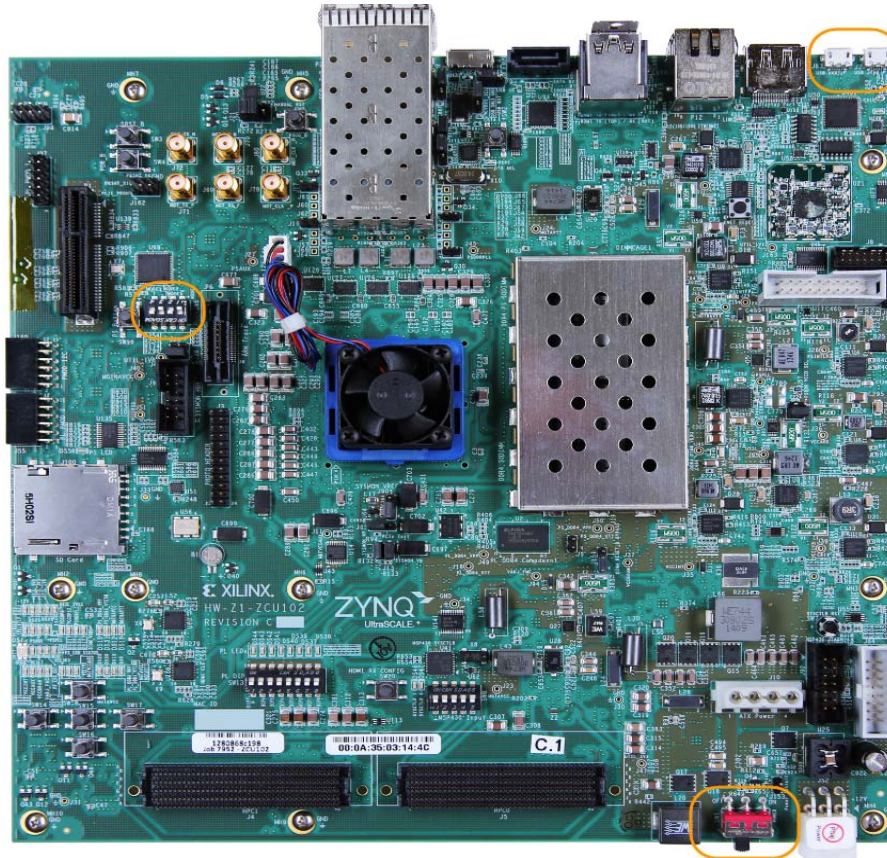



Figure 3-2: ZCU102 Board Power Switch

**Note:** If the Vitis IDE is already running, jump to step 6.

5. Open the Vitis IDE and set the workspace path to your project file, which in this example is `C:\edt`

Alternately, you can open the Vitis IDE with a default workspace and later switch it to the correct workspace by selecting **File > Switch Workspace** and then selecting the workspace.

6. Open a serial communication utility for the COM port assigned on your system. The Vitis IDE provides a serial terminal utility, which will be used throughout the tutorial; select **Window > Show View > Other > Terminal** to open it.
7. Click the **Connect** button  to set the serial configuration and connect it.

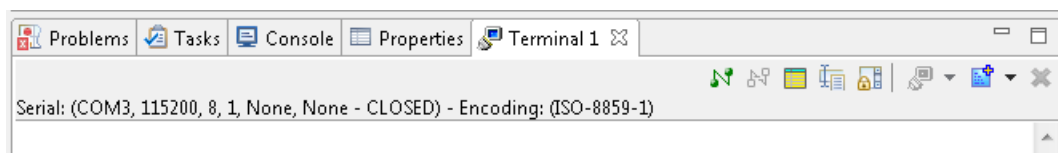



Figure 3-3: Terminal Window Header Bar

8. To modify, disconnect the connection by clicking the **Disconnect** button.

9. Click the **Settings** button  to open the Terminal Settings dialog box.
10. Verify the port details in the device manager.

UART-0 terminal corresponds to COM port with Interface-0. For this example, UART-0 terminal is set by default, so for the COM port, select the port with interface-0.

The following figure shows the standard configuration for the Zynq UltraScale+ MPSoC Processing System.

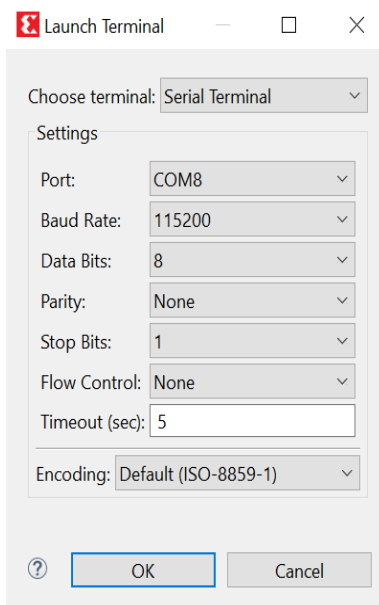


Figure 3-4: Terminal Settings Dialog Box

11. Select **File > New > Application Project**.

The new Project wizard opens.

12. Use the information in the table below to make your selections in the wizard screens.



Table 3-1: New Application Project Settings for Standalone APU Application

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	test_a53
	Use Default Location	Select this option
	System Project	Create New
	Platform	edt_zcu102_wrapper
	Domain	standalone on psu_cortexa53_0
	Language	C
	CPU	psu_cortexa53_0
	OS	Standalone
Templates	Available Templates	Hello World

The Vitis IDE creates the test\_a53 application project and test\_a53\_system project under the Project Explorer. Right-click on the test\_a53 application project and select build option to build the application.

13. Right-click **test\_a53** and select **Run as > Run Configurations**.
14. Right-click **Xilinx Application Debugger** and click **New Configuration**.

The Vitis IDE creates the new run configuration, named **Debugger\_test\_a53-Default**.

The configurations associated with the application are pre-populated in the Main tab of the launch configurations.

15. Click the **Target Setup** tab and review the settings.
  - Note:** The board should be in JTAG boot mode before power cycling.
16. Power cycle the board.
17. Click **Run**.

"Hello World" appears on the serial communication utility in Terminal 1, as shown in the following figure.

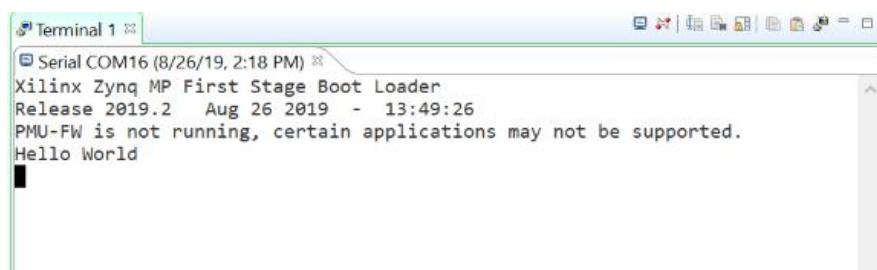


Figure 3-5: Output on Serial Terminal

**Note:** There was no bitstream download required for the above software application to be executed on the Zynq UltraScale+ evaluation board. The Arm Cortex-A53 quad core is already present in the processing system. Basic initialization of this system to run a simple application is done by the Device initialization Tcl script.

18. Power cycle the board and retain same connections and board settings for the next section.

## What Just Happened?

The application software sent the "Hello World" string to the UART0 peripheral of the PS section.

From UART0, the "Hello world" string goes byte-by-byte to the serial terminal application running on the host machine, which displays it as a string.

---

## Example Project: Running the “Hello World” Application from Arm Cortex-R5

In this example, you will learn how to manage the board settings, make cable connections, connect to the board through your PC, and run a simple hello world software application from Arm Cortex-R5 in JTAG mode using System Debugger in the Vitis IDE.

**Note:** If you have already set up the board, skip to step 5.

1. Connect the power cable to the board.
2. Connect a USB Micro cable between the Windows Host machine and the **J2 USB JTAG** connector on the Target board.
3. Connect a USB cable to connector **J83** on the target board with the Windows Host machine. This is used for USB to serial transfer.
4. Power on the ZCU102 board using the switch indicated in [Figure 3-1](#).



---

**IMPORTANT:** Ensure that the SW6 switch is set to JTAG boot mode as shown in [Figure 3-1](#).

---

**Note:** If the Vitis IDE is already open, jump to step 6.

5. Launch the Vitis IDE and set the workspace path to your project file, which in this example is

`c:\edt\.`

Alternately, you can open the Vitis IDE with a default workspace and later switch it to the correct workspace by selecting **File > Switch Workspace** and then selecting the workspace.

- Open a serial communication utility for the COM port assigned on your system. The Vitis IDE provides a serial terminal utility, which will be used throughout the tutorial; select **Window > Show View > Terminal** to open it.

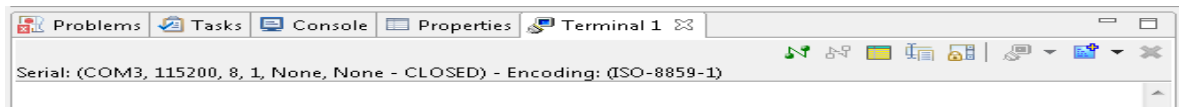




Figure 3-6: Terminal Window Header Bar

- Click the **Connect** button  to set the serial configuration and connect it.
- Click the **Settings** button  to open the Terminal Settings dialog box.

The Com -port details can be found in the device manager on host machine. UART-0 terminal corresponds to Com-Port with Interface-0. For this example, UART-0 terminal is set by default, so for the Com-port, select the port with interface-0.

The following figure shows the standard configuration for the Zynq UltraScale+ MPSoC Processing System.

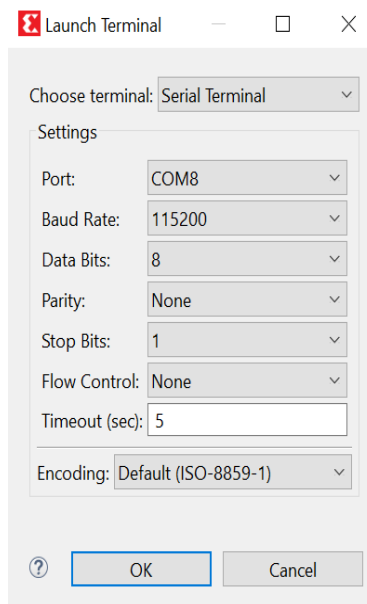


Figure 3-7: Terminal Settings Dialog Box

- In the Vitis IDE, switch back from Debug perspective to C/C++ perspective. For this you have to click **Windows > Open Perspective > C/C++**.

Ignore this step, if the Vitis IDE is in C/C++ perspective already.

- Select **File > New > Application Project**.

The New Project wizard opens.

11. Use the information in the following table to make your selections in the wizard screens.

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	hello_world_r5
	Use Default Location	Select this option
	System Project	Create New
	Platform	edt_zcu102_wrapper
	Domain	psu_cortexr5_0
	Language	<b>C</b>
	CPU	psu_cortexr5_0
	OS	standalone
Templates	Available Templates	<b>Hello World</b>

The Vitis IDE creates the `hello_world_r5` application project and `hello_world_r5_system` project under the Project Explorer. You have to compile the application manually.

12. Right-click `hello_world_r5` and select **Run as > Run Configurations**.

13. Right-click **Xilinx Application Debugger** and click **New Configuration**.

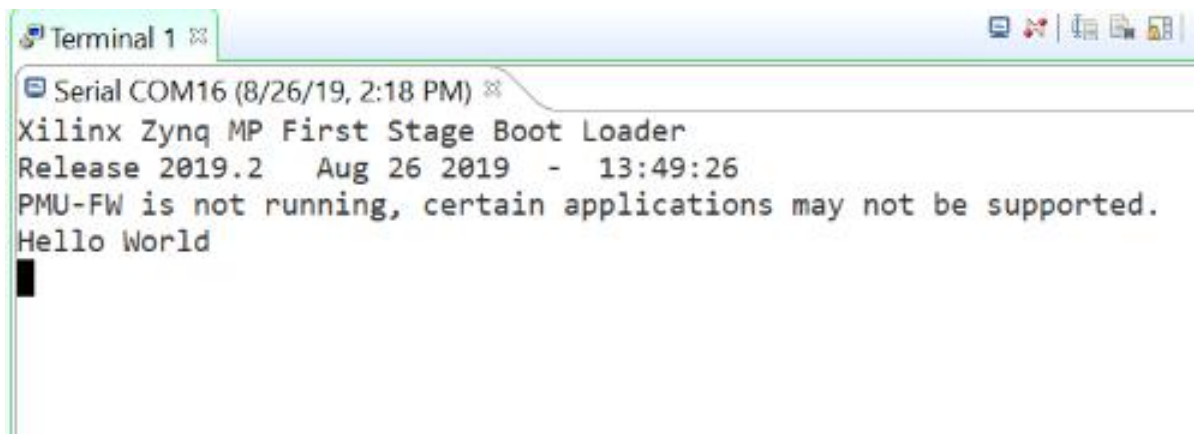
The Vitis IDE creates the new run configuration, named **Debugger\_hello\_world\_r5-Default**. The configurations associated with the application are pre-populated in the Main tab of the launch configurations.

14. Click the **Target Setup** tab and review the settings.

This file is exported when you create the platform using the Vitis IDE; it contains the initialization information for the processing system.

15. Click **Run**.

"**Hello World**" appears on the serial communication utility in Terminal 1, as shown in the following figure.



```
Terminal 1
Serial COM16 (8/26/19, 2:18 PM)
Xilinx Zynq MP First Stage Boot Loader
Release 2019.2   Aug 26 2019   -   13:49:26
PMU-FW is not running, certain applications may not be supported.
Hello World
```

Figure 3-8: Output on Serial Terminal

**Note:** There was no bitstream download required for the above software application to be executed on the Zynq UltraScale+ evaluation board. The Arm Cortex-R5 dual core is already present on the board. Basic initialization of this system to run a simple application is done by the Device initialization Tcl script.

## What Just Happened?

The application software sent the "Hello World" string to the UART0 peripheral of the PS section.

From UART0, the "Hello world" string goes byte-by-byte to the serial terminal application running on the host machine, which displays it as a string.

---

## Additional Information

### Domain

A domain can refer to the settings and files of a standalone BSP, a Linux OS, a third-party OS/BSP like FreeRTOS, or a component like the device tree generator.

### Board Support Package

The board support package (BSP) is the support code for a given hardware platform or board that helps in basic initialization at power up and helps software applications to be run on top of it. It can be specific to some operating systems with boot loader and device drivers.



**TIP:** To reset the BSP source, double-click **platform.prj**, select a BSP in a domain, and click **Reset BSP Source**. This action only resets the source files while settings are not touched.

To change the target domain after an application project creation, double-click the **project.prj** file in Explorer view. In the Application Project Settings, select **Domain --> Domain change option --> Drop-down Domain**, then select available domains for this application.

## Standalone BSP

Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts, and exceptions, as well as the basic processor features of a hosted environment. These basic features include standard input/output, profiling, abort, and exit. It is a single threaded semi-hosted environment.

---

## Example Project: Create a Bare-Metal Application Project in the Vitis IDE

For this example, you will launch the Vitis IDE and create a bare-metal application using the hardware platform for Zynq UltraScale+ created using the Vivado Design Suite. [Figure 3-13](#) shows the New Application Project dialog box and possible options for creating bare-metal (Standalone) applications for processing subsystems in Zynq UltraScale+ devices.

### Create Bare-Metal Application for Arm Cortex-A53 based APU

Now that the FSBL is created, you will now create a simple bare-metal application targeted for an Arm A53 Core 0.

For this example, you will use the test\_a53 application that you created in [Example Project: Running the "Hello World" Application from Arm Cortex-A53](#).

In test\_a53, you selected a simple Hello World application. This application can be loaded on APU by FSBL running on either APU or RPU.

also provides few other bare-metal applications templates to make it easy to start running applications on Zynq UltraScale+ devices. Alternatively, you can also select the Empty Application template and copy or create your custom application source code in the application folder structure.

### Modify the Application Source Code

1. In the Project Explorer, click `test_a53 > src > helloworld.c`.

This opens the `helloworld.c` source file for the `test_a53` application.

2. Modify the arguments in the `print` command, as shown below.

```
Print("Hello World from APU\n\r");

int main()
{
    init_platform();
    print("Hello World from APU\n\r");
    cleanup_platform();
    return 0;
}
```

Figure 3-9: Application Source Code Snippet: Print Command

3. Type **Ctrl + S** to save the changes.
4. Right-click the `test_a53` project and select **Build Project**.
5. Verify that the application is compiled and linked successfully and the `test_a53.elf` file is generated in the `test_a53 > Debug` folder.

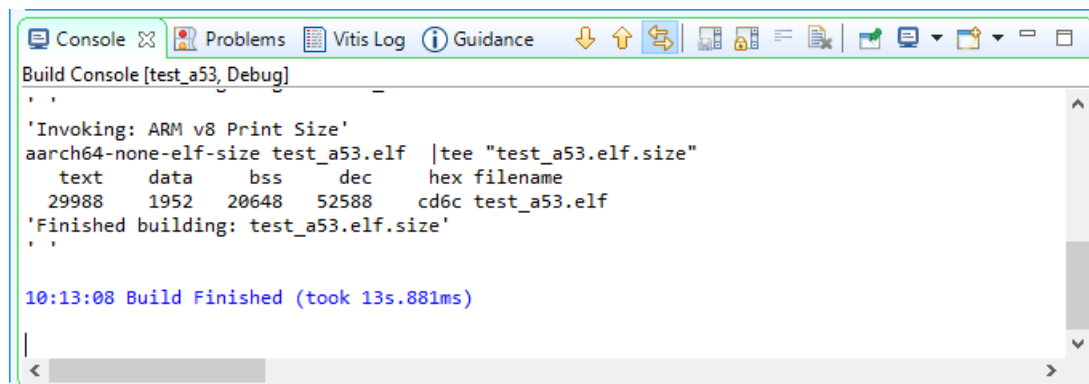


Figure 3-10: CDT Build Console

## Create Bare-Metal Application for Arm Cortex-R5 based RPU

In this example, you will create a bare-metal application project for Arm Cortex-R5 based RPU. For this project, you will need to import the application source files available in the Design Files ZIP file released with this tutorial. For information about locating these design files, refer to [Design Files for This Tutorial in Appendix B](#).

### Creating the Application Project

1. In the Vitis IDE, select **File > New > Application Project** to open the New Project wizard.
2. Use the information in the following table to make your selections in the wizard.

Table 3-2: Settings to Create New RPU Application Project

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	<code>testapp_r5</code>
	Use Default Location	Select this option
	System Project	Create New
	Platform	<b>edt_zcu102_wrapper</b>
	Domain	<code>psu_cortexr5_0</code>
	Language	<b>C</b>
	CPU	<code>psu_cortexr5_0</code>
	OS	Standalone
Templates	Available Templates	<b>Empty Application</b>

3. Click **Finish**.

The New Project wizard closes and the Vitis IDE creates the **testapp\_r5** application project, which can be found in the Project Explorer.

4. In the Project Explorer tab, expand the **testapp\_r5** project.

5. Right-click the **src** directory, and select **Import** to open the Import dialog box.

6. Expand **General** in the Import dialog box and select **File System**.

7. Click **Next**.

8. Select **Browse** and navigate to the design files folder, which you saved earlier (see [Design Files for This Tutorial in Appendix B](#)).

9. Click **OK**.

10. Select the `testapp.c` file.

11. Click **Finish**.

12. Open `testapp.c` to review the source code for this application. The application configures the UART interrupt and sets the Processor to WFI mode. This application is reused and explained during run time in [Chapter 5, Boot and Configuration](#).

### Modifying the Linker Script

1. In the Project Explorer, expand the **testapp\_r5** project.

2. In the **src** directory, double-click `lscript.ld` to open the linker script for this project.

3. In the linker script, in Available Memory Regions, modify following attributes for `psu_r5_ddr_0_MEM_0`:

- **Base Address:** `0x70000000`
- **Size:** `0x10000000`



The linker script modification is shown in following figure. The following figure is for representation only. Actual memory regions may vary in case of Isolation settings.

**Linker Script: lscript.ld**

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

**Available Memory Regions**

Name	Base Address	Size
psu_ocm_ram_0_MEM_0	0xFFFC0000	0x40000
psu_qspi_linear_0_MEM_0	0xC0000000	0x20000000
psu_r5_0_atcm_MEM_0	0x0	0x10000
psu_r5_0_btcm_MEM_0	0x20000	0x10000
psu_r5_0_ddr_0_MEM_0	0x70000000	0x10000000
psu_r5_tcm_ram_0_MEM_0	0x0	0x40000

**Stack and Heap Sizes**

Stack Size

Heap Size

Figure 3-11: Linker Script Modification

This modification in the linker script ensures that the RPU bare-metal application resides above 0x70000000 base address in the DDR, and occupies no more than 256 MB of size.

4. Type **Ctrl + S** to save the changes.
5. Right-click the `testapp_r5` project and select **Build Project**.
6. Verify that the application is compiled and linked successfully and that the `testapp_r5.elf` file was generated in the `testapp_r5 > Debug` folder.

### Modifying the Board Support Package

The ZCU102 Evaluation kit has a USB-TO-QUAD-UART Bridge IC from Silicon Labs (CP2108). This enables you to select a different UART port for applications running on A53 and R5 Cores. For this example, let A53 use the UART 0 by default, and send and receive RPU serial data over UART 1. This requires a small modification in the `r5_bsp` file.

1. Navigate to `psu_cortexr5_0` domain BSP settings, click **modify BSP settings > standalone**. Change the `stdin` and `stdout` to `psu_uart_1`.
2. Click **Standalone**.
3. Modify the `stdin` and `stdout` values to `psu_uart_1`, as shown in the figure below.

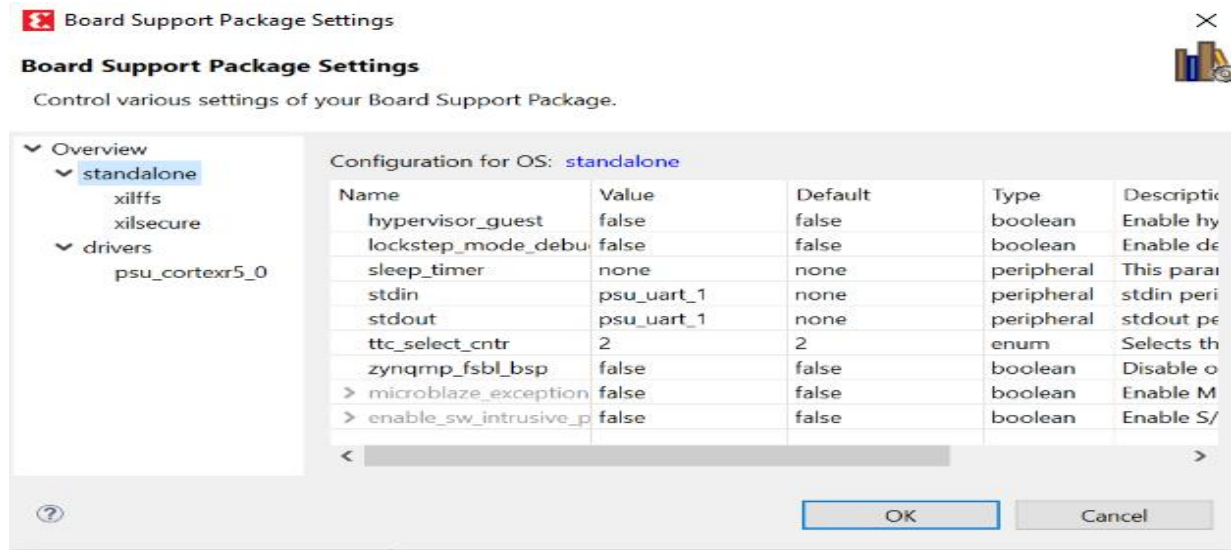


Figure 3-12: Board Support Package Settings for psu\_cortexr5\_0 domain

4. Click **OK**.
5. Build psu\_cortexr5\_0 domain and testapp\_r5 application.
6. Verify that the application is compiled and linked successfully and that the testapp\_r5.elf was generated in the testapp\_r5 > Debug folder.

## Reviewing Software Projects in the Platform

### Review of FSBL in Platform

To review the FSBL in platform, follow these steps:

1. Click the **edt\_zcu102\_wrapper** platform drop-down menu, then click the drop-down menu next to the **zynqmp\_fsbl** to see the FSBL source code for zynqmp. You can edit this source for customizations.
2. The platform generated FSBL is involved in PS initialization while launching standalone applications using JTAG.
3. This FSBL is created for the **psu\_cortexa53\_0**, but you can also re-target the FSBL to **psu\_cortexr5\_0** using the **re-target to psu\_cortexr5\_0** option in the zynqmp\_fsbl domain settings.

### Review of PMU Firmware in Platform

To review the FSBL in platform, follow these steps:

1. Click the desired platform drop-down menu to see the **zynqmp\_pmufw** software project that was created within the platform by default.
2. **zynqmp\_pmufw** contains the source code of PMU firmware for **psu\_pmu\_0**. Compile and run the firmware on **psu\_pmu\_0**.

The **psu\_pmu\_0** processor domain is created automatically for **zynqmp\_pmufw**.

## Create First Stage Boot Loader for Arm Cortex-A53-Based APU

FSBL can load the required application or data to memory and launch applications on the target CPU core. One FSBL has been provided in the platform project but you can create an additional FSBL application as a general application for further modification or debugging purposes.

In this example, you will create an FSBL image targeted for Arm Cortex-A53 core 0.

1. Launch the Vitis IDE if it is not already open.
2. Set the Workspace path based on the project you created in Chapter 2. For example, **C:\edt**.
3. Select **File > New > Application Project**.

The New Project dialog box opens.

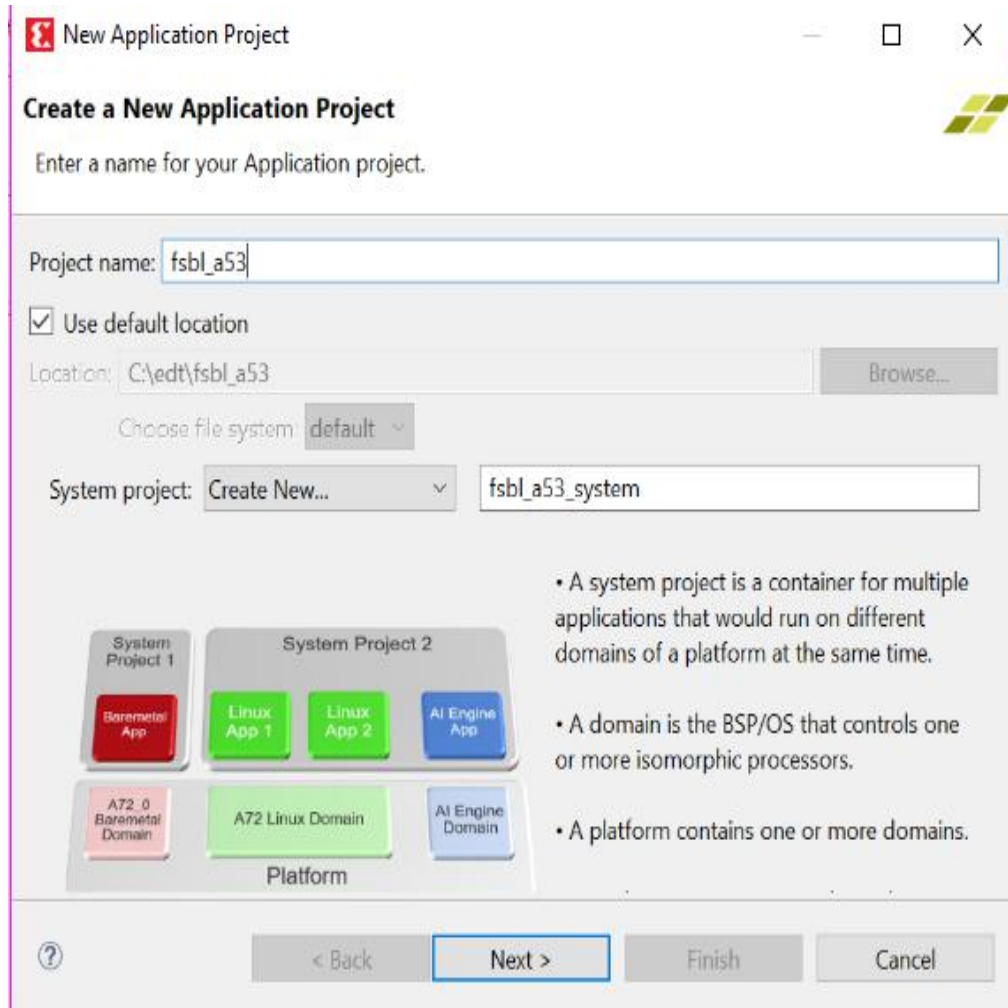


Figure 3-13: Application Project Page of New Project Wizard

- Use the information in the following table to make your selections in the New Project wizard:

Table 3-3: Settings to Create New Application Project - FSBL\_A53

Wizard Screen	System Properties	Setting or Command
Application Project	Project Name	fsbl_a53
	Use Default Location	Select this option
	System Project	Create New
	Platform	edt_zcu102_wrapper
	Domain	Standalone on psu_cortexa53_0
	Language	C
	CPU	psu_cortexa53_0
	OS	Standalone

Table 3-3: Settings to Create New Application Project - FSBL\_A53 (Cont'd)

Wizard Screen	System Properties	Setting or Command
Click <b>Next</b>		
Templates	Available Templates	<b>Zynq MP FSBL</b>

5. In the Templates page, select **Zynq MP FSBL**:

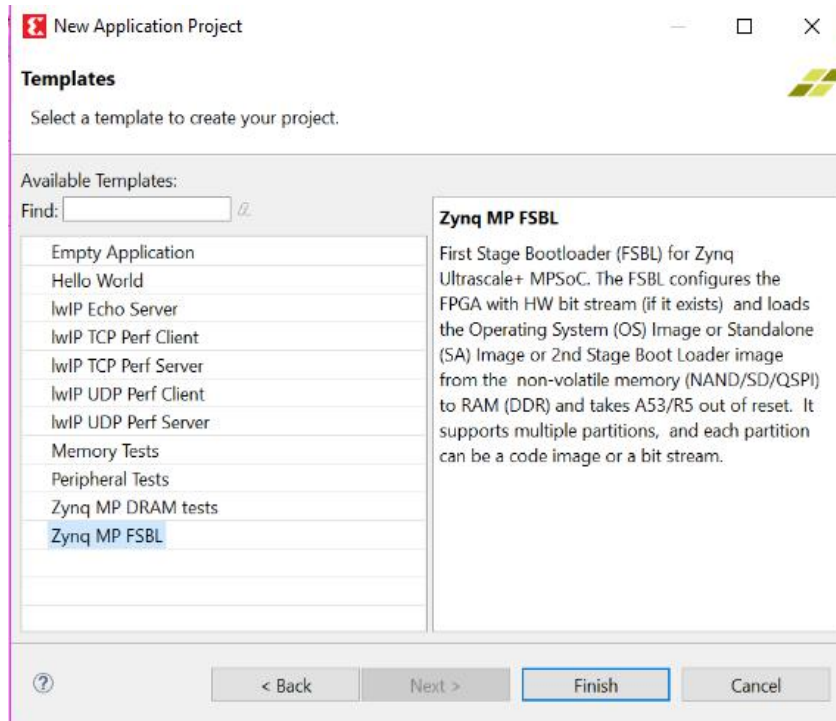


Figure 3-14: Templates Page of the New Project Wizard


6. Click **Finish**.

The Vitis IDE creates the System project and FSBL application.

By default, the FSBL is configured to show basic print messages. Next, you will modify the FSBL build settings to enable debug prints.

For a list of the possible debug options for FSBL, refer to the `fsbl_a53 > src > xfsbl_debug.h` file.

For this example, enable FSBL\_DEBUG\_INFO by doing the following:

1. In the **Project Explorer** folder, right-click the **fsbl\_a53** application.
2. Click **C/C++ Build Settings**.
3. Select **Settings > ARM V8 gcc compiler > Symbols**.
4. Click the **Add** button .
5. Enter **FSBL\_DEBUG\_INFO**.

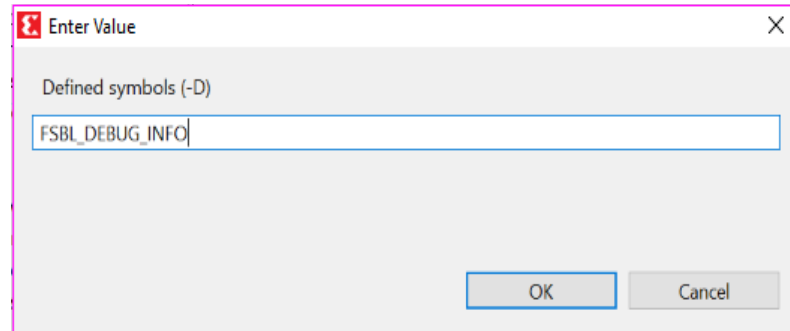


Figure 3-15: Enter Value Dialog Box

The Symbols settings are as shown in the following figure.

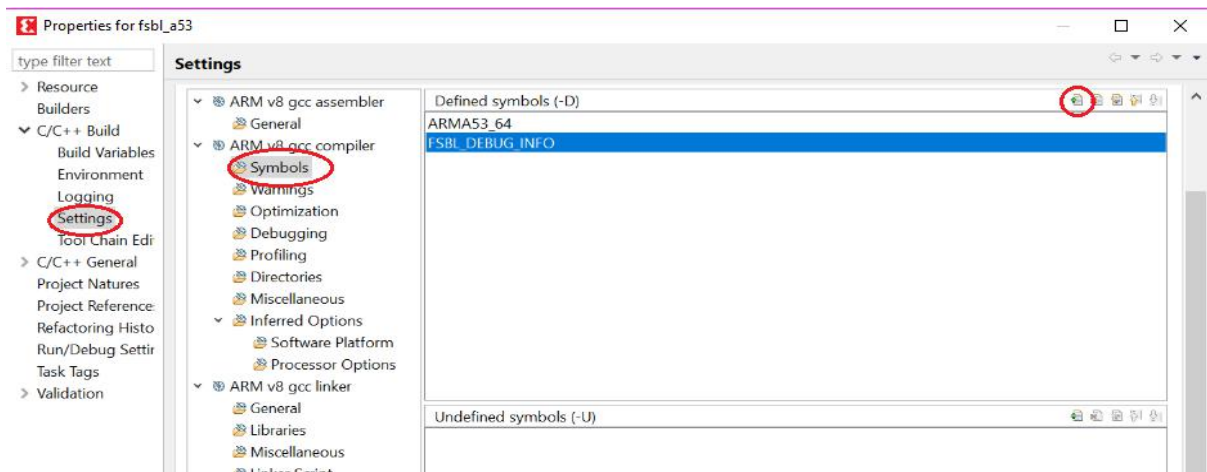


Figure 3-16: Symbols Settings for fsbl\_a53 Application

6. Click **OK** to accept the changes and close the Settings dialog box.
7. Navigate BSP settings. Under **Overview > Drivers > psu\_cortexa53\_0 > extra\_compiler\_flags**, edit extra\_compiler\_flags to append " -Os -flto -ffat-lto-objects".
8. Right-click the **fsbl\_a53** application and select **Build Project**.
9. The FSBL executable is now saved as **fsbl\_a53 > debug > fsbl\_a53.elf**.

In this tutorial, the application name `fsbl_a53` is to identify that the FSBL is targeted for APU (the Arm Cortex-A53 core).

**Note:** If the system design demands, the FSBL can be targeted to run on RPU, which can then load rest of the software stack on RPU and APU.

---

## Example Project: Create Linux Images using PetaLinux

The earlier example highlighted creation of the bootloader images and bare-metal applications for APU, RPU, and PMU using the Vitis IDE. In this chapter, you will configure and build Linux Operating System Platform for Arm Cortex A53 core based APU on Zynq UltraScale+. The PetaLinux tool flow, along with the board-specific BSP, can be used to configure and build Linux images.



---

**IMPORTANT:** This example needs a Linux Host machine. *PetaLinux Tools Documentation: Reference Guide (UG1144) [Ref 7]* for information about dependencies for PetaLinux 2019.2.

---



---

**IMPORTANT:** This example uses the ZCU102 PetaLinux BSP to create a PetaLinux project. Ensure that you have downloaded the ZCU102 BSP for PetaLinux as instructed in *PetaLinux Tools, page 11*.

---

1. Create a PetaLinux project using the following command:

```
$petalinux-create -t project -s <path to the directory that has  
xilinx-zcu102-v2019.2-final.bsp>
```

**Note:** `xilinx-zcu102-v2019.2-final.bsp` is the PetaLinux BSP for ZCU102 Production Silicon Rev1.0 Board. Use `xilinx-zcu102-ZU9-ES2-Rev1.0-v2019.2-final.bsp`, if you are using ES2 Silicon on Rev 1.0 board.

The above step creates a PetaLinux Project Directory, such as:

```
xilinx-zcu102-2019.2.
```

2. Change to the PetaLinux project directory using the following command:

```
$ cd xilinx-zcu102-2019.2
```

The **ZCU102 PetaLinux-BSP** is the default ZCU102 Linux BSP. For this example, you reconfigure the PetaLinux Project based on the Zynq UltraScale+ hardware platform that you configured using Vivado Design Suite in [Chapter 2](#).

3. Copy the hardware platform `edt_zcu102_wrapper.xsa` to the Linux Host machine.
4. Reconfigure the project using the following command:

```
$ petalinux-config --get-hw-description=<path containing edt_zcu102_wrapper.xsa>/
```

This command opens the PetaLinux Configuration window. If required, make changes in the configuration. For this example, the default settings from the BSP are sufficient to generate required boot images.

The following steps will verify if PetaLinux is configured to create Linux and boot images for SD Boot.

5. Select **Subsystem AUTO Hardware Settings**.
6. Select **Advanced Bootable Images Storage Settings**.
  - a. Select **boot image settings**.
  - b. Select **Image Storage Media**.
  - c. Select **primary sd** as the boot device.
7. Under the **Advanced Bootable Images Storage Settings** submenu, do the following:
  - a. Select **kernel image settings**.
  - b. Select **Image Storage Media**.
  - c. Select **primary sd** as the storage device.
8. Under **Subsystem AUTO Hardware Settings**, select **Memory Settings** and set the System Memory Size to **0x6FFFFFFF**
9. Save the configuration settings and exit the Configuration wizard.
10. Wait until PetaLinux reconfigures the project.

The following steps will build the Linux images, verify them, and generate the boot image.

11. Modify Device Tree to disable Heartbeat LED and SW19 push button, from the device tree. Due to this the RPU R5-0 can use PS LED and SW19 switch for other designs in this tutorial. This can be done by adding the following to the `system-user.dtsi` which can be found in the following location:

```
<PetaLinux-project>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

12. Add the following to `system-user.dtsi`, so that it looks like:

```
/include/ "system-conf.dtsi"
/ {
    gpio-keys {
        sw19 {
            status = "disabled";
        };
    };
    leds {
        heartbeat_led {
            status = "disabled";
        };
    };
};
```



```
};  
};  
&uart1  
{  
    status = "disabled";  
};
```

13. In `<PetaLinux-project>`, build the Linux images using the following command:

```
$ petalinux-build
```

14. After the above statement executes successfully, verify the images and the timestamp in the `images` directory in the PetaLinux project folder using the following commands:

```
$ cd images/linux/  
$ ls -al
```

15. Generate the Boot image using the following command:

```
$ petalinux-package --boot --fsbl zynqmp_fsbl.elf --u-boot
```

This creates a `BOOT.BIN` image file in the following directory:

```
<petalinux-project>/images/linux/BOOT.BIN
```

The Logs indicate that the above command includes `PMU_FW` and `ATF` in `BOOT.BIN`. You can also add `--pmufw <PMUFW_ELF>` and `--atf <ATF_ELF>` in the above command. Refer `$ petalinux-package --boot --help` for more details.

**Note:** The option to add bitstream, that is `--fpga` is missing from above command intentionally. This is because the hardware configuration so far is only based on PS with no design in PL. In case a bitstream is present in the design, `--fpga` can be added in the `petalinux-package` command as shown below:

```
petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit --pmufw pmufw.elf  
--atf bl31.elf --u-boot u-boot.elf
```

## Verify the Image on the ZCU102 Board

To verify the image:

1. Copy files `BOOT.BIN` and `image.ub` to an SD card.
2. Load the SD card into the ZCU102 board, in the J100 connector.
3. Connect a Micro USB cable from ZCU102 Board USB UART port (J83), to USB port on the host Machine.

- Configure the Board to Boot in SD-Boot mode by setting switch SW6 as shown in the following figure.

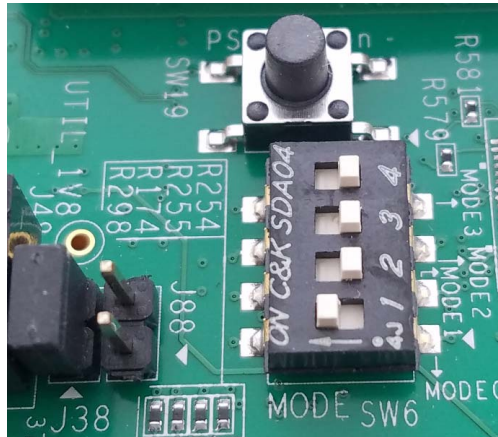


Figure 3-17: SW6 Switch Settings for SD Boot Mode

- Connect 12V Power to the ZCU102 6-Pin Molex connector.
- Start a terminal session, using Tera Term or Minicom depending on the host machine being used. set the COM port and baud rate for your system, as shown in the following figure.

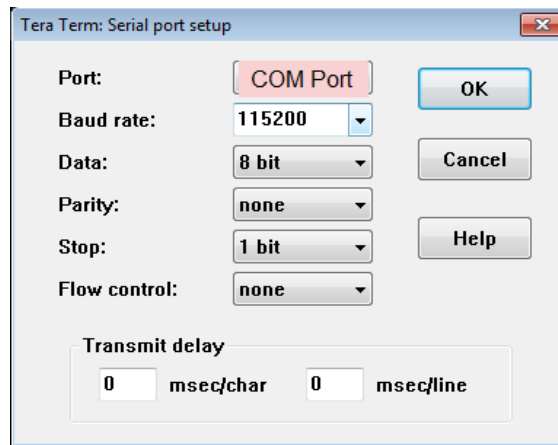


Figure 3-18: COM Port Set Up

- For port settings, verify COM port in the device manager and select the COM port with interface-0.
- Turn on the ZCU102 Board using SW1, and wait until Linux loads on the board.

## Create Linux Images using PetaLinux for QSPI Flash

The earlier example highlighted creation of the Linux Images and Boot images to boot from an SD card. This section explains the configuration of PetaLinux to generate Linux images

for QSPI flash. For more information about the dependencies for PetaLinux 2019.2, see the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 7].

1. Before starting this example, create a backup of the boot images created for SD card setup using the following commands:

```
$ cd <Petalinux-project-path>/xilinx-zcu102-2019.2/images/linux/  
$ mkdir sd_boot  
$ cp image.ub sd_boot/  
$ cp u-boot.elf sd_boot/  
$ cp BOOT.BIN sd_boot/
```

2. Change the directory to the PetaLinux Project root directory:

```
$ cd <Petalinux-project-path>/xilinx-zcu102-2019.2
```

3. Launch the top level system configuration menu:

```
$ petalinux-config
```

The Configuration wizard opens.

4. Select **Subsystem AUTO Hardware Settings**.
5. Select **Advanced bootable images storage Settings**.
  - a. Select **boot image settings**.
  - b. Select **image storage media**.
  - c. Select **primary flash** as the boot device.
6. Under the **Advanced bootable images storage Settings** submenu, do the following:
  - a. Select **kernel image settings**.
  - b. Select **image storage media**.
  - c. Select **primary flash** as the storage device.
7. One level above, that is, under **Subsystem AUTO Hardware Settings**,
  - a. Select **Flash Settings** and notice the entries listed in the partition table.
  - b. Note that some memory (0x1E00000 + 0x40000) is set aside for initial Boot partitions and U-Boot settings. These values can be modified on need basis.
  - c. Based on this, the offset for Linux Images is calculated as 0x1E40000 in QSPI Flash device. This will be used in Chapter 5, while creating Boot image for QSPI Boot-mode.

The following steps will set the Linux System Memory Size to about 1.79 GB.

8. Under **Subsystem AUTO Hardware Settings**, do the following:
  - a. Select **Memory Settings**
  - a. Set **System Memory Size** to **0x6FFFFFFF**

9. Save the configuration settings and exit the Configuration wizard.
10. Rebuild using the `petalinux-build` command.
11. Take a backup of `u-boot.elf` and the other images. These will be used in [Chapter 5](#).

**Note:** For more information, refer to the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [\[Ref 7\]](#)

In this chapter, you learned how to configure and compile Software blocks for Zynq UltraScale+ devices using Xilinx tools. You will use these images in [Chapter 6](#) to create Boot images for a specific design example.

Next, you will debug software for Zynq UltraScale+ devices using the Vitis IDE in [Chapter 4, Debugging with the Vitis Debugger](#).

# Debugging with the Vitis Debugger

This chapter describes debug possibilities with the design flow you have already been working with. The first option is debugging with software using the Vitis debugger.

The Vitis debugger provides the following debug capabilities:

- Supports debugging of programs on Arm® Cortex® -A53, Arm Cortex-R5, and MicroBlaze™ processor architectures (heterogeneous multi-processor hardware system debugging)
- Supports debugging of programs on hardware boards
- Supports debugging on remote hardware systems
- Provides a feature-rich IDE to debug programs
- Provides a Tool Command Language (Tcl) interface for running test scripts and automation

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

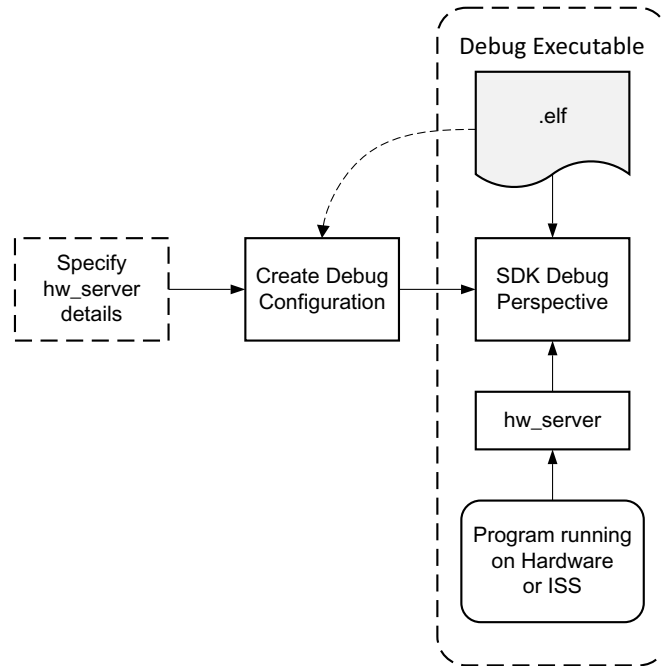
The Vitis debugger supports debugging through Xilinx System Debugger.

---

## Xilinx System Debugger

The Xilinx System Debugger uses the Xilinx hw\_server as the underlying debug engine. The Vitis IDE translates each user interface action into a sequence of Target Communication Framework (TCF) commands. It then processes the output from System Debugger to display the current state of the program being debugged. It communicates to the processor on the hardware using Xilinx hw\_server.

The debug workflow is described in the following figure.



X16794-041816

Figure 4-1: System Debugger Flow

The workflow is made up of the following components:

- **Executable ELF File:** To debug your application, you must use an Executable and Linkable Format (ELF) file compiled for debugging. The debug ELF file contains additional debug information for the debugger to make direct associations between the source code and the binaries generated from that original source. To manage the build configurations, right-click the software application and select **Build Configurations > Manage**.
- **Debug Configuration:** To launch the debug session, you must create a debug configuration in the Vitis debugger. This configuration captures options required to start a debug session, including the executable name, processor target to debug, and other information. To create a debug configuration, right-click your software application and select **Debug As > Debug Configurations**.
- **Vitis Debug Perspective:** Using the Debug perspective, you can manage the debugging or running of a program in the Workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables. To view the Debug Perspective, select **Window > Open Perspective > Debug**.

You can repeat the cycle of modifying the code, building the executable, and debugging the program in the Vitis debugger.

**Note:** If you edit the source after compiling, the line numbering will be out of step because the debug information is tied directly to the source. Similarly, debugging optimized binaries can also cause unexpected jumps in the execution trace.

---

## Debugging Software Using the Vitis Debugger

In this example, you will walk through debugging a hello world application.

If you did not create a hello world application on APU or RPU, follow the steps in [Create Bare-Metal Application for Arm Cortex-A53 based APU](#) to create a new hello world application.

After you create the Hello World Application, work through below example to debug the software using the Vitis debugger.

1. Follow the steps in [Example Project: Running the “Hello World” Application from Arm Cortex-A53](#) to set the target in JTAG mode and power ON.
2. In the C/C++ Perspective, right-click the `test_a53` Project and select **Debug As > Launch on Hardware (Application Debugger)**.

**Note:** The above step launches the Application Debugger in the Debug perspective based on the project settings. Alternatively, you can also create a Debug configuration which looks like [Figure 4-2](#).

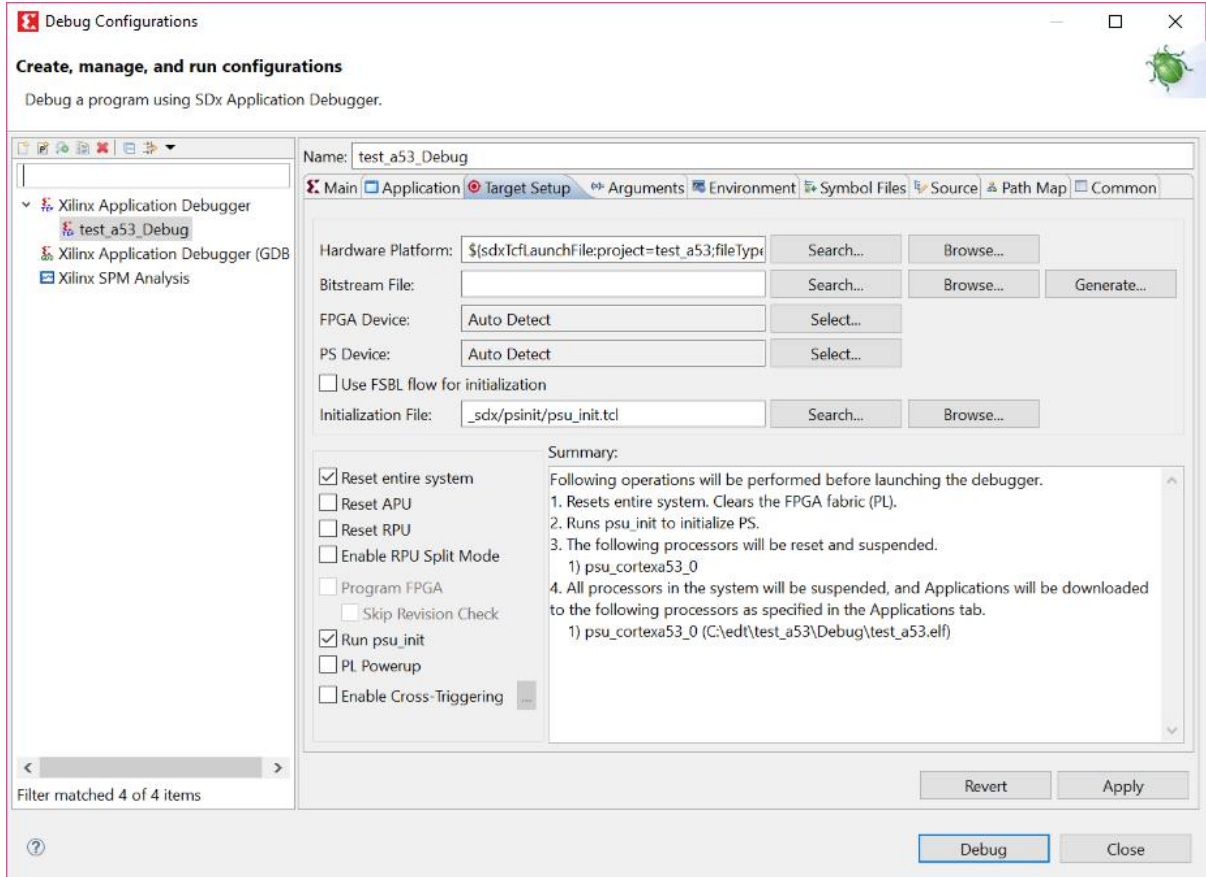


Figure 4-2: Debug Configurations

If the **Confirm Perspective Switch** popup window appears, click **Yes**. The Debug Perspective opens.

**Note:** If the Debug Perspective window does not automatically open, select **Window > Perspective > Open Perspective > Other**, then select **Debug** in the Open Perspective wizard.



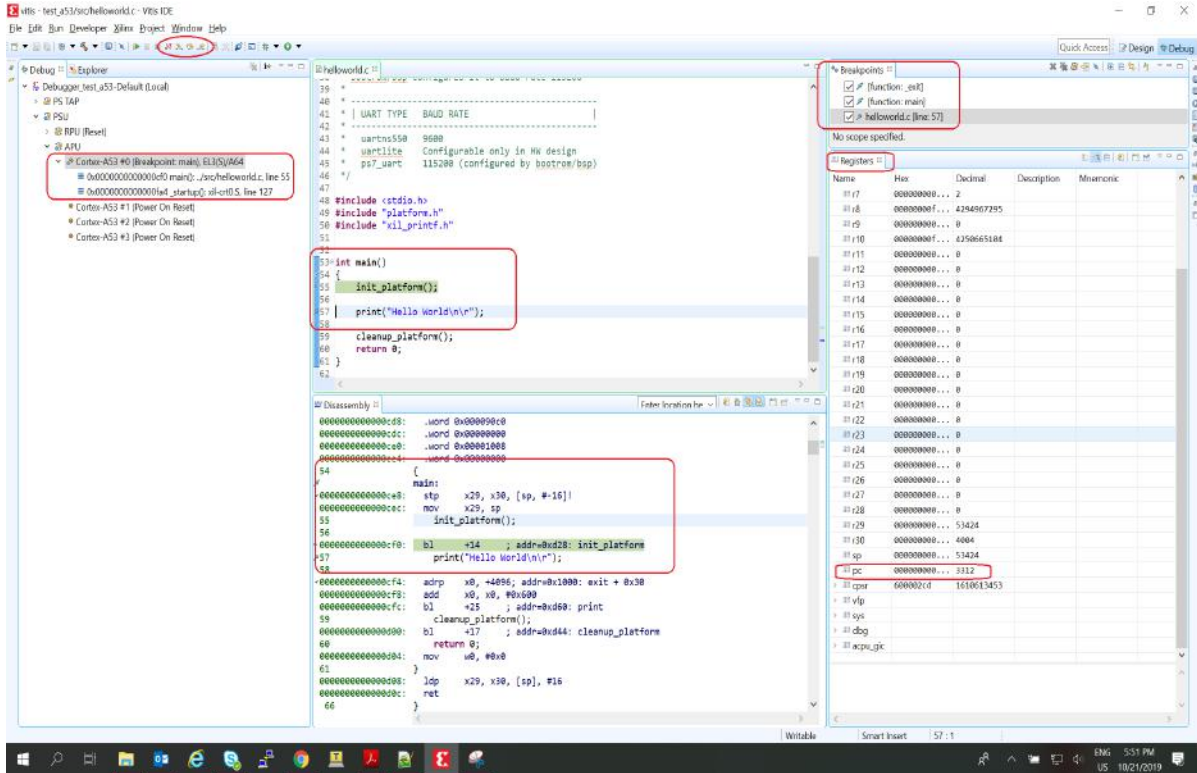


Figure 4-3: Application Debug Perspective

**Note:** The addresses shown on this page might slightly differ from the addresses shown on your system.

The processor is currently sitting at the beginning of `main()` with program execution suspended at line `0x0000000000000980`. You can confirm this information in the Disassembly view, which shows the assembly-level program execution also suspended at `0x0000000000000980`.

**Note:** If the Disassembly view is not visible, select **Window > Show View > Disassembly**.

3. The `helloworld.c` window also shows execution suspended at the first executable line of C code. Select the Registers view to confirm that the program counter, pc register, contains `0x0000000000000980`.

**Note:** If the Registers window is not visible, select **Window > Show View > Registers**.

4. Double-click in the margin of the `helloworld.c` window next to the line of code that reads `printf("Hello World\n\r");`. This sets a breakpoint at the `printf` command. To confirm the breakpoint, review the Breakpoints window.

**Note:** If the Breakpoints window is not visible, select **Window > Show View > Breakpoints**.

5. Select **Run > Step Into** to step into the `init_platform()` routine.

Program execution suspends at location `0x00000000000009c8`. The call stack is now two levels deep.

6. Select **Run > Resume** to continue running the program to the breakpoint.

Program execution stops at the line of code that includes the `printf` command. The Disassembly and Debug windows both show program execution stopped at `0x0000000000000984`.

**Note:** The execution address in your debugging window might differ if you modified the hello world source code in any way.

7. Select **Run > Resume** to run the program to conclusion.

When the program completes, the Debug window shows that the program is suspended in a routine called `exit`. This happens when you are running under control of the debugger.

8. Re-run your code several times. Experiment with single-stepping, examining memory, breakpoints, modifying code, and adding print statements. Try adding and moving views.



---

**TIP:** You can use the Vitis debugger debugging shortcuts for step-into (F5), step-return (F7), step-over (F6), and resume (F8).

---

---

## Debugging Using XSCT

You can use the previous steps to debug bare-metal applications running on RPU and PMU using Vitis Application Debugger GUI.

Additionally, you can debug in the command line mode using XSDB, which is encapsulated as a part of XSCT. In this example, you will debug the bare-metal application `testapp_r5` using XSCT.

Following steps indicate how to load a bare-metal application on R5 using XSCT.


This example is just to demonstrate the command line debugging possibility using XSDB/XSCT. Based on the requirement, you can choose to debug the code using either the System Debugger graphical interface or the command line debugger in XSCT. All XSCT commands are scriptable and this applies to the commands covered in this example.

### Set Up Target

1. Connect a USB cable between USB-JTAG J2 connector on target and the USB port on the host machine.
2. Set the board in JTAG Boot mode, where SW6 is set as shown in following figure.



Figure 4-4: SW6 Switch Settings for JTAG Boot Mode

3. Power on the Board using switch SW1.
4. Open XSCT Console, click the **XSCT Console** button  in the tool bar.  
Alternatively, you can also open the XSCT console from **Xilinx > XSCT Console**.
5. In the XSCT Console, connect to the target over JTAG using the `connect` command:

```
xscct% connect
```

The `connect` command returns the channel ID of the connection.

6. Command `targets` lists the available targets and allows you to select a target through its ID.

The targets are assigned IDs as they are discovered on the JTAG chain, so the target IDs can change from session to session.

For non-interactive usage such as scripting, the `-filter` option can be used to select a target instead of selecting the target through its ID:

```
xscct% targets
```

The targets are listed as shown in the following figure.

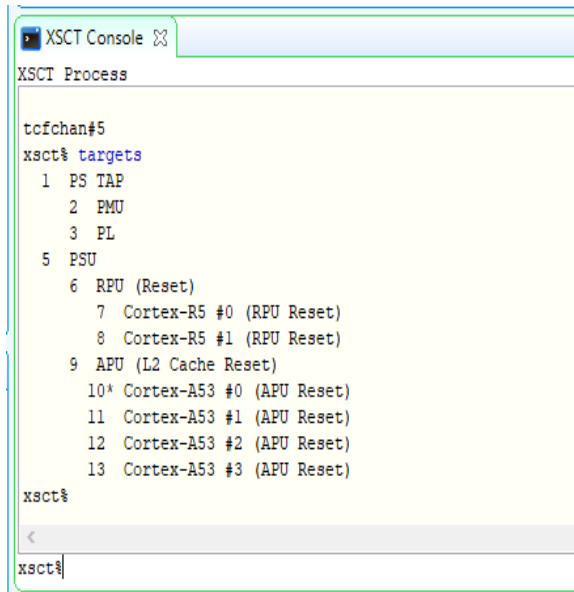


Figure 4-5: Target List

- Now select PSU target. The Arm APU and RPU clusters are grouped under PSU.

```
xsct% targets -set -filter {name =~ "Cortex-A53 #0"}
```

The command `targets` now lists the targets and also shows the selected target highlighted with an asterisk (\*) mark. You can also use target number to select a Target, as shown in the following figure.

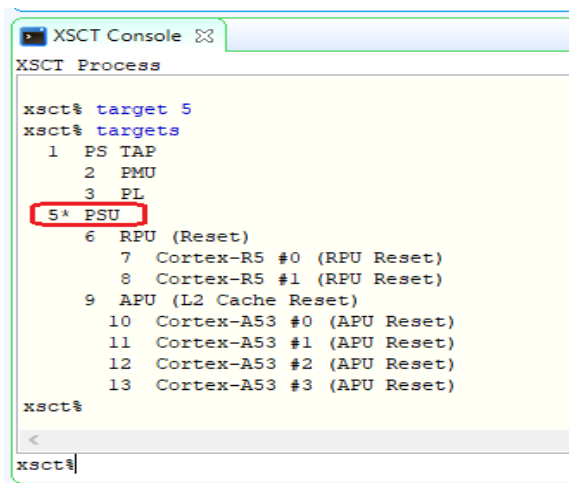


Figure 4-6: PSU Target Selected

- Load the fsb on Cortex-A53 #0. Fsb initialize the processing system of Zynq® UltraScale+™.

```

xsct% dow {C:\edt\fsbl_a53\Debug\fsbl_a53.elf}
xsct% con
xsct% stop

```

Note the `{}` used in above command. These are required on windows machine to enable backward slash (`\`) in paths. These braces can be avoided by using forward `/` in paths. Considering Linux paths, use forward `/` because the paths in XSCT in Linux can work as is, without any braces.

## Load the Application Using XSCT

1. Now download the `testapp_r5` application on Arm R5 Core 0.
2. Check and select RPU Cortex-R5 Core 0 target ID

```
xsct% targets
xsct% targets -set -filter {name =~ "Cortex-R5 #0"}
xsct% rst -processor
```

The command `rst -processor` clears the reset on an individual processor core.

This step is important, because when Zynq MPSoC boots up JTAG boot mode, all the A53 and R5 cores are held in reset. You must clear the resets on each core, before debugging on these cores. The `rst` command in XSDB can be used to clear the resets.

**Note:** The command `rst -cores` clears resets on all the processor cores in the group (such as APU or RPU), of which the current target is a child. For example, when A53 #0 is the current target, `rst -cores` clears resets on all the A53 cores in APU.

```
xsct% dow {C:\edt\testapp_r5\Debug\testapp_r5.elf}
```

Or

```
xsct% dow {C:/edt/testapp_r5/Debug/testapp_r5.elf}
```

At this point, you can see the sections from the ELF file downloaded sequentially. The XSCT prompt can be seen after successful download.

Now, configure a serial terminal (Tera Term, Mini com, or the Serial Terminal interface for UART-1 USB-serial connection).

## Serial Terminal Configuration

1. Start a terminal session, using Tera Term or Mini com depending on the host machine being used, and the COM port and baud rate as shown in following figure.

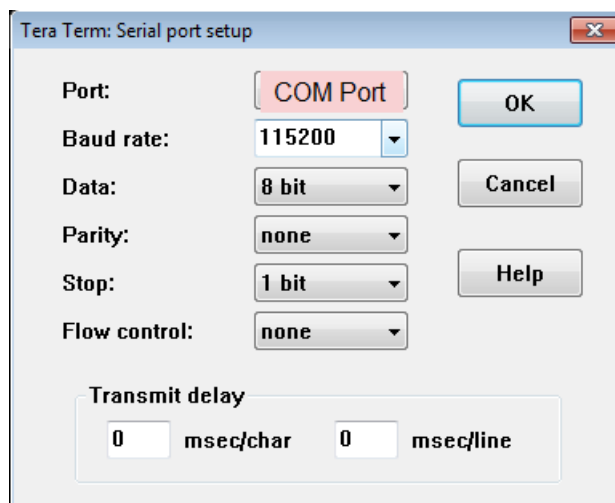


Figure 4-7: COM Port Set Up

2. For port settings, verify the COM port in the device manager. There are four USB UART interfaces exposed by the ZCU102 board. Select the COM port associated with the interface with the lowest number. So in this case, for UART-0, select the COM port with interface-0.
3. Similarly, for UART-1, select COM port with interface-1. Remember that R5 BSP has been configured to use UART-1, and so R5 application messages will appear on the COM port with UART-1 terminal.

## Run and Debug Application Using XSCT

1. Now before you run the application, set a breakpoint at `main()`.

```
xsc% bpadd -addr &main
```

This command returns the breakpoint ID.

You can verify the breakpoints planted using command `bp list`.

For more details on breakpoints in XSCT, type `help breakpoint` in XSCT,

2. Now resume the processor core.

```
xsc% con
```

The following informative messages will be displayed when the core hits the breakpoint.

```
xsc% Info: Cortex-R5 #0 (target 7) Stopped at 0x10021C (Breakpoint)
```

3. At this point, you can view registers when the core is stopped.

```
xsct% rrd
```

4. View local variables

```
xsct% locals
```

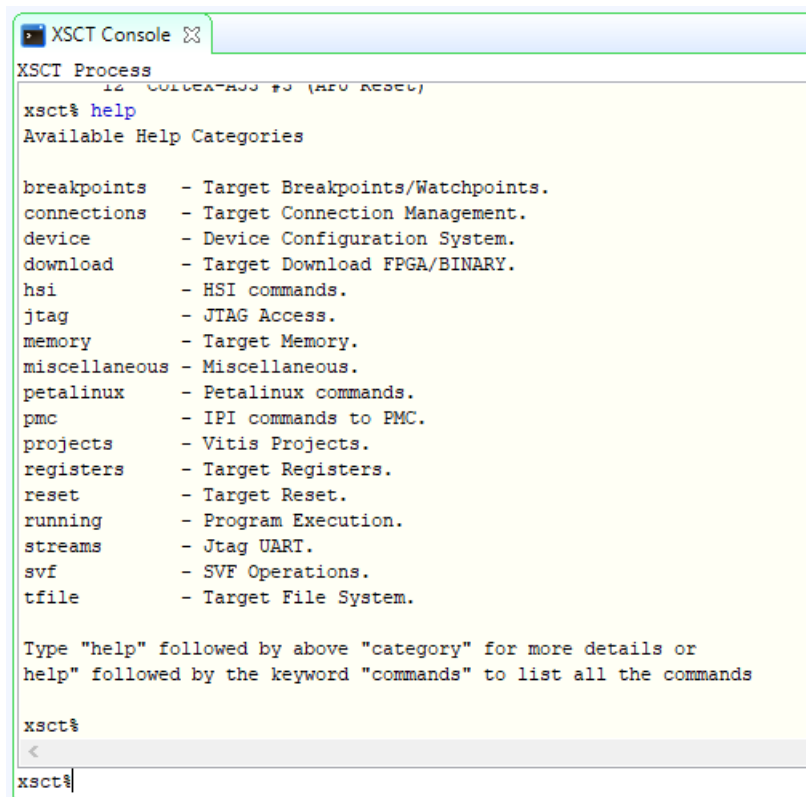
5. Step over a line of the source code and view the stack trace.

```
xsct% nxt
```

```
Info: Cortex-R5 #0 (target 6) Stopped at 0x100490 (Step)
```

```
xsct% bt
```

You can use the `help` command to find other options:



```
XSCT Console
XSCT Process
12 Cortex-R5 #0 (AFU RESE)
xsct$ help
Available Help Categories

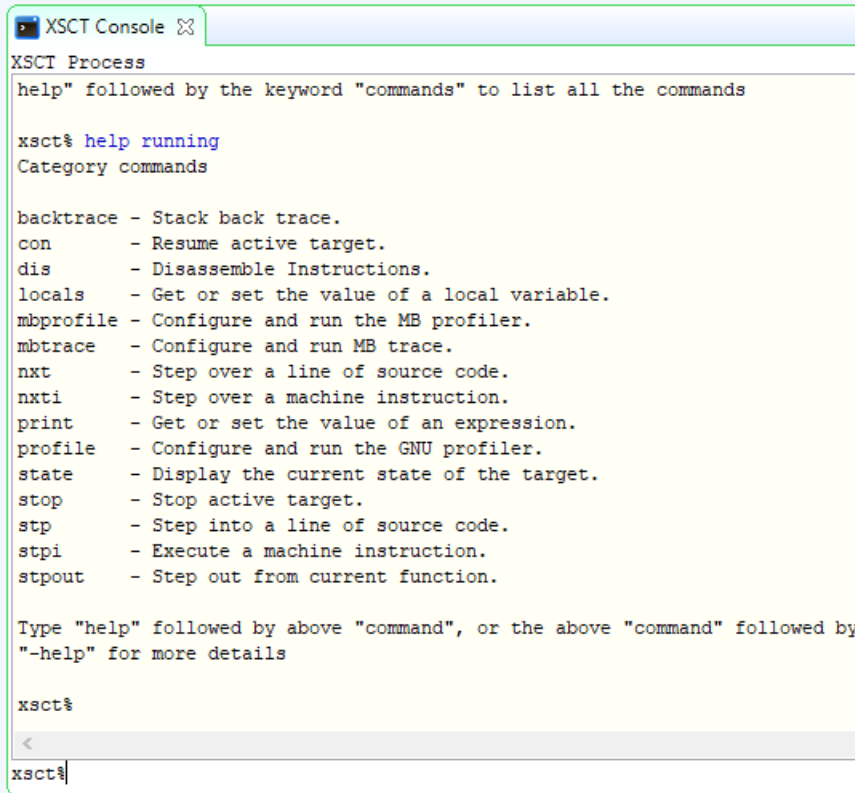
breakpoints - Target Breakpoints/Watchpoints.
connections - Target Connection Management.
device - Device Configuration System.
download - Target Download FPGA/BINARY.
hsi - HSI commands.
jtag - JTAG Access.
memory - Target Memory.
miscellaneous - Miscellaneous.
petalinux - Petalinux commands.
pmc - IPI commands to PMC.
projects - Vitis Projects.
registers - Target Registers.
reset - Target Reset.
running - Program Execution.
streams - Jtag UART.
svf - SVF Operations.
tfile - Target File System.

Type "help" followed by above "category" for more details or
help" followed by the keyword "commands" to list all the commands

xsct$
<
xsct$
```

Figure 4-8: XSCT Help Categories

You can use the `help running` command to get a list of possible options for running or debugging an application using XSCT.



```

XSCT Console
XSCT Process
help" followed by the keyword "commands" to list all the commands

xsct% help running
Category commands

backtrace - Stack back trace.
con       - Resume active target.
dis       - Disassemble Instructions.
locals    - Get or set the value of a local variable.
mbprofile - Configure and run the MB profiler.
mbtrace   - Configure and run MB trace.
nxt       - Step over a line of source code.
nxti      - Step over a machine instruction.
print     - Get or set the value of an expression.
profile   - Configure and run the GNU profiler.
state     - Display the current state of the target.
stop      - Stop active target.
stp       - Step into a line of source code.
stpi      - Execute a machine instruction.
stpout    - Step out from current function.

Type "help" followed by above "command", or the above "command" followed by
"-help" for more details

xsct%

```

Figure 4-9: XSCT Help for Debugging Program Execution

6. You can now run the code:

```
xsct% con
```

At this point, you can see the R5 application print message on UART-1 terminal.

## Debugging FSBL using the Vitis Debugger

The FSBL is built with Size Optimization and Link Time Optimization Flags, that is `-Os` and `LTO` optimizations by default in the Vitis debugger. This reduces the memory footprint of FSBL. This needs to be disabled for debugging FSBL.

Removing optimization can lead to increased code size, resulting in failure to build the FSBL. To disable the optimization (for debugging), some FSBL features (that are not required), need to be disabled in `xfsb1_config.h` file of FSBL.

Now, create a new FSBL for this section instead of modifying the FSBL created in [Chapter 3, Build Software for PS Subsystems](#). This is to avoid disturbing the `FSBL_a53` project, which will be used extensively in rest of the chapters in this tutorial.



## Create and Modify FSBL

Use the following steps to create an FSBL project.

1. Launch the Vitis debugger if it is not already open.
2. Set the Workspace path based on the project you created in [Chapter 3, Build Software for PS Subsystems](#). For example, `C:\edt`.
3. Select **File > New > Application Project**.

The New Project dialog box opens.

4. Use the information in the following table to make your selections in the New Project dialog box.

Table 4-1: Settings to Create FSBL\_debug Project

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	fsbl_debug
	Use Default Location	Select this option
	System Project	Create New
	Platform	<b>edt_zcu102_wrapper</b>
	Domain	standalone on psu_cortexa53_0
	Language	<b>C</b>
	CPU	psu_cortexa53_0
	OS	standalone
Click <b>Next</b>		
Templates	Available Templates	<b>Zynq MP FSBL</b>

5. Click **Finish**.

The Vitis debugger creates the System project and an FSBL application. Now disable Optimizations as shown below.

1. In the Project Explorer folder, right-click the **fsbl\_debug** application.
2. Click **C/C++ Build Settings**.
3. Select **Settings > Tool Settings** tab > **Arm v8 gcc Compiler > Miscellaneous**
4. Remove **-flto -ffat-lto-objects** from other flags, as shown below.

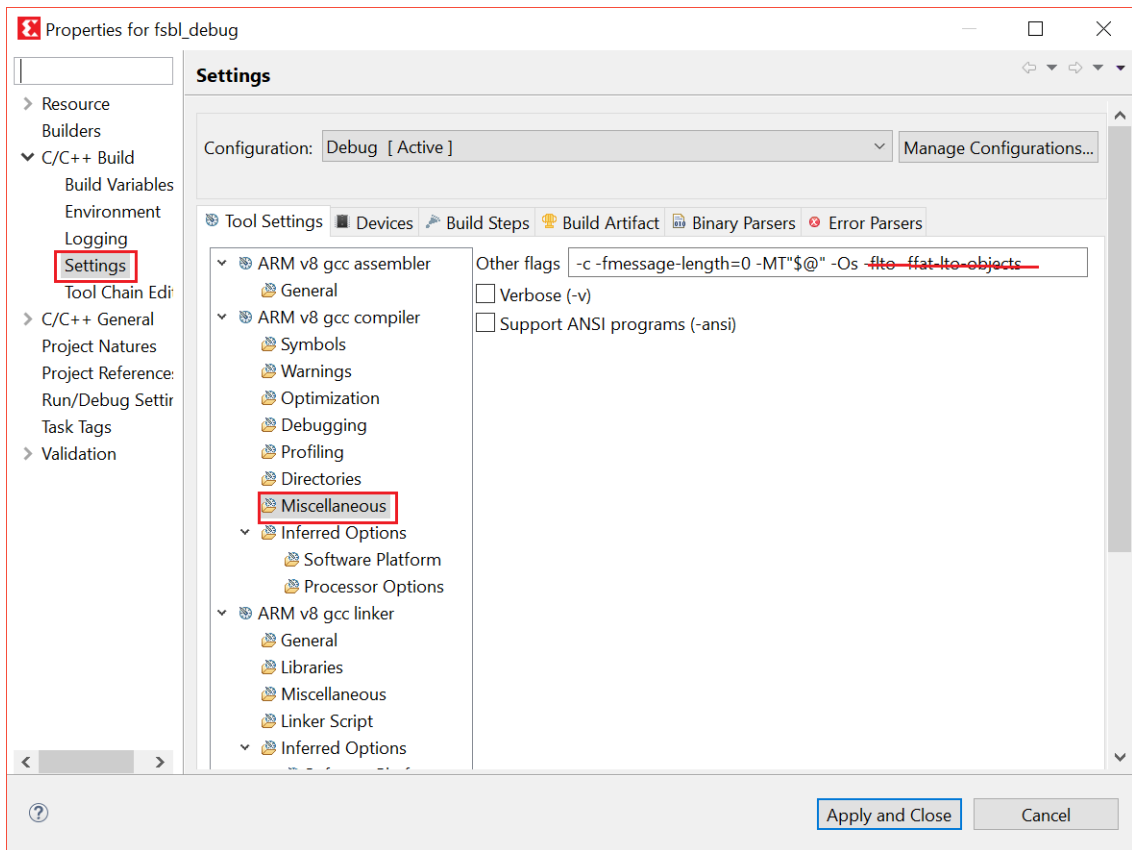


Figure 4-10: Modify FSBL BSP Build Settings

Similarly, the fsbl\_debug\_bsp needs to be modified to disable optimization.

5. Right-click **fsbl\_debug\_bsp** and select **Board Support Package Settings**.
6. Under **Overview > Drivers > psu\_cortexa53\_0 > extra\_compiler\_flags**, edit extra\_compiler\_flags to ensure extra compiler has this value "-g -Wall -Wextra -Os" as shown below.

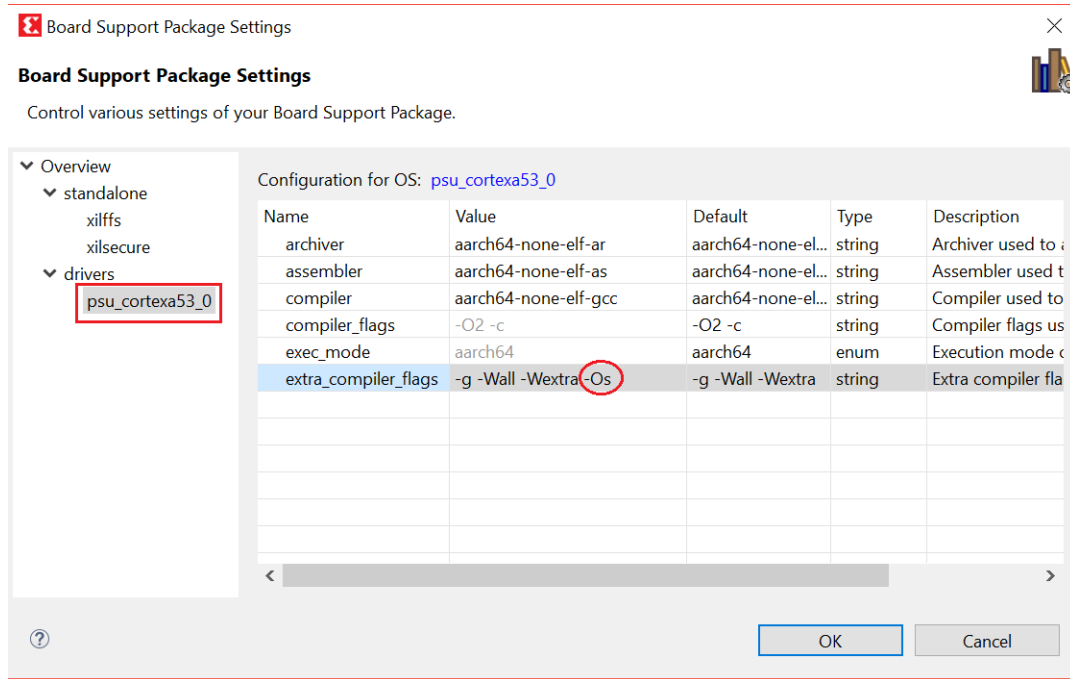


Figure 4-11: Modify FSBL BSP Build Settings

7. Click **OK**, to save these settings. BSP re-builds automatically after this.
8. Go to the `fsbl_debug>src>fsbl_config.h` file. In the FSBL code include the options and disable the following:
  - `#define FSBL_NAND_EXCLUDE_VAL (1U)`
  - `#define FSBL_SECURE_EXCLUDE_VAL(1U)`

**Note:** '1' is disable and '0' is enable.

At this point FSBL is ready to be debugged.

You can either debug the FSBL like any other standalone application (as shown in [Debugging Software Using the Vitis Debugger](#) and [Debugging Using XSCT](#)), or debug FSBL as a part of a Boot image by using the 'Attach to running target' mode of System Debugger.

# Boot and Configuration

This chapter shows integration of components to create a Zynq UltraScale+ system. The purpose of this chapter is to understand how to integrate and load Boot loaders, bare-metal applications (For APU/RPU), and Linux Operating System for a Zynq UltraScale+ system.

The following important points are covered in this chapter:

- System Software: FSBL, U-Boot, Arm® trusted firmware (ATF)
- Application Processing Unit (APU): Configure SMP Linux for APU
- Real-time Processing Unit (RPU): Configure Bare-metal for RPU in Lock-step
- Create Boot Image for the following Boot sequence:
  - a. APU
  - b. RPU Lock-step
- Create and load Secure Boot Image

**Note:** For more information on RPU Lock-step, see *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 5].

This boot sequence also includes loading the PMU Firmware for the Platform Management Unit (PMU). You can achieve the above configurations using the Vitis IDE and PetaLinux Tool flow. While [Chapter 3](#) focused only on creating software blocks for each processing unit in the PS, this chapter explains how these blocks can be loaded as a part of a bigger system.

The Create Boot Image wizard (Bootgen - Command Line tool) from the Vitis IDE is used in generating Boot Image. Create Boot Image Wizard's or Bootgen's principle function is to integrate the partitions (hardware-bitstream and software), and allow you to specify the security options in the design. It can also create the cryptographic keys.

Functionally, Bootgen uses a Bootgen Image Format (BIF) file as an input, and generates a single file image in binary BIN or MCS format. Bootgen outputs a single file image which is loaded into NVM (QSPI, SD Card). The Bootgen GUI facilitates the creation of the BIF input file.

This chapter makes use of Processing System block. [Design Example 1: Using GPIOs, Timers, and Interrupts](#), covers Boot-image which will include the PS partitions used in this chapter and a bitstream targeted for PL fabric.

## System Software

The following system software blocks cover most of the Boot and Configuration for this chapter. For detailed boot flow and various Boot sequences, refer to the “System Boot and Configuration” chapter in the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [Ref 6].

### First Stage Boot Loader

In non-secure Boot mode, the platform management unit (PMU) releases the reset of the configuration security unit, and enters the PMU server mode to monitor power. At this stage the configuration security unit loads the first stage boot loader (FSBL) into on-chip memory (OCM). The FSBL can be run from either APU A53\_0 or RPU R5\_0 or RPU R5\_lockstep. In this example, the FSBL is targeted for APU A53 Core 0. The last 512 bytes of this region is used by FSBL to share the hand-off parameters corresponding to applications which ATF hands off.

The First Stage Boot Loader initializes important blocks in the processing subsystem. This includes clearing the reset of the processors, initializing clocks, memory, UART, and so on before handing over the control of the next partition in DDR, to either RPU or APU. In this example, the FSBL loads bare-metal application in DDR and handsoff to RPU R5 in Lockstep mode, and similarly loads U-Boot to be executed by APU A53 Core-0. For more information, see the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [Ref 6].

For this chapter, you can use the FSBL executable that you created in [Chapter 3](#). In FSBL application, the `xfsb1_translation_table.s` differs from `translation_table.s` (of A53) in only one aspect, to mark DDR region as reserved. This is to avoid speculative access to DDR before it is initialized. Once the DDR initialization is done in FSBL, memory attributes for DDR region is changed to “Memory” so that it is cacheable.

### Platform Management Unit Firmware

The platform management unit (PMU) and the configuration security unit manage and perform the multi-staged booting process. The PMU primarily controls the pre-configuration stage that executes PMU ROM to set up the system. The PMU handles all of the processes related to reset and wake-up. The Vitis IDE provides PMU Firmware that can be built to run on the PMU. For more details on the Platform Management and PMU Firmware, see the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [Ref 6].

The PMU Firmware can be loaded in the following ways:

1. Using BootROM to load PMU Firmware, as described in [Boot Sequence for SD-Boot](#)
2. Using FSBL to load PMU Firmware, as described in [Boot Sequence for QSPI Boot Mode](#)

3. Load PMU Firmware in JTAG boot mode, as described in [Boot Sequence for QSPI-Boot Mode Using JTAG](#).

For more information, see the [PMU Firmware Xilinx Wiki](#).

## U-Boot

The U-Boot acts as a secondary boot loader. After the FSBL handoff, the U-Boot loads Linux on Arm A53 APU. After FSBL, the U-Boot configures the rest of the peripherals in the processing system based on board configuration. U-Boot can fetch images from different memory sources like eMMC, SATA, TFTP, SD, and QSPI. For this example, U-Boot and all other images are loaded from the SD card. Therefore, for this example, the Board will be set to SD-boot mode.

U-Boot can be configured and built using the PetaLinux tool flow. For this example, you can use the U-Boot image that you created in [Chapter 3](#) or from the design files shared with this document. See [Design Files for This Tutorial, page 167](#) for information about downloading the design files for this tutorial.

## Arm Trusted Firmware

The Arm Trusted Firmware (ATF) is a transparent bare-metal application layer executed in Exception Level 3 (EL3) on APU. The ATF includes a Secure Monitor layer for switching between secure and non-secure world. The Secure Monitor calls and implementation of Trusted Board Boot Requirements (TBBR) makes the ATF layer a mandatory requirement to load Linux on APU on Zynq UltraScale+.

The FSBL loads ATF to be executed by APU, which keeps running in EL3 awaiting a service request. The ATF starts at 0xFFFEA000. The FSBL also loads U-Boot in DDR to be executed by APU, which loads Linux OS in SMP mode on APU. It is important to note that the PL Bitstream should be loaded before ATF is loaded. The reason is FSBL uses the OCM region which is reserved for ATF for holding a temporary buffer in the case where bitstream is present in `.BIN` file. Because of this, if bitstream is loaded after ATF, FSBL will overwrite the ATF image with its temporary buffer, corrupting ATF image. Hence, bitstream should be positioned in `.BIF` before ATF and preferably immediately after FSBL and PMUFW.

The ATF (`b131.e1f`) is built by default in PetaLinux and can be found in the PetaLinux Project images directory.

For more details on ATF, refer to the "Arm Trusted Firmware" section in the "Security" chapter of the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [\[Ref 6\]](#).

---

## Linux on APU and Bare-Metal on RPU

Now that the system software is configured, create Linux Images using PetaLinux Toolflow. You already created the PetaLinux images in [Chapter 3](#). For this example, the PetaLinux is configured to build images for SD-boot. This is the default boot setting in PetaLinux.

The images can be found in the `$<PetaLinux_Project>/images/linux/` directory.

For loading Linux on APU, the following images will be used from PetaLinux:

- ATF - `b131.elf`
- U-Boot - `u-boot.elf`
- Linux images - `image.ub`, which contains:
  - Kernel image
  - Device Tree `system.dtb`
  - Filesystem - `rootfs.cpio.gz.u-boot`

In addition to Linux on APU, this example also loads a bare-metal Application on RPU R5 in Lockstep mode.

For this example, refer the `testapp_r5` application that you created in [Create Bare-Metal Application for Arm Cortex-R5 based RPU, page 39](#).

Alternatively you can also find the `testapp_r5.elf` executable in the design files that accompany this tutorial. See [Design Files for This Tutorial, page 167](#) for information about downloading the design files for this tutorial.

---

## Boot Sequence for SD-Boot

Now that all the individual images are ready, let's create the boot image to load all of these components on Zynq UltraScale+. This can be done using the Create Boot Image wizard in the Vitis IDE, using the following steps:

1. In the Vitis IDE, select **Xilinx > Create Boot Image**.
2. Select all the partitions referred in earlier sections in this chapter, and set them as shown in the following figure.

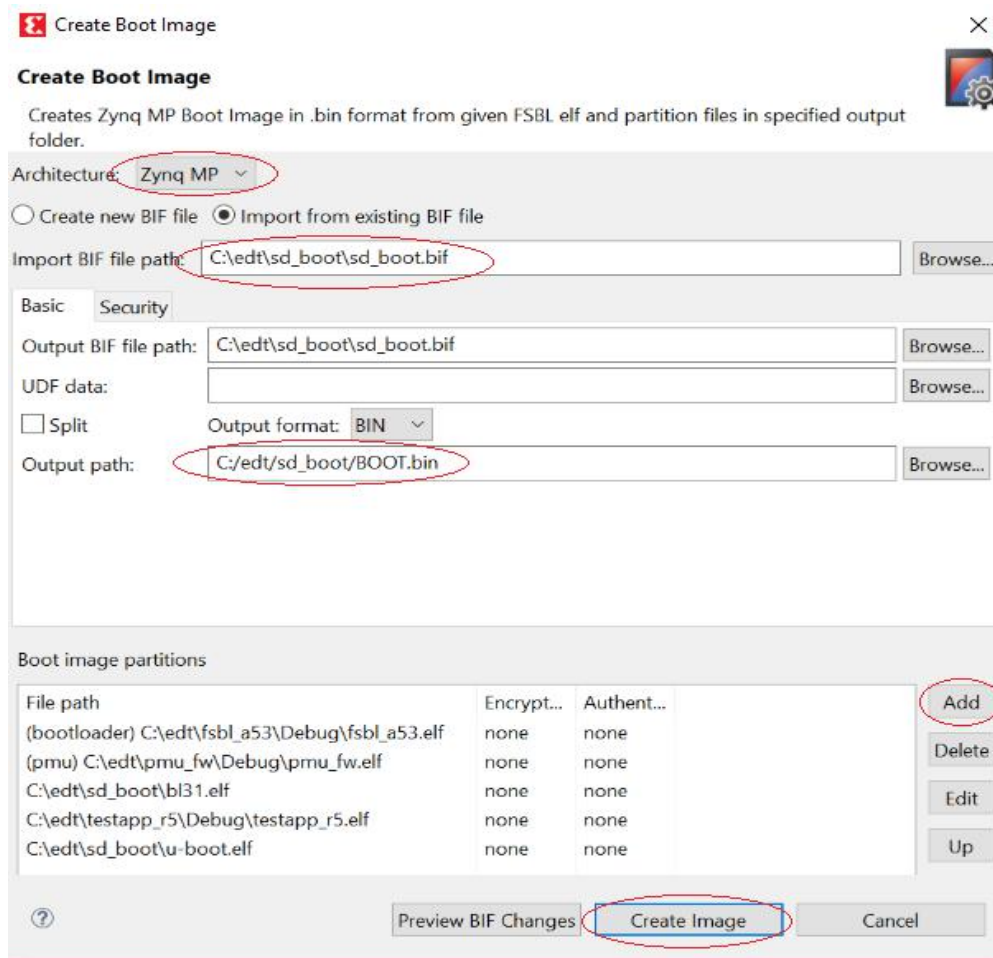


Figure 5-1: Create Boot Image for SD Boot Mode

First, add the FSBL partition.

1. In the Create Boot Image dialog box, click **Add** to open the Add partition dialog box.
2. In the Add Partition dialog box, click **Browse** to select the FSBL executable.
3. For FSBL, ensure that the partition type is selected as `bootloader` and the correct destination CPU is selected by the tool. The tool is configured to make this selection based on the FSBL executable.

**Note:** Ignore the Exception Level drop down, as FSBL is set to EL3 by default. Also, leave the Trustzone setting unselected for this example.



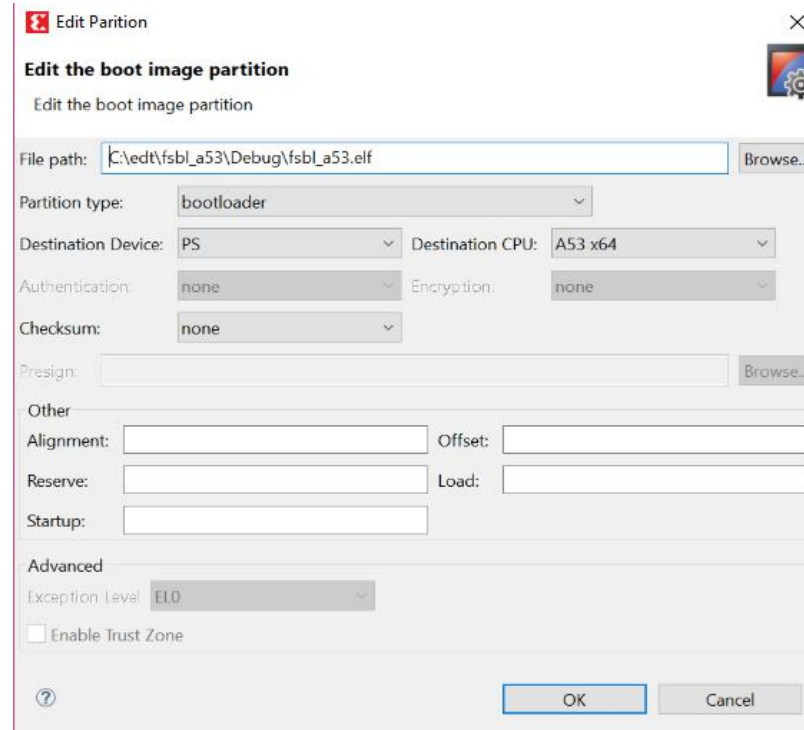


Figure 5-2: Add New Boot Image Partition Dialog Box

4. Click **OK** to select FSBL and go back to Create Boot Image wizard.

Next, add the PMU and ATF firmware partitions.

1. Click **Add** to open the Add Partition dialog box, shown in the following figure.

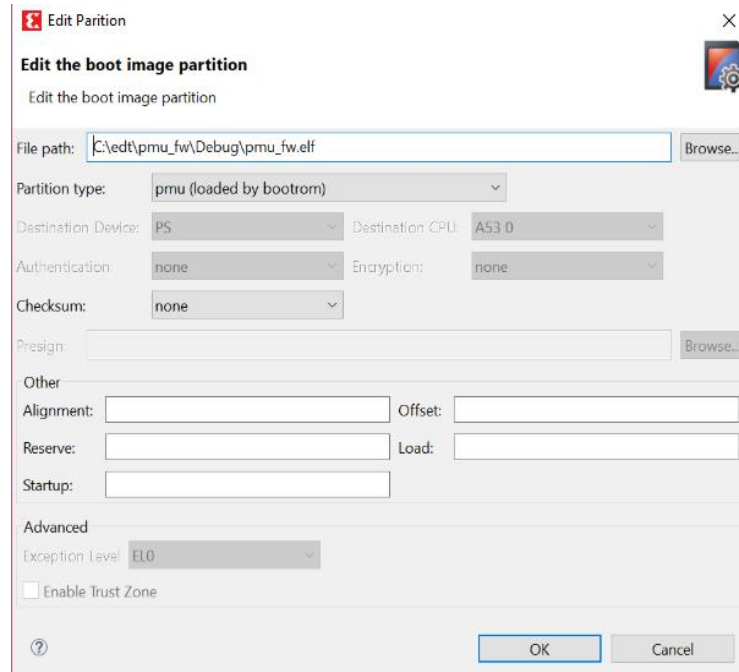


Figure 5-3: Add PMUFW Partition

2. Add the PMU firmware partition.
  - a. Browse to and select the PMU Firmware executable.
  - b. For this partition, select **pmu** as the partition type.
 

**Note:** The **pmu** partition type also implies that the executable is targeted for PMU. Therefore, the **Destination Device** and **Destination CPU** are grayed out for this setting.
3. Leave the Exception Level and Trustzone settings unselected.
4. Click **OK**.
5. Click **Add** to open Add Partition dialog box.
6. Add the **ATF firmware** `b131.e1f` partition.
 

**Note:** ATF Firmware (`b131.e1f`) can be found in <PetaLinux Project>/image/linux/. Alternatively, you can also use `b131.e1f` from [Design Files for This Tutorial](#).

  - a. For this partition, select **datafile** as the partition type.
  - b. Set the Destination Device as **PS**.
  - c. Set the Destination CPU as **A53 0**.
  - d. Set the Exception Level to **EL3** and select Enable **Trustzone**.

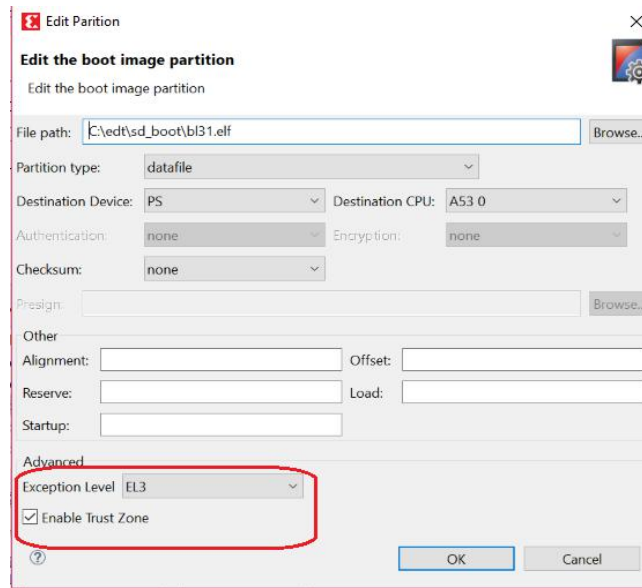


Figure 5-4: Add ATF partition

7. Click **OK**.

Next, add the R5 executable and enable it in lockstep mode.

1. Click **Add** to add the R5 bare-metal executable.

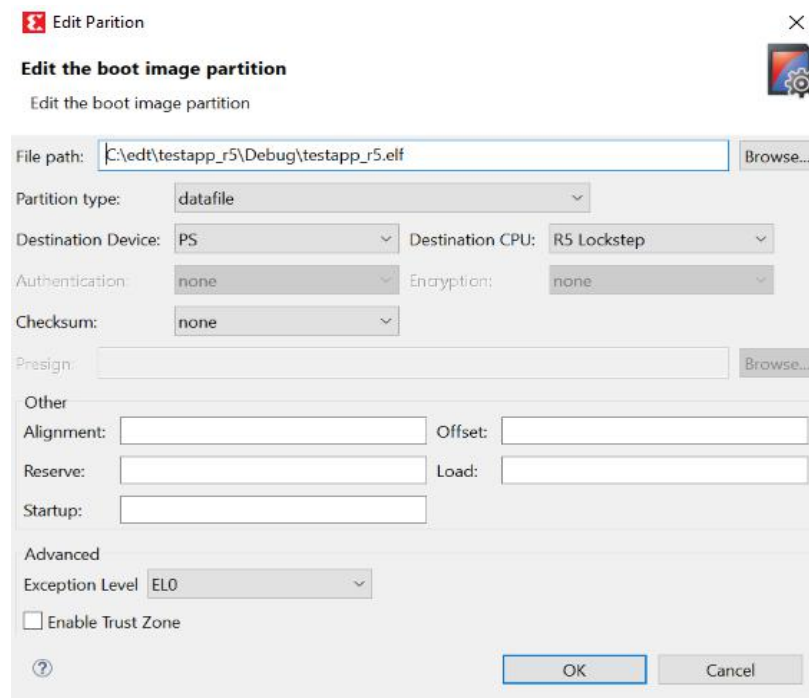


Figure 5-5: Add RPU Image Partition

2. Set the Destination Device as **PS**.

3. Set the Destination CPU as **R5 Lockstep**.

This sets the RPU R5 cores to run in Lockstep mode.

4. Leave Exception Level and Trustzone unselected.
5. Click **OK**.

Now, add the U-Boot partition. You can find `u-boot.elf` for `sd_boot` mode in `<PetaLinux_project>/images/linux/sd_boot`

1. Click **Add** to add the `u-boot.elf` partition.
2. For U-Boot, select the Destination Device as PS.
3. Select the Destination CPU as **A53 0**.
4. Set the Exception Level to **EL2**.

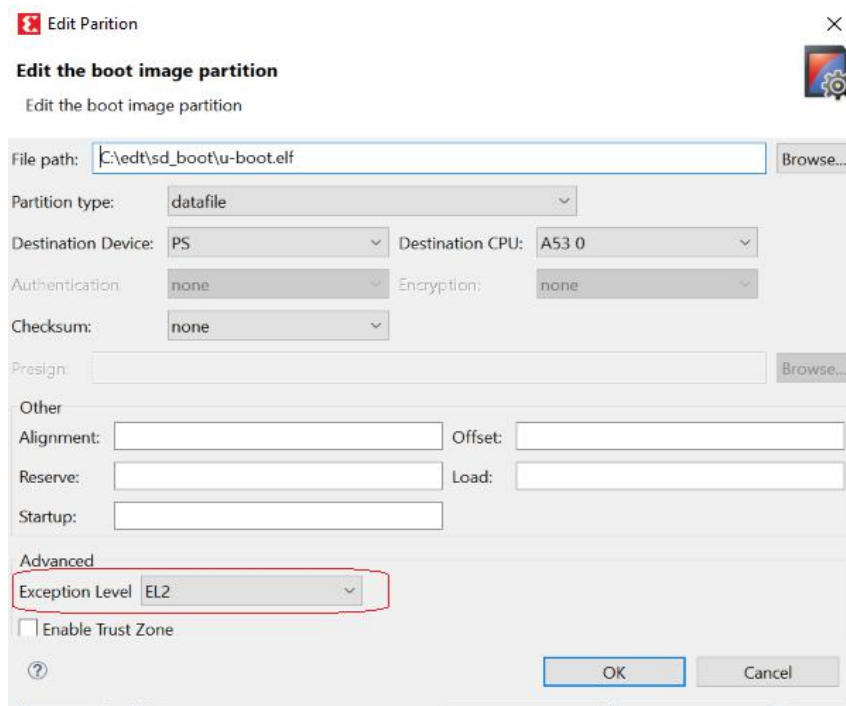


Figure 5-6: Add U-Boot Partition

5. Click **OK** to return to the Create Boot Image wizard.
6. Click **Create Image** to close the wizard and create the boot image.

You can also create BOOT.bin images using the BIF attributes and the `Bootgen` command.

For this configuration, the BIF file contains following attributes:

```
the_ROM_image:
{
```

```
[fsbl_config]a53_x64
[bootloader]C:\edt\fsbl_a53\Debug\fsbl_a53.elf
[pmufw_image]C:\edt\pmu_fw\Debug\pmu_fw.elf
[destination_cpu = a53-0, exception_level = e1-3, trustzone]C:\edt\sd_boot\bl31.elf
[destination_cpu = r5-lockstep]C:\edt\testapp_r5\Debug\testapp_r5.elf
[destination_cpu = a53-0, exception_level = e1-2]C:\edt\sd_boot\u-boot.elf
}
```

The Vitis IDE calls the following Bootgen command to generate the BOOT.bin image for this configuration:

```
bootgen -image sd_boot.bif -arch zynqmp -o C:\edt\sd_boot\BOOT.bin
```

## Running the Image on the ZCU102 Board

1. Copy the `BOOT.bin` and `image.ub` images to an SD card.
2. Load the SD card into the ZCU102 board, in the J100 connector.
3. Connect a micro USB cable from ZCU102 Board USB UART port (J83), to the USB port on the host Machine.
4. Configure the Board to Boot in SD-Boot mode by setting switch SW6 to 1-ON, 2-OFF, 3-OFF and 4-OFF, as shown in following figure.

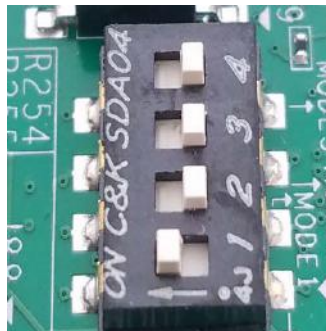


Figure 5-7: SW6 Switch Settings for SD Boot Mode

5. Connect 12V Power to the ZCU102 6-Pin Molex connector.

6. Start a terminal session, using Tera Term or Minicom depending on the host machine being used, as well as the COM port and baud rate for your system, as shown in following figure.

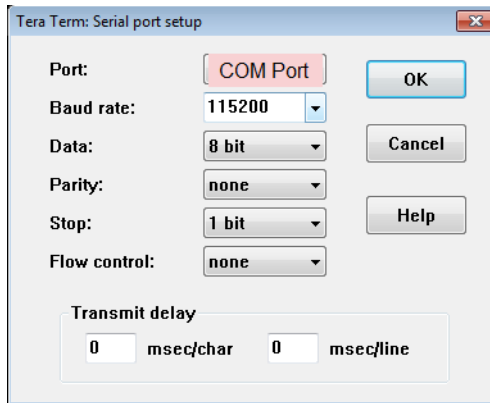


Figure 5-8: COM Port Set Up

7. For port settings, verify COM port in device manager.

There are four USB-UART interfaces exposed by the ZCU102 Board.

8. Select the COM port associated with the interface with the lowest number. In this case, for UART-0, select the COM port with interface-0.
9. Similarly, for UART-1, select COM port with interface-1.

Remember that the R5 BSP has been configured to use UART-1, and so R5 application messages will appear on the COM port with the UART-1 terminal.

10. Turn on the ZCU102 Board using SW1, and wait until Linux loads on the board.

At this point, you can see the initial Boot sequence messages on your Terminal Screen representing UART-0.

You can see that the terminal screen configured for UART-1 also prints a message. This is the print message from the R5 bare-metal Application running on RPU, configured to use UART-1 interface. This application is loaded by the FSBL onto RPU.

The bare-metal application has been modified to include the UART interrupt example. This application now waits in the waiting for interrupt (WFI) state until a user input is encountered from Keyboard in UART-1 terminal.



Figure 5-9: Hello World Displayed on UART-1 from R5-0

Meanwhile, the boot sequence continues on APU and the images loaded can be understood from the messages appearing on the UART-0 terminal. The messages are highlighted in the following figure.

```

Xilinx Zynq MP First Stage Boot Loader
Release 2019.2 Aug 27 2019 - 16:10:19
Reset Mode : System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU9EG
FMC UADJ Configuration Successful
Board Configuration successful
Processor Initialization Done
===== In Stage 2 =====
SD1 with level shifter Boot Mode
SD: rc= 0
File name is BOOT.BIN
Multiboot Reg : 0x0
Image Header Table Offset 0x8C0
*****Image Header Table Details*****
Boot Gen Ver: 0x1020000
No of Partitions: 0x5
Partition Header Address: 0x440
Partition Present Device: 0x0
Initialization Success
===== In Stage 3, Partition No:1 =====
UnEncrypted data Length: 0x31DE
Data word offset: 0x31DE
Total Data word length: 0x31DE
Destination Load Address: 0xFFFA000
Execution Address: 0xFFFA000
Data word offset: 0x10DB0
Partition Attributes: 0x117
Partition 1 Load Success
===== In Stage 3, Partition No:2 =====
UnEncrypted data Length: 0x128
Data word offset: 0x128
Total Data word length: 0x128
Destination Load Address: 0x0 FSBL on ARM Cortex-A53
Execution Address: 0x3C
Data word offset: 0x13F90
Partition Attributes: 0x71E
Initializing TCM ECC
Address 0xFFE00000, Length 40000, ECC initialized
Partition 2 Load Success
===== In Stage 3, Partition No:3 =====
UnEncrypted data Length: 0x1B74
Data word offset: 0x1B74
Total Data word length: 0x1B74
Destination Load Address: 0x70000000
Execution Address: 0x0
Data word offset: 0x140C0
Partition Attributes: 0x71E
Partition 3 Load Success
===== In Stage 3, Partition No:4 =====
UnEncrypted data Length: 0x33309
Data word offset: 0x33309
Total Data word length: 0x33309
Destination Load Address: 0x10080000
Execution Address: 0x10080000
Data word offset: 0x15C40
Partition Attributes: 0x114
Partition 4 Load Success
All Partitions Loaded
===== In Stage 4 =====
Protection configuration applied
CPU 0x700 reset release. Exec State 0x8. HandoffAddress: 3C
ATF running on XCZU9EG/silicon v4/RTL5.1 at 0xfffa000
NOTICE: BL31: Secure code at 0x0
NOTICE: BL31: Non secure code at 0x10080000 ARM Trusted Firmware
NOTICE: BL31: v2.0(release):xilinx-v2019.1-13-gc776b5f7
NOTICE: BL31: Built : 02:59:42, Aug 27 2019
PMUFW: v1.1
PMU Firmware Loaded

U-Boot 2019.01 (Aug 27 2019 - 03:00:29 +0000)
Model: ZynqMP ZCU102 Rev1.0
Board: Xilinx ZynqMP
DRAM: 4 GiB U-boot on ARM Cortex-A53
EL Level: EL2
Chip ID: zu9eg
MMC: mmc0ff170000: 0
Loading Environment from SPI Flash... SF: Detected n25q512a with page
*** Warning - bad CRC, using default environment
  
```

Figure 5-10: Messages from APU During Zynq UltraScale+ Boot Sequence

The U-Boot then loads Linux Kernel and other images on Arm Cortex® -A53 APU in SMP mode. The terminal messages indicate when U-Boot loads Kernel image and the kernel start up to getting a user interface prompt in Target Linux OS. The Kernel loading and starting sequence can be seen in the following figure.

```

Erase Group Size: 512 Bytes
24748260 bytes read in 1658 ms (14.2 MiB/s)
## Loading kernel from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'kernel@1' kernel subimage
  Description: Linux kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x10000100
  Data Size: 18082304 Bytes = 17.2 MiB
  Architecture: AArch64
  OS: Linux
  Load Address: 0x00080000
  Entry Point: 0x00080000
  Hash algo: sha1
  Hash value: 6777fff165c8ff2e473887685f9d1a31b9552167
Verifying Hash Integrity ... sha1+ OK
## Loading ramdisk from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'ramdisk@1' ramdisk subimage
  Description: petalinux-user-image
  Type: RAMDisk Image
  Compression: gzip compressed
  Data Start: 0x11148e64
  Data Size: 6622447 Bytes = 6.3 MiB
  Architecture: AArch64
  OS: Linux
  Load Address: unavailable
  Entry Point: unavailable
  Hash algo: sha1
  Hash value: 48abe1bb666e13715f2c2b61280f784f88eb7f2a
Verifying Hash Integrity ... sha1+ OK
## Loading fdt from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'fdt@system-top.dtb' fdt subimage
  Description: Flattened Device Tree blob
  Type: Flat Device Tree
  Compression: uncompressed
  Data Start: 0x1113ec0c
  Data Size: 41361 Bytes = 40.4 KiB
  Architecture: AArch64
  Hash algo: sha1
  Hash value: c024eecd7345889f0eb4961600333183b202dfe6
Verifying Hash Integrity ... sha1+ OK
Booting using the fdt blob at 0x1113ec0c
Loading Kernel Image ... OK
Loading Ramdisk to 789af000, end 78ffcef ... OK
Loading Device Tree to 0000000007ff2000, end 0000000007fff190 ... OK

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x00000000 [0x410fd034]
[ 0.000000] Linux version 4.19.0-xilinx-v2019.2 (oe-user@oe-host) (gcc version 8.2.0)
[ 0.000000] Machine model: ZynqMP ZCU102 Rev1.0
[ 0.000000] earlycon: cdns0 at MMIO 0x00000000ff000000 (options '115200n8')
[ 0.000000] bootconsole [cdns0] enabled
[ 0.000000] efi: Getting EFI parameters from FDT:
[ 0.000000] efi: UEFI not found.
    
```

Figure 5-11: Kernel Loading and Start Sequence



---

## Boot Sequence for QSPI Boot Mode

The ZCU102 board also comes with dual parallel QSPI Flashes adding up to 128 MB size. In this example, you will create a boot image and load the images on Zynq UltraScale+ in QSPI boot mode. The images can be configured using the Create Boot Image wizard in the Vitis IDE. This can be done by doing the following steps.

**Note:** This section assumes that you have created PetaLinux Images for QSPI Boot mode by following steps from [Create Linux Images using PetaLinux for QSPI Flash](#).

1. If the Vitis IDE is not already running, start it and set the workspace as indicated in [Chapter 3](#).
2. Select **Xilinx > Create Boot Image**.
3. Select **Zynq MP** as the Architecture.
4. Select the **Create new BIF file** option.
5. Ensure that the **Output format** is set to **BIN**.
6. In the Basic tab, browse to and select the **Output BIF file path** and **Output path**.

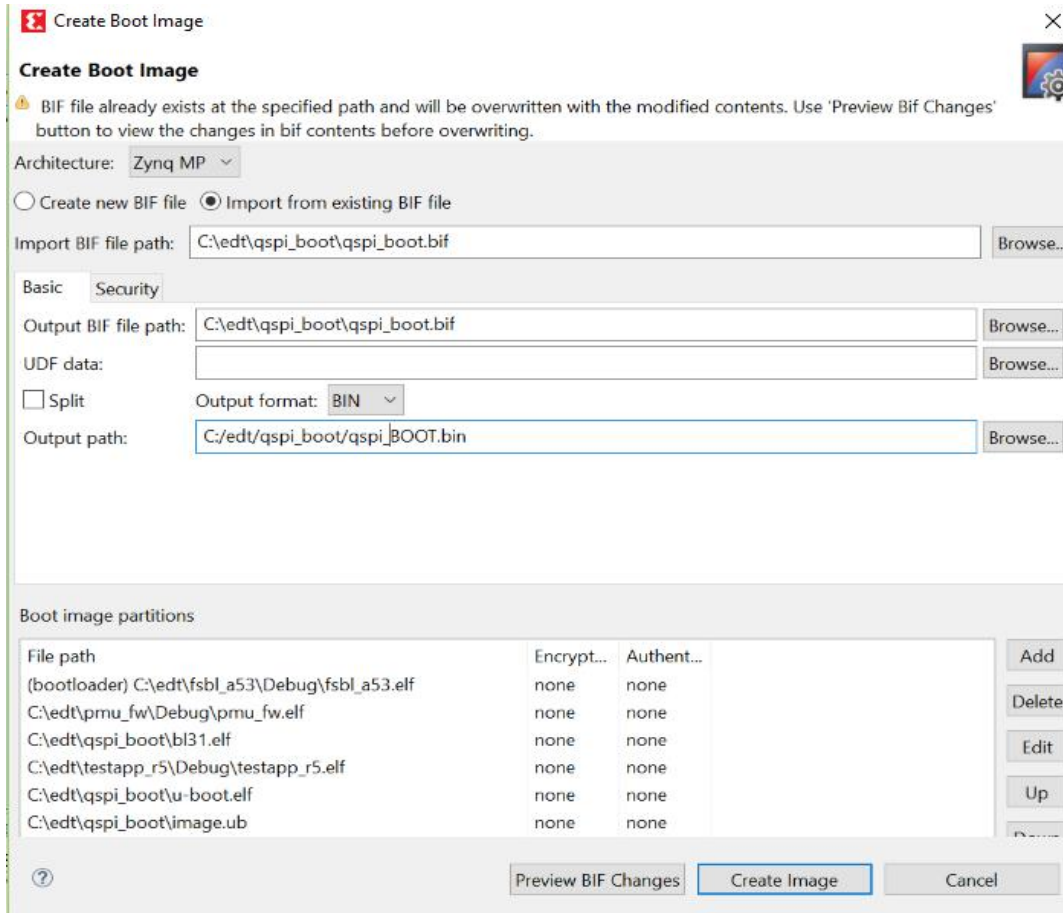


Figure 5-12: Create Boot Image for QSPI Boot Mode

Next, add boot partitions.

1. Click **Add** to open the Add Partition dialog box.
2. In the Add Partition dialog box, click the **Browse** button to select the FSBL executable.
  - a. For FSBL, ensure that the **Partition type** is selected as **bootloader** and the correct destination CPU is selected by the tool. The tool is configured to make this selection based on the FSBL executable.

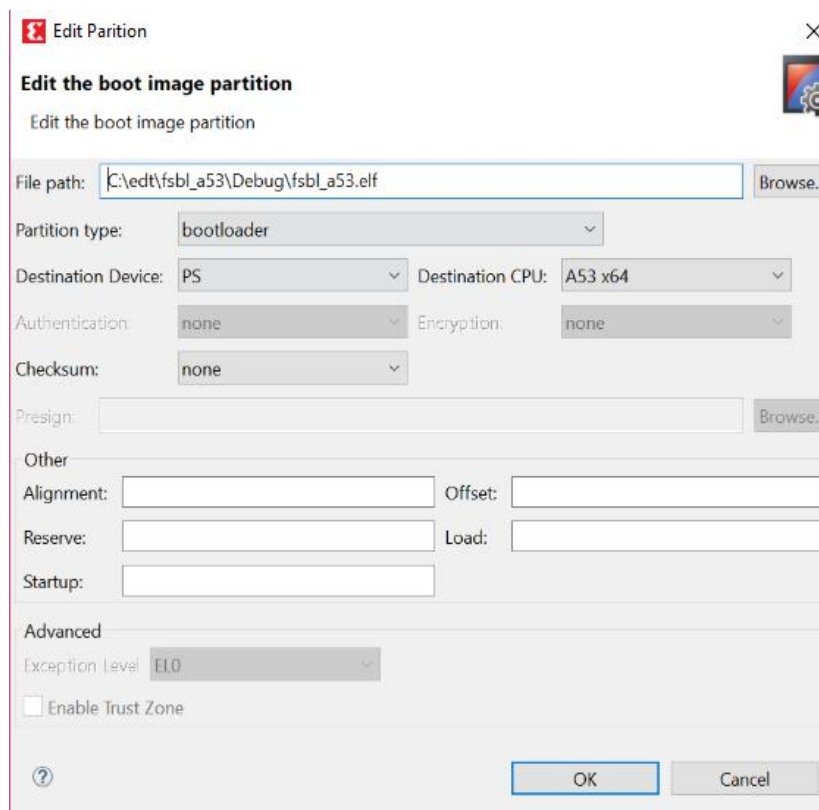


Figure 5-13: Add New Boot Image Partition Dialog Box

- b. Ignore the Exception Level, as FSBL is set to EL3 by default. Also, leave the Trustzone setting unselected for this example.
  - c. Click **OK** to select the FSBL and go back to the Create Boot Image wizard.
3. Click **Add** to open the Add Partition window to add the next partition.
4. The next partition is the PMU firmware for the Platform Management Unit.
  - a. Select the **Partition type** as **datafile** and the **Destination Device** as **PS**.
  - b. Select **PMU** for Destination CPU.
  - c. Click **OK**.

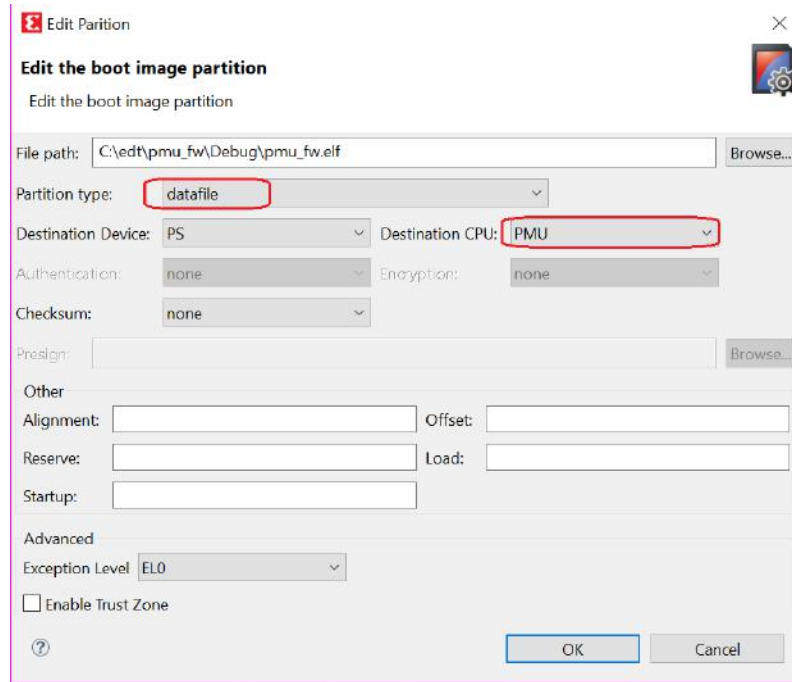


Figure 5-14: Add PMU Partition Details

5. The next partition to be added is the **ATF firmware**. For this, set the **Partition type** to **datafile**.
  - a. The ATF executable `bl31.elf` can be found in the PetaLinux images folder `<PetaLinux_project>/images/linux/`.
  - b. Select the **Destination Device** as **PS** and the **Destination CPU** as **A53 0**.
  - c. Set the Exception Level to **EL3** and select **Enable Trustzone**.

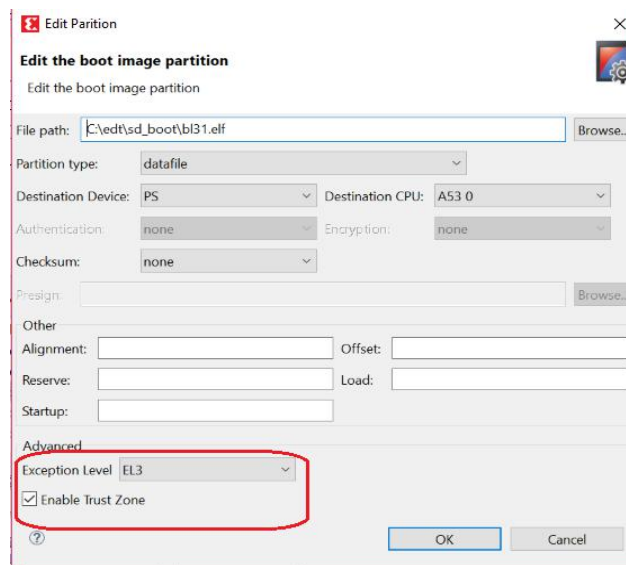


Figure 5-15: Add ATF Partition

- d. Click **OK**.
6. Click **Add** to add the R5 bare-metal executable.
  - a. Add the R5 executable and enable it in **lockstep** mode, as shown in the following image.
  - b. Click **OK**.

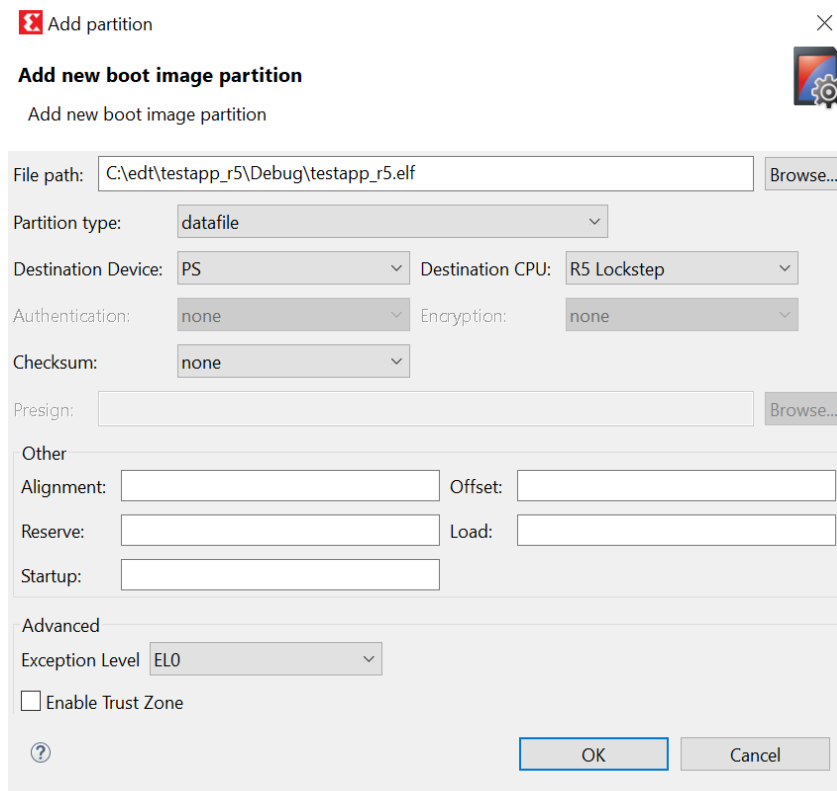


Figure 5-16: Add RPU Lockstep Image Partition

7. Click **Add** to add the U-boot partition. `u-boot.elf` can be found in `<PetaLinux_Project>/images/linux/`
  - a. For U-Boot, make the following selections:
    - Set the **Partition Type** to **datafile**.
    - Set the **Destination Device** to **PS**.
    - Set the **Destination CPU** to **A53 0**.
    - Set the Exception Level to **EL2**.

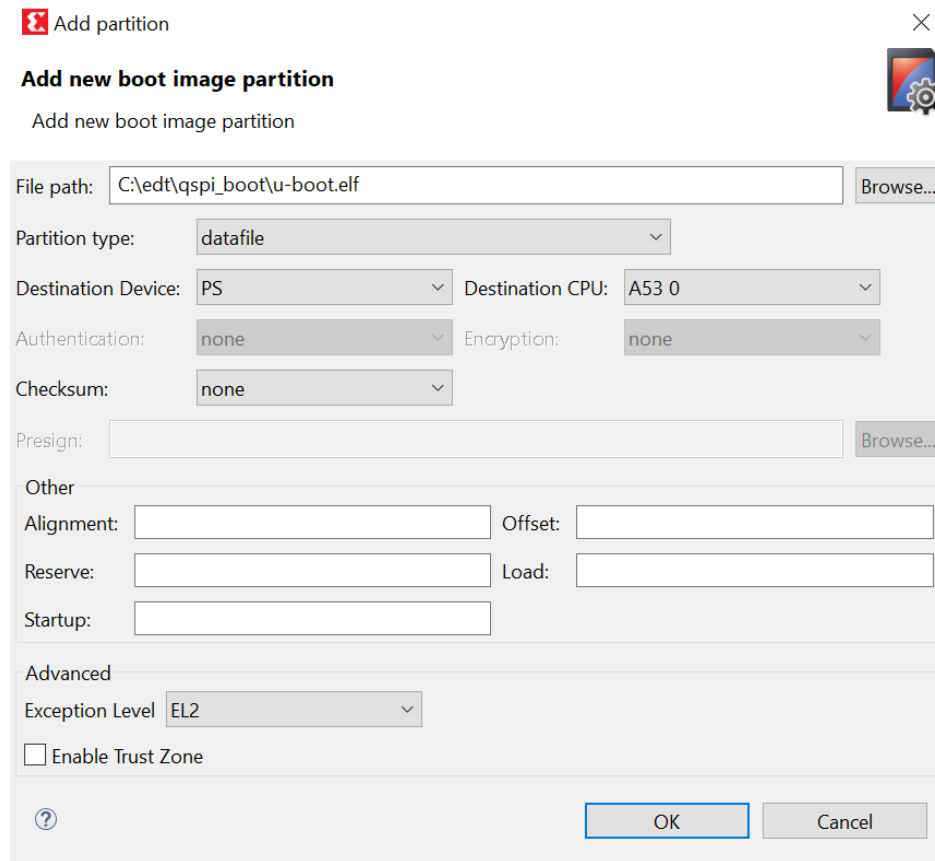


Figure 5-17: Add U-Boot Partition

- b. Click **OK**.
8. Click **Add** to add the `image.ub` Linux image file.
  - a. The `image.ub` image file can be found in PetaLinux project in the `images/Linux` directory.
  - b. For `image.ub`, make the following selections:
    - Set **Partition Type** to **datafile**.
    - Set the **Destination Device** to **PS**.
    - Set the **Destination CPU** to **A53 0**.
  - c. Enter `0x1E40000` as the **Offset**.
  - d. Leave Exception Level and Trustzone unselected.



**TIP:** See [Create Linux Images using PetaLinux for QSPI Flash](#), to understand the offset value.

9. Click **OK** to go back to Create Boot Image wizard.
10. Click **Create Image** to create the `qspi_BOOT.bin` image.

You can also create `qspi_BOOT.bin` images using the BIF attributes and the Bootgen command. You can view the BIF attributes for this configuration by clicking **Preview BIF Changes**. For this configuration, the BIF file contains following attributes:

```
the_ROM_image:
{
  [fsbl_config]a53_x64
  [bootloader]C:\edt\fsbl_a53\Debug\fsbl_a53.elf
  [destination_cpu = pmu]C:\edt\pmu_fw\Debug\pmu_fw.elf
  [destination_cpu = a53-0, exception_level = e1-3,
  trustzone]C:\edt\qspi_boot\bl31.elf
  [destination_cpu = r5-lockstep]C:\edt\testapp_r5\Debug\testapp_r5.elf
  [destination_cpu = a53-0, exception_level = e1-2]C:\edt\qspi_boot\u-boot.elf
  [offset = 0x1E40000, destination_cpu = a53-0]C:\edt\qspi_boot\image.ub
}
```

The Vitis IDE calls the following Bootgen command to generate the `qspi_BOOT.bin` image for this configuration.

```
bootgen -image qspi_boot.bif -arch zynqmp -o C:\edt\qspi_BOOT.bin
```

**Note:** In this boot sequence, the First Stage Boot Loader (FSBL) loads PMU firmware. This is because the PMU firmware was added as a **datafile** partition type. Ideally, the Boot ROM code can load the PMU Firmware for PMU as witnessed in the earlier section. For more details on PMU Firmware, refer to the “Platform Management” chapter in the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [Ref 6].

## Running the Image in QSPI Boot Mode on ZCU102 Board

To test the image in this example, you will load the Boot image (`qspi_BOOT.bin`) onto QSPI on the ZCU102 board using the Program Flash utility in the Vitis IDE. Alternately, you can use the XSDB debugger.

1. In the Vitis IDE, select **Xilinx > Program Flash**.
2. In the Program Flash wizard, browse to and select the `qspi_BOOT.bin` image file that was created as a part of this example.
3. Select **qspi-x8-dual\_parallel** as the Flash type.
4. Set the **Offset** as **0** and select the FSBL ELF file (`fsbl_a53.elf`).
5. Ensure that a USB cable is connected between the USB-JTAG connector on ZCU102 target and the USB port on the Host machine using the following steps.
  - a. Set the **SW6 Boot mode switch** as shown in the following figure.
  - b. Turn on the board.

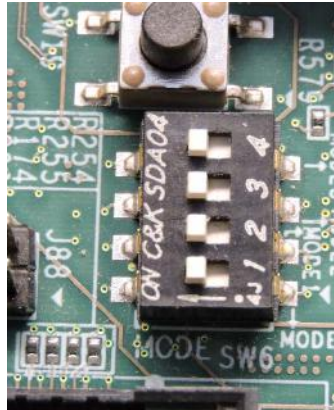


Figure 5-18: SW6 Switch Settings for JTAG Boot Mode

6. Click **Program** to start the process of programming the QSPI Flash with the `qspi_BOOT.bin` image.

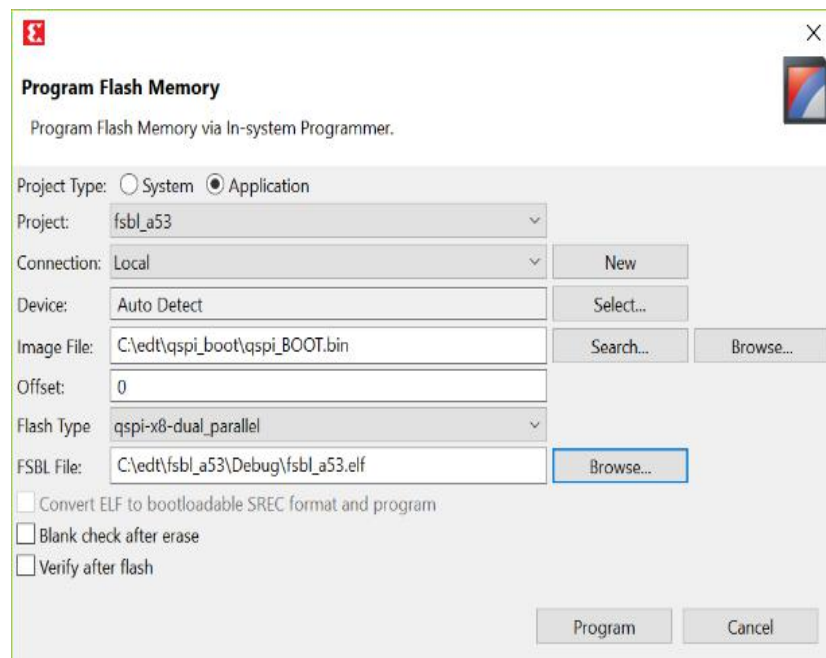
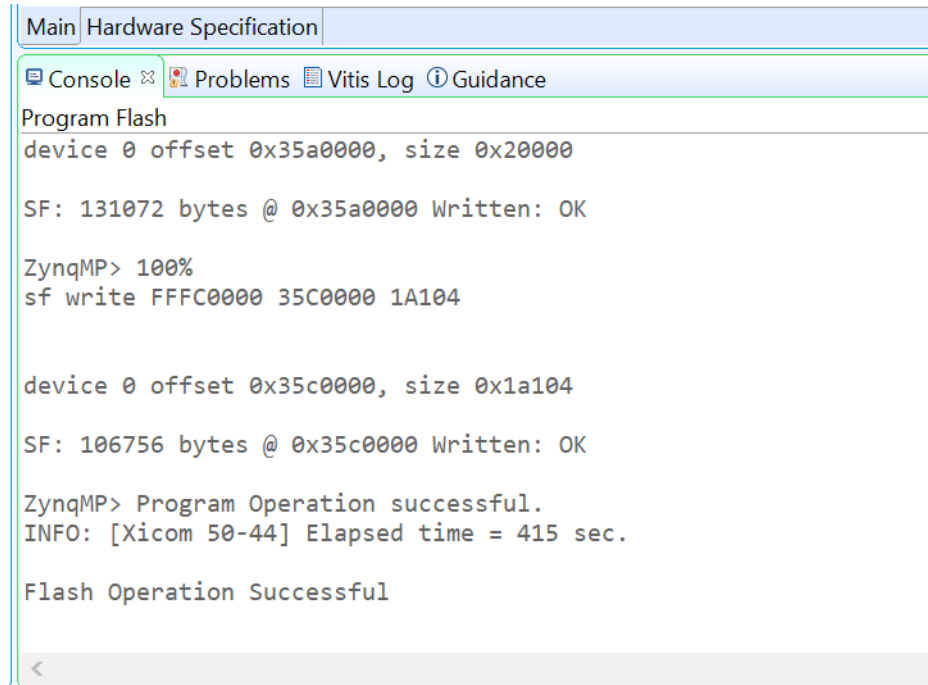


Figure 5-19: Program Flash Memory Dialog Box

Wait until you see the message “Flash Operation Successful” in the console, as shown in the following image.



The screenshot shows the Vitis IDE interface with the 'Console' tab selected. The console output displays the following messages:

```
Program Flash
device 0 offset 0x35a0000, size 0x20000

SF: 131072 bytes @ 0x35a0000 Written: OK

ZynqMP> 100%
sf write FFFC0000 35C0000 1A104

device 0 offset 0x35c0000, size 0x1a104

SF: 106756 bytes @ 0x35c0000 Written: OK

ZynqMP> Program Operation successful.
INFO: [Xicom 50-44] Elapsed time = 415 sec.

Flash Operation Successful
```

Figure 5-20: Vitis IDE Console Program Flash Messages

### Set Up the ZCU102 Board

1. Connect Board USB-UART on Board to Host machine. Connect the Micro USB cable into the ZCU102 Board Micro USB port **J83**, and the other end into an open USB port on the host Machine.
2. Configure the Board to Boot in QSPI-Boot mode by switching **SW6** as shown in following figure.

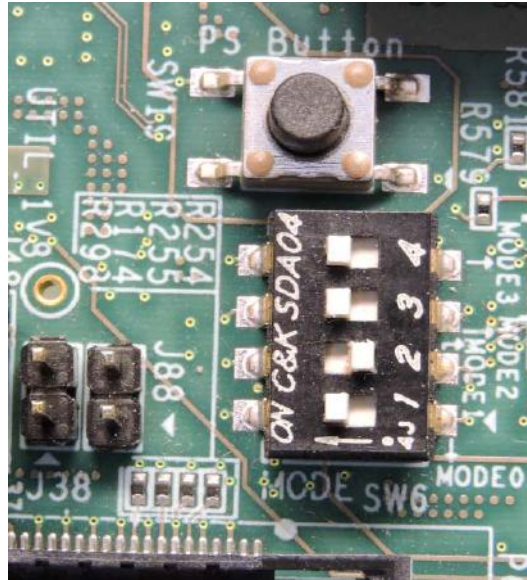


Figure 5-21: SW6 Switch Settings for QSPI Boot Mode

3. Connect 12V Power to the ZCU102 6-Pin Molex connector.
4. Start a terminal session, using Tera Term or Mini com, depending on the host machine being used, and the COM port and baud rate as shown in Figure 5-22.
5. For port settings, verify the COM port in the device manager. There are four USB UART interfaces exposed by the ZCU102.
6. Select the COM port associated with the interface with the lowest number. In this case, for UART-0, select the COM port with interface-0.
7. Similarly, for UART-1, select COM port with interface-1.

Remember, R5 BSP has been configured to use UART-1, so R5 application messages will appear on the COM port with UART-1 terminal.

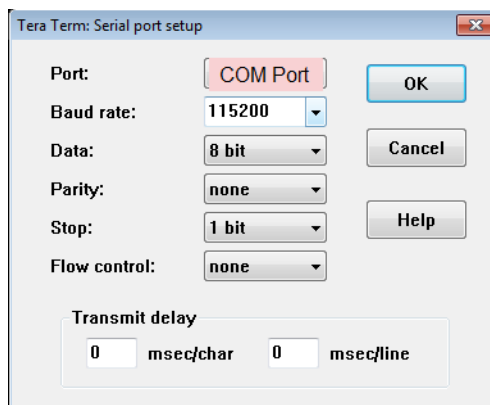


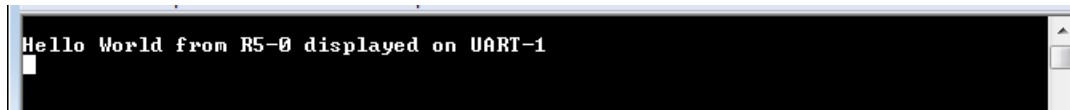
Figure 5-22: COM Port Settings for UART-1 Terminal

8. Turn on the ZCU102 Board using SW1.

At this point, you can see initial Boot sequence messages on your Terminal Screen representing UART-0.

You can see that the terminal screen configured for UART-1 also prints a message. This is the print message from the R-5 bare-metal Application running on RPU, configured to use UART-1 interface. This application is loaded by the FSBL onto RPU.

The bare-metal application has been modified to include the UART interrupt example. This application now waits in the WFI state until a user input is encountered from Keyboard in UART-1 terminal.



*Figure 5-23: Hello World Displayed on UART-1 From R5-0*

Meanwhile, the boot sequence continues on APU and the images loaded can be understood from the messages appearing on the UART-0 terminal. The messages are highlighted in the following figure.

```

Xilinx Zynq MP First Stage Boot Loader
Release 2019.2 Aug 27 2019 - 16:10:19
Reset Mode : System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU9EG
PMC UADJ Configuration Successful
Board Configuration successful
Processor Initialization Done
===== In Stage 2 =====
QSPI 32 bit Boot Mode
QSPI is in Dual Parallel connection
QSPI is using 4 bit bus
FlashID=0x20 0xBB 0x20 QSPI Boot Mode
MICRON 512M Bits
Multiboot Reg : 0x0
QSPI Reading Src 0x0, Dest FFFF1C40, Length EC0
Image Header Table Offset 0x8C0
QSPI Reading Src 0x8C0, Dest FFFE0088, Length 40
*****Image Header Table Details*****
Boot Gen Ver: 0x1020000
No of Partitions: 0x9
Partition Header Address: 0x440
Partition Present Device: 0x0
QSPI Reading Src 0x1100, Dest FFFE00C8, Length 40
.QSPI Reading Src 0x1140, Dest FFFE0108, Length 40
.QSPI Reading Src 0x1180, Dest FFFE0148, Length 40
.QSPI Reading Src 0x11C0, Dest FFFE0188, Length 40
.QSPI Reading Src 0x1200, Dest FFFE01C8, Length 40
.QSPI Reading Src 0x1240, Dest FFFE0208, Length 40
.QSPI Reading Src 0x1280, Dest FFFE0248, Length 40
.QSPI Reading Src 0x12C0, Dest FFFE0288, Length 40
.QSPI Reading Src 0x1300, Dest FFFE02C8, Length 40
Initialization Success
===== In Stage 3, Partition No:1 =====
UnEncrypted data Length: 0x5C57
Data word offset: 0x5C57
Total Data word length: 0x5C57
Destination Load Address: 0xFFDC0000
Execution Address: 0xFFDCFF58
Data word offset: 0x8F00
Partition Attributes: 0x83E
QSPI Reading Src 0x23C00, Dest FFDC0000, Length 1715C
Partition 1 Load Success
===== In Stage 3, Partition No:2 =====
UnEncrypted data Length: 0x249
Data word offset: 0x249
Total Data word length: 0x249
Destination Load Address: 0xFFDDB188
Execution Address: 0x0
Data word offset: 0xEB60
Partition Attributes: 0x83E
QSPI Reading Src 0x3AD80, Dest FFDDDB188, Length 924
Partition 2 Load Success
===== In Stage 3, Partition No:3 =====
UnEncrypted data Length: 0x100
Data word offset: 0x100
Total Data word length: 0x100
Destination Load Address: 0xFFDDF6E0
Execution Address: 0x0
Data word offset: 0xEDB0
Partition Attributes: 0x83E
QSPI Reading Src 0x3B6C0, Dest FFDDF6E0, Length 400
.PMU Firmware 2019.2 Aug 27 2019 14:56:56
PMU_ROM Version: xphr-v8.1.0-0

```

Figure 5-24: Messages Appearing on UART-0 Terminal

The U-Boot then loads Linux Kernel and other images on Arm Cortex-A53 APU in SMP mode. The terminal messages indicate when U-Boot loads Kernel image and the kernel start up to getting a user interface prompt in Linux Kernel. The Kernel loading and starting sequence can be seen in following figure.

```
Erase Group Size: 512 Bytes
24748260 bytes read in 1658 ms (14.2 MiB/s)
## Loading kernel from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'kernel@1' kernel subimage
  Description: Linux kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x10000108
  Data Size: 18082304 Bytes = 17.2 MiB
  Architecture: AArch64
  OS: Linux
  Load Address: 0x00080000
  Entry Point: 0x00080000
  Hash algo: sha1
  Hash value: 6777fff165c8ff2e473887685f9d1a31b9552167
Verifying Hash Integrity ... sha1+ OK
## Loading ramdisk from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'ramdisk@1' ramdisk subimage
  Description: petalinux-user-image
  Type: RAMDisk Image
  Compression: gzip compressed
  Data Start: 0x11148e64
  Data Size: 6622447 Bytes = 6.3 MiB
  Architecture: AArch64
  OS: Linux
  Load Address: unavailable
  Entry Point: unavailable
  Hash algo: sha1
  Hash value: 48abe1bb666e13715f2c2b61280f784f88eb7f2a
Verifying Hash Integrity ... sha1+ OK
## Loading fdt from FIT Image at 10000000 ...
Using 'conf@system-top.dtb' configuration
Trying 'fdt@system-top.dtb' fdt subimage
  Description: Flattened Device Tree blob
  Type: Flat Device Tree
  Compression: uncompressed
  Data Start: 0x1113ec0c
  Data Size: 41361 Bytes = 40.4 KiB
  Architecture: AArch64
  Hash algo: sha1
  Hash value: c024eecd7345889f0eb4961600333183b202dfe6
Verifying Hash Integrity ... sha1+ OK
Booting using the fdt blob at 0x1113ec0c
Loading Kernel Image ... OK
Loading Ramdisk to 789af000, end 78fffc00 ... OK
Loading Device Tree to 000000007ff2000, end 000000007fff190 ... OK

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x000000000 [0x410fd034]
[ 0.000000] Linux version 4.19.0-xilinx-v2019.2 (oe-user@oe-host) (gcc version 8.2.0)
[ 0.000000] Machine model: ZynqMP ZCU102 Rev1.0
[ 0.000000] earlycon: cdns0 at MMIO 0x00000000ff00000 (options '115200n8')
[ 0.000000] bootconsole [cdns0] enabled
[ 0.000000] efi: Getting EFI parameters from FDT:
[ 0.000000] efi: UEFI not found.
```

Figure 5-25: Kernel Loading and Starting Sequence

9. Wait until Linux loads on the Board.

## Boot Sequence for QSPI-Boot Mode Using JTAG

Zynq UltraScale+ MPSoC supports many ways to load the boot image. One way is using the JTAG interface. This example XSC T session demonstrates how to download a BOOT image file (`qspi_BOOT.bin`) in QSPI using the XSDB debugger. After the QSPI is loaded, the `qspi_BOOT.bin` image executes in the same way as QSPI Boot mode in Zynq UltraScale+. You can use the same XSC T session or the System Debugger for debugging similar Boot flows.

The following sections demonstrate the basic steps involved in this Boot mode.


### Setting Up the Target

1. Connect a USB cable between the USB-JTAG J2 connector on the target and the USB port on the host machine.
2. Set the board to JTAG Boot mode by setting the **SW6** switch, as shown in the following figure.



Figure 5-26: SW6 Switch Settings for JTAG Boot Mode

3. Power on the Board using switch **SW1**.

Open the XSC T Console in the Vitis IDE by clicking the XSC T button . Alternatively, you can also open the XSC T console by selecting **Xilinx > XSC T Console**.

4. In the XSC T console, connect to the target over JTAG using the `connect` command:

```
xsc t% connect
```

The `connect` command returns the channel ID of the connection.

- The `targets` command lists the available targets and allows you to select a target using its ID.

The targets are assigned IDs as they are discovered on the JTAG chain, so the IDs can change from session to session.

**Note:** For non-interactive usage such as scripting, you can use the `-filter` option to select a target instead of selecting the target using its ID.

```
xsct% targets
```

The targets are listed as shown in the following figure.

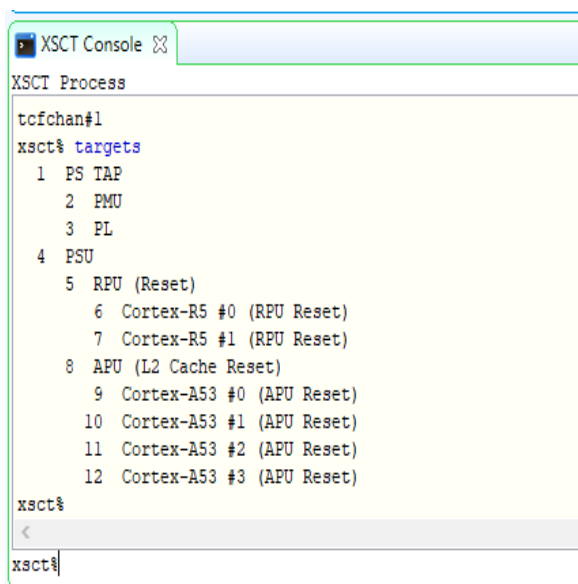


Figure 5-27: XSC Targets

## Load U-Boot Using XSC/XSDB

- Download the `u-boot` application on Cortex-A53 #0 using the following commands:

By default JTAG Security gates are enabled. Disable the security gates for DAP, PL TAP and PMU (this will make PMU MB target visible to Debugger).

```
xsct% targets -set -filter {name =~ "PSU"}
xsct% mwr 0xffca0038 0x1ff
xsct% targets
```

Verify if the PMU MB target is listed under the PMU device. Now, load and run PMUFW

```
xsct% targets -set -filter {name =~ "MicroBlaze PMU"}
xsct% dow {C:\edt\pmu_fw\Debug\pmu_fw.elf}
xsct% con
```

Now, reset APU Cortex-A53 Core 0 to load and run FSBL

```
xsct% targets -set -filter {name =~ "Cortex-A53 #0"}
xsct% rst -processor
```



**TIP:** `rst -processor` clears the reset on an individual processor core.

This step is important, because when Zynq UltraScale+ boots up in JTAG bootmode, all the APU and RPU cores are held in reset. You must clear resets on each core before performing debugging on these cores. You can use the `rst` command in XSCCT to clear the resets.

**Note:** `rst -cores` clears resets on all the processor cores in the group (such as APU or RPU) of which the current target is a child. For example, when A53 #0 is the current target, `rst -cores` clears resets on all the A53 cores in APU.

Load and run FSBL

```
xsct% dow {C:\edt\fsbl_a53\Debug\fsbl_a53.elf}
xsct% con
```

Verify FSBL messages on Serial Terminal and stop FSBL after couple of seconds

```
xsct% stop
```

Load and Run ATF

```
xsct% dow {C:\edt\design_files\b131.elf}
xsct% con
xsct% stop
```

2. Configure a serial terminal (Tera Term, Mini com, or the Serial Terminal interface for UART-0 USB-serial connection).
3. For serial terminal settings, see [Figure 5-22](#).

```
xsct% dow {C:\edt_zcu102\u-boot.elf}
Downloading Program -- C:/edt_zcu102/u-boot.elf
    section, .data: 0x08000000 - 0x0806c70f
100%  0MB  0.1MB/s  00:03
Setting PC to Program Start Address 0x08000000
Successfully downloaded C:/edt_zcu102/u-boot.elf
xsct%
```

Figure 5-28: Verify the Image on the ZCU102 Board

4. Load and run U-Boot

```
xsct% dow {C:\edt\design_files\sd_boot\u-boot.elf}
```

5. Run U-Boot, using the `con` command in XSDB.

```
xsct% con
```

6. In the target serial terminal, press any key to stop the U-Boot auto boot.



7. Stop the core using the `stop` command in XSDB.

```
xsct% stop
```

## Load Boot.bin in DDR Using XSDB

1. Download the Boot.bin binary into DDR on ZCU102. Use the same Boot.bin created for QSPI boot mode.

```
xsct% dow -data {C:\edt\qspi_boot\qspi_BOOT.bin} 0x2000000
```

2. Now continue the U-Boot again, using the `con` command in XSDB.

```
xsct% con
```

## Load the Boot.bin Image in QSPI Using U-Boot

1. Execute the following commands in the U-Boot console on the target terminal. These commands erase QSPI and then write the Boot.bin image from DDR to QSPI.

```
ZynqMP> sf probe 0 0 0  
ZynqMP> sf erase 0 0x4000000  
ZynqMP> sf write 0x2000000 0 0x4000000
```

2. After successfully writing the image to QSPI, turn off the board and set up the ZCU102 board as described in [Set Up the ZCU102 Board, page 89](#).

You can see Linux loading on the UART-0 terminal and the R5 application executing in the UART-1 terminal.

This chapter focused mostly on system boot and different components related to system boot. In the next chapter, you will focus on applications, Linux and Standalone (bare-metal) applications which will make use of PS peripherals, PL IPs, and processing power of APU Cores and RPU cores.

---

## Boot Sequence for USB Boot Mode

Zynq UltraScale+ MPSoC also supports USB Slave Boot Mode. This is using the USB DFU Device Firmware Upgrade (DFU) Device Class Specification of USB. Using a standard update utility such as [OpenMoko's DFU-Util](#), you will be able to load the newly created image on Zynq UltraScale+ via the USB Port. The following steps list the required configuration to load Boot images using this Boot mode. The DFU Utility is also shipped with the Vitis unified software platform and PetaLinux.

## Configure FSBL to Enable USB Boot Mode

There are few changes required in FSBL to enable USB Boot Mode. USB boot mode support increases the footprint of FSBL (by approximately 10 KB). Since it is intended mostly during

initial development phase, its support is disabled by default to conserve OCM space. In this section, you will modify the FSBL to enable the USB Boot Mode. Considering the FSBL project is used extensively throughout this tutorial, we will not modify the existing FSBL project. Instead, this section will make use of new FSBL project.

### Create First Stage Boot Loader for Arm Cortex-A53-Based APU

1. In the Vitis IDE, select **File > New > Application Project** to open the New Project wizard.
2. Use the information in the table below to make your selections in the wizard.

Table 5-1: Wizard Properties and Commands

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	fsbl_usb_boot
	Use Default Location	Select this option
	System Project	Create New
	Platform	edt_zcu102_wrapper
	Domain	standalone on psu_cortexa53_0
	Language	C
	CPU	psu_cortexa53_0
	OS	Standalone
Templates	Available Templates	Zynq MP FSBL

3. Click **Finish**.
4. In the Project Explorer tab, expand the fsbl\_usb\_boot project and open **xfsbl\_config.h** from:

```
fsbl_usb_boot > src > xfsbl_config.h
```

5. In **xfsbl\_config.h** change or set following settings:

```
#define FSBL_QSPI_EXCLUDE_VAL (1U)
#define FSBL_SD_EXCLUDE_VAL (1U)
#define FSBL_USB_EXCLUDE_VAL (0U)
```

6. Use **CTRL + S** to save these changes.
7. Build FSBL (fsbl\_usb\_boot).

## Creating Boot Images for USB Boot

In this section, you will create the Boot Images to be loaded, via USB using DFU utility. Device Firmware Upgrade (DFU) is intended to download and upload firmware to/from devices connected over USB. In this boot mode, the Boot loader (FSBL) and the PMUFW which are loaded by Boot ROM are copied to Zynq Ultrascale+ On Chip Memory (OCM) from Host Machine USB port using DFU Utility. The size of OCM (256 KB) limits the size of

boot image downloaded by BootROM in USB boot mode. Considering this, and subject to size requirement being met, only FSBL and PMUFW are stitched into the first `Boot.bin`, which is copied to OCM. Rest of the Boot partitions will be stitched in another Boot image and copied to DDR to be loaded by the FSBL which is already loaded and running at this stage. Follow the below steps to create Boot images for this boot mode.

1. In the Vitis IDE, select **Xilinx > Create Boot Image**.
2. Select `fsbl_usb_boot.elf` and `pmu_fw.elf` partitions and set them as shown in the following figure.

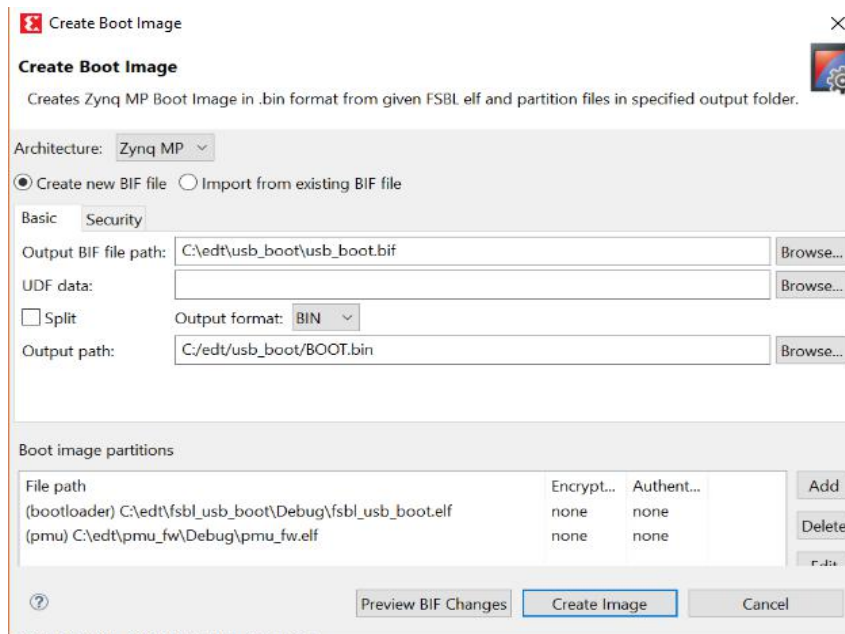


Figure 5-29: Create Boot Image for USB Boot

3. Ensure that PMU Partition is set to be loaded by BootROM.
4. Click on Create Image to generate `BOOT.bin`.

### Modifying PetaLinux U-Boot

Modify PetaLinux U-Boot so that it can load the `image.ub` image. The Device tree needs to be modified to set USB in the Peripheral mode. The default PetaLinux configuration is set for the USB in Host mode.

For this, follow the below steps to modify `system-user.dtsi` in the Petalinux Project `<PetaLinux-project>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi`.

1. Add the following to `system-user.dtsi`, so that it looks like:

```
/include/ "system-conf.dtsi"
/ {
    gpio-keys {
        sw19 {
            status = "disabled";
        };
    };

    leds {
        heartbeat_led {
            status = "disabled";
        };
    };

    &uart1
    {
        status = "disabled";
    };

    &dwc3_0 {
        dr_mode = "peripheral";
        maximum-speed = "super-speed";
    };
};
```

The modified `system-user.dtsi` file can be found in `<Design Files>/usb_boot` released with tutorial.

2. Build Petalinux with the following changes.

```
$ petalinux-build
```

The following steps describe how to create a `usb_boot.bin` comprising rest of the partitions.

**Note:** Copy the newly generated U-Boot to `C:\edt\usb_boot\`. The `u-boot.elf` is also available in [Design Files for This Tutorial](#).

1. In the Vitis IDE, select **Xilinx > Create Boot Image**.
2. Select FSBL and rest of the partitions and set them as shown in the following figure. For this you can also choose to import the BIF File from SD Boot Sequence.

**Note:** Ensure that you have set the correct exception levels for ATF (EL-3, Trustzone) and U-Boot (EL-2) partitions. These settings can be ignored for other partitions.

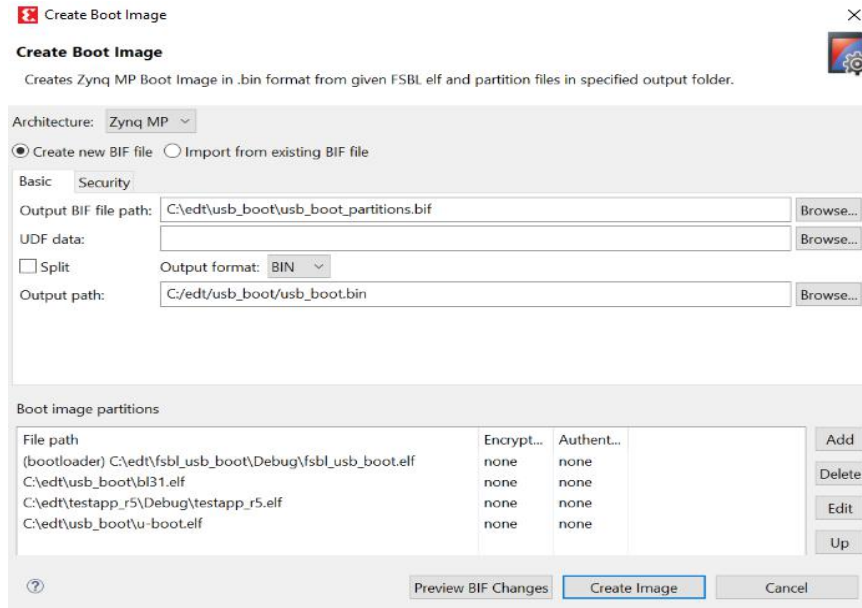


Figure 5-30: Create Boot Image with Rest of the Partitions

3. Notice that PMUFW partition is not required in this image, as it will be loaded by the Boot ROM before this image (`usb_boot.bin`) is loaded.
4. Click on Create Image to generate `usb_boot.bin`.

**Note:** In addition to `BOOT.bin` and `usb_boot.bin`, the Linux image like `image.ub` is required to boot till Linux. This `image.ub` will be loaded by DFU utility separately.

## Boot using USB Boot

In this section you will load the boot images on ZCU102 target using DFU utility. Before you start, set the board connections as shown below:

1. Set ZCU102 for USB Boot mode by setting SW6 (1-OFF, 2-OFF, 3-OFF, and 4-ON), as shown below:

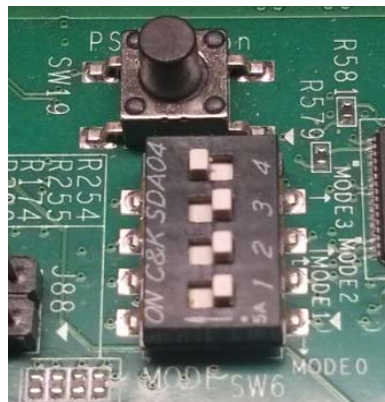


Figure 5-31: SW6 Settings for USB Boot Mode

2. Connect a USB 3.0 Cable to J96 USB 3 ULPI Connector, and the other end of the Cable to USB port on Host Machine.
3. Connect a USB Micro cable between USB-UART port on Board (J83) and Host Machine.
4. Start a terminal session, using Tera Term or Minicom depending on the host machine being used, as well as the COM port and baud rate for your system, as shown in [Figure 5-31](#).
5. Power ON the board.

The following steps will load the boot images via USB using DFU utility, which can be found in `VITIS\2019.1\tps\lnx64\dfu-util-0.9`.

Alternatively you can also install DFU utility on Linux using Package Manager supported by Linux Distribution being used.

### **Boot Commands for Linux Host Machine**

1. Check if the DFU can detect the USB target.

```
$ sudo dfu-util -l
```

The USB device should be enumerated with VendorId : ProductId which is 03fd:0050. You should see something like below:

```
Found DFU: [03fd:0050] ver=0100, devnum=30, cfg=1, intf=0, alt=0, name="Xilinx DFU Downloader", serial="2A49876D9CC1AA4"
```

**Note:** If you do not see the 'Found DFU' message, verify the connection and retry.

2. Now download the `BOOT.bin` that was created in [Creating Boot Images for USB Boot](#).

```
$ sudo dfu-util -d 03fd:0050 -D <USB_Boot_Image_Path>/Boot.bin
```

Verify from Serial Terminal if the FSBL is loaded successfully.

3. Now download the `usb_boot.bin`. Before this start another terminal session for UART-1 serial console.

```
$ sudo dfu-util -d 03fd:0050 -D <USB_Boot_Image_Path>/usb_boot.bin
```

Check UART 0 terminal and wait until U-Boot loads.

4. On U-Boot prompt, type **Enter** to terminate autoboot. Verify from the UART1 console that the R5 application is also loaded successfully.
5. In U-Boot console start DFU\_RAM to enable downloading Linux Images

```
U-boot> run dfu_ram
```

6. Download Linux Image `Image.ub` using following Command from Host Machine Terminal:

```
$ sudo dfu-util -d 03fd:0300 -D <PetaLinux_project>/images/linux/image.ub -a 0
```

7. Now execute CTRL+C on U-Boot console to stop dfu\_ram.
8. Run bootm command from U-Boot Console.

```
U-boot> bootm
```

9. Verify that Linux loads successfully on the target

**Note:** In this example, `image.ub` is copied to DDR location based on `#define DFU_ALT_INFO_RAM` settings in U-Boot configuration. The same can be modified to copy other image files to DDR location. Then, if required, these images can be copied to QSPI using U-Boot commands listed in [Boot Sequence for QSPI-Boot Mode Using JTAG](#).

### **Boot Commands for Windows Host Machine**

1. In the Vitis IDE, Select **Xilinx > Launch Shell**.
2. In Shell, use Check if the DFU can detect the USB target

```
> dfu-util.exe -l
```

**Note:** `dfu-util.exe` can be found in  
`<VITIS_Installation_path>\VITIS\2019.2\tps\Win64\dfu-util-0.9\dfu-util.exe`

3. The USB device should be enumerated with VendorId : ProductId which is 03fd:0050

**Note:** If you do not see the message starting with "Found DFU", download and install "zadig" software. Open the software and click on options and select "List all devices". Select device "Xilinx Dfu Downloader" and click on Install driver tab.

4. Now download the Boot.bin that was created in [Creating Boot Images for USB Boot](#).

```
$ dfu-util.exe -d 03fd:0050 -D BOOT.bin
```

5. Verify from Serial Terminal (UART 0) that FSBL is loaded successfully.
6. Now download the `usb_boot.bin`. Before this start another terminal session for UART-1 serial console.

```
$ dfu-util.exe -d 03fd:0050 -D usb_boot.bin
```

7. On U-Boot prompt type Enter to terminate auto-boot. Verify from UART1 console that the R5 application is also loaded successfully.

**Note:** At this point, use Zadig utility to install drivers for "Usb download gadget" with device ID 03fd:0300. Without this, zadig software does not show "Xilinx DFU Downloader" after booting U-Boot on target.

8. In U-Boot console start DFU\_RAM to enable downloading Linux Images

```
U-boot> run dfu_ram
```

9. Download Linux Image `image.ub` using following Command from Host Machine Terminal

```
$ dfu-util.exe -d 03fd:0300 -D image.ub -a 0
```

10. Run bootm command from U-Boot Console.

```
U-boot> bootm
```

11. Verify that Linux loads successfully on the target.

---

## Secure Boot Sequence

The secure boot functionality in Zynq UltraScale+ MPSoC allows you to support confidentiality, integrity, and authentication of partitions. Secure boot is accomplished by combining the Hardware Root of Trust (HROT) capabilities of the Zynq UltraScale+ device with the option of encrypting all boot partitions. The HROT is based on the RSA-4096 asymmetric algorithm in conjunction with SHA-3/384, which is hardware accelerated, or SHA-2/256, implemented as software. Confidentiality is provided using 256 bit Advanced Encryption Standard - Galois Counter Mode (AES-GCM). This section focuses on how to use and implement the following:

- Hardware Root of Trust with Key Revocation
- Partition Encryption with Differential Power Analysis (DPA) Countermeasures
- Black Key Storage using the Physically Unclonable Function (PUF)

The section [Secure Boot System Design Decisions](#) outlines high level secure boot decisions which should be made early in design development. The [Hardware Root of Trust](#) section discusses the use of a Root of Trust (RoT) in boot. The [Boot Image Confidentiality and DPA](#) section discusses methods to use AES encryption.

The [Boot Image Confidentiality and DPA](#) section discusses the use of the operational key and key rolling techniques as countermeasures to a DPA attack. Changing the AES key reduces the exposure of both the key and the data protected by the key.

A red key is a key in unencrypted format. The [Black Key Storage](#) section provides a method for storing the AES key in encrypted, or black format. Black key store uses the physically unclonable function (PUF) as a Key Encryption Key (KEK).

The [Practical Methods in Secure Boot](#) section provides steps to develop and test systems that use AES encryption and RSA authentication.

## Secure Boot System Design Decisions

The following are device level decisions affecting Secure Boot:

- Boot Mode
- AES Key Storage Location
- AES Storage State (encrypted or unencrypted)
- Encryption and Authentication requirements



- Key Provisioning

The boot modes which support secure boot are Quad Serial Peripheral Interface (QSPI), SD, eMMC, and NAND. The AES key is stored in either eFUSES (encrypted or unencrypted), Battery Backed Random Access Memory (BBRAM) (unencrypted only), or in external NVM (encrypted only).

In Zynq UltraScale+ MPSoC, partitions can be encrypted and/or authenticated on a partition basis. Xilinx generally recommends that all partitions be RSA authenticated. Partitions that are open source (U-Boot, Linux), or do not contain any proprietary or confidential information, typically do not need to be encrypted. In systems in which there are multiple sources/suppliers of sensitive data and/or proprietary IP, encrypting the partitions using unique keys may be important.

DPA resistance requirements are dictated by whether the adversary has physical access to the device.

[Table 5-2](#) can be a good reference while deciding on features required to meet a specific secure system requirement. Next sections will discuss the features in more detail.

**Table 5-2: System Level Security Requirements**

System Consideration/ Requirement	Zynq UltraScale+ Feature
Ensure that only the users SW and HW runs on the device	HWROT
Guarantee that the users SW and HW are not modified	HWROT
Ensure that an adversary cannot clone or reverse engineer SW/HW	Boot Image Confidentiality
Protect sensitive data and proprietary Intellectual Property (IP)	Boot Image Confidentiality
Ensure that Private Key (AES key) is protected against side channel attacks	DPA Protections
Private/Secret keys (AES key) is stored encrypted at rest	Black Key Storage

## Hardware Root of Trust

Root of trusts are security primitives for storage (RTS), integrity (RTI), verification (RTV), measurement (RTM), and reporting (RTR). RoT consists of hardware, firmware, and software. The HROT has advantages over software RoTs because the HROT is immutable, has a smaller attack surface, and the behavior is more reliable.

The HROT is based on the CSU, eFUSES, BBRAM, and isolation elements. The HROT is responsible for validating that the operating environment and configuration have not been modified. The RoT acts as an anchor for boot, so an adversary can not insert malicious code before detection mechanisms start.

Firmware and software run on the HROT during boot. Zynq UltraScale provides immutable BootROM code, a first stage boot loader, device drivers, and the XILSKEY and XILSECURE libraries which run on the HROT. These provide a well-tested, proven in use API so that developers do not create security components from scratch with limited testing.

## Data Integrity

Data integrity is the absence of corruption of hardware, firmware and software. Data integrity functions verify that an adversary has not tampered with the configuration and operating environment.

Zynq UltraScale+ verifies the integrity of partition(s) using both symmetric key (AES-GCM) and asymmetric key (RSA) authentication. RSA uses a private/public key pair. The fielded embedded system only has the public key. Theft of the public key is of limited value since it is not possible, with current technology, to derive the private key from the public key. Encrypted partitions are also authenticated using the Galois Counter Mode (GCM) mode of AES.

In the secure boot flow, partitions are first authenticated and then decrypted if necessary.

## Authentication

Figure 5-32 shows RSA signing and verification of partitions. From a secure facility, the Bootgen tool signs partitions, using the private key. In the device, the ROM verifies the FSBL and either the FSBL or U-Boot verifies the subsequent partitions, using the public key. Primary and secondary private/public key pairs are used. The function of the primary private/public key pair is to authenticate the secondary private/public key pair. The function of the secondary key is to sign/verify partitions.

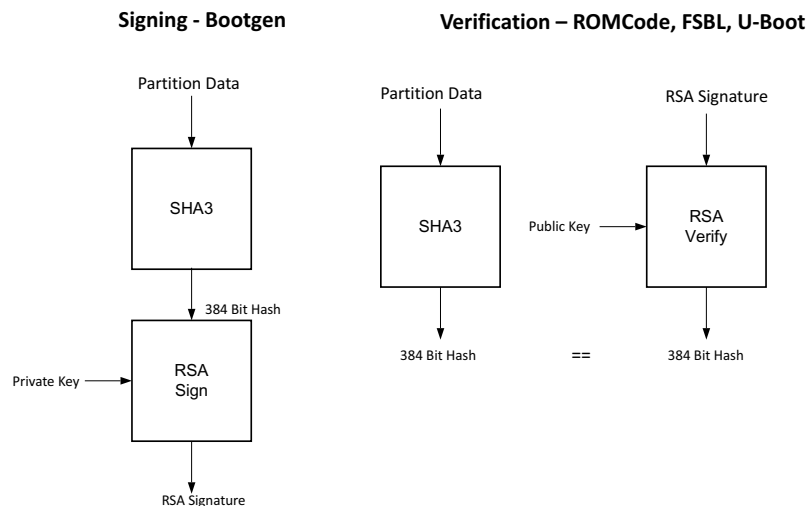


Figure 5-32: Zynq UltraScale+ RSA Authentication

To sign a partition, Bootgen first calculates the SHA3 of the partition data. The 384 bit hash is then RSA signed using the private key. The resulting RSA signature is placed in the authentication certificate. In the image, each signed partition has partition data followed by an authentication certificate which includes the RSA signature.

Verification of the FSBL is handled by the CSU ROM code. To verify the subsequent partitions, the FSBL or U-Boot uses the XILSECURE library.

There is a debug mode for authentication called bootheader authentication. In this mode of authentication, the CSU ROM code does not check the primary public key digests, the session key ID or the key revocation bits stored in the device eFUSES. Therefore, this mode is not secure. However, it is useful for testing and debugging as it does not require programming of eFUSES. This tutorial uses this mode. However, fielded systems should not use boot header authentication. The example BIF file for a fully secured system is included at the end of this section.

### **Boot Image Confidentiality and DPA**

AES is used to ensure confidentiality of sensitive data and IP. Zynq UltraScale+ uses AES Galois Counter Mode (GCM). Zynq UltraScale+ uses a 256 AES bit key. The principle AES enhancements provided by Zynq UltraScale+ are increased resistance to Differential Power Analysis (DPA) attacks and the availability of AES encryption/decryption post boot.

Bootgen and FSBL software support AES encryption. Private keys are used in AES encryption, and AES encryption is done by Bootgen using the key files. The key files can be generated by Bootgen or OpenSSL. The use of the operational key limits the exposure of the device key. The use of the operational key in key rolling is discussed in the next section. To maintain Boot image confidentiality, Encrypted Boot images can be created using Bootgen. Software examples to program keys to BBRAM and eFUSE are also available in the Vitis IDE. One such example is discussed in [Practical Methods in Secure Boot](#).

## DPA Protections

Key rolling is used for DPA resistance. Key rolling and black key store can be used in the same design. In key rolling, software and bitstream is broken up into multiple data blocks, each encrypted with a unique AES key. The initial key is stored in BBRAM or eFUSE NVM. Keys for successive data blocks are encrypted in the previous data block. After the initial key, the key update register is used as the key source.

A 96 bit initialization vector is included in the NKY key file. The IV uses 96 bits to initialize AES counters. When key rolling is used, a 128 bit IV is provided in the bootheader. The 32 least significant bits define the block size of data to decrypt using the current key. The block sizes following the initial block defined in the IV are defined as attributes in the Bootgen Image Format (BIF) file.

An efficient method of key rolling uses the operational key. With the operational key, Bootgen creates an encrypted secure header with the user specified operational key and the first block IV. The AES key in eFUSE or BBRAM is used only to decrypt the 384 bit secure header with the 256 bit operational key. This limits the exposure of the device key to DPA attacks.

## Black Key Storage

The PUF enables storing the AES key in encrypted (black) format. The black key can be stored either in eFUSES or in the bootheader. When needed for decryption, the encrypted key in eFUSES or the bootheader is decrypted using the PUF generated key encrypting key (KEK).

There are two steps in using the PUF for black key storage. In the first, PUF registration software is used to generate PUF helper data and the PUF KEK. The PUF registration data allows the PUF to re-generate the identical key each time the PUF generates the KEK. For more details on the use of PUF registration software, see [PUF Registration - Boot Header Mode](#). For more information on PUF Registration - eFUSE mode, see *Programming BBRAM and eFUSES* (XAPP1319) [Ref 12].

The helper data and encrypted user key must both be stored in eFUSES if the PUF eFUSE mode is used, and in the bootheader if the PUF Bootheader mode is used. The procedure for the PUF bootheader mode is discussed in [Using PUF in Bootheader Mode](#). For the procedure to use PUF in eFUSE mode, see *Programming BBRAM and eFUSES* (XAPP1319) [Ref 12].

This tutorial uses PUF Bootheader Mode as it does not require programming of eFUSES, and is therefore useful for test and debug. However, the most common mode is PUF eFUSE mode, as the PUB Bootheader mode requires a unique run of bootgen for each and every device. The example BIF file for a fully secured system is included at the end of the [Secure Boot Sequence](#) section demonstrates the PUF eFUSE mode.

## Practical Methods in Secure Boot

This section outlines the steps to develop secure boot in a Zynq UltraScale+ system. Producing a secure embedded system is a two-step process. In the first phase, the cryptographic keys are generated and programmed into NVM. In the second phase, the secure system is developed and tested. Both steps use the Vitis IDE to create software projects, generate the image, and program the image. For the second phase, a test system can be as simple as `fsbl.elf` and `hello.elf` files. In this section, you will use the same images used in [Boot Sequence for SD-Boot](#), but this time the images will be assembled together, and have the secure attributes enabled as part of the secure boot sequence.

This section starts by showing how to generate AES and RSA keys. Following key generation, systems using the advanced AES and RSA methods are developed and tested. Keys generated in this section are also included in the [Design Files for This Tutorial](#), released with this tutorial.

The methods used to develop AES functionality are provided in the following sections:

- [Generating all of the AES keys](#)
- [Enabling Encryption Using Key Rolling](#)
- [Enable use of an Operational Key](#)
- AES key in eFUSE
- [Using the PUF](#)

The [Creating RSA Private/Public Key Pairs](#) section provides the steps to authenticate all partitions loaded at boot. This section also shows how to revoke keys.

A requirement in the development of a secure system is to add security attributes which are used in image generation. Bootgen generates a Boot Image Format (BIF) file. The BIF file is a text file. In its simplest form, the BIF is a list of partitions to be loaded at boot. Security attributes are added to the BIF to specify cryptographic functionality. In most cases, the Bootgen GUI (Create Boot Image wizard) is used to generate the BIF file. In some cases, adding security attributes requires editing the Bootgen generated BIF file. In Create Boot Image Wizard in the Vitis IDE, after the Security tab is selected, the Authentication and Encryption tabs are used to specify security attributes.

After implementing AES and RSA cryptography in secure boot, a Boot test is done. The system loads successfully and displays the FSBL messages on the terminal. These messages indicate the cryptographic operations performed on each partition. [Appendix A, Debugging Problems with Secure Boot](#) provides steps that are required to use, if the secure boot test fails.

### Sample Design Overview

The sample design demonstrates loading various types of images into the device. It includes loading a FSBL, PMU Firmware, U-Boot, Linux, RPU software and a PL configuration image. In this sample, all of these images are loaded by the FSBL which performs all authentication and decryption. This is not the only means of booting a system. However, it is the simple and secure method, as of 2019.1.

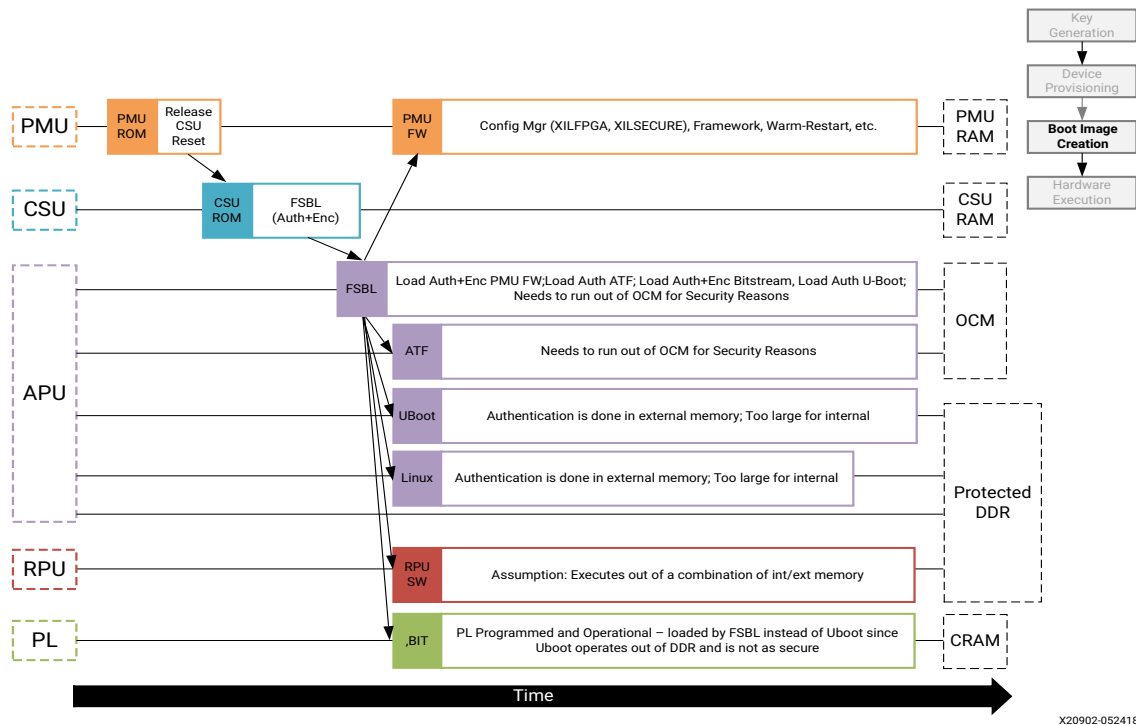


Figure 5-33: Sample Design Overview

Different sections within the boot image have different levels of security and are loaded into different locations. The following table explains the contents of the final boot image.

Table 5-3: Final Boot Image with Secure Attributes

Binary	RSA Authenticated	AES Encrypted	Exception Level	Loader
FSBL	Yes	Yes	EL3	CSU ROM
PMU Firmware	Yes	Yes	NA	FSBL
PL Bitstream	Yes	Yes	NA	FSBL
Arm Trusted Firmware (ATF)	Yes	No	EL3	FSBL
R5 Software	Yes	Yes	NA	FSBL
U-Boot	Yes	No	EL2	FSBL

Table 5-3: Final Boot Image with Secure Attributes (Cont'd)

Binary	RSA Authenticated	AES Encrypted	Exception Level	Loader
Linux	Yes	No	EL1	FSBL

**Notes:**

1. In a Secure boot sequence PMU image is loaded by FSBL. Using the BootROM/CSU to load the PMU firmware introduces a security weakness as the key/IV combination is used twice. First to decrypt the FSBL and then again to decrypt the PMU image. This is not allowed for the secure systems.
2. As of 2019.1, U-Boot does not perform a secure authenticated loading of Linux. So instead of U-Boot, FSBL loads the Linux images to memory address and then uses U-Boot to jump to that memory address.

This tutorial demonstrates assembling the binaries that are created using [Chapter 6, System Design Examples](#) in a boot image with all the security features enabled. This section also shows how PL bitstream can be added as a part of secure boot flow. Follow [Chapter 6, System Design Examples](#) till the section [Modifying the Build Settings](#) to create all the necessary files and then switch back.

Enabling the security features in boot image is done in two different methods. During the first method, the BIF file is manually created using a text editor and then using that BIF file to have Bootgen create keys. This enables you to identify the sections of the BIF file that are enabled which security features. The second method uses the Create Boot Image wizard in the Vitis IDE. It demonstrates the same set of security features. The second method reuses the keys from the first method for convenience.

### Generating Keys for Authentication

There are multiple methods of generating keys. These include, but are not limited to, using bootgen, customized key files, OpenSSL and Hardware Security Modules (HSMs). This tutorial covers methods using bootgen. The bootgen created files can be used as templates for creating files containing user-specified keys from the other key sources.

The creation of keys using bootgen commands requires the generation and modification of the BIF files. The key generation section of this tutorial creates these bif files "by hand" using a text editor. The next section, building your boot image demonstrates how to create these BIF files using the Bootgen GUI (create Boot Image Wizard).

### Creating RSA Private/Public Key Pairs

For this example, you will create the Primary and Secondary keys in the PEM format. The keys are generated using Bootgen command-line options. Alternately, you can create the keys using external tools like OpenSSL.

The following steps describe the process of creating the RSA Private/Public Key Pairs:

1. Launch the shell for the Vitis IDE.
2. Select **Xilinx > Launch Shell**.
3. Create a file named `key_generation.bif`.

**Note:** The `key_generation.bif` file will be used to create both the asymmetric keys in these steps and the symmetric keys in later steps.

```
the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [fsbl_config]a53_x64
  [bootloader]fsbl_a53.elf
  [destination_cpu = pmu]pmu_fw.elf
  [destination_device = pl]edt_zcu102_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone] bl31.elf
  [destination_cpu = r5-0]tmr_psled_r5.elf
  [destination_cpu = a53-0, exception_level = el-2]u-boot.elf
  [load = 0x1000000, destination_cpu = a53-0]image.ub
}
```

4. Save the `key_generation.bif` file in the `C:\edt\secure_boot_sd\keys` directory.
5. Copy all of the ELF, BIF and UB files built in [Chapter 6, System Design Examples](#) to `C:\edt\secure_boot_sd\keys` directory.
6. Navigate to the folder containing the BIF file.

```
cd C:\edt\secure_boot_sd\keys
```

7. Run the following command to generate the keys:

```
bootgen -p zu9eg -arch zynqmp -generate_keys auth pem -image key_generation.bif
```

8. Verify that the files `psk0.pem` and `ssk0.pem` are generated at the location specified in the BIF file (`C:\edt\secure_boot_sd\keys`).

### Generate SHA3 of Public Key in RSA Private/Public Key Pair

The following steps are required only for RSA Authentication with eFUSE mode, and can be skipped for RSA authentication with bootheader mode. The 384 bits from `sha3.txt` can be programmed to eFUSE for RSA Authentication with the eFUSE Mode. For more information, see *Programming BBRAM and eFUSES (XAPP1319)* [Ref 12].

1. Perform the steps from the prior section.
2. Now that the PEM files have been defined, add `authentication = rsa` attributes as shown below to `key_generation.bif`

```
the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [fsbl_config]a53_x64
  [bootloader, authentication = rsa]fsbl_a53.elf
  [destination_cpu = pmu, authentication = rsa]pmu_fw.elf
  [destination_device = pl, authentication = rsa]edt_zcu102_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
```



```
[destination_cpu = r5-0, authentication = rsa]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}
```

3. Use the bootgen command to calculate the hash of the PPK:

```
bootgen -p zcu9eg -arch zynqmp -efuseppkbits ppk0_digest.txt -image
key_generation.bif
```

4. Verify that the file `ppk0_digest.txt` is generated at the location specified (`c:\edt\secure_boot_sd\keys`).

### Additional RSA Private/Public Key Pairs

The steps in this section to generate Secondary RSA Private/Public key pair required for Key Revocation, which requires programming of eFUSE. For more information, see *Programming BBRAM and eFUSES* (XAPP1319) [Ref 12]. You can skip this section if you do not intend to use Key Revocation.

Repeat steps from [Creating RSA Private/Public Key Pairs](#) and [Generate SHA3 of Public Key in RSA Private/Public Key Pair](#) to generate the second RSA private/public key pair and generate the SHA3 of the second PPK.

1. Perform the steps from the prior section but with replacing `psk0.pem`, `ssk0.pem`, and `ppk0_digest.txt` with `psk1.pem`, `ssk1.pem` and `ppk1_digest.pem` respectively. Save this file as `key_generation_1.bif`. That `.bif` file will look like:

```
the_ROM_image:
{
  [pskfile]psk1.pem
  [sskfile]ssk1.pem
  [auth_params]spk_id = 1; ppk_select = 1
  [fsbl_config]a53_x64
  [bootloader]fsbl_a53.elf
  [destination_cpu = pmu]pmu_fw.elf
  [destination_device = pl]edt_zcu102_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone]bl31.elf
  [destination_cpu = r5-0]tmr_psled_r5.elf
  [destination_cpu = a53-0, exception_level = el-2]u-boot.elf
  [load = 0x1000000, destination_cpu = a53-0]image.ub
}
```

2. Run the bootgen command to create the RSA private/public key pairs.

```
bootgen -p zu9eg -arch zynqmp -generate_keys auth pem -image key_generation_1.bif
```

3. Add `authentication = rsa` attributes to the `key_generation_1.bif` file. The `.bif` file will look like:

```
the_ROM_image:
{
  [pskfile]psk1.pem
  [sskfile]ssk1.pem
  [auth_params]spk_id = 1; ppk_select = 1
  [fsbl_config]a53_x64
  [bootloader, authentication = rsa]fsbl_a53.elf
}
```

```

[destination_cpu = pmu, authentication = rsa]pmu_fw.elf
[destination_device = pl, authentication = rsa]edt_zcu102_wrapper.bit
[destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
[destination_cpu = r5-0, authentication = rsa]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}

```

4. Run the bootgen command to generate the hash of the primary RSA public key.

```

bootgen -p zcu9eg -arch zynqmp -efuseppkbits ppk1_digest.txt -image
key_generation_1.bif

```

5. Verify that the files `ppk1.pem`, `spk1.pem`, and `ppk1_digest.txt` are all generated at the location specified (`c:\edt\secure_boot\keys`).

### Enabling Boot Header Authentication

Boot header authentication is a mode of authentication that instructs the ROM to skip the checks of the eFUSE hashes for the PPKs, the revocation status of the PPKs and the Session IDs for the secondary keys. This mode is useful for testing and debugging as it does not require programming of eFUSES. This mode can be permanently disabled for a device by programming the RSA\_EN eFUSES which forces RSA Authentication with the eFUSE checks. Fielded systems should use the RSA\_EN eFUSE to force the eFUSE checks and disable Boot Header Authentication.

Add the `bh_auth_enable` attribute to the `[fsbl_config]` line so that the bif file appears as following:

```

the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [fsbl_config]a53_x64, bh_auth_enable
  [bootloader, authentication = rsa]fsbl_a53.elf
  [destination_cpu = pmu, authentication = rsa]pmu_fw.elf
  [destination_device = pl, authentication = rsa]edt_zcu102_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
  [destination_cpu = r5-0, authentication = rsa]tmr_psled_r5.elf
  [destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
  [load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}

```

### Generating Keys for Confidentiality

Image confidentiality is discussed in [Boot Image Confidentiality and DPA](#) section. In this section you will modify the `.bif` file from the authentication section by adding the attributes required to enable image confidentiality, using the AES-256-GCM encryption algorithm. At the end, a bootgen command will be used to create all of the required AES-256 keys.

## Using AES Encryption

1. Enable image confidentiality by specifying the key source for the initial encryption key (bbram\_red\_key for now) using the [keysrc\_encryption] bbram\_red\_key attribute
2. On several of the partitions enable confidentiality by adding the encryption = aes attribute of the partitions. Also specify a unique key file for each partition. Having a unique key file for each partition allows each partition to use a unique set of keys which increases security strength by not reusing keys and reducing the amount of information encrypted on any one key. The `key_generation.bif` file should now look as follows:

```
the_ROM_image:
{
[pskfile]psk0.pem
[sskfile]ssk0.pem
[auth_params]spk_id = 0; ppk_select = 0
[keysrc_encryption]bbram_red_key
[fsbl_config]a53_x64, bh_auth_enable
[bootloader, authentication = rsa, encryption = aes, aeskeyfile =
=fsbl_a53.nky]fsbl_a53.elf
[destination_cpu = pmu, authentication = rsa, encryption = aes, aeskeyfile =
pmu_fw.nky]pmu_fw.elf
[destination_device = pl, authentication = rsa, encryption = aes, aeskeyfile =
edt_zcu102_wrapper.nky]edt_zcu102_wrapper.bit
[destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
[destination_cpu = r5-0, authentication = rsa, encryption = aes, aeskeyfile =
tmr_psled_r5.nky]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}
```

## Enabling DPA Protections

This section provides the steps to use an operational key and key rolling effective countermeasures against the differential power analysis (DPA).

### Enable use of an Operational Key

Use of an operational key limits the amount of information encrypted using the device key. Enable use of the operational key by adding the `opt_key` attribute to the [fsbl\_config] line of the bif file. The `key_generation.bif` file should now look like as shown below:

```
the_ROM_image:
{
[pskfile]psk0.pem
[sskfile]ssk0.pem
[auth_params]spk_id = 0; ppk_select = 0
[keysrc_encryption]bbram_red_key
[fsbl_config]a53_x64, bh_auth_enable, opt_key
[bootloader, authentication = rsa, encryption = aes, aeskeyfile =
fsbl_a53.nky]fsbl_a53.elf
[destination_cpu = pmu, authentication = rsa, encryption = aes, aeskeyfile =
pmu_fw.nky]pmu_fw.elf
```

```

[destination_device = pl, authentication = rsa, encryption = aes, aeskeyfile =
edt_zcu102_wrapper.nky]edt_zcu102_wrapper.bit
[destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
[destination_cpu = r5-0, authentication = rsa, encryption = aes, aeskeyfile =
tmr_psled_r5.nky]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}
    
```

## Enabling Encryption Using Key Rolling

Use of key rolling limits the amount of information encrypted using any of the other keys. Key-rolling is enabled on a partition-by-partition basis using the `blocks` attribute in the `bif` file. The `blocks` attribute allows specifying the amount of information in bytes to encrypt with each key. For example, `blocks=4096,1024(3),512(*)` would use the first key for 4096 bytes, the 2nd through 4th keys for 1024 bytes and all remaining keys for 512 bytes. In this example, the `block` command will be used to limit the life of each key to 1728 bytes.

Enable use of the key rolling by adding the `blocks` attribute to each of the encrypted partitions. The `key_generation.bif` file should now look like.

```

the_ROM_image:
{
[pskfile]psk0.pem
[sskfile]ssk0.pem
[auth_params]spk_id = 0; ppk_select = 0
[keysrc_encryption]bbam_red_key
[fsbl_config]a53_x64, bh_auth_enable, opt_key
[bootloader, authentication = rsa, encryption = aes, aeskeyfile = fsbl_a53.nky,
blocks = 1728(*)]fsbl_a53.elf
[destination_cpu = pmu, authentication = rsa, encryption = aes, aeskeyfile =
pmu_fw.nky, blocks = 1728(*)]pmu_fw.elf
[destination_device = pl, authentication = rsa, encryption = aes, aeskeyfile =
edt_zcu102_wrapper.nky, blocks = 1728(*)]edt_zcu102_wrapper.bit
[destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
[destination_cpu = r5-0, authentication = rsa, encryption = aes, aeskeyfile =
tmr_psled_r5.nky, blocks = 1728(*)]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}
    
```

## Generating all of the AES keys

Once all desired encryption features have been enabled, you can generate all key files by running `Bootgen`. Some of the source files (for example, ELF) contain multiple sections. These individual sections will be mapped to separate partitions, and each partition will have a unique key file. In this case, the key file will be appended with a ".1.". For example, if the `pmu_fw.elf` file contains multiple sections, both a `pmu_fw.nky` and a `pmu_fw.1.nky` file will be generated.

1. Create all of the necessary NKY files by running the `bootgen` command that creates the final `BOOT.bin` image.

```
bootgen -p zcu9eg -arch zynqmp -image key_generation.bif
```

2. Verify that the NKY files were generated. These file should include `edt_zcu102_wrapper.nky`, `fsbl_a53.nky`, `pmu_fw.nky`, `pmu_fw.1.nky`, `pmu_fw.2.nky`, `tmr_psled_r5.nky`, and `tmr_psled_r5.1.nky`.

### Using Key Revocation

Key revocation allows you to revoke a RSA primary or secondary public key. Key revocation may be used due to elapsed time of key use or if there is an indication that the key is compromised. The primary and secondary key revocation is controlled by onetime programmable eFUSES. The Xilinx Secure Key Library is used for key revocation, allowing key revocation in fielded devices. Key revocation is discussed further in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 5].

### Using the PUF

In this section, the PUF is used for black key storage in the PUF Bootheader mode. RSA authentication is required when the PUF is used. In PUF Bootheader mode, the PUF helper data and the encrypted user's AES key are stored in the Bootheader. This section shows how to create a BIF for using the PUF. Because the helper data and encrypted user key will be unique for each and every board, the bootgen image created will only work on the board from which the helper data originated.

At the end of the [Secure Boot Sequence](#) section, a different BIF file demonstrates using the PUF in eFUSE mode. In PUF eFUSE mode, the PUF helper data and encrypted user's AES key are stored in eFUSES. In PUF eFUSE mode, a single boot image can be used across all boards.

### PUF Registration - Boot Header Mode

The PUF registration software is included in the XILSKEY library. The PUF registration software operates in a Bootheader mode or eFUSE mode. The Bootheader mode allows development without programming the OTP eFUSES. The eFUSE mode is used in production. This lab runs through PUF registration in Bootheader Mode only. For PUF registration using eFUSE, see *Programming BBRAM and eFUSES* (XAPP1319) [Ref 12].

The PUF registration software accepts a red (unencrypted) key as input, and produces syndrome data (helper data), which also contains CHASH and AUX, and a black (encrypted) key. When the PUF Bootheader mode is used, the output is put in the bootheader. When the PUF eFUSE mode is used, the output is programmed into eFUSES.

1. In the Vitis IDE, right click `tmr_psled_r5_bsp` and click **Board Support Package Settings**.
2. Ensure that xilskey and the xilsecure libraries are enabled.

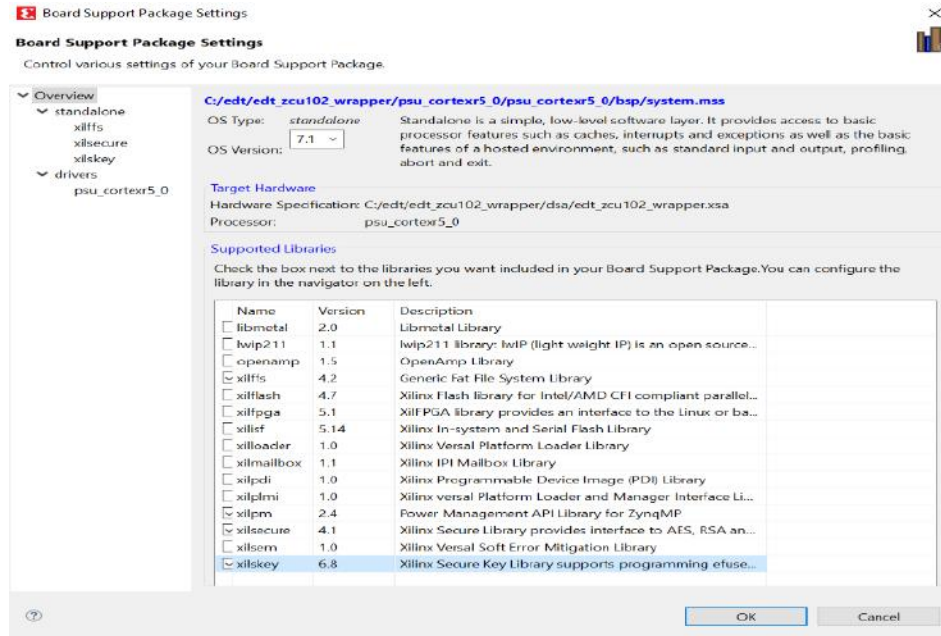


Figure 5-34: Select Xilskkey and Xilsecure Libraries

3. Click **OK**. Re-build the hardware platform for changes to apply. Navigate to `tmr_psled_r5_bsp` settings.
4. Scroll to the Libraries section. Click on **xilskkey 6.8 Import Examples**.
5. In the dialog box, select the **xilskkey\_puf\_registration** example. Click **OK**.

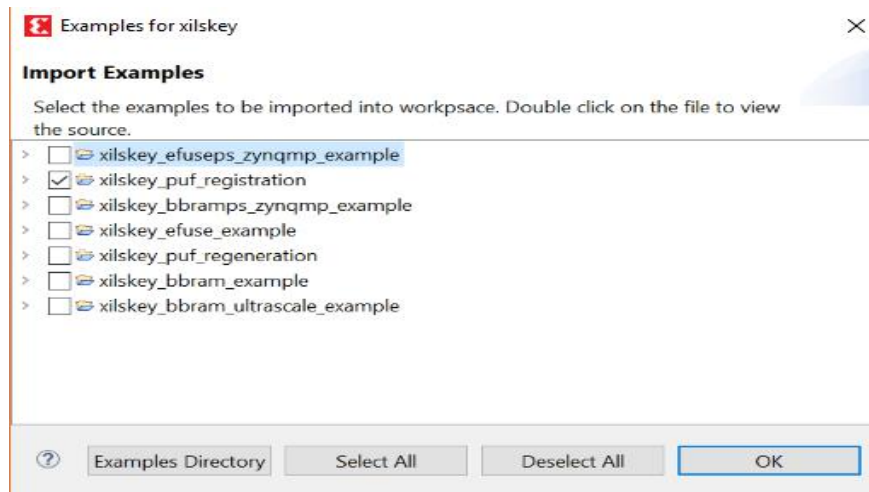


Figure 5-35: Import PUF Registration Example

6. In the project explorer, verify that the `xilskkey_puf_example_1` application is created under `system_bsp_example_1`.
7. In Project Explorer, `xilskkey_puf_example_1 'Src'` double click `xilskkey_puf_registration.h` to open in the Vitis IDE.

8. Edit `xilskey_puf_registration.h` as follows:
  - a. Change `#define XSK_PUF_INFO_ON_UART` from `FALSE` to `TRUE`
  - b. Ensure that `#define XSK_PUF_PROGRAM_EFUSE` is set to `FALSE`
  - c. Set `XSK_PUF_AES_KEY` (the 256 bit key).

The key is to be entered in HEX format and should be `Key 0` from the `fsb1_a53.nky` file that you generated in [Generating all of the AES keys](#). You can find a sample key below:

```
#define XSK_PUF_AES_KEY
"68D58595279ED1481C674383583C1D98DA816202A57E7FE4F67859CB069CD510"
```

**Note:** Do not copy this key. Refer to the `fsb1_a53.nky` file for your key.

- d. Set the `XSK_PUF_BLACK_KEY_IV`. The initialization vector IV is a 12 byte data of your choice.

```
#define XSK_PUF_BLACK_KEY_IV "E1757A6E6DD1CC9F733BED31"
```

```
/* Following parameters should be configured by user */
#define XSK_PUF_INFO_ON_UART TRUE
#define XSK_PUF_PROGRAM_EFUSE FALSE
#define XSK_PUF_IF_CONTRACT_MANUFACTURER FALSE

/* For programming/reading secure bits of PUF */
#define XSK_PUF_READ_SECUREBITS FALSE
#define XSK_PUF_PROGRAM_SECUREBITS FALSE

#if (XSK_PUF_PROGRAM_SECUREBITS == TRUE)
#define XSK_PUF_SYN_INVALID FALSE
#define XSK_PUF_SYN_WRLK FALSE
#define XSK_PUF_REGISTER_DISABLE FALSE
#define XSK_PUF_RESERVED FALSE
#endif

#define XSK_PUF_AES_KEY "EA7D39616B5C1CEDB8DE9F6C97E86B7AFD07EBA6DCBB39B740E576F9E92AE570"
#define XSK_PUF_BLACK_KEY_IV "D0514B92D8774770A73A846F"
```

Figure 5-36: PUF Registration in Bootheader Mode

9. Save the file and exit.
10. In Project Explorer, right-click on the `xilskey_puf_example_1` project and select **Build Project**.
11. In the Vitis IDE, select **Xilinx > Create Boot Image**.
12. Select Zynq MP in the Architecture dialog box.
13. In the Output BIF file path: dialog box, specify
 

```
c:\edt\secureboot_sd\puf_registration\puf_registration.bif
```
14. In the Output Path dialog box, specify
 

```
c:\edt\secureboot_sd\puf_registration\BOOT.bin
```

- In the Boot Image Partitions pane, click **Add**. Add the partitions and set the destination CPU of the xilskey\_puf\_example\_1 application to R5-0:

```
C:\edt\fsbl_a53\Debug\fsbl_a53.elf
C:\edt\xilskey_puf_example_1\Debug\xilskey_puf_example_1.elf
```

- Click on **Create Image** to create the Boot Image for PUF registration

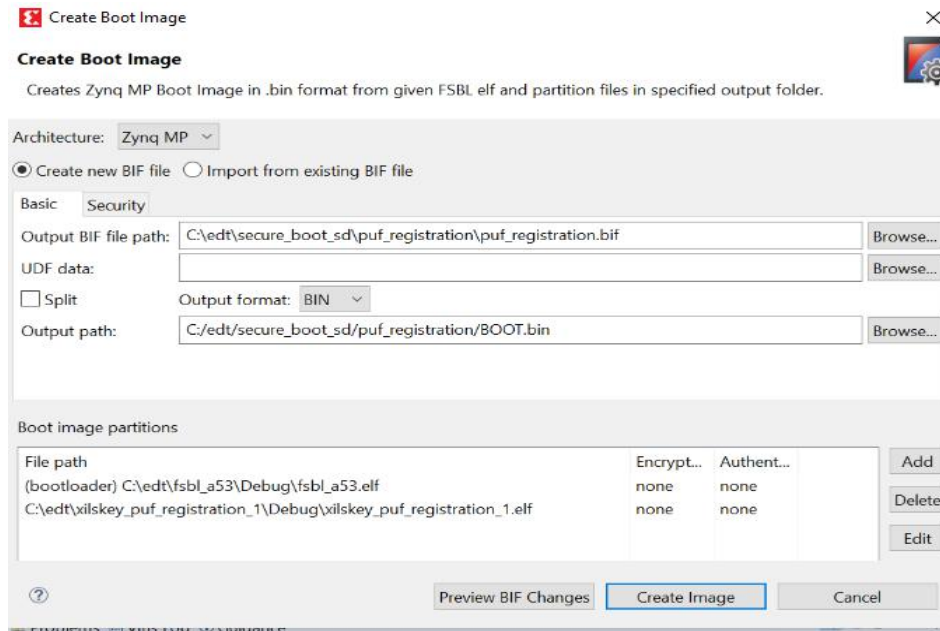


Figure 5-37: PUF Registration Software

- Insert a SD card into the PC SD card slot.
- Copy `C:\edt\secureboot_sd\puf_registration\BOOT.bin` to the SD Card
- Move the SD card from the PC SD card slot to the ZCU102 card slot.
- Start a terminal session, using Tera Term or Minicom depending on the host machine being used, as well as the COM port and baud rate for your system, as shown in [Figure 3-18](#).
- In the communication terminal menu bar, select **File > Log**. Enter `C:\edt\secureboot_sd\puf_registration\puf_registration.log` in the dialog box.
- Power cycle the board.
- After the `puf_registration` software has run, exit the communication terminal.
- The `puf_registration.log` content is used in [Using PUF in Bootheader Mode](#). Open `puf_registration.log` in a text editor.
- Save the PUF Syndrome data that starts after App PUF Syndrome data Start!!!; and ends at PUF Syndrome data End!!!, non-inclusive, to a file named `helperdata.txt`.
- Save the black key IV identified by App: Black Key IV - to a file named `black_iv.txt`.



27. Save the black key to a file named `black_key.txt`.
28. The files `helperdata.txt`, `black_key.txt`, and `black_iv.txt` can be saved in `C:\edt\secure_boot_sd\keys`

### Using PUF in Bootheader Mode

The following steps describes the process to update the `.bif` file from the previous sections to include using the PUF in Boot Header mode. This section will make use of the Syndrome data and Black Key created during PUF registration process.

1. Enable use of the PUF by adding all of the fields and attributes indicated in bold to the `bif` file (`key_generation.bif`) shown below.

```

the_ROM_image:
{
[pskfile]psk0.pem
[sskfile]ssk0.pem
[auth_params]spk_id = 0; ppk_select = 0
[keysrc_encryption]bh_blk_key
[bh_key_iv]black_iv.txt
[bh_keyfile]black_key.txt
[puf_file]helperdata.txt
[fsbl_config]a53_x64, bh_auth_enable, opt_key, puf4kmode,
shutter=0x0100005E,pufhd_bh
[bootloader, authentication = rsa, encryption = aes, aeskeyfile = fsbl_a53.nky,
blocks = 1728(*)]fsbl_a53.elf
[destination_cpu = pmu, authentication = rsa, encryption = aes, aeskeyfile =
pmu_fw.nky, blocks = 1728(*)]pmu_fw.elf
[destination_device = pl, authentication = rsa, encryption = aes, aeskeyfile =
edt_zcu102_wrapper.nky, blocks = 1728(*)]edt_zcu102_wrapper.bit
[destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
[destination_cpu = r5-0, authentication = rsa, encryption = aes, aeskeyfile =
tmr_psled_r5.nky, blocks =1728(*)]tmr_psled_r5.elf
[destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
[load = 0x1000000, destination_cpu = a53-0, authentication = rsa]image.ub
}

```

2. The above `.bif` file can be used for creating a final boot image using an AES key encrypted in the boot image header with the PUF KEK. This would be done using the following `bootgen` command.

```
bootgen -p zcu9eg -arch zynqmp -image key_generation.bif -w -o BOOT.bin
```

**Note:** The above steps can also be executed with PUF in eFUSE mode. In this case you can repeat the previous steps, using the PUF in eFUSE mode. This requires enabling the programming of eFUSES during PUF registration by setting the `XSK_PUF_PROGRAM_EFUSE` macro in the `xilskkey_puf_registration.h` file used to build the PUF registration application. Also, the BIF would need to be modified to use the encryption key from eFUSE and removing the helper data and black key files. PUF in eFUSE mode is not covered in this tutorial in order to avoid programming the eFUSES on development or tutorial systems.

```

[keysrc_encryption]efuse_blk_key
[bh_key_iv]black_iv.txt

```

### System Example Using the Vitis IDE Create Boot Image Wizard

The prior sections enabled the various security features (authentication, confidentiality, DPA protections, and black key storage) by hand editing the BIF file. This section performs the same operations, but uses the Bootgen Wizard as a starting point. The Bootgen Wizard creates a base BIF file, and then adds the additional security features that are not supported by the wizard using a text editor.

1. Change directory to the `bootgen_files` directory.

```
cd c:\edt\secure_boot_sd\bootgen_files
```

2. Copy the below data from the prior example to this example.

```
cp ../keys/*nky .
cp ../keys/*pem .
cp ../keys/black_iv.txt .
cp ../keys/helperdata.txt .
cp ../keys/*.elf .
cp ../keys/edt_zcu102_wrapper.bit .
cp ../keys/image.ub .
cp ../keys/black_key.txt.
```

3. Click **Programs > Xilinx Design Tools > Vitis 2019.2 > Xilinx Vitis 2019.2** to launch the Vitis IDE.
4. Click **Xilinx Tools > Create Boot Image** from the menu bar to launch the Create Boot Image wizard.
5. Select **Zynq MP** as the Architecture.
6. Enter the Output BIF file path as  
`c:\edt\secure_boot_sd\bootgen_files\design_bh_bkey_keyrolling.bif`.
7. Select BIN as the output format.
8. Enter the output path `c:\edt\secure_boot_sd\bootgen_files\BOOT.bin`.
9. Enable authentication.
  - a. Click the **Security** tab.
  - b. Check the **Use Authentication** check box.
  - c. Browse to select the `psk0.pem` file for the PSK and the `ssk0.pem` for the SSK.
  - d. Ensure PPK select is 0.
  - e. Enter SPK ID as 0.
  - f. Check the Use BH Auth checkbox.

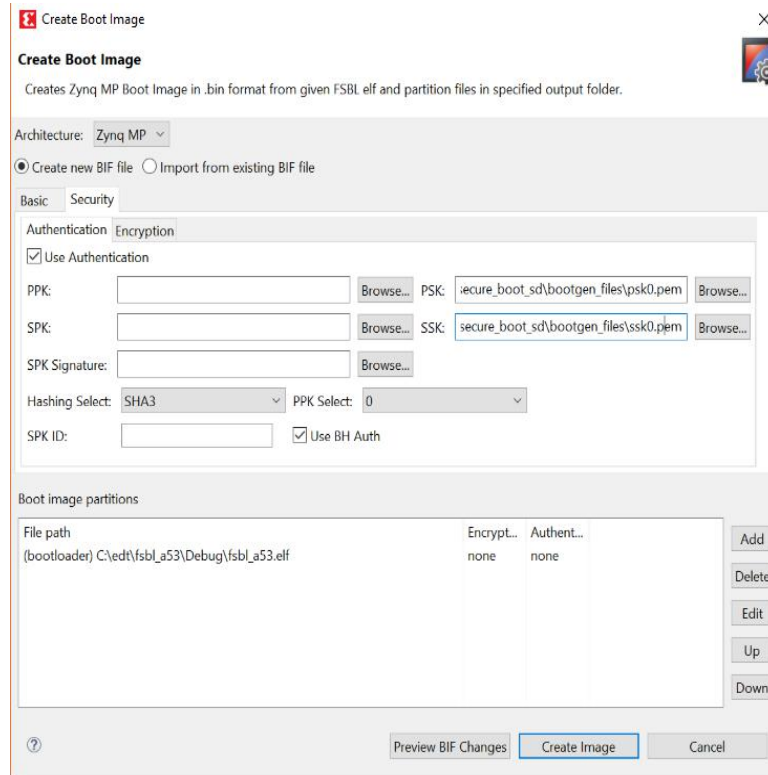


Figure 5-38: Enable Authentication

10. Enable encryption.

- a. Click the **Encryption** tab.
- b. Check the **Use Encryption** checkbox.
- c. Use the browse button to select `fsbl_a53.nky` as the key file.
- d. Check the **Operational Key** checkbox.

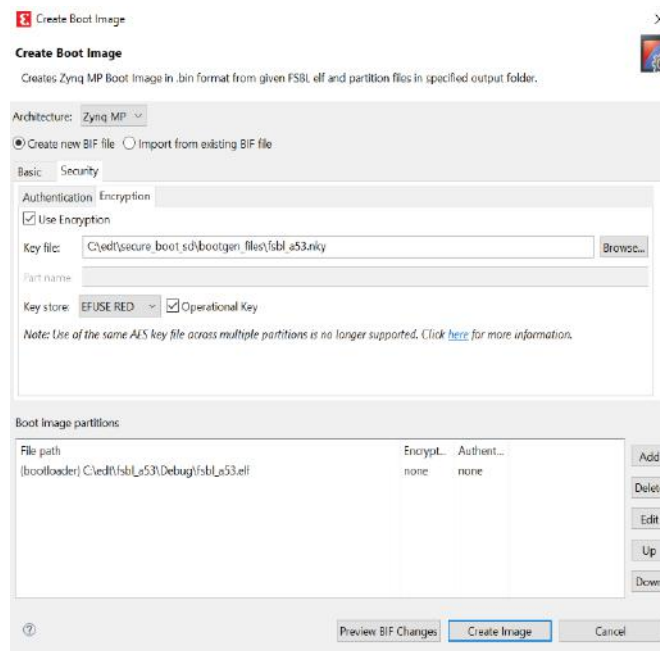


Figure 5-39: Enable Encryption

11. Click the **Basic** tab.
12. Add the FSBL binary to the boot image.
  - a. Click **Add**.
  - b. Use the browse button to select the `fsbl_a53.elf` file.
  - c. Make sure the partition-type is bootloader and the destination CPU is a53x64.
  - d. Change authentication to RSA.
  - e. Change encryption to AES.
  - f. Click **OK**.

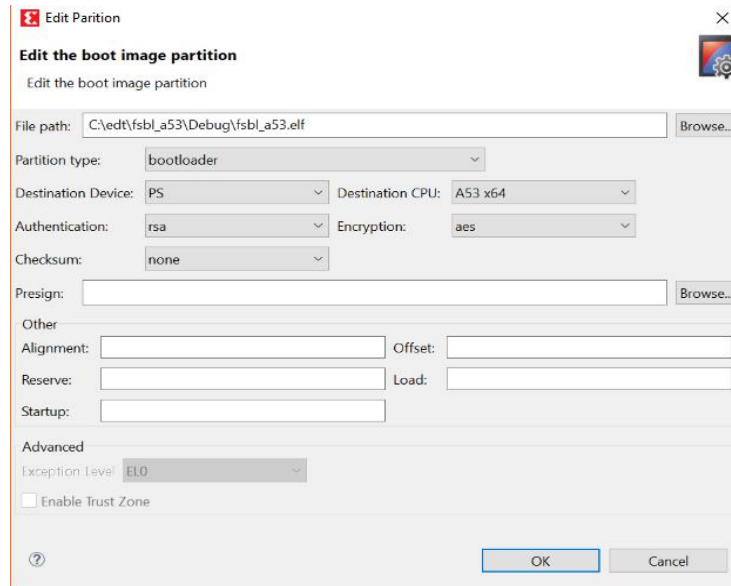


Figure 5-40: Adding FSBL Binary

13. Add the PMW firmware binary to the boot image.
  - a. Click **Add**.
  - b. Use the browse button to select the `pmu_fw.elf` file.
  - c. Make sure the partition-type is datafile.
  - d. Change the destination CPU to PMU.
  - e. Change authentication to RSA.
  - f. Change encryption to AES.
  - g. Click **OK**.

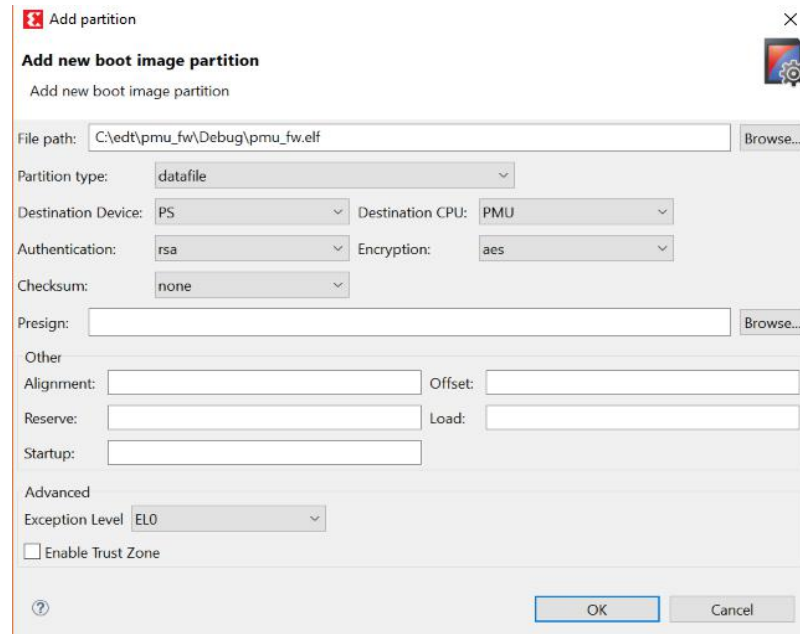


Figure 5-41: Adding PMU Firmware Binary

14. Add the PL Bitstream to the boot image.
  - a. Click the **Add**.
  - b. Use the browse button to select the `edt_zcu102_wrapper.bit` file.
  - c. Make sure the partition-type is datafile.
  - d. Make sure the destination device is PL.
  - e. Change authentication to RSA.
  - f. Change encryption to AES.
  - g. Click **OK**.

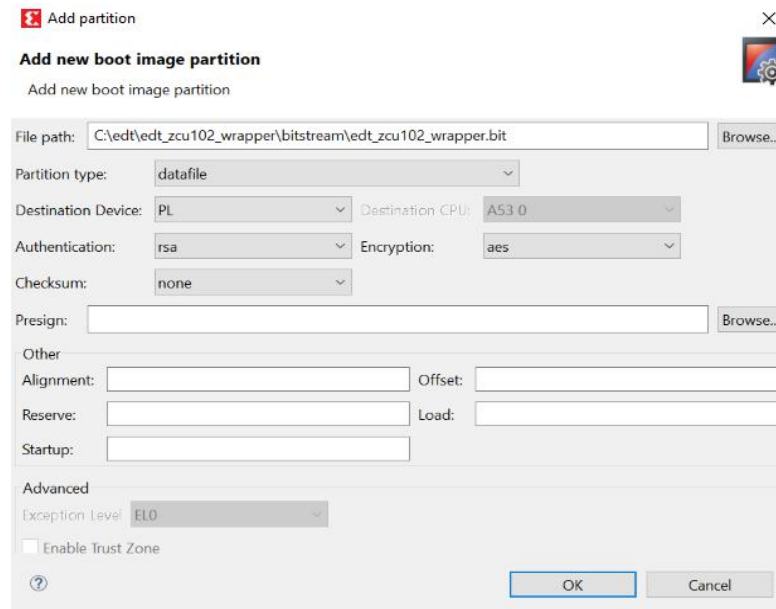


Figure 5-42: Adding PL Bitstream

15. Add the Arm Trusted Firmware (ATF) binary to the image.
  - a. Click **Add**.
  - b. Use the browse button to select the `b131.e1f` file.
  - c. Make sure the partition-type is datafile.
  - d. Make sure the destination CPU is A53 0.
  - e. Change authentication to RSA.
  - f. Make sure the encryption is none.
  - g. Make sure the Exception Level is EL3 and enable Trust Zone.
  - h. Click **OK**.

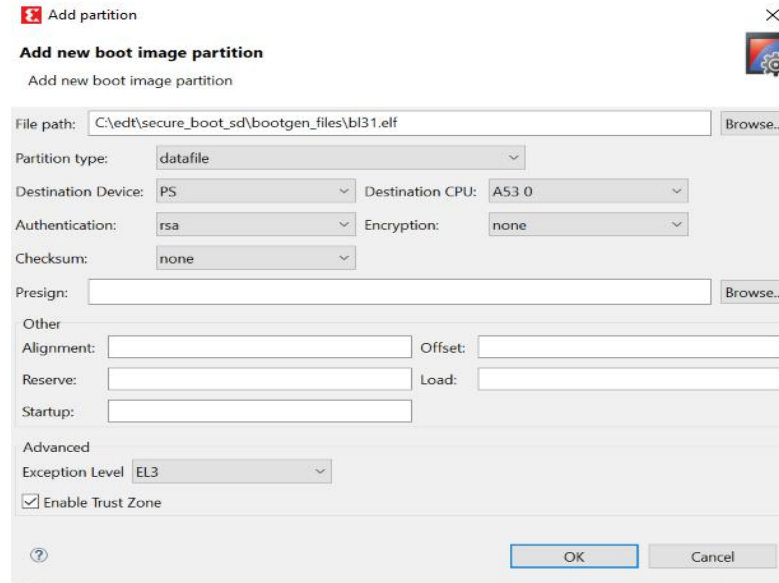


Figure 5-43: Adding Arm Trusted Firmware

16. Add the R5 software binary to the boot image.
  - a. Click **Add**.
  - b. Use the browse button to select the `tmr_psled_r5.elf` file.
  - c. Make sure the partition-type is datafile.
  - d. Make sure the destination CPU is R5 0.
  - e. Change authentication to RSA.
  - f. Change encryption to AES.
  - g. Click **OK**.



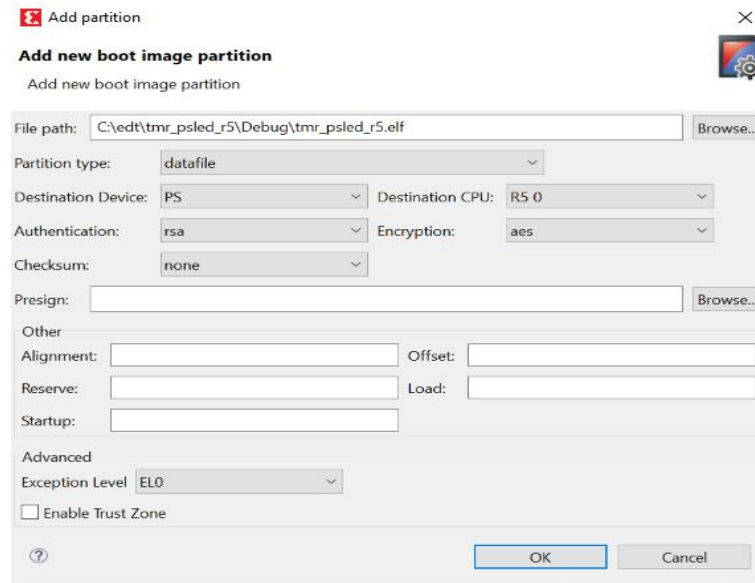


Figure 5-44: Adding R5 Application Binary

17. Add the U-Boot software binary to the boot image.
  - a. Click **Add**.
  - b. Use the browse button to select the `u-boot.elf` file.
  - c. Make sure the partition-type is datafile.
  - d. Make sure the destination CPU is A53 0.
  - e. Change authentication to RSA.
  - f. Make sure that encryption is none.
  - g. Change the Exception Level to EL2.
  - h. Click **OK**.

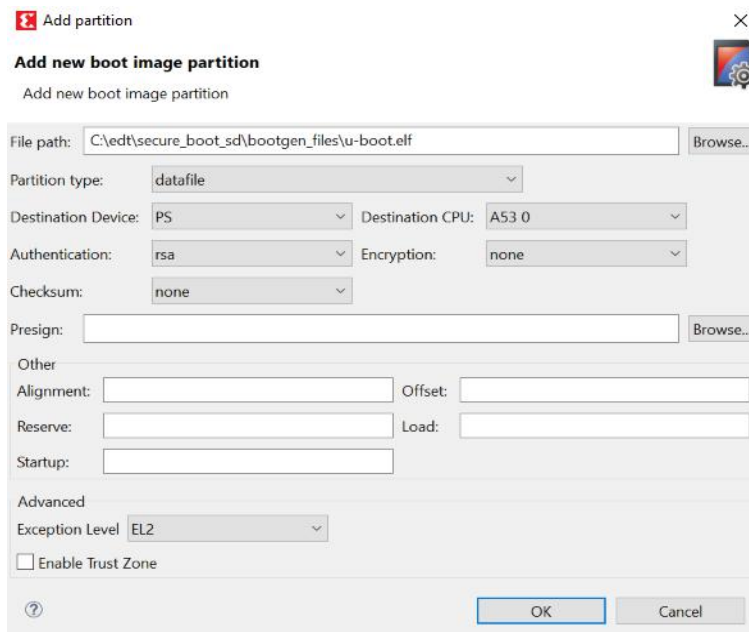


Figure 5-45: Adding U-Boot Software

18. Add the Linux image to the boot image.
  - a. Click **Add**.
  - b. Use the browse button to select the `image.ub` file.
  - c. Make sure the partition-type is datafile.
  - d. Make sure the destination CPU is A53 0.
  - e. Change authentication to RSA.
  - f. Make sure that encryption is none.
  - g. Update the load field to 0x2000000.
  - h. Click **OK**.

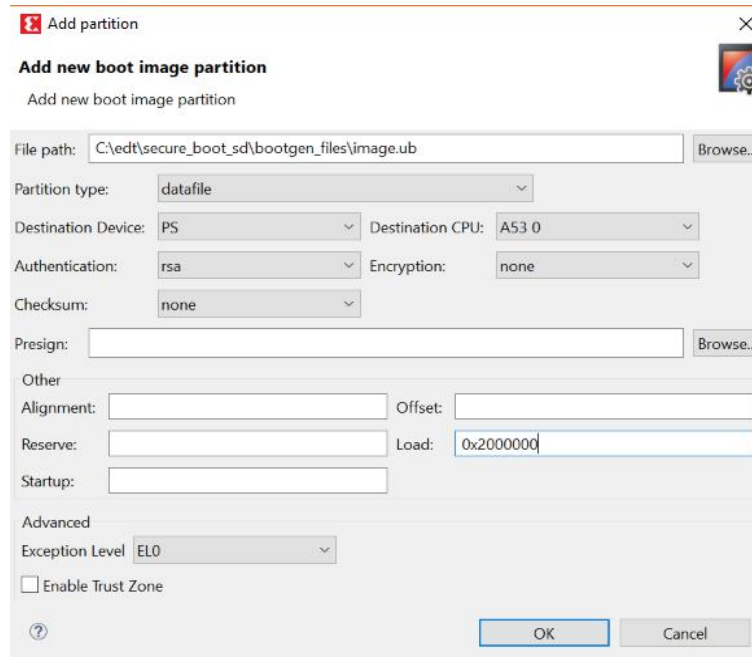


Figure 5-46: Adding Linux Boot Image

19. Click **Create image**

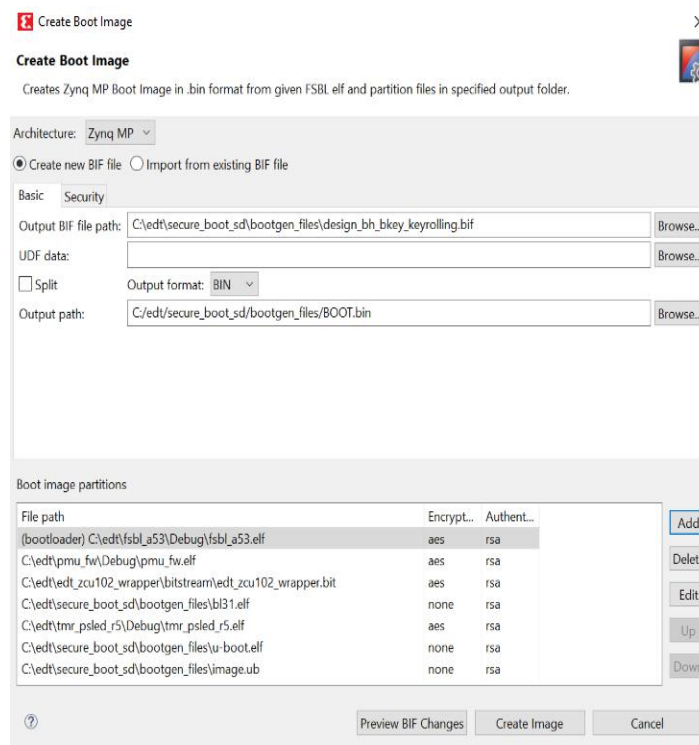


Figure 5-47: Creating a Final Boot Image

20. The `design_bh_bkey_keyrolling.bif` file should look similar to the following:

```

the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [aeskeyfile]fsbl_a53.nky
  [keysrc_encryption]efuse_red_key
  [fsbl_config]a53_x64, bh_auth_enable, opt_key
  [bootloader, encryption = aes, authentication = rsa]fsbl_a53.elf
  [encryption = aes, authentication = rsa, destination_cpu = pmu]pmu_fw.elf
  [encryption = aes, authentication = rsa, destination_device =
pl]edt_zcu102_wrapper.bit
  [authentication = rsa, destination_cpu = a53-0, exception_level = e1-3]bl31.elf
  [encryption = aes, authentication = rsa, destination_cpu = r5-0]tmr_psled_r5.elf
  [authentication = rsa, destination_cpu = a53-0, exception_level = e1-2]u-boot.elf
  [authentication = rsa, load = 0x2000000, destination_cpu = a53-0]image.ub
}
    
```

21. This BIF file is still missing several security features that are not supported by the Create Boot Image wizard. These are features are per-partition nky files, key rolling and black key store.
22. Add black key store by changing the `keysrc_encryption` and adding the other additional items so that the BIF file looks like the following:

```

the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [aeskeyfile]fsbl_a53.nky
  [keysrc_encryption]bh_blk_key
  [bh_key_iv]black_iv.txt
  [bh_keyfile]black_key.txt
  [puf_file]helperdata.txt
  [fsbl_config]a53_x64, bh_auth_enable, opt_key, puf4kmode, shutter=0x0100005E,
pufhd_bh
  [bootloader, encryption = aes, authentication = rsa]fsbl_a53.elf
  [encryption = aes, authentication = rsa, destination_cpu = pmu]pmu_fw.elf
  [encryption = aes, authentication = rsa, destination_device =
pl]edt_zcu102_wrapper.bit
  [authentication = rsa, destination_cpu = a53-0, exception_level = e1-3]bl31.elf
  [encryption = aes, authentication = rsa, destination_cpu = r5-0]tmr_psled_r5.elf
  [authentication = rsa, destination_cpu = a53-0, exception_level = e1-2]u-boot.elf
  [authentication = rsa, load = 0x2000000, destination_cpu = a53-0]image.ub
}
    
```

23. Specify unique AES key files for each encrypted partition by updating the BIF file to look like the following:

```

the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [keysrc_encryption]bh_blk_key
  [bh_key_iv]black_iv.txt
  [bh_keyfile]black_key.txt
  [puf_file]helperdata.txt
    
```

```
[fsbl_config]a53_x64, bh_auth_enable, opt_key, puf4kmode, shutter=0x0100005E,
pufhd_bh
[bootloader, encryption = aes, aeskeyfile = fsbl_a53.nky, authentication =
rsa]fsbl_a53.elf
[encryption = aes, aeskeyfile = pmu_fw.nky, authentication = rsa, destination_cpu =
pmu]pmu_fw.elf
[encryption = aes, aeskeyfile = edt_zcu102_wrapper.nky, authentication = rsa,
destination_device = pl]edt_zcu102_wrapper.bit
[authentication = rsa, destination_cpu = a53-0, exception_level = el-3]bl31.elf
[encryption = aes, aeskeyfile = tmr_psled_r5.nky, authentication = rsa,
destination_cpu = r5-0]tmr_psled_r5.elf
[authentication = rsa, destination_cpu = a53-0, exception_level = el-2]u-boot.elf
[authentication = rsa, load = 0x2000000, destination_cpu = a53-0]image.ub
}
```

24. Enable key rolling by adding the block attributes to the encrypted partitions. The updated BIF file should now look like the following:

```
the_ROM_image:
{
[pskfile]psk0.pem
[sskfile]ssk0.pem
[auth_params]spk_id = 0; ppk_select = 0
[keysrc_encryption]bh_blk_key
[bh_key_iv]black_iv.txt
[bh_keyfile]black_key.txt
[puf_file]helperdata.txt
[fsbl_config]a53_x64, bh_auth_enable, opt_key, puf4kmode, shutter=0x0100005e,
pufhd_bh
[bootloader, encryption = aes, aeskeyfile = fsbl_a53.nky, authentication = rsa,
blocks = 1728(*)]fsbl_a53.elf
[encryption = aes, aeskeyfile = pmu_fw.nky, authentication = rsa, destination_cpu =
pmu, blocks = 1728(*)]pmu_fw.elf
[encryption = aes, aeskeyfile = edt_zcu102_wrapper.nky, authentication = rsa,
destination_device = pl, blocks = 1728(*)]edt_zcu102_wrapper.bit
[authentication = rsa, destination_cpu = a53-0, exception_level = el-3]bl31.elf
[encryption = aes, aeskeyfile = tmr_psled_r5.nky, authentication = rsa,
destination_cpu = r5-0, blocks = 1728(*)]tmr_psled_r5.elf
[authentication = rsa, destination_cpu = a53-0, exception_level = el-2]u-boot.elf
[authentication = rsa, load = 0x2000000, destination_cpu = a53-0]image.ub
}
```

25. Generate the boot image by running the following command. Note that the `-encryption_dump` flag has been added. This flag causes the log file `aes_log.txt` to be created. The log file details all encryption operations that were used. This allows you to see which keys and IVs were used on which sections of the boot image.

```
bootgen -p zcu9eg -arch zynqmp -image design_bh_bkey_keyrolling.bif -w -o BOOT.bin
-encryption_dump
```

### **Booting the system using a Secure Boot Image**

This section demonstrates how to use the `BOOT.bin` boot image created in prior sections to perform a secure boot using the ZCU102.

1. Copy the `BOOT.bin` image and the `ps_pl_linux_app.elf` over to the SD card from `c:\edt\secure_boot_sd\bootgen_files`.

2. Insert the SD card into the ZCU102.
3. Set SW6 of the ZCU102 for SD boot mode (1=ON; 2,3,4=OFF).

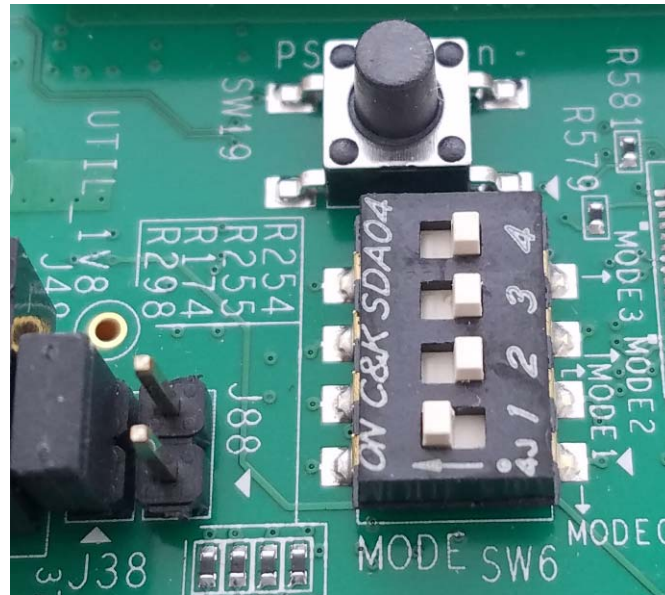


Figure 5-48: SW6 Switch Settings for SD boot Mode

4. Connect Serial terminals to ZCU102 (115200, 8 data bits, 1 stop bit, no parity)
5. Power on the ZCU102
6. When the terminal reaches the U-boot ZynqMP> prompt, type bootm 0x2000000.

```

Manufacturer ID: 3
OEM: 5344
Name: SS04G
Bus Speed: 100000000
Mode : UHS SDR50 (100MHz)
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 3.7 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
** Unable to read file image.ub **
ZynqMP> bootm 0x2000000

```

Figure 5-49: U-Boot Prompt

7. Login into Linux using the following credentials:

Login: root;

password: root

```

dropbear.
Starting internet superserver: inetd.
Starting syslogd/klogd: done
Starting tcf-agent: OK

Petalinux 2019.2 xilinx-zcu102-2019_2 /dev/ttyPS0
xilinx-zcu102-2019_2 login: root
Password:
root@xilinx-zcu102-2019_2:~# █

```

Figure 5-50: Linux Login

Run the Linux Application as described in [Design Example 1: Using GPIOs, Timers, and Interrupts](#).

### Running the Linux Application

Use the following steps to run a Linux application:

1. Copy the application from SD card mount point to /tmp.

```
# cp /run/media/mmcblk0p1/ps_pl_linux_app.elf /tmp
```

**Note:** Mount the SD card manually if you fail to find SD card contents in this location.

```
# mount /dev/mmcblk0p1 /media/
```

2. Copy the application to /tmp.

```
# cp /media/ps_pl_linux_app.elf /tmp
```

3. Run the application.

```
# /tmp/ps_pl_linux_app.elf
```

### Sample BIF for a fielded system

The following BIF file is an example for a fielded system. In order for this bif file to work on a board it requires the RSA\_EN, PPK0 Digest, black AES key and PUF helper data to all be programmed in the eFUSES. Since programming these eFUSES severely limits the use of the device or board for testing and debugging, it is only included here as a reference. It is not part of the tutorial.

The following changes are made to the final `generation.bif` file reach the following result:

1. Change from PUF Bootheader mode to PUF eFUSE mode.
  - a. Change the `keysrc_encryption` attribute to `efuse_blk_key`.
  - b. Remove the `bh_keyfile` and `puf_file` lines.
  - c. Remove the `puf4kmode` and `pufhd_bh` attributes from the `fsbl_config` line.
2. Change from boot header authentication to eFUSE authentication.
  - a. Remove the `bh_auth_enable` attribute from the `fsbl_config` line.

```

the_ROM_image:
{
  [pskfile]psk0.pem
  [sskfile]ssk0.pem
  [auth_params]spk_id = 0; ppk_select = 0
  [keysrc_encryption]efuse_blk_key
  [bh_key_iv]black_iv.txt
  [fsbl_config]a53_x64, opt_key, shutter=0x0100005E
  [aeskeyfile]fsbl_a53.nky
  [bootloader, authentication = rsa, encryption = aes, blocks = 1728(*)]fsbl_a53.elf
  [aeskeyfile]pmu_fw.nky
  [destination_cpu = pmu, authentication = rsa, encryption = aes, blocks =
1728(*)]pmu_fw.elf
  [aeskeyfile]edt_zcu102_wrapper.nky
  [destination_device = pl, authentication = rsa, encryption = aes, blocks =
1728(*)]edt_zcu102_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone, authentication =
rsa]bl31.elf
  [aeskeyfile]tmr_psled_r5.nky
  [destination_cpu = r5-0, authentication = rsa, encryption = aes, blocks =
1728(*)]tmr_psled_r5.elf
  [destination_cpu = a53-0, exception_level = el-2, authentication = rsa]u-boot.elf
  [load = 0x2000000, destination_cpu = a53-0, authentication = rsa]image.ub
}
    
```



# System Design Examples

This chapter guides you through building a system based on Zynq® UltraScale+™ devices using available tools and supported software blocks. This chapter highlights how you can use the software blocks you configured in [Chapter 3](#) to create a Zynq UltraScale+ system. It does not discuss domain-specific designs, but rather highlights different ways to use low-level software available for Zynq UltraScale+ devices.

---

## Design Example 1: Using GPIOs, Timers, and Interrupts

The Zynq ZCU102 UltraScale+ Evaluation Board comes with a few user configurable Switches and LEDs. This design example makes use of bare-metal and Linux applications to toggle these LEDs, with the following details:

- The Linux applications configure a set of PL LEDs to toggle using a PS Dip Switch, and another set of PL LEDs to toggle using a PL Dip Switch (SW17).
- The Linux APU A-53 Core 0 hosts this Linux application, while the RPU R5-0 hosts another bare-metal application.
- The R5-Core 0 application uses an AXI Timer IP in Programmable logic to toggle PS LED (DS50). The application is configured to toggle the LED state every time the timer counter expires, and the Timer in the PL is set to reset periodically after a user-configurable time interval. The system is configured such that the APU Linux Application and RPU Bare-metal Application run simultaneously.

### Configuring Hardware

The first step in this design is to configure the PS and PL sections. This can be done in Vivado IP integrator. You start with adding the required IPs from the Vivado IP catalog and then connect the components to blocks in the PS subsystem.

1. If the Vivado Design Suite is already open, start from the block diagram (shown in [Figure 2-2](#)) and jump to [step 4](#).
2. Open the Vivado Project that you created:

```
C:/edt/edt_zcu102/edt_zcu102.xpr
```

3. In the Flow Navigator, under **IP Integrator**, click **Open Block Design** and select **edt\_zcu102.bd**.

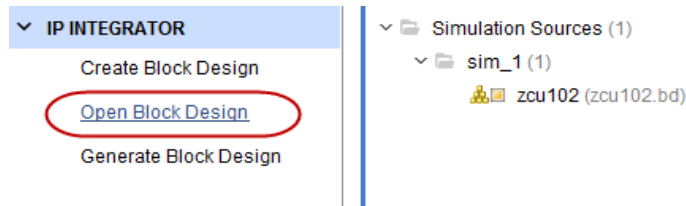


Figure 6-1: Open Block Design

4. Right click in the block diagram and select **Add IP** from the IP catalog.

### Adding and Configuring IPs

1. In the catalog, select **AXI Timer**.

The IP Details information displays, as shown in the following figure.

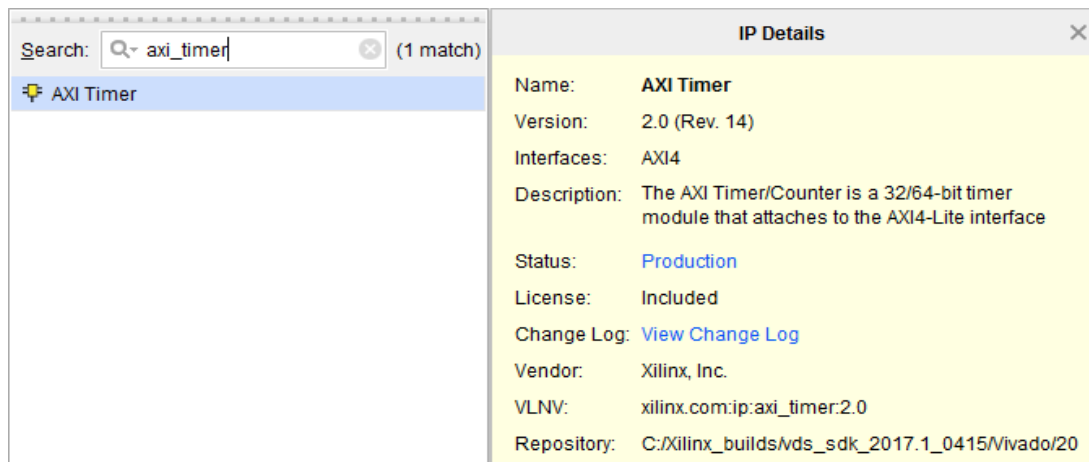


Figure 6-2: IP Details Information

2. Double-click the **AXI Timer** IP to add it to the design.

3. Double-click the **AXI Timer** IP again to configure the IP, as shown in following figure.

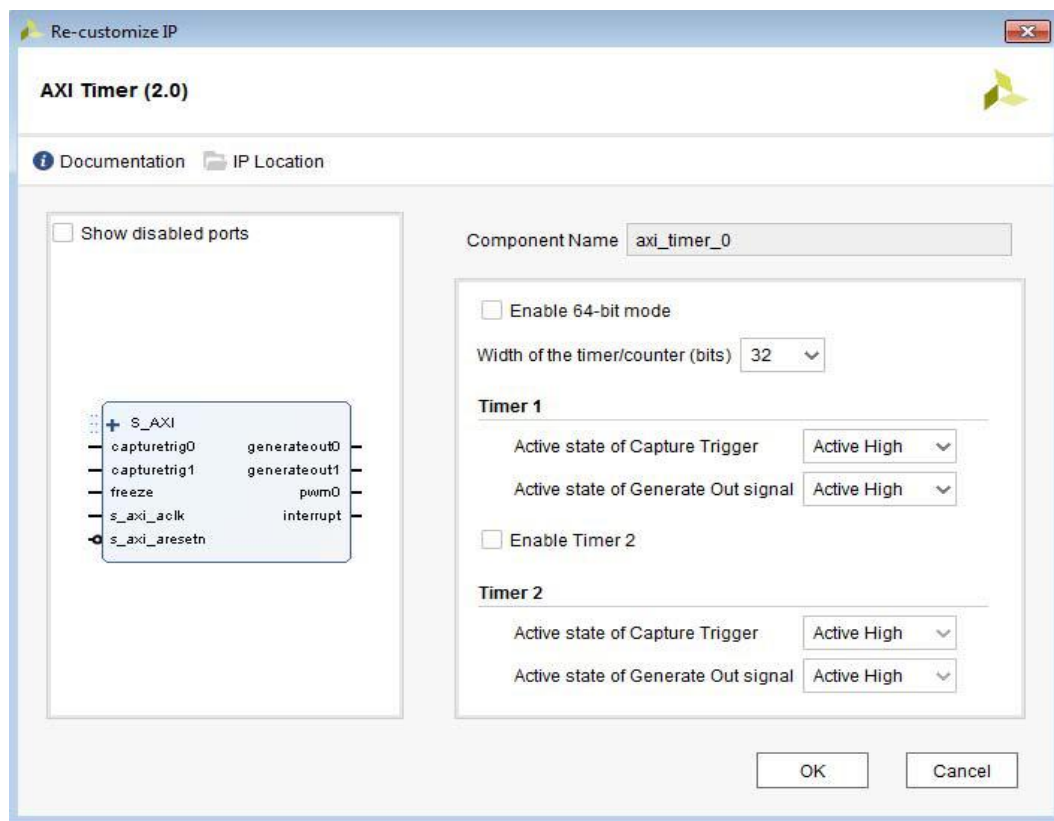


Figure 6-3: Re-customize IP Dialog Box for AXI Timer

4. Click **OK**.
5. Again, right-click in the block diagram and select **Add IP**.
6. Search for "AXI GPIO" and double-click the **AXI GPIO** IP to add it to the design.
7. Repeat [step 5](#) and [step 6](#) to add another instance of AXI GPIO IP.
8. Double-click **axi\_gpio\_0** and select **Push button 5bits** from the GPIO Board Interface drop-down list.

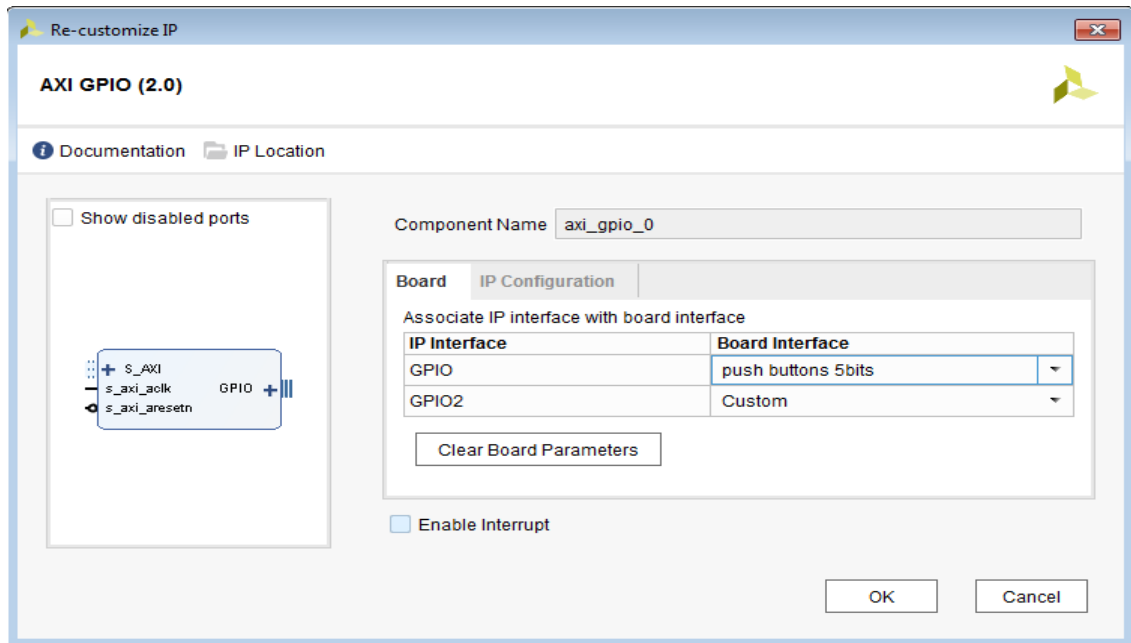


Figure 6-4: Re-customize IP Dialog Box for AXI GPIO

9. Click **OK** to configure the AXI\_GPIO for Push buttons.
10. Double-click on `axi_gpio_1`.
11. Configure `axi_gpio_1` for PL LEDs by selecting `led_8bits` from the **GPIO Board Interface** drop-down list, as shown in the following figure.

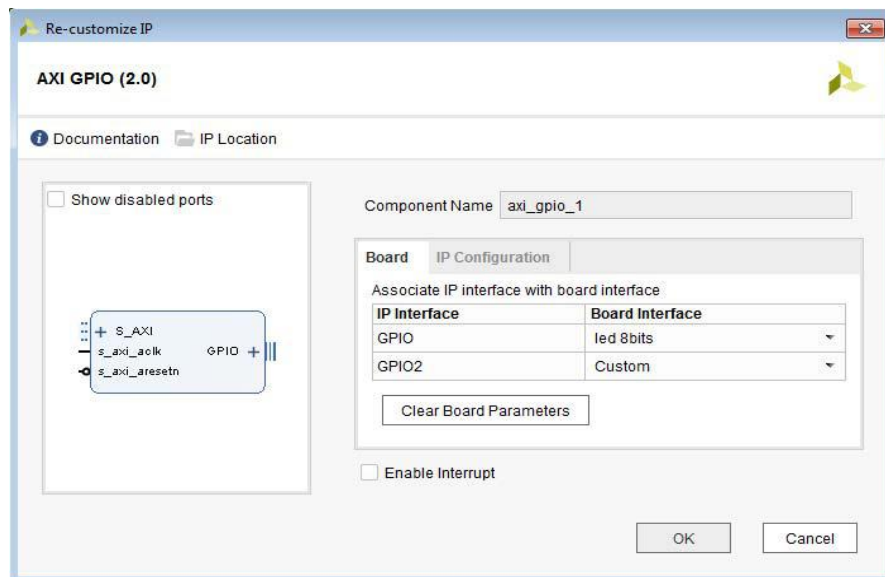


Figure 6-5: Configuring GPIO for led\_8bits

12. Click **OK** to configure the AXI\_GPIO for LED.

### Connecting IP Blocks to Create a Complete System

Make the initial connections using Board presets. To do this, follow the below steps:

1. Double-click the **Zynq UltraScale+ IP** Block, and select a PL-PS interrupt as shown in [Figure 6-6](#) (Ignore and move to the next step, if this is selected by default).

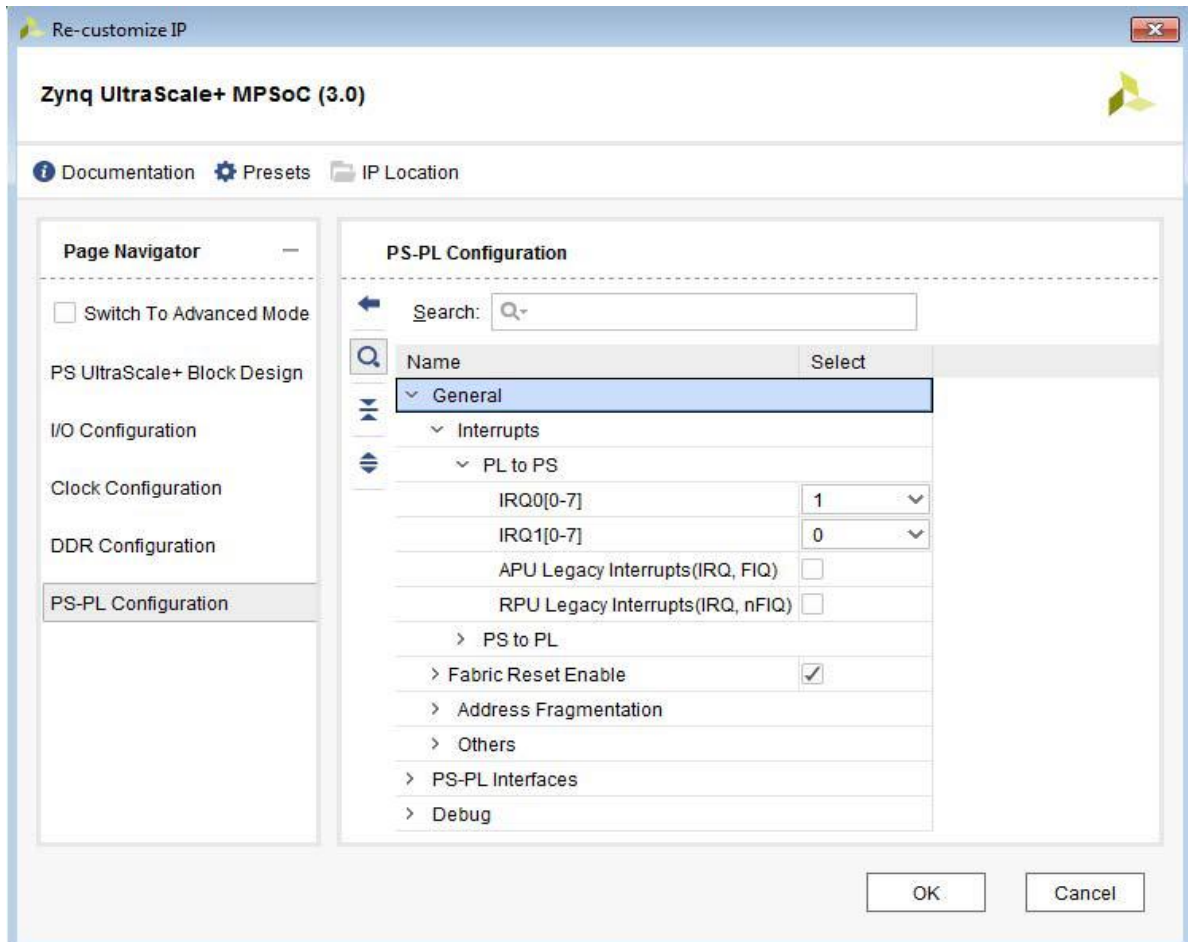


Figure 6-6: Selecting PL to PS Interrupt

2. In PS-PL Configuration, expand **PS-PL Interfaces** and expand the **Master Interface**.

- Expand **AXI HPM0 LPD** and set the **AXI HPM0 LPD Data Width** drop-down to **128 bit**, as shown in [Figure 6-7](#).

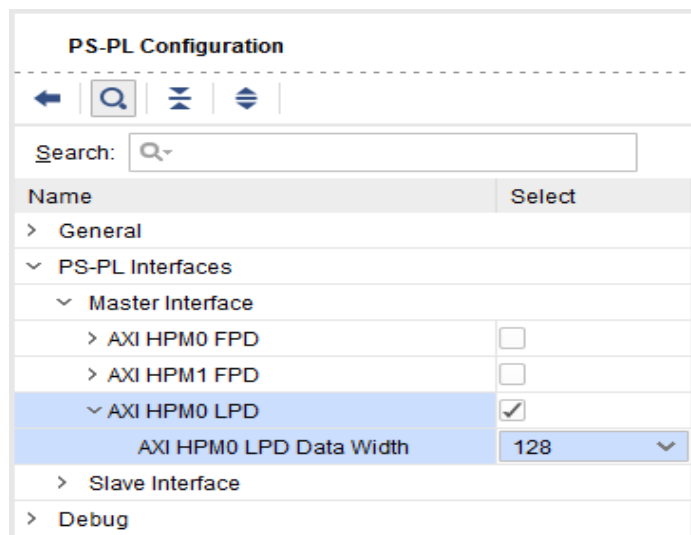


Figure 6-7: Set Data Width for AXI HPM0 LPD

- Click **OK** to complete the configuration and return to the block diagram.
- In the diagram view, connect the interrupt port from `axi_timer_0` to `pl_ps_irq[0:0]`.
- Click **Run Connection Automation**. Do not click on Run Block Automation.

 Designer Assistance available. [Run Block Automation](#) [Run Connection Automation](#)

Figure 6-8: Run Connection Automation Link

- In the Run Connection Automation dialog box, click **All Automation**.
- Click **OK**.
- In the Address Editor view, verify that the corresponding IPs are allocated the same Address Map, as shown in the following figure. If not, set the offset address such that they match the following figure.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits : 0x0080000000 [ 512M ])					
axi_gpio_0	S_AXI	Reg	0x00_8000_1000	4K	0x00_8000_1FFF
axi_gpio_1	S_AXI	Reg	0x00_8000_2000	4K	0x00_8000_2FFF
axi_timer_0	S_AXI	Reg	0x00_8000_0000	4K	0x00_8000_0FFF

Figure 6-9: Address Map for PL IPs

10. Validate the design and generate the output files for this design, as described in the following sections.

### Validating the Design and Generating Output

1. Return to the block diagram view and save the Block Design (press **Ctrl + S**).
2. Right-click in the white space of the Block Diagram view and select **Validate Design**.

Alternatively, you can press the **F6** key.

A message dialog box opens and states "Validation successful. There are no errors or critical warnings in this design."

3. Click **OK** to close the message.
4. In the Block Design view, click the **Sources** tab.
5. Click **Hierarchy**.
6. In the Block Diagram, Sources window, under Design Sources, expand `edt_zcu102_wrapper`.
7. Right-click the top-level block diagram, titled `edt_zcu102_i : edt_zcu102` (`edt_zcu102.bd`) and select **Generate Output Products**.

The Generate Output Products dialog box opens, as shown [Figure 6-10](#).

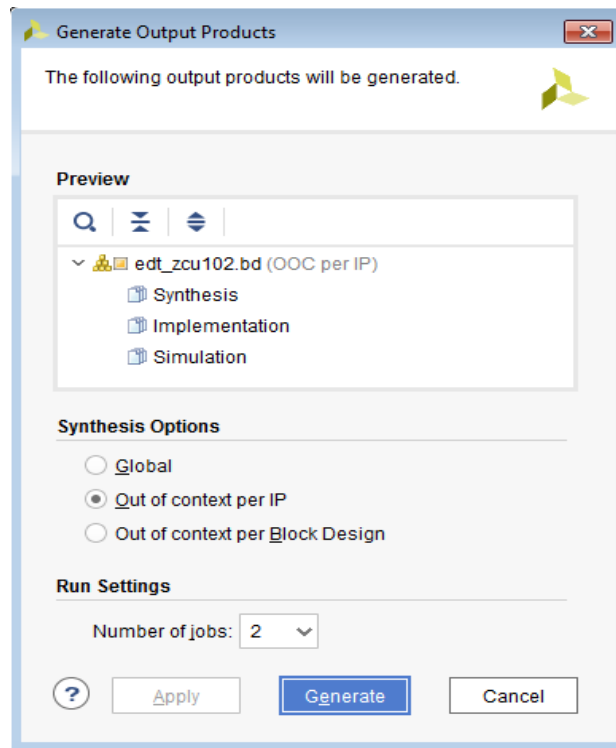


Figure 6-10: Generate Output Products Dialog Box

**Note:** If you are running the Vivado Design Suite on a Linux host machine, you might see additional options under Run Settings. In this case, continue with the default settings.

8. Click **Generate**.
9. When the Generate Output Products process completes, click **OK**.
10. In the Block Diagram Sources window, click the **IP Sources** tab. Here you can see the output products that you just generated, as shown in the following figure.



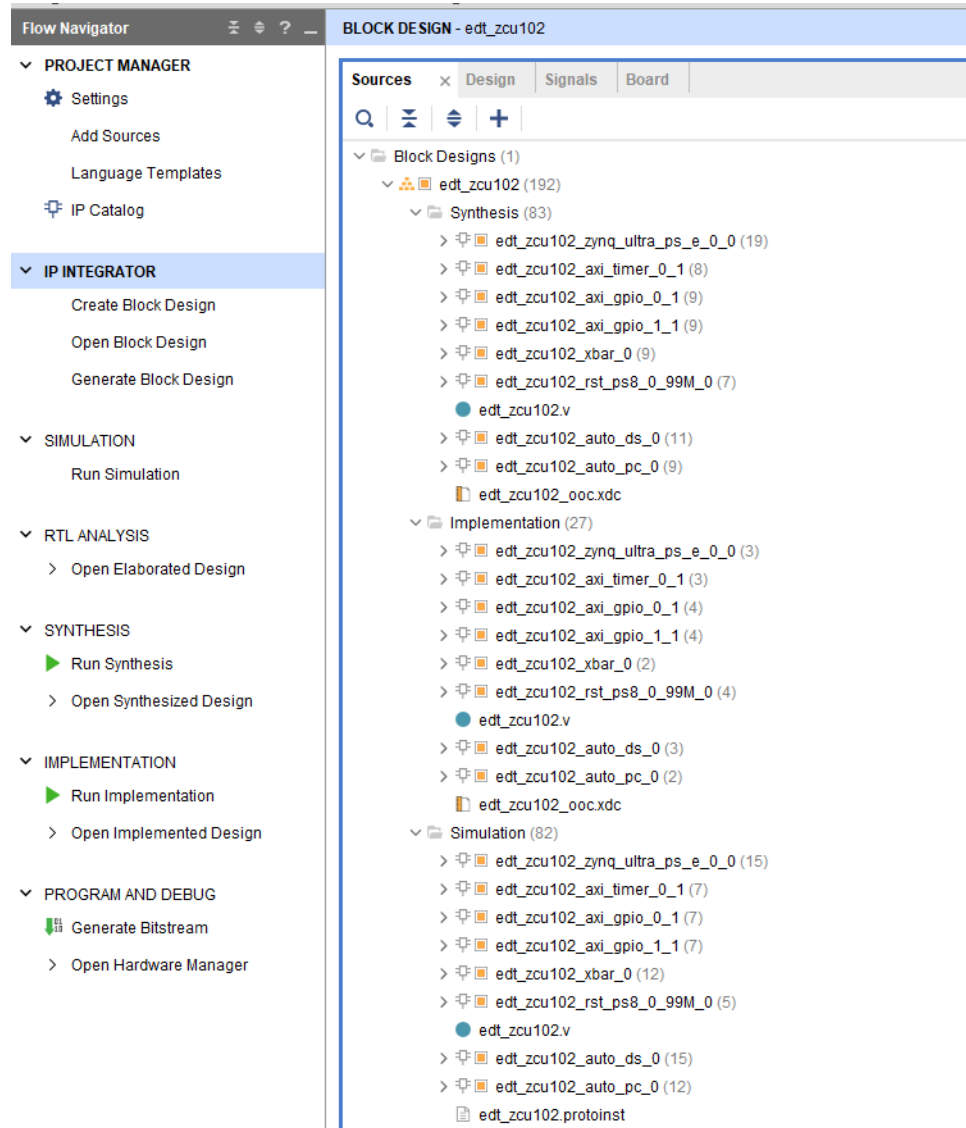


Figure 6-11: Outputs Generated Under IP Sources

### ***Synthesizing the Design, Running Implementation, and Generating the Bitstream***

1. You can now synthesize the design. In the Flow Navigator pane, under **Synthesis**, click **Run Synthesis**.

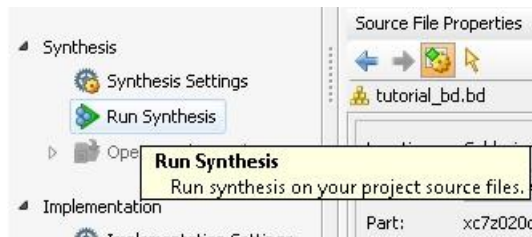


Figure 6-12: Run Synthesis Button

2. If Vivado prompts you to save your project before launching synthesis, click **Save**.

While synthesis is running, a status bar displays in the upper right-hand window. This status bar spools for various reasons throughout the design process. The status bar signifies that a process is working in the background.



Figure 6-13: Status Bar

When synthesis completes, the Synthesis Completed dialog box opens.

3. Select Run Implementation and click **OK**.

Again, notice that the status bar describes the process running in the background. When implementation completes, the Implementation Completed dialog box opens.

4. Select Generate Bitstream and click **OK**.

When Bitstream Generation completes, the Bitstream Generation Completed dialog box opens.

5. Click **Cancel** to close the window.
6. After the Bitstream generation completes, export the hardware to the Vitis IDE.

### Exporting Hardware

1. From the Vivado toolbar, select **File > Export > Export Hardware**.

The Export Hardware dialog box opens. Make sure that the **Include bitstream** check box is checked (only when design has PL design and bitstream generated), and that the **Export to** field is set to the local path.

2. Click **OK**.

3. To update the hardware in the Vitis IDE, go to **edt\_zcu102\_wrapper** platform project. Right-click and select **Update the Hardware**, then provide the path to the hardware exported from Vivado.

## Configuring Software

This use case has a bare-metal application running on an R5 core and a Linux Application running on APU Linux Target. Most of the software blocks will remain the same as mentioned in [Chapter 3](#). The software for this design example requires additional drivers for components added in the PL Logic. For this reason, you will need to generate a new Bare-metal BSP in the Vitis IDE using the Hardware files generated for this design. Linux also requires the Linux BSP to be reconfigured in sync with the new hardware design file (HDF). Before you configure the software, first look at the application design scheme. The system has a bare-metal application on RPU, which starts with toggling the PS LEDs for a user configurable period. The LEDs are set to toggle in synchronization with PL AXI Timer running in the PL block. The application sets the AXI Timer in the generate mode and generates an interrupt every time the Timer count expires. The application is designed to toggle the PS LED state after handling the Timer interrupt. The application runs in an infinite while loop and sets the RPU in WFI mode after toggling the LEDs for the user-configured time period. This LED toggling sequence can be repeated again by getting the RPU out of WFI mode using an external interrupt. For this reason, the UART interrupt is also configured and enabled in the same application. While this application runs on the RPU, the Linux target also hosts another Linux application. The Linux application uses user Input from PS or PL switches to toggle PL LEDs. This Linux application also runs in an infinite while loop, waiting for user input to toggle PL LEDs. The next set of steps show how to configure System software and build user applications for this design.

### ***Configure and Build Linux using PetaLinux***

First, create the Linux images using PetaLinux. The Linux images must be created in sync with the hardware configuration for this design. You will also need to configure PetaLinux to create images for SD boot.

See the [Example Project: Create Linux Images using PetaLinux in Chapter 3](#), and repeat steps from [step 2](#) to [step 13](#) to update the device tree and build Linux images using PetaLinux. Alternatively, you can also use the Linux image files shared with this tutorial. The images for this section can be found in `<design_files>/design`.

Follow [step 15](#) to verify the images. The next step is to create a Bare-metal Application targeted for Arm Cortex-R5 based RPU.

For this design example, you must import the application source files available in the Design Files ZIP file released with this tutorial. For information about locating these design files, see the [Design Files for This Tutorial in Appendix B](#).

## Creating the Bare-Metal Application Project

1. In the Vitis IDE, select **File > New > Application Project**.

The New Project wizard opens.

2. Use the information in the table below to make your selections in the wizard.

**Table 6-1: Settings to Create Timer-Based RPU Application Project**

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	<code>tmr_psled_r5</code>
	Use Default Location	Select this option.
	System Project	Create New
	Platform	<code>edt_zcu102_wrapper</code>
	Domain	<code>psu_cortexr5_0</code>
	Language	<b>C</b>
	CPU	<code>psu_cortexr5_0</code>
	OS	Standalone
Templates	Available Templates	<b>Empty Application</b>

3. Click **Finish**.

The New Project wizard closes and the Vitis IDE creates the `tmr_psled_r5` application project, which you can view in the Project Explorer.

4. In the Project Explorer tab, expand the `tmr_psled_r5` project.
5. Right-click the `src` directory, and select **Import** to open the Import dialog box.
6. Expand **General** in the Import dialog box and select **File System**.
7. Click **Next**.
8. Select **Browse** and navigate to the `design-files/design1` folder, which you saved earlier (see [Additional Resources and Legal Notices in Appendix B](#)).
9. Click **OK**.
10. Select and add the `timer_psled_r5.c` file.
11. Click **Finish**.

The Vitis IDE automatically builds the application and displays the status in the console window.

### Modifying the Linker Script

1. In the Project Explorer, expand the `tmr_psled_r5` project.
2. In the `src` directory, double-click `lscript.ld` to open the linker script for this project.

- In the linker script in Available Memory Regions, modify following attributes for psu\_r5\_ddr\_0\_MEM\_0 :

Base Address: **0x70000000**

Size: **0x10000000**

The following figure shows the linker script modification. The following figure is for representation only. Actual memory regions may vary in case of isolation settings.

### Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

#### Available Memory Regions

Name	Base Address	Size
psu_ocm_ram_0_MEM_0	0xFFFC0000	0x40000
psu_qspi_linear_0_MEM_0	0xC0000000	0x20000000
psu_r5_0_atcm_MEM_0	0x0	0x10000
psu_r5_0_btcm_MEM_0	0x20000	0x10000
psu_r5_ddr_0_MEM_0	0x70000000	0x10000000
psu_r5_tcm_ram_0_MEM_0	0x0	0x40000

#### Stack and Heap Sizes

Stack Size

Heap Size

Figure 6-14: Linker Script Modification

This modification in the linker script ensures that the RPU bare-metal application resides above 0x70000000 base address in the DDR, and occupies no more than 256 MB of size.

- Type **Ctrl + S** to save the changes.
- Right-click the `tmr_ps1ed_r5` project and select **Build Project**.
- Verify that the application is compiled and linked successfully and that the `tmr_ps1ed_r5.elf` file was generated in the `tmr_ps1ed_r5\Debug` folder.
- Verify that the BSP is configured for UART\_1. For more information, see the [Modifying the Board Support Package in Chapter 3](#).

### Creating the Linux Domain for Linux Applications

To create a Linux domain for Linux applications, follow these steps:

1. Go to the explorer in the Vitis IDE and expand the **edt\_zcu102\_wrapper** platform project.
2. Open the hardware by double-clicking **platform.spr**. The Platform explorer opens.
3. Click + button on the top-right corner to add the domain.

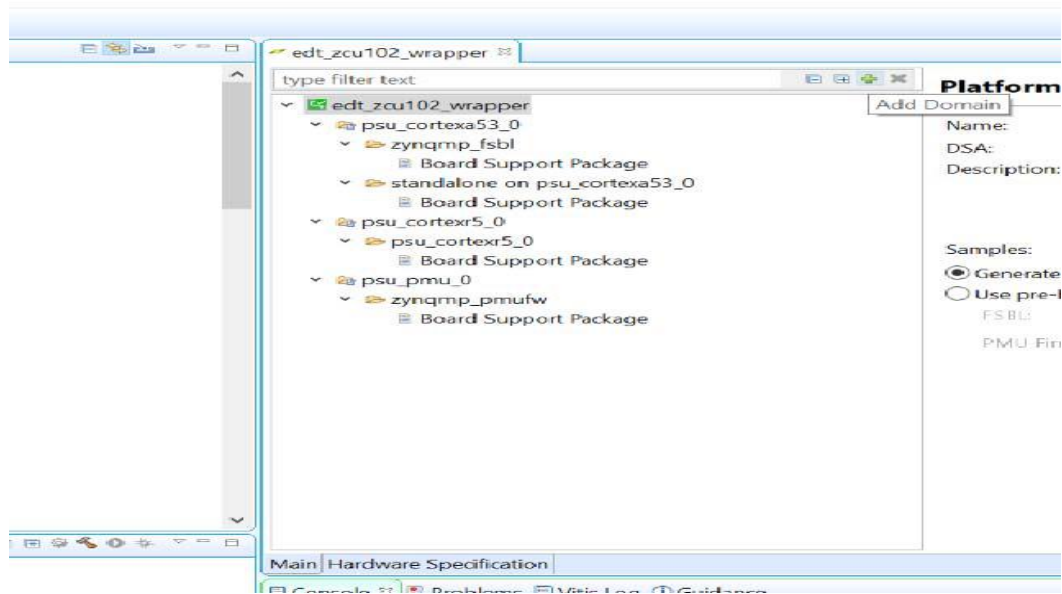


Figure 6-15: Adding a Linux Domain

4. Enter the following details in the new domain window.

Table 6-2: Settings to Create New Linux Domain

System Properties	Setting or Command to Use
Name	Linux Domain
Display Name	Linux_application_domain
OS	Linux
Processor	psu_cortexa53_0
Supported Run times	C/C++

5. Select Use pre-built software components. Then do the following:
  - a. Create a boot directory and copy the boot components (FSBL, PMU firmware from the Vitis IDE, ATF, u-boot.elf, and image.ub from PetaLinux) generated from the Vitis tool and the PetaLinux tool.
  - b. Copy the bif file which is supplied with the design files located in design1 and provide the bif file path.

6. Click **OK** to finish.

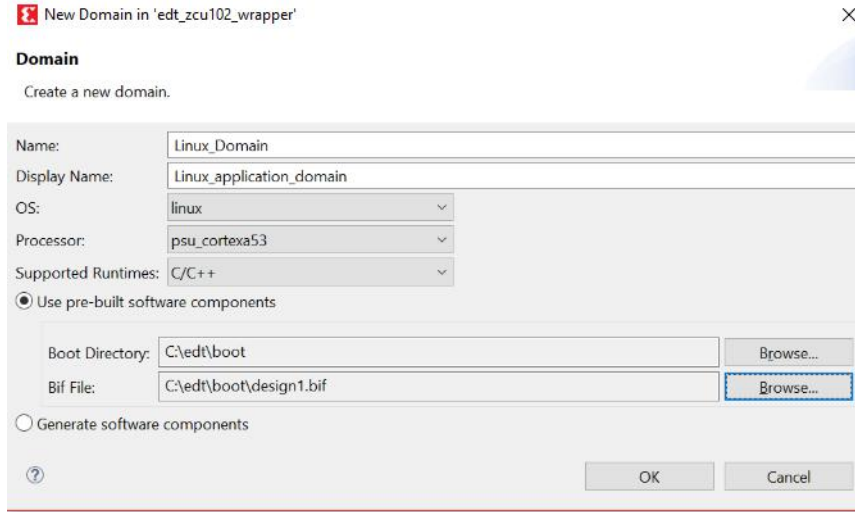


Figure 6-16: Creating a Linux Application Domain

You can see the Linux domain properties in the window as shown below.

7. Add image.ub file path in the image file path.

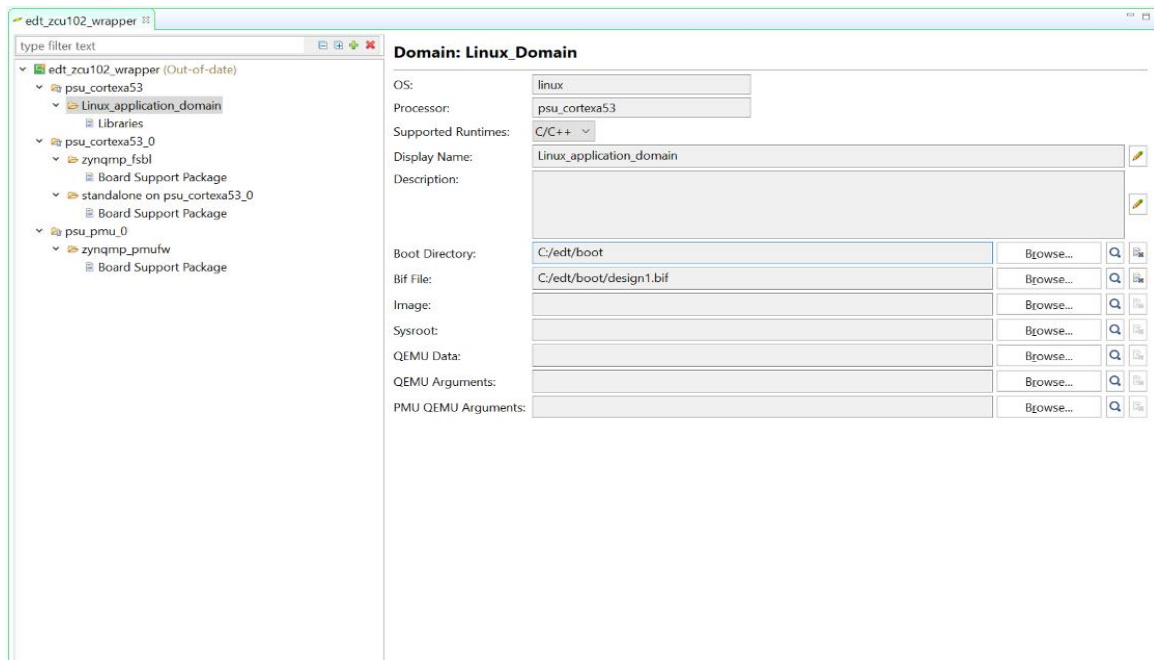


Figure 6-17: Linux Domain Properties

The Linux domain is ready and you can create Linux applications.

## Creating the Linux Application Project

1. In the Vitis IDE, select **File > New > Application Project**.

The New Project wizard opens.

2. Use the information in the table below to make your selections in the wizard.

**Table 6-3: Settings to Create New Linux Application Project**

Wizard Screen	System Properties	Setting or Command to Use
Application Project	Project Name	<code>ps_pl_linux_app</code>
	Use Default Location	Select this option
	System Project	Create New
	Platform	<code>edt_zcu102_wrapper</code>
	Domain	Linux_application_domain
	Language	C
	CPU	cortex-a53
	OS	Linux
Templates	Available Templates	<b>Linux Empty Application</b>

3. Click **Finish**.

The New Project wizard closes and the Vitis IDE creates the `ps_pl_linux_app` application project, which can be found in the Project Explorer.

4. In the Project Explorer tab, expand the `ps_pl_linux_app` project.
5. Right-click the `src` directory, and select **Import** to open the Import dialog box.
6. Expand **General** in the Import dialog box and select **File System**.
7. Click **Next**.
8. Select **Browse** and navigate to the `design-files/design1` folder, which you saved earlier (see [Design Files for This Tutorial in Appendix B](#)).
9. Click **OK**.
10. Select and add the `ps_pl_linux_app.c` file.

**Note:** The application might fail to build because of a missing reference to the pthread Library. The next section shows how to add the pthread library.



### Modifying the Build Settings

This application makes use of Pthreads from the pthread library. Add the pthread library as follows:

1. Right-click `ps_pl_linux_app`, and click on **C/C++ Build Settings**.
2. Refer to the following figures to add the `pthread` library.

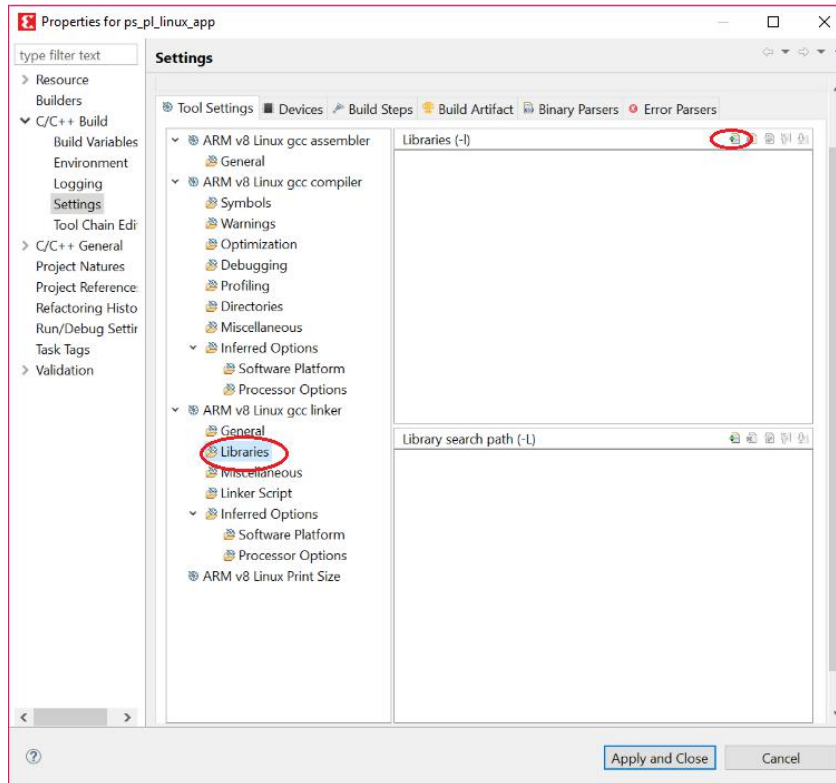


Figure 6-18: C/C++ Build Settings

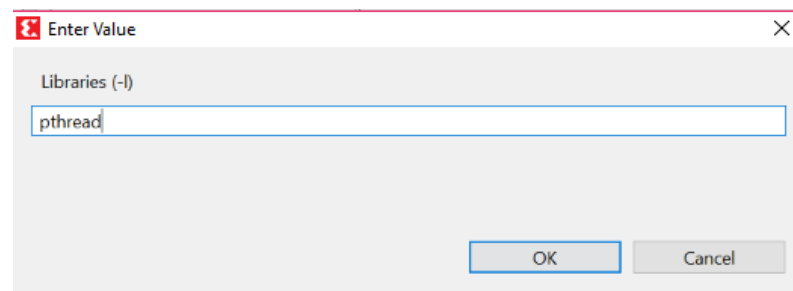


Figure 6-19: Add pthread Library

3. Click **OK** in both the windows.

Then build the application by right-clicking on the application selecting **Build option**.

### Creating a Boot Image

Now that all the individual images are ready, you will create the boot image to load all of these components on a Zynq UltraScale+ device. This can be done using the Create Boot Image wizard in the Vitis IDE, using the following steps. This example creates a Boot Image BOOT.bin in `c:\edt\design1`.

1. Launch the Vitis IDE, if it is not already running.
2. Set the workspace based on the project you created in [Chapter 2](#). For example:  
`c:\edt`
3. Select **Xilinx > Create Boot Image**.
4. See [Figure 6-20](#) for settings in the Create Boot Image wizard.
5. Add the partitions as shown in the following figure.

**Note:** For detailed steps on how to add partitions, see [Boot Sequence for SD-Boot](#).

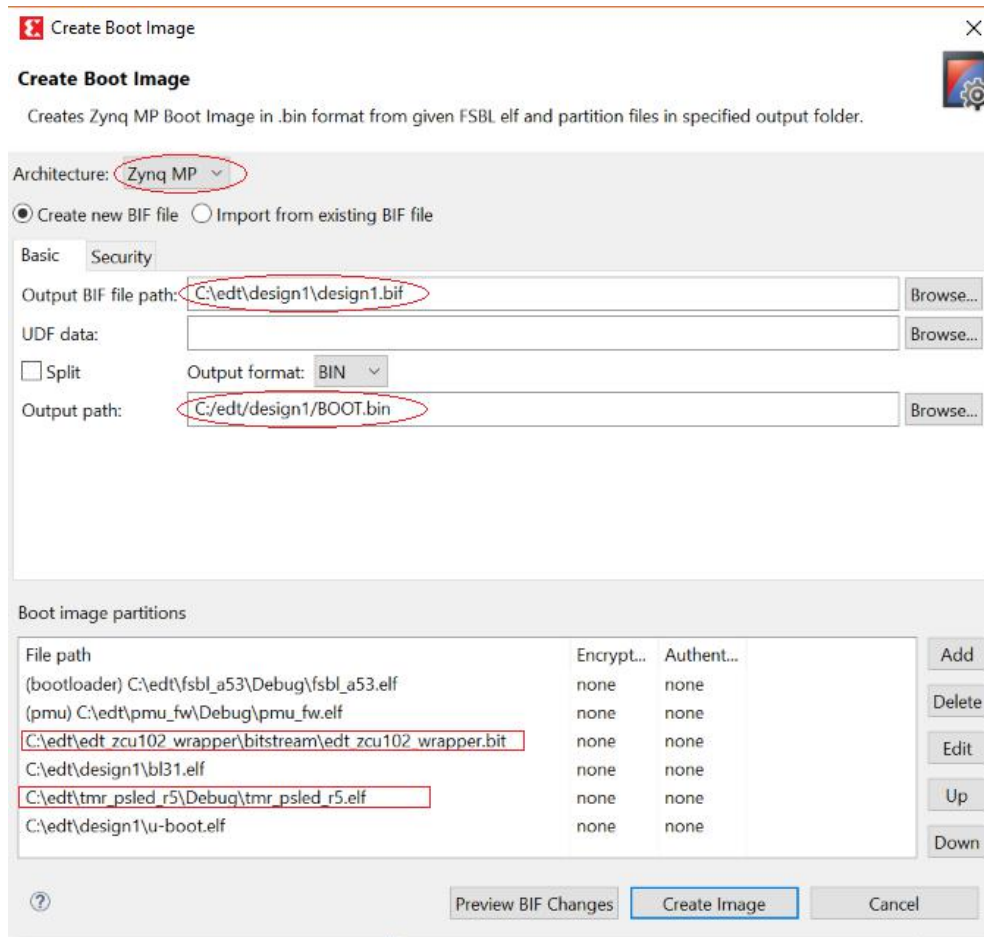


Figure 6-20: Create Boot Image for SD Boot Mode

**Note:** This Boot image requires PL bitstream `edt_zcu102_wrapper.bit` (Partition Type - Datafile, Destination Device - PL). The Bitstream Partition needs to be added right after the Bootloader while you create the boot image. Also note that the R5 application `tmr_ps1ed_r5.elf` is added as partition in this boot image.

- After adding all the partitions, click **Create Image**.



**IMPORTANT:** Ensure that you have set the correct exception levels for ATF (EL-3, Trustzone) and U-Boot (EL-2) partitions. These settings can be ignored for other partitions.

## Running the Image on a ZCU102 Board

### Prepare the SD Card

Copy the images and executables on an SD card and load it in the SD card slot in the Board.

- Copy files `BOOT.BIN` and `image.ub` to an SD card.

**Note:** `BOOT.BIN` is located in `C:\edt\design1`.

- Copy the Linux application, `ps_pl_linux_app.elf`, to the same SD Card. The application can be found in:

`c:\edt\ps_pl_linux_app\Debug`

### Target Setup

- Load the SD card into the ZCU102 board, in the J100 connector.
- Connect the USB-UART on the Board to the Host machine.
- Connect the Micro USB cable into the ZCU102 Board Micro USB port J83, and the other end into an open USB port on the host Machine.
- Configure the Board to Boot in SD-Boot mode by setting switch SW6 as shown in the following figure.

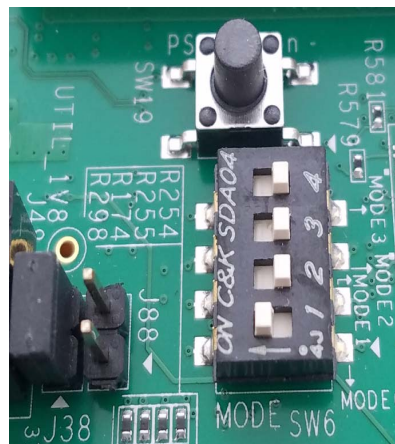


Figure 6-21: SW6 Switch Settings for SD Boot Mode

5. Connect 12V Power to the ZCU102 6-Pin Molex connector.
6. Start a terminal session, using TeraTerm or Minicom depending on the host machine being used, as well as the COM port and baud rate for your system, as shown in [Figure 5-8](#).
7. For port settings, verify the COM port in the device manager.

There are four USB-UART interfaces exposed by the ZCU102 Board.

8. Select the COM Port associated with the interface with the lowest number. In this case, for UART-0, select the COM port with interface-0.
9. Similarly, for UART-1, select COM port with interface-1.

Remember that the R5 BSP has been configured to use UART-1, and so R5 application messages will appear on the COM port with the UART-1 terminal.

### **Power ON Target and Run Applications**

1. Turn on the ZCU102 Board using SW1, and wait until Linux loads on the board.

You can see the initial Boot sequence messages on your Terminal Screen representing UART-0.

You can see that the terminal screen configured for UART-1 also prints a message. This is the print message from the R-5 bare-metal Application running on RPU, configured to use UART-1 interface. This application is loaded by the FSBL onto RPU.

2. Now that this application is running, notice the PS LED being toggled by the application, and follow the instructions in the application terminal.

```
PS LED Turned ON
PS LED Turned OFF
PS LED Turned ON
PS LED Turned OFF
PS LED Turned ON
RPU in WFI mode. Press any key to repeat the sequence
```

Figure 6-22: R5-0 Bare Metal Application

## Running Linux Applications

After Linux is up on the ZCU102 system, log in to the Linux target with login: **root** and password: **root**. The Linux target is now ready for running applications.

Run the Linux application using following steps.

1. Copy the application from SD card mount point to `/tmp`

```
# cp /run/media/mmcblk0p1/ps_pl_linux_app.elf /tmp
```

**Note:** Mount the SD card manually if you fail to find SD card contents in this location.

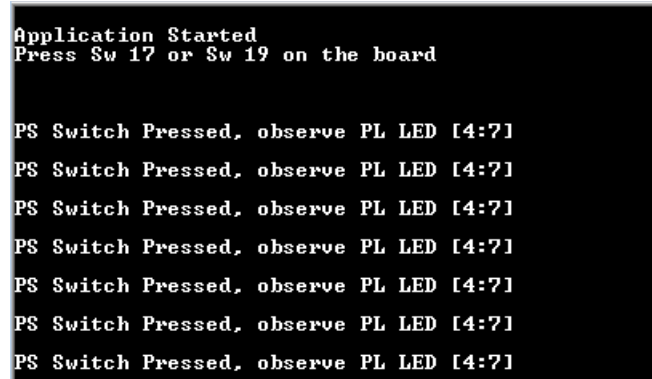
```
# mount /dev/mmcblk0p1 /media/
```

Copy the application to `/tmp`.

```
# cp /media/ps_pl_linux_app.elf /tmp
```

2. Run the application.

```
# /tmp/ps_pl_linux_app.elf
```



```
Application Started
Press Sw 17 or Sw 19 on the board

PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
PS Switch Pressed, observe PL LED [4:7]
```

Figure 6-23: Linux Terminal

## Design Example 2: Example Setup for Graphics and Display Port Based Sub-System

This design example is primarily based on the Graphics Processing Unit and the Display Port on a Zynq UltraScale+ MPSoC device. The main idea behind this example is to demonstrate the configurations, packages, and tool flow required for running designs based on GPU and DP on a Zynq UltraScale+ MPSoC device. This design example can be broken down into the following sections:

1. Configuring the hardware.
2. Configuring PetaLinux RootFS to include the required packages:
  - a. GPU related packages
  - b. X Window System and dependencies
3. Building Boot images and Linux images using PetaLinux.
4. Building a Graphics OpenGL ES application targeted for Mali GPU. This application is based on the X Window System.
5. Loading Linux on the ZCU102 board and running the Graphics Application on the target to see the result on the display port.

### Configuring the Hardware

In this section, you will configure the processing system to set Dual lower GT lanes for the display port. The hardware configuration in this section is based on the same Vivado project that you created in [Design Example 1: Using GPIOs, Timers, and Interrupts](#).

#### *Configuring Hardware in Vivado IP Integrator*

1. Ensure that the `edt_zcu102` project and the block design are open in Vivado.
2. Double-click the Zynq UltraScale+ Processing System block in the Block Diagram window and wait till the Re-customize IP dialog box opens.
3. In Re-customize IP window, click on **I/O Configuration > High Speed**
4. De-select PCIe peripheral connection
5. Expand Display Port, and set Lane Selection to Dual Lower, as shown in following figure:

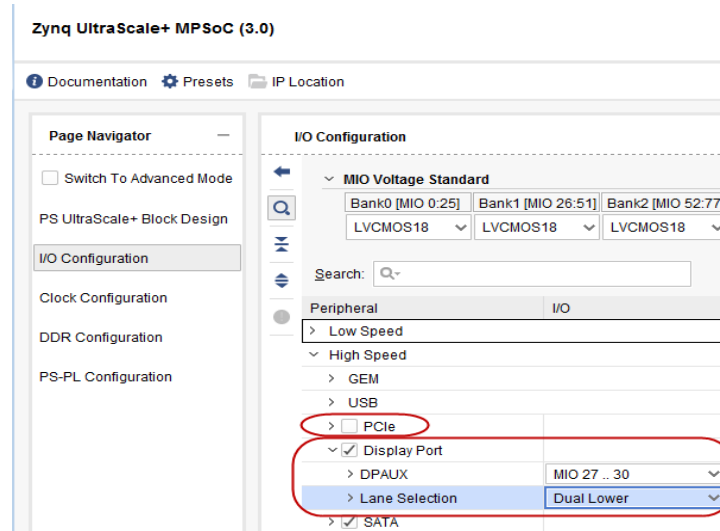


Figure 6-24: Display Port Lane Selection

**Note:** The Display port lane selection is set to Dual lane to support UHD@30 resolution in the design example of this tutorial. This configuration will lock display for UHD@30 as well as lower resolution like 1080p 60 and others, for corresponding monitors.

6. Click **OK** to close the Re-customize IP wizard.



**CAUTION!** Do not click the Run Block Automation link. Clicking the link will reset the design as per board preset and disable the design updates you made using in this section.

7. Click **File > Save Block Design** to save the block design. Alternatively, you can press **CTRL + S** to save the block design.
8. Click **Generate Bitstream**, to re-synthesize the design and generate the Bitstream.
9. After the Bitstream is generated successfully, click **File > Export > Export Hardware** to export the hardware design.
10. Select **Include Bitstream**.
11. Click **OK**.

Vivado generates the .xsa hardware file in the specified location.

12. Copy the XSA file to a Linux Host machine.

The next section describes steps to build Linux for your Hardware configuration and also add additional software packages for GPU and the X Window System.

## Modifying the Configuration and Building Linux Images using PetaLinux

Re-configure the PetaLinux BSP in sync with the new hardware changes. This section uses the PetaLinux project you created in [Example Project: Create Linux Images using PetaLinux](#).

1. Change to the PetaLinux directory using the following command:

```
$ cd xilinx-zcu102-2019.2
```

2. Copy the hardware platform `edt_zcu102_wrapper.xsa` in the Linux Host machine.
3. Reconfigure the BSP using the following command:

```
$ petalinux-config --get-hw-description=<path containing edt_zcu102_wrapper.xsa>/
```

The PetaLinux configuration wizard opens.

4. Save and exit the wizard without any additional configuration settings.

Wait until PetaLinux reconfigures the project.

5. Clean the existing Bootloader image. This is to ensure that the bootloader is recreated in sync with new hardware design.

```
$ petalinux-build -c bootloader -x distclean
```

## Building the Mali OpenGL ES Application

This section leads you through building a Triangle-based Cube application. This application is written in OpenGL ES and is based on the X Window System. For more details and for the application source code, refer to `tricube` in the `design_files` folder of the zip file that accompanies this tutorial. See [Design Files for This Tutorial](#).

Use the following steps to build the OpenGL ES application:

1. Copy the entire application source directory of `tricube` to the Linux host machine in the `recipe-apps` directory of the PetaLinux project.

```
<PetaLinux-Project>/project-spec/meta-user/recipes-apps/tricube
```

2. Add the newly created `tricube` in `user-rootfsconfig`, which is located in

```
<plnx_project>/project-spec/meta-user/conf/user-rootfsconfig
```

With this addition, the file will look like below. Notice the new application in bold.

```
CONFIG_gpio-demo
CONFIG_peekpoke
CONFIG_tricube
```

3. Refer to recipe `tricube/tricube.bb` for detailed instructions and libraries used for building this application. The X Window System (X11) packages included while building



the above application is application dependent. Libraries included in `tricube.bb` recipe are based on the packages that were used in the application.

## Enable GPU Libraries and Other Packages in RootFS

In this section, you will use the PetaLinux rootfs Configuration wizard to add the Mali GPU libraries. PetaLinux is shipped with Mali GPU libraries and device drivers for Mali GPU. By default, the Mali driver is enabled in the kernel tree, but Mali user libraries need to be configured (on an as-needed basis) in the rootfs. In addition to this, you will use the same wizard to include the X Window System libraries.

1. Open the PetaLinux rootfs Configuration wizard -

```
$ petalinux-config -c rootfs
```

2. Navigate to and enable the following packages:

```
Filesystem Packages ---> libs ---> libmali-xlnx ---> libmali-xlnx
Filesystem Packages ---> libs ---> libmali-xlnx ---> libmali-xlnx-dev
```

These packages enable you to build and Run OpenGL ES applications targeted for Mali GPU in the Zynq UltraScale+ MPSoC device.

3. Add X11 package groups to add X window related packages:

```
Petalinux Package Groups > packagegroup-petalinux-x11 >packagegroup-petalinux-x11
Petalinux Package Groups > packagegroup-petalinux-x11 >
packagegroup-petalinux-x11-dev
```

4. Add the OpenGL ES application created in the earlier section:

```
User Packages ---> [*]tricube
```

5. After enabling all the packages, save the config file and exit the rootfs configuration settings.

6. Build the Linux images using the following command:

```
$ petalinux-build
```

**Note:** If the PetaLinux build fails, use the following commands to build again:

```
$ petalinux-build -x mrproper
$ petalinux-build
```

7. Verify that the `image.ub` Linux image file is generated in the `images/linux` directory.

8. Generate the Boot image for this design example as follows:

```
$ petalinux-package --boot --fsbl images/linux/zynqmp_fsbl.elf --pmufw
images/linux/pmufw.elf --atf images/linux/bl31.elf --fpga images/linux/system.bit
--u-boot images/linux/u-boot.elf
```

A `BOOT.BIN` Boot image is created. It is composed of the FSBL boot loader, the PL bitstream, PMU firmware and ATF, and U-Boot. Alternatively, see the steps in [Creating a Boot Image](#) to create this boot image.



**IMPORTANT:** This example uses the GPU packages based on X window system, which is the default setting in PetaLinux 2019.2. To enable Frame Buffer fbdev based GPU Packages in PetaLinux 2019.2, add the following line in `<PetaLinux_project>/project-spec/meta-user/conf/petalinuxbsp.conf`.

```
DISTRO_FEATURES_remove_zynqmp = " x11 "
```

See example `eglfbdev` application (based on `fdev`) available in [Design Files for This Tutorial](#). For more information, see the Xilinx Answer [68821](#).

## Loading Linux and Running the OpenGL Application on the Target and Viewing the Result on the Display Port

### Preparing the SD Card

Now that the Linux images are built and the application is also built, copy the following images in an SD card and load the SD card in ZCU102 board.

- `BOOT.BIN`
- `image.ub`

### Running the Application on a Linux Target

#### Setting Up the Target

Do the following to set up the Target:

1. Load the SD card into the J100 connector of the ZCU102 board.
2. Connect the Micro USB cable into the ZCU102 Board Micro USB port J83, and the other end into an open USB port on the host Machine.

Also, make sure that the JTAG cable is disconnected. If the cable is not disconnected, the system might hang.

3. Connect a Display Port monitor to the ZCU102 Board. The display port cable from the DP monitor can be connected to the display port connector on the ZCU102 board.

**Note:** These images were tested on a UHD@30 Hz and a FullHD@60 Hz Display Port capable monitor.

- Configure the Board to Boot in SD-Boot mode by setting switch SW6 as shown in the following figure.

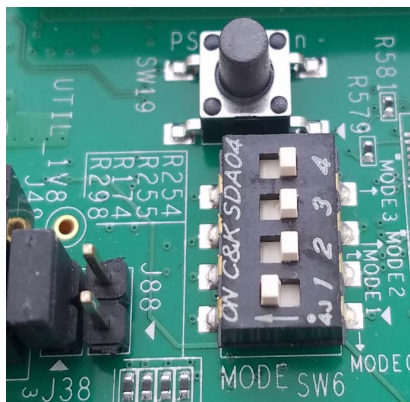


Figure 6-25: SW6 Switch Settings for SD Boot Mode

- Connect 12V Power to the ZCU102 6-Pin Molex connector.
- Start a terminal session, using TeraTerm or Minicom depending on the host machine being used, as well as the COM port and baud rate for your system, as shown in [Figure 5-8](#).
- For port settings, verify the COM port in the device manager.

There are four USB-UART interfaces exposed by the ZCU102 Board. Select the COM port associated with the interface with the lowest number. In this case, for UART-0, select the COM port with interface-0.

### Powering On the Target and Running the Applications

- Turn on the ZCU102 Board using SW1, and wait until Linux loads on the board.
- After Linux loads, log in to the target Linux console using `root` for the login and password.
- Set the display parameters and start Xorg with the correct depth.

```
# export DISPLAY=:0.0
# /usr/bin/Xorg -depth 16&
```

- Run the tricube application.

```
# tricube
```

At this point, you can see a rotating multi-colored cube and a rotating triangle on the display port. Notice that the cube is also made of multi-colored triangles.

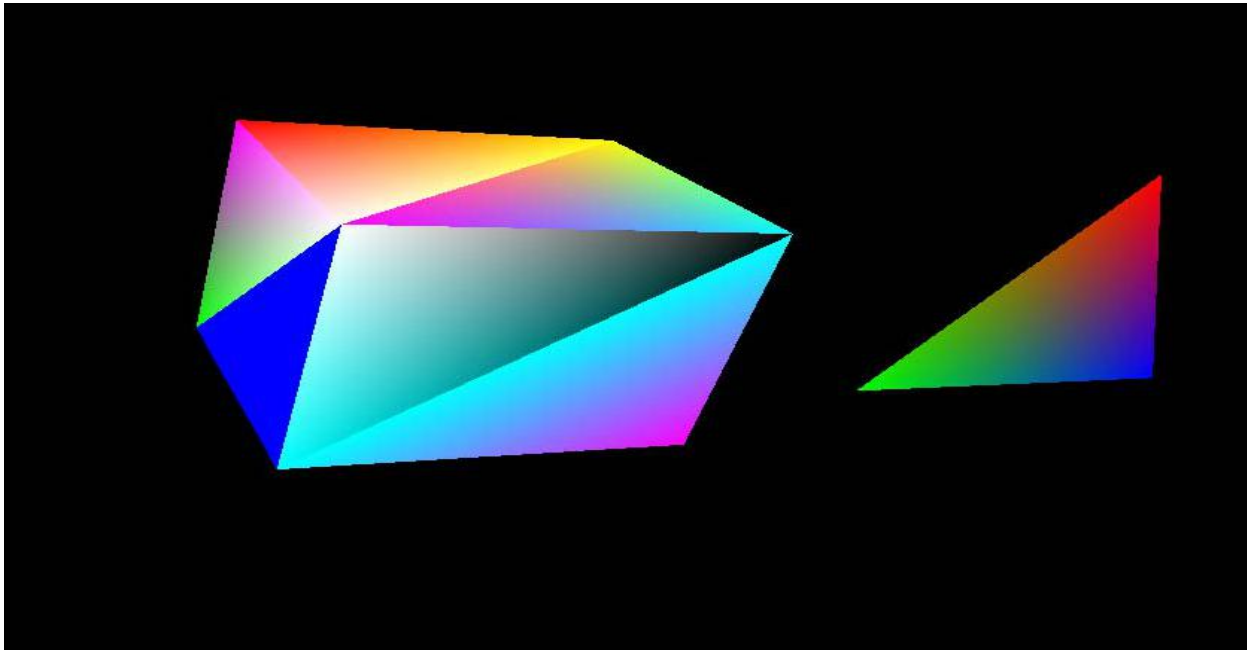


Figure 6-26: Rotating Cube and Triangle

# Debugging Problems with Secure Boot

This appendix describes how to debug security failures. One procedure determines if PUF registration has been run on the device. A second procedure checks the value of the Boot Header in the boot image.

---

## Determine if PUF Registration is Running

The following steps can be used to verify if the PUF registration software has been run on the device:

1. In the Vitis IDE, select **Xilinx > XSCT Console**.
2. Enter the following commands at the prompt:

```
xsct% connect
xsct% targets
xsct% targets -set -filter {name =~ "Cortex-A53 #0"}
xsct% rst -processor
xsct% mrd -force 0xFFCC1050 (0xFFCC1054)
```

3. This location contains the CHASH and AUX values. If non-zero, PUF registration software has been run on the device.

---

## Read the Boot Image

You can use the Bootgen Utility to verify the header values and the partition data used in the Boot Image.

1. Change to the directory containing **BOOT.bin**.
2. From an XSCT prompt, run the following command.

```
bootgen_utility -bin BOOT.bin -out myfile -arch zynqmp
```

3. Look for "BH" in myfile.

# Additional Resources and Legal Notices

---

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

---

## Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

---

## Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

**Note:** For more information on Documentation Navigator, see the [Documentation Navigator](#) page on the Xilinx website.

Available in Documentation Navigator, design hubs provide quick access to documentation, training, and information for specific design tasks. The following design hubs are applicable to embedded development and the methods described in this guide:

- [PetaLinux Tools Design Hub](#)
- [Vitis Embedded Software Development Flow Documentation](#)

---

## Design Files for This Tutorial

The ZIP file associated with this document contains the design files for the tutorial. You can download the [reference design files](#) from the Xilinx website.

To view the contents of the ZIP file, download and extract the contents from the ZIP file to `C:\edt`. The design files contain the HDF files, source code and prebuilt images for all the sections.

---

## Xilinx Resources

The following Xilinx Vivado Design Suite and Zynq® UltraScale+™ guides are referenced in this document.

1. *Vivado Design Suite User Guide: Getting Started* ([UG910](#))
2. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* ([UG940](#))
3. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
4. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
5. *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#))
6. *Zynq UltraScale+ MPSoC Software Developer Guide* ([UG1137](#))
7. *PetaLinux Tools Documentation: Reference Guide* ([UG1144](#))
8. *Zynq UltraScale+ Processing System Product Guide* ([PG201](#))
9. *Measured Boot of Zynq-7000 SoCs* ([XAPP1309](#))
10. *Secure Boot of Zynq-7000 SoC* ([XAPP1175](#))
11. *Changing the Cryptographic Key in Zynq-7000 SoC* ([XAPP1223](#))
12. *Programming BBRAM and eFUSES* ([XAPP1319](#))
13. [Vitis Embedded Software Development Flow Documentation](#)

## Support Resources

14. [Xilinx Zynq UltraScale+ MPSoC Solution Center](#)
15. The Software Zone:  
<https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html#docsdownload>

## Additional Resources

16. The Effect and Technique of System Coherence in Arm Multicore Technology by John Goodacre, Senior Program Manager, Arm Processor Division  
(<http://www.mpsoc-forum.org/previous/2008/slides/8-6%20Goodacre.pdf>)
  17. Xilinx GitHub website: <https://github.com/xilinx>
  18. The Linux Kernel Module Programming Guide:  
<http://ltdp.org/LDP/lkmpg/2.6/html/index.html>
- 

## Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related videos:

1. [Vivado Design Suite QuickTake Video Tutorials](#)



## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

### **AUTOMOTIVE APPLICATIONS DISCLAIMER**

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2017-2019 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, and MPCore are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.