

# Zynq-7000 SoC: Embedded Design Tutorial

# A Hands-On Guide to Effective System Design

UG1165 (2019.2) October 30, 2019





### **Revision History**

The following table shows the revision history for this document.

Date	Version	Revision	
10/30/2019	2019.2	Added support for the Vitis™ software platform.	
11/23/2017	2017.3	Verified for 2017.3 version of Vivado® Design Suite, Xilinx® SDK, and PetaLinux Tools.	



# **Table of Contents**

Revision History	. 2
Chapter 1: Introduction	
About This Guide	5
How Zynq Devices Simplify Embedded Processor Design	7
How the Vivado Tools Expedite the Design Process	8
What You Need to Set Up Before Starting	8
Chapter 2: Using the Zynq SoC Processing System	
Embedded System Configuration	12
Example Project: Creating a New Embedded Project with Zynq SoC	13
Example Project: Running the "Hello World" Application	29
Additional Information	32
Chapter 3: Using the GP Port in Zynq Devices	
Adding IP in PL to the Zynq SoC Processing System	34
Standalone Application Software for the Design.	50
Chapter 4: Debugging with the Vitis Software Platform	
Xilinx System Debugger	52
Debugging Software Using the Vitis Software Platform	54
Chapter 5: Using the HP Slave Port with AXI CDMA IP	
Integrating AXI CDMA with the Zynq SoC PS HP Slave Port	57
Standalone Application Software for the Design.	
Linux OS Based Application Software for the CDMA System	66
Running Linux CDMA Application Using the Vitis Software Platform	67
Chapter 6: Linux Booting and Debug in the Vitis Software Platform	
Requirements	80
Booting Linux on a Zynq SoC Board	81

www.xilinx.com



Chapter 7:	Creating	<b>Custom</b>	P and	Device	Driver	for Linux
------------	----------	---------------	-------	--------	--------	-----------

Requirements	103
Creating Peripheral IP	104
Integrating Peripheral IP with PS GP Master Port	109
Linux-Based Device Driver Development	112
Loading Module into Running Kernel and Application Execution	114

### **Chapter 8: Software Profiling Using the Vitis Software Platform**

Profiling an Application in the Vitis Software Platform with System Debugger	118
Additional Design Support Options	120

#### Chapter 9: Linux OS Aware Debugging Using the Vitis Software Platform

Setting Up Linux OS Aware Debugging	121
Debugging Linux Processes and Threads Using OS Aware Debug	124

#### Appendix A: Additional Resources and Legal Notices

Xilinx Resources	133
Solution Centers	133
Documentation Navigator and Design Hubs	133
Design Files for This Tutorial	134
Xilinx Resources	134
Training Resources	135
Please Read: Important Legal Notices	135





# Introduction

### **About This Guide**

This document provides an introduction to using the Xilinx® Vivado® Design Suite flow for using the Zynq®-7000 SoC device. The examples are targeted for the Xilinx ZC702 Rev 1.0 evaluation board and the tools used are the Vivado® Design Suite and the Vitis<sup>™</sup> unified software platform.

The examples in this document were created using the Xilinx tools running on Windows 7, 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.

### **Document Audience and Scope**

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

- Tutorials for creating a system with the Zynq-7000 SoC processing system (PS) and the programmable logic (PL)
- Tutorials on booting the Linux OS on the Zynq SoC board and application development with PetaLinux tools
- Tutorials on debugging in the Vitis integrated design environment (IDE)
- System design examples

### **Example Project**

The best way to learn a tool is to use it. So, 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**

#### Vivado Design Suite, System Edition

Xilinx offers a broad range of development system tools, collectively called the Vivado Design Suite. Various Vivado Design Suite editions can be used for embedded system development. In this guide, you will use the System Edition. The Vivado Design Suite editions are shown in the following figure.

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
Accelerating Implementat	ion				
Synthesis and Place and Route	•	•		•	•
Partial Reconfiguration*				•	
Accelerating Verification					
Vivado Simulator	•	•		•	•
Vivado Device Programmer	•	•	•	•	•
Vivado Logic Analyzer				1.61	
Vivado Serial I/O Analyzer		•		•	
Debug IP (ILA/VIO/IBERT)	•	•		•	•
Accelerating High Level De	esign				
Vivado High-Level Synthesis	•	•		•	•
Vivado IP Integrator	•	•		•	•
System Generator for DSP					

#### Vivado Design Suite - HLx Editions

\* Can be purchased as an option.



#### **Other Vivado Components**

Other Vivado components include:



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

#### Vitis Unified Software Platform

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. For more information about the Eclipse development environment, see http://www.eclipse.org.

#### PetaLinux Tools

For more information, see the Embedded Design Tools web page.

The PetaLinux Tools design hub provides information and links to documentation specific to the PetaLinux Tools. For more information, see Embedded Design Hub - PetaLinux Tools.

### How Zynq Devices Simplify Embedded Processor Design

Embedded systems are complex. Hardware and software portions of an embedded design are projects in themselves. Merging the two design components so that they function as one system creates additional challenges. Add an FPGA design project to the mix, and your design has the potential to become complicated.

The Zynq SoC solution reduces this complexity by offering an Arm<sup>®</sup> Cortex<sup>®</sup> -A9 dual core, along with programmable logic, all within a single SoC.

To simplify the design process, Xilinx offers the Vivado Design Suite and the Vitis software platform. 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 offers hardware and software application design, debugging capability, code execution, and transfer of the design onto actual boards for verification and validation.



### How the Vivado 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 microprocessor, peripherals, and the interconnection of these components, along with their respective detailed configuration.

The Vitis software platform is used for software development, and can be installed and used without any other Xilinx tools installed on the machine on which it is loaded. The Vitis software platform can also be used to debug software applications.

The Zynq SoC Processing System (PS) can be booted and made to 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 in the Vitis software platform.

For more information on the embedded design process, see the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) [Ref 5].

### 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 those required for the "Example Project" sections of this guide.

### Hardware Requirements for this Guide

This tutorial targets the Zynq ZC702 Rev 1.0 evaluation board, and can also be used for Rev 1.0 boards. To use this guide, you need the following hardware items, which are included with the evaluation board:

- The ZC702 evaluation board
- AC power adapter (12 VDC)
- USB Type-A to USB Mini-B cable (for UART communications)
- USB Type-A to 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



### **Installation Requirements**

### Vitis Software Platform and Vivado Design Suite

Ensure that you have both the Vitis software platform and the Vivado Design Suite installed. Visit the Xilinx Support Page to ensure that you download the latest software version. To install the Vitis software platform, follow the instructions in the Installation section of the *Vitis Embedded Software Development Flow Documentation* (UG1400) [Ref 10]. When you install the Vitis software platform, the Vivado Design Suite is installed automatically.

To install Vivado by itself, see the Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973) [Ref 6].

😢 Vitis IDE 2019.2 Installer - Vitis Development Environments	
Vitis Development Environments Customize your instaliation by (de)selecting items in the tree below. Noving the cursor over the selections below provides additional informabi	👷 🎗 XILINX
Choose the right devices for your target device.	- 10-
Design Tools     Vits Development Environment IDE     Dockav     Dockav     Process     Total levices for Alveo and Xilinx edge acceleration platforms     Devices     Socs     Zynd-7000     Zynd-7000     Zynd UtraScale+ MPSoC     Zynd UtraScale+ MPSoC     Zynd UtraScale+ MPSoC     Zynd UtraScale+ MPSoC     Jostal Lesie Chriefs of Custom Platforms     brotalistion Options     More and Xilinx edge acceleration platforms     brotalistion Options     Jostal Cable Chriefs of Custom Platforms     brotalistion Options     Jostal Kole Vite MUST disconnect all Xilinx Platform Cable USE II cables before proceeding)     Acquire or Manage a License Key     Enable WebTak for Vivado to send usage statistics to Xilinx (Always enabled for WebPACK license)     Jostal VincPacific Deference Hardware Co-Smutbation     Launch configuration manager to associate System Generator for DSP with MATLAB	
Download Size: 14,08 GB Disk Space Required: 66,87 GI	Reset to Defaults
Copyright © 1986-2019 Xilinx, Inc. All rights reserved.	<gack next=""> Cancel</gack>

Figure 1-2: Vitis Software Platform 2019.2 Installer - Select Development Environment



### PetaLinux Tools

The PetaLinux tool offers a full Linux distribution which includes the Linux OS as well as a complete configuration, build, and deploy environment for Xilinx silicon.

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 7.2, 7.3, 7.4, 7.5 (64-bit)
- CentOS 7.2, 7.3, 7.4, 7.5 (64-bit)
- Ubuntu Linux 16.04.3, 16.04.4 (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 software (version 2019.2) from the Xilinx Website.
- Install the PetaLinux (version 2019.2) release package.
- Add common system packages and libraries to the workstation or virtual machine. For more details, see the Installation Requirements from *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 8].

#### Prerequisites

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

#### Extract the PetaLinux Package

By default, the installer installs the package as a subdirectory within the current directory. Alternatively, you can specify an installation path. Run the downloaded PetaLinux installer.

*Note:* Ensure that the PetaLinux installation path is kept short. The PetaLinux build will fail if the path exceeds 255 characters.

bash> ./petalinux-v2019.2-final-installer.run

PetaLinux is installed in the petalinux-v2019.2-final directory, directly underneath the working directory of this command. If the installer is placed in the home directory /home/user, PetaLinux is installed in /home/user/petalinux-v2019.2-final.

Refer to Chapter 6, Linux Booting and Debug in the Vitis Software Platform for additional information about the PetaLinux environment setup, project creation, and project usage examples. A detailed guide on PetaLinux Installation and usage can be found in the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 8].





### 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 6].

### **Tutorial Design Files**

See Design Files for This Tutorial, page 134 for information about downloading the design files for this tutorial.





### Chapter 2

# Using the Zynq SoC Processing System

Now that you have been introduced to the Xilinx<sup>®</sup> Vivado<sup>®</sup> Design Suite, you will begin looking at how to use it to develop an embedded system using the Zynq<sup>®</sup>-7000 SoC Processing System (PS).

The Zynq SoC consists of Arm<sup>®</sup> Cortex<sup>®</sup>-A9 cores, many hard intellectual property components (IPs), and programmable logic (PL). This offering can be used in two ways:

- The Zynq SoC 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 PS as a PS+PL combination.

### **Embedded System Configuration**

Creation of a Zynq device 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.





# Example Project: Creating a New Embedded Project with Zynq SoC

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.

Wizard Screen	System Property	Setting or Command to Use	
Project Name	Project name	edt_tutorial	
	Project Location	C:/designs	
	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.	RTL Project	
	Do not specify sources at this time 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		Boards	
	Board	ZYNQ-7 ZC702 Evaluation Board	
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 an Embedded Processor Project**

Perform the following steps to create an embedded processor project.

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



IP Integrator      Create Block Design	Gimulation Sources (1) Gimulation Sources (1) Gimulation Sources (1)
Solution State Block D Create Block	ock Design
Create a	nd add an IP subsystem to the project.

Figure 2-1: Create Block Design Button

The Create Block Design wizard opens.

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

Wizard Screen	System Property	Setting or Command to Use	
Create Block Design	Design Name	tutorial_bd	
	Directory	<local project="" to=""></local>	
	Specify Source Set	Design Sources	

3. 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.

- 4. Click the **Add IP** button 🤔.
- 5. In the search box, type zyng to find the Zyng device IP options.
- 6. Double-click the **ZYNQ7 Processing System** IP to add it to the Block Design.

The Zynq SoC processing system IP block appears in the Diagram view, as shown in Figure 2-2.



Dlagram × Address Editor ×	2 0 0
Q, Q, X, X, O, Q, X, ⊕ + ∞, P, B, P, C, G, = Default View ∨	0
* Designer Assistance available. Run Block Automation	
processing_system7_0	
DDR +	
- MAXEGAD YORK SAND +	
FCLK_CLK0	
FCLK_RESETO_N 🔈	
ZYNQ7 Processing System	

Figure 2-2: Zynq SoC Processing System IP Block

### Managing the Zynq7 Processing System in Vivado

Now that you have added the processor system for the Zynq SoC to the design, you can begin managing the available options.

1. Double-click the **ZYNQ7 Processing System** block in the Block Diagram window.



The Re-customize IP dialog box opens, as shown Figure 2-3. Notice that by default, the processor system does not have any peripherals connected.

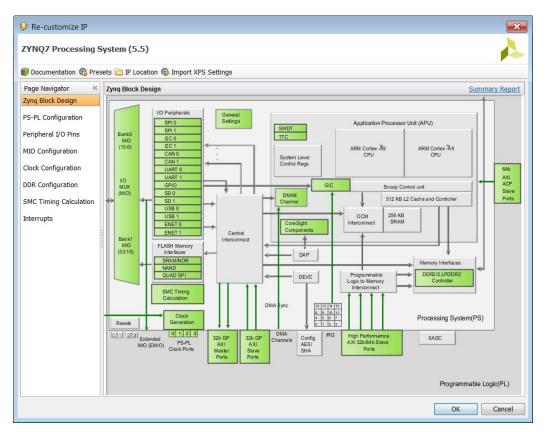


Figure 2-3: Re-Customize IP Dialog Box

2. You will use a preset template created for the ZC702 board. In the Re-customize IP window, click the **Presets** button and select **ZC702**.

This configuration wizard enables many peripherals in the Processing System with some multiplexed I/O (MIO) pins assigned to them as per the board layout of the ZC702 board. For example, UART1 is enabled and UART0 is disabled. This is because UART1 is connected to the USB-UART connector through UART to the USB converter chip on the ZC702 board.

Note the check marks that appear next to each peripheral name in the Zynq device block diagram that signify the **I/O Peripherals** that are active.

www.xilinx.com



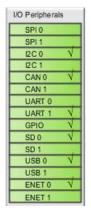


Figure 2-4: I/O Peripherals with Active Peripherals Identified

3. In the block diagram, click one of the green I/O Peripherals. The MIO Configuration window opens for the selected peripheral.

Documentation 🔞 Pr	resets 🛅 IP Location ෯ Import XP	PS Settings					
Page Navigator 🛛 🔍	MIO Configuration				<u>5</u> .	ummary R	eport.
Ynq Block Design 25-PL Configuration	Bank 0 I/O Voltage LVCMOS 1.	8V 👻	Bank 1	I/O Voltage	LVCMOS 1.8V		
Peripheral I/O Pins	😝 Peripheral	ю		Signal	ІО Туре		Spee
4IO Configuration	Memory Interfaces						
lock Configuration		MIO 16 27	*	]			
DR Configuration	🕀 🔲 ENET 1						
MC Timing Calculation	🕀 🔽 USB 0	MIO 28 39	*	1			
	- 🔲 USB 1						
Interrupts	🕀 - 🔽 SD 0	MIO 40 45	*	1			
	🕀 🔲 SD 1						
	🕀 🥅 UART O						
	🕀 💟 UART 1	MIO 48 49	•				
	😑 💟 12C 0	MIO 50 51	*				
	- 📄 Interrupt						
	I2C 0	MIO 50		scl	LVCMOS 1.8V	*	slov
	I2C 0	MIO 51		sda	LVCMOS 1.8V	*	slov
	🕀 📄 I2C 1						
	🖶 🔲 SPI 0						
	🕀 📄 SPI 1						
	🕀 🔽 CAN 0	MIO 46 47	*				
	🕀 📄 CAN 1						
	GPIO						

*Figure 2-5:* **MIO Configuration Window** 

4. Click **OK** to close the Re-customize IP wizard. Vivado implements the changes that you made to apply the ZC702 board presets.



In the Block Diagram window, notice the message stating that Designer assistance is available, as shown in the following figure.



Figure 2-6: Run Block Automation Link

5. Click the **Run Block Automation** link.

The Run Block Automation dialog box opens.

Note that Cross Trigger In and Cross Trigger Out are disabled. For a detailed tutorial with information about cross trigger set-up, refer to the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) [Ref 5].

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

### Validating the Design and Connecting Ports

Now, 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 critical error message appears, indicating that the M\_AXI\_GP0\_ACLK must be connected.

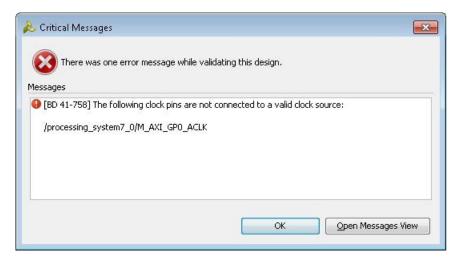


Figure 2-7: Critical Message Dialog Box

3. Click **OK** to clear the message.



- In the Block Diagram view of the ZYNQ7 Processing System, locate the M\_AXI\_GP0\_ACLK port. Hover your mouse over the connector port until the pencil icon appears.
- 5. Click the **M\_AXI\_GP0\_ACLK** port and drag to the **FCLK\_CLK0** input port to make a connection between the two ports.

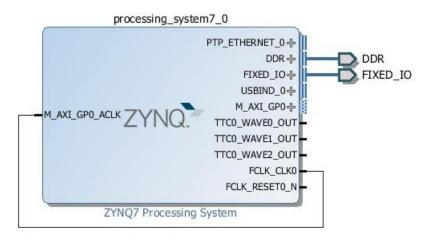


Figure 2-8: ZYNQ7 Processing System with Connection

6. Validate the design again to ensure there are no other errors. To do this, right-click in the white space of the Block Diagram view and select **Validate Design**.

A message dialog box opens and states "Validation successful. There are no errors or critical warnings in this design."

- 7. Click **OK** to close the message.
- 8. In the Block Design view, click the **Sources** tab.
- 9. Click Hierarchy.
- 10. Under Design Sources, right-click tutorial\_bd 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.

- 11. Select Let Vivado manage wrapper and auto-update and click OK.
- 12. In the Block Diagram, Sources window, under **Design Sources**, expand **tutorial\_bd\_wrapper**.



13. Right-click the top-level block diagram, titled **tutorial\_bd\_i - tutorial\_bd** (tutorial\_bd.bd) and select Generate Output Products.

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

revie	w		
Q,	Ŧ	\$	
× a	e tute	orial_bd.bd (OOC per IP)	
	🗇 S	ynthesis	
	🗐 Ir	nplementation	
	<b>回</b> S	imulation	
	₿H	w_Handoff	
ynth	esis Ol		
		f context per IP	
-		f context per <u>B</u> lock Design	
0	ouro	reenten per <u>p</u> roce o congri	
tun S	etting	5.	
		of jobs: 2	

Figure 2-9: Generate Output Products Dialog Box

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.

14. 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 sub-system when **Generate Output Products** is selected.

- 15. When the Generate Output Products process completes, click **OK**.
- 16. 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.

### **E** XILINX<sub>®</sub>

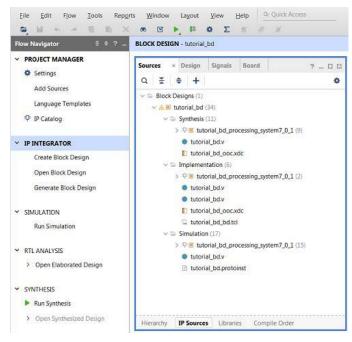


Figure 2-10: Outputs Generated Under IP Sources

# 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**.

×	SY	NTHESIS
	•	Run Synthesis
	>	Open Synthesized Design

Figure 2-11: Run Synthesis Button

2. If Vivado prompts you to save your project before launching synthesis, click Save.

While synthesis is running, a status circle displays in the upper right-hand window. This status circle spools for various reasons throughout the design process. The status circle signifies that a process is working in the background.

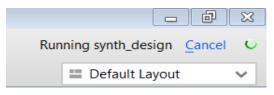


Figure 2-12: Status Bar

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 and launch the Vitis<sup>™</sup> unified software platform.

### Exporting Hardware to the Vitis Software Platform

1. From the Vivado toolbar, select **File > Export > Export Hardware**.

The Export Hardware dialog box opens. Make sure that the **Export to** field is set to the default option of C:/designs/edt\_tutorial/.

*Note:* Only check the **Include bitstream** option if your design has a programmable logic design and is bitstream generated. Otherwise, leave it unchecked.

2. Click **OK**.

Export hardv developmen	vare platform for software	2
_		
Include Export to:	bitstream C:/designs/edt_tutorial/.	

Figure 2-13: Export Hardware



**TIP:** The hardware is exported in a ZIP file (<project wrapper>.xsa).

- Launch the Vitis IDE by using the desktop shortcut or by double-clicking the C:\Xilinx\Vitis\2019.2\bin\vitis.bat file. The Eclipse Launcher dialog box opens.
- 4. Select the workspace location as C:\designs\workspace. Create the workspace folder if it is not already created.



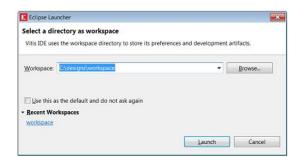
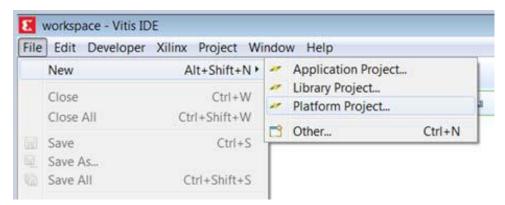


Figure 2-14: Vitis IDE Eclipse Launcher Dialog Box

Click Launch. The Vitis integrated design environment (IDE) opens. Click File > New > Platform Project to create platform project from the output of Vivado Xilinx Shell Archive (XSA).



*Figure 2-15:* **Create New Platform Project** 

6. When the New Platform Project dialog box opens, enter the project name as hw\_platform, as shown in following figure. Keep the **Use default location option** checked. Click **Next**.



New Platform Project	- 0 💌
Create new platform project Enter a name for your platform project	
Project name: hw_platform	
Location: C:\designs\workspace\hw_platform	Browse
System Project 1 Bremedia A72_0 Baremetal Domain Platform	<ul> <li>A system project is a container for multiple applications that would run on different domains of a platform at the same time.</li> <li>A domain is the BSP/OS that controls one or more isomorphic processors.</li> <li>A platform contains one or more domains.</li> <li>A workspace can contain unlimited platforms and unlimited system projects</li> </ul>
(2) < Back	Next > Finish Cancel

Figure 2-16: Enter Project Name

- 7. Select Create from hardware specification (XSA/DSA). Click Next.
- 8. In the Platform Project Specification window, browse to the hardware specification file and select the XSA file C:\designs\edt\_tutorial\tutorial\_bd\_wrapper.xsa. When the XSA file is selected, the Software Specification fields (Operating system and Processor) are updated to standalone and ps7\_cortexa9\_0 respectively, as shown in the following figure. Click **Finish**.



New Platform Proje	ect				
Platform Project S	Specification				
Provide the hardware	e and software specification	n for the new platf	orm project		
Hardware Specification	on				
XSA/DSA file: C:\c	designs\edt_tutorial\tutoria	l_bd_wrapper.xsa	1		Browse
Software Specificatio	n				
Operating system:	standalone	•			
Processor:	ps7_cortexa9_0	•			
0		< <u>B</u> ack	<u>N</u> ext >	<u>F</u> inish	Cancel

Figure 2-17: Software Specification

9. The platform project is created. Double-click on **Project Explorer > platform.spr** to view the platform view as shown in the following figure.



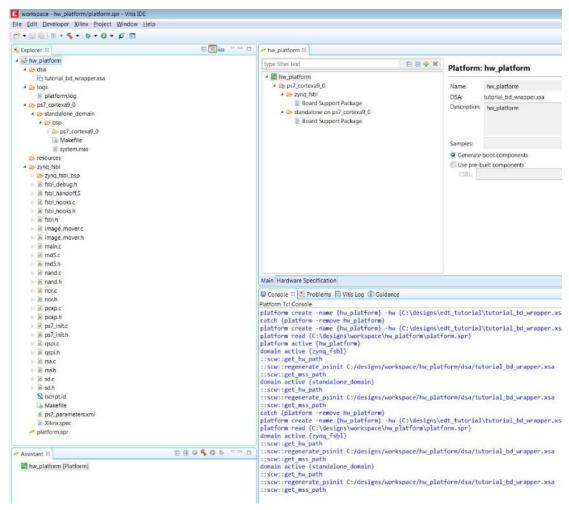


Figure 2-18: Platform View

10. The tutorial\_bd\_wrapper.xsa tab shows the address map for the entire processing system, as shown in the following figure.



Design Information						
	10.00074					
Target FPGA Device:						
	xc7z020clg48					
Created With:						
Created On:	Mon Aug 26 1	15:24:39 2019				
Address Map for pro	cessor ps7_cor	rtexa9[0-1]				
Filter:	Search:		36 Loaded - 36	5 Shown - 0 Selected		
Cell		Base Address	High Address	Slave Interface	Addr Range Type	1
ps7_intc_dist_0		0xf8f01000	0xf8f01fff		register	
ps7_gpio_0		0xe000a000	0xe000afff	-	register	
ps7_scutimer_0		0x18f00600	0xf8f0061f	-	register	
ps7_slcr_0		0xf8000000	0xf8000fff	-	register	1
ps7_scuwdt_0		0xf8f00620	0xf8f006ff	-	register	
hat Treation of			a carinacci		en mistar	
ps7_l2cachec_0		0xf8f02000	0xf8t02fff		register	
the second se		0xf8f02000 0xf8f00000	0x18102111 0x181000fc	:	register	
ps7_l2cachec_0	N.			•	1 Co. 7 Co. 107	
ps7_l2cachec_0 ps7_scuc_0	0.;	0xf8f00000	0xf8f000fc		register	
ps7_l2cachec_0 ps7_scuc_0 ps7_qspi_linear_0	01 01	0xf8f00000 0xfc000000	0xf8f000fc 0xfcffffff		register flash	
ps7_l2cachec_0 ps7_scuc_0 ps7_qspi_linear_0 ps7_pmu_0	N.	0xf8100000 0xfc000000 0xf8893000	0xf8f000fc 0xfcffffff 0xf8893fff	)) ¥	register flash register	
ps7_l2cachec_0 ps7_scuc_0 ps7_qspi_linear_0 ps7_pmu_0 ps7_afi_1 ps7_afi_0 ps7_qspi_0	N	0xf8f00000 0xfc000000 0xf8893000 0xf8009000 0xf8008000 0xe000d000	Oxf81000fc Oxfcfffff Oxf8893fff Oxf8009fff Oxf8008fff Oxe000dfff	• •	register flash register register register register	
ps7_J2cachec_0 ps7_scuc_0 ps7_qspi_linear_0 ps7_pmu_0 ps7_afi_1 ps7_afi_0 ps7_qspi_0 ps7_usp_0	N.	0xf8100000 0xfc000000 0xf8893000 0xf8009000 0xf8008000 0xe000d000 0xe0002000	Oxf81000fc Oxfcfffff Oxf8893fff Oxf8009fff Oxf8008fff Oxe000dfff Oxe0002fff	•	register flash register register register register register	
ps7_J2cachec_0 ps7_scuc_0 ps7_qspi_linear_0 ps7_pmu_0 ps7_afi_1 ps7_afi_0 ps7_qspi_0 ps7_qspi_0 ps7_usb_0 ps7_afi_3	M	0xf8100000 0xfc000000 0xf8893000 0xf8009000 0xf8008000 0xe000d000 0xe000d000 0xe0002000 0xf800b000	Oxf81000fc Oxfcfffff Oxf8893fff Oxf8009fff Oxf8008fff Oxe000dfff Oxe0002fff Oxf800bfff	* * *	register flash register register register register register register	
ps7_J2cachec_0 ps7_scuc_0 ps7_qspi_linear_0 ps7_pmu_0 ps7_afi_1 ps7_afi_0 ps7_qspi_0 ps7_usp_0		0xf8100000 0xfc000000 0xf8893000 0xf8009000 0xf8008000 0xe000d000 0xe0002000	Oxf81000fc Oxfcfffff Oxf8893fff Oxf8009fff Oxf8008fff Oxe000dfff Oxe0002fff	• • •	register flash register register register register register	

Figure 2-19: Address Map for Processing System

11. Build the platform project either by clicking the hammer icon or by right-clicking on the platform project and selecting **Build Project** as shown in following figure.

		x <u>Xilinx P</u> roject <u>W</u> indow <u>H</u> elp <b>X</b> ▼ <b>I Ø</b> I <b>D</b> I <b>☆</b> ▼ <b>O</b> ▼	
Sector Explorer			
hw_platfor	m		
👂 🇁 dsa		New	
<ul> <li>logs</li> <li>ps7_cor</li> <li>resource</li> </ul>		Paste Delete Refresh	Ctrl+V
b B zynq_fsl platforr	2 2	Import Sources Export as Archive	
		Build Project	
		Clean Project	
	· B	Make Project Portable Update Hardware Specification	
		Run As	
		Debug As	,
		Properties	

*Figure 2-20:* **Build Project** 



12. As the project builds, you can see the output in the console window. When the build completes, observe that the top-level platform XML file, hw\_platform.xpfm, has been generated, and the Build Finished log appears in the console window as shown in the following figure.

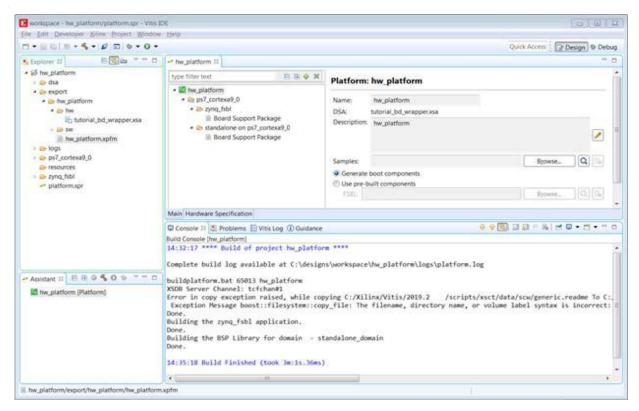


Figure 2-21: Build Finished Log

13. Close the Vitis IDE.

### What Just Happened?

Vivado has exported the hardware design to the hardware specification file (XSA). Using the Vitis IDE, you have created a platform project and exported the XSA file to the workspace in C:\designs\workspace. The export operation generated a standalone domain with a ps7\_cortexa9\_0 processor and an FSBL application project. You have built the platform project to generate a Xilinx platform definition file (hw\_platform.xpfm) which can be used as a platform for the applications that you create in the Vitis IDE.

### What's Next?

Now you can start developing the software for your project using the Vitis software platform. The next sections help you create a software application for your hardware platform.



### Example Project: Running the "Hello World" Application

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 in JTAG mode using System Debugger in the Vitis IDE.

*Note:* If you 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 Target board with the following SW10 switch settings:

```
Bit-1 is 0
Bit-2 is 1
```

- **Note:** 0 = switch is open. 1 = switch is closed.
- 3. Connect a USB cable to connector **J17** on the target board with the Windows Host machine. This is used for USB to serial transfer.
- 4. Power on the ZC702 board using the switch indicated in the figure below.



**IMPORTANT:** Ensure that jumper s J27 and J28 are placed on the side farther from the SD card slot and change the SW16 switch setting as shown in Figure 2-22.







Figure 2-22: ZC702 Board Power Switch

5. Open the Vitis software platform and set the workspace path to your project file, which in this example is C:\designs\workspace.

Alternatively, you can open the Vitis software platform 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 software platform provides a serial terminal utility, which will be used throughout the tutorial; select **Window > Show View > Terminal** to open it.

Jerminal 🛛		• • • • • • • • • • • • • • • • • • •
	Figure 2-23: Terminal Window Heade	r Bar

7. Click the **Launch Terminal** button  $\blacksquare$  to open the Launch Terminal dialog box.



Select Serial for Choose Terminal, click **Terminal Settings**, and click **OK**. This figure shows the standard configuration for the Zynq SoC processing system. The following figure shows the standard configuration for the Zynq SoC processing system.

Settings Port:	COM85 -
Baud Rate:	115200 -
Data Bits:	8 -
Parity:	None
Stop Bits:	1
Flow Control:	None •
Timeout (sec):	5
Encoding: Def	ault (ISO-8859-1)

Figure 2-24: Launch Terminal Dialog Box

#### 8. Select File > New > Application Project.

The New Application Project wizard opens.

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

Wizard Screen	System Properties	Setting or Command to Use
	Project Name	hello_world
	Use Default Location	Select this option
Application Project	System Project	hello_world_system
Application Project	Platform	hw_platform
	Domain	standalone_domain <sup>(1)</sup>
	Language	C
Templates	Available Templates	Hello World

#### Notes:

1. By default, this step sets the CPU and OS to ps7\_cortexa9\_0 and standalone respectively.

The Vitis software platform creates the hello\_world application project under the Project Explorer. Right-click on the **hello\_world** application and select **Build Project** to generate the hello\_world.elf binary file.

- 10. Right-click **hello\_world** and select **Run as > Run Configurations**.
- 11. Right-click Single Application Debug and click New Configuration.

The Vitis software platform creates the new run configuration, named Debugger\_hello\_world-Default.

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

- 12. Click the **Target Setup** tab and review the settings. The default choice is the Tcl script.
- 13. Click **Run**.

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

<b>E M</b>   1		E D	3ª -	
)/17/19, <mark>4</mark> :	:51 PM) ສິ			*
9			9/17/19, 4:51 PM) 🛛	■ **   1

Figure 2-25: Output on Serial Terminal

**Note:** There was no bitstream download required for the above software application to be executed on the Zynq SoC evaluation board. The Arm Cortex A9 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 UART1 peripheral of the PS section.

From UART1, 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 or Board Support Package**

A domain or board support package (BSP) is a collection of software drivers and, optionally, the operating system on which to build your application. It 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. You can create multiple applications to run on the domain. A domain is tied to a single processor in the platform.

### **Standalone OS**

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.



**IMPORTANT:** The application you ran in this chapter was created on top of the standalone OS. The domain/BSP that your software application targets is selected during the New Platform Project creation process.



### Chapter 3

# Using the GP Port in Zynq Devices

One of the unique features of using the Xilinx<sup>®</sup> Zynq<sup>®</sup>-7000 SoC as an embedded design platform is in using the Zynq SoC Processing System (PS) for its Arm<sup>®</sup> Cortex<sup>®</sup>-A9 dual core processing system as well as the Programmable Logic (PL) available on it.

In this chapter, you will create a design with:

- AXI GPIO and AXI Timer in fabric (PL) with interrupt from fabric to PS section
- Zynq SoC PS GPIO pin connected to the fabric (PL) side pin via the EMIO interface

The flow of this chapter is similar to that in Chapter 2 and uses the Zynq device as a base hardware design. It is assumed that you understand the concepts discussed in Chapter 2 regarding adding the Zynq device into a Vivado<sup>®</sup> IP integrator block diagram design. If you skipped that chapter, you might want to look at it because we will continually refer to the material in Chapter 2 throughout this chapter.

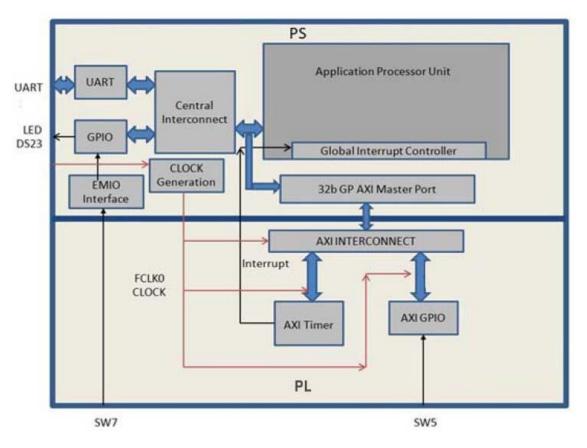
### Adding IP in PL to the Zynq SoC Processing System

There is no restriction on the complexity of an intellectual property (IP) that can be added in fabric to be tightly coupled with the Zynq SoC PS. This section covers a simple example with the AXI GPIO, AXI Timer with interrupt, and the PS section GPIO pin connected to PL side pin via the EMIO interface.

In this section, you will create a design to check the functionality of the AXI GPIO, AXI Timer with interrupt instantiated in fabric, and PS section GPIO with EMIO interface. The block diagram for the system is as shown in the Figure 3-1.







*Figure 3-1:* Block Diagram

You can use the system created in Chapter 2 and continue after Creating an Embedded Processor Project, page 13.

In the examples in this chapter, we will expand on the design in Chapter 2. You will make the following design changes:

- The fabric-side AXI GPIO is assigned a 1-bit channel width and is connected to the SW5 push-button switch on the ZC702 board.
- The PS GPIO ports are modified to include a 1-bit interface that routes a fabric pin (via the EMIO interface) to the SW7 push-button switch on the board.
- In the PS section, another 1-bit GPIO is connected to the DS23 LED on the board, which is on the MIO port.
- The AXI timer interrupt is connected from fabric to the PS section interrupt controller. The timer starts when you press any of the selected push buttons on the board. After the timer expires, the timer interrupt is triggered.
- Along with making the above hardware changes, you will write the application software code. The code will function as follows:



- A message appears in the serial terminal and asks you to select the push button switch to use on the board (either SW7 or SW5).
- When the appropriate button is pressed, the timer automatically starts, switches OFF LED DS23, and waits for the timer interrupt to happen.
- After the Timer Interrupt, LED DS23 switches ON and execution starts again and waits for you to again select the push button switch in the serial terminal.

### **Example Project: Validate Instantiated Fabric IP Functionality**

In this example, you will add the AXI GPIO, AXI Timer, the interrupt instantiated in fabric, and the EMIO interface. You will then validate the fabric additions.

- 1. Open the Vivado® Design Suite.
- 2. Under the Recent Projects column, click the **edt\_tutorial** design that you created in Chapter 2.
- 3. Under IP Integrator, click Open Block Design.
- 4. In the Diagram window, right-click in the blank space and select Add IP.
- 5. In the search box, type AXI GPIO and double-click the **AXI GPIO** IP to add it to the Block Design.

The AXI GPIO IP block appears in the Diagram view.

- 6. In the Diagram window, right-click in the blank space and select Add IP.
- 7. In the search box, type AXI Timer and double-click the **AXI Timer** IP to add it to the Block Design. The AXI Timer IP block appears in the Diagram view.
- You must also edit the EMIO configuration of the ZYNQ7 SoC Processing system and enable interrupts. Right-click the **ZYNQ7 Processing System** IP block and select **Customize Block**.

*Note:* You can also double-click the IP block to customize.

The Re-customize IP dialog box opens, as shown in the Figure 3-2.



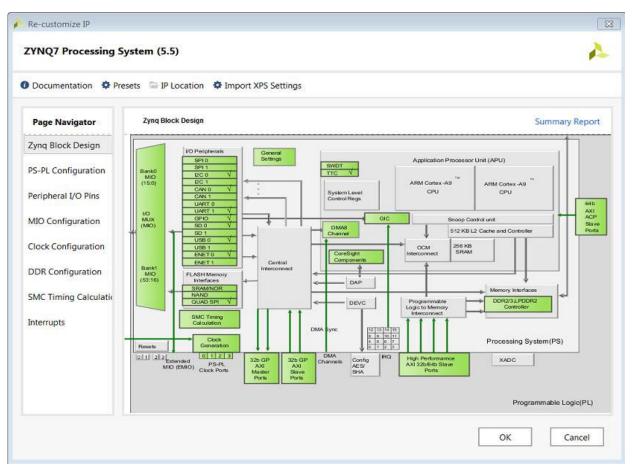


Figure 3-2: Re-customize IP Dialog Box

- 9. Click MIO Configuration.
- 10. Expand I/O Peripherals > GPIO and select the EMIO GPIO (Width) check box.
- 11. Change the EMIO GPIO (Width) to 1.
- 12. With the ZYNQ7 Processing System configuration options still open, navigate to **Interrupts > Fabric Interrupts > PL-PS Interrupt Ports**.
- 13. Check the **Fabric Interrupts** box and also check **IRQ\_F2P[15:0]** to enable PL-PS interrupts in the IP Core.
- 14. Click **OK** to accept the changes to the ZYNQ7 Processing System IP. The Diagram looks like Figure 3-3.



Diagram × Address Editor ×	? 🗆 🖸
Q Q X X Q Ξ 0 Q Ξ 0 + N Y Y C Y = Default View ∨	0
Designer Assistance available. Run Connection Automation	
ski,gplo,0 + \$,AXI aui_state_0 + \$,AXI aui_state_0 + \$,AXI +	

Figure 3-3: ZYNQ7 Processing System IP

- 15. Click the **Run Connection Automation** link at the top of the page to automate the connection process for the newly added IP blocks.
- 16. In the Run Connection Automation dialog box, select the check box next to **All Automation**, as shown in Figure 3-4.

ace pin on the left panel to view its options

Figure 3-4: Run Connection Automation Dialog Box

17. Click **OK**.

Upon completion, the updated diagram looks like Figure 3-5.



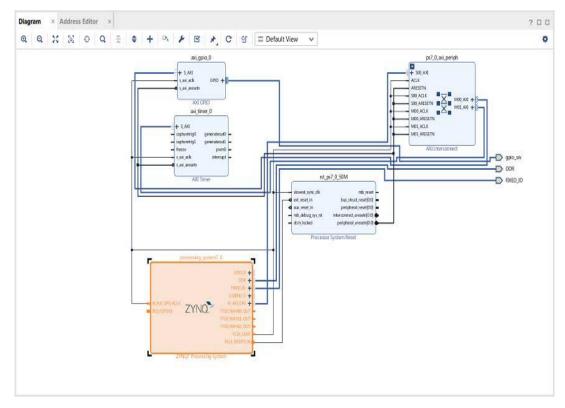


Figure 3-5: Updated Block Diagram with Connections

18. Right-click the AXI GPIO IP block and select Customize Block.

Note: You can also double-click the IP block to make customizations.

- 19. Under the **Board** tab, make sure that both **GPIO** and **GPIO2** are set to **Custom**.
- 20. Select the **IP Configuration** tab. In the GPIO section, change the **GPIO Width** to **1** because you only need one GPIO port. Also ensure that **All Inputs** and **All Outputs** are both unchecked.
- 21. Click **OK** to accept the changes.
- 22. Notice that the Interrupt port is not automatically connected to the AXI Timer IP Core. In the Block Diagram view, locate the IRQ\_F2P[0:0] port on the ZYNQ7 Processing System.
- 23. Scroll your mouse over the connector port until the pencil icon appears, then click the **IRQ\_F2P[0:0]** port and drag to the **interrupt** output port on the AXI Timer IP core to make a connection between the two ports.



24. Notice that the ZYNQ7 Processing System GPIO\_0 port is not connected. Right-click the **GPIO\_0** output port on the **ZYNQ7 Processing System** and select **Make External**.

The pins are external but do not have the needed constraints for our board. In order to constrain your hardware pins to specific device locations, follow the steps below. These steps can be used for any manual pin placements.

25. Click **Open Elaborated Design** under RTL Analysis in the Flow Navigator view.

Y R	TL ANALYSIS
~	Open Elaborated Design
	Report Methodology
	Report DRC
	Schematic

Figure 3-6: Open Elaborated Design

26. When the Elaborate Design message box opens, as shown in the following figure, click **OK**.

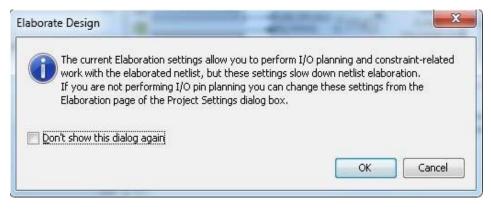


Figure 3-7: Elaborate Design Message Box

**TIP:** The design might take a few minutes to elaborate. If you want to do something else in Vivado while the design elaborates, you can click the Background button to have Vivado continue running the process in the background.

27. Select **I/O Planning** from the drop-down menu, as shown in the following figure, to display the **I/O Ports** window.

■ + + E E :	( ∞ ▶ ₩ ₩ Θ Φ Σ K # X	🖾 Default Layout 🗸 🗸
Flow Navigator 😤	ELABORATED DESIGN - xc7z020clg484-1	I Default Layout
PROJECT MANAGER	~	I/O Planning
Settings	Sources Netlist × ? _ □ □ Project Summary × Schematic ×	I Floorplanning
Add Sources	₹ 31	+ - C 3 Cells 132 I/O Save As New Layout
Add Sources	R tutorial bd wrapper	Reset Layout F!
Language Templates	> 🗅 Nets (138)	0 00,00 0 00,00
P IP Catalog	> 🚍 Leaf Cells (2)	O may O may

Figure 3-8: I/O Planning Selection



28. Under the I/O Ports tab view at the bottom of the Vivado window (as seen in the following figure), expand the GPIO\_0\_18048 and gpio\_sw\_18048 ports to check the site (pin) map.

I/O Ports × Log Reports	Design Runs
Q ≍ ♦ 📢 + 2	a 👔
Name	Direction
V 🖨 All ports (132)	
> @ DDR_18048 (71)	INOUT
> 5 FIXED_IO_18048 (59)	INOUT
✓ ™ GPIO_0_18048 (1)	INOUT
GPIO_0_0_tri_io (1)	INOUT
GPIO_0_tri_io[0]     GPIO_0_tri_io[0]     GPIO_0_0_tri_io[0]     GPIO_0_tri_io[0]     GPIO_0_0_tri_io[0]     GPIO_0_tri_io[0]      GPIO_0_tri_io[0]     GPIO_0_tri_io[0]     GPIO_0_tri_io[0]	INOUT
Scalar ports (0)	
✓ ☎ gpio_sw_18048 (1)	INOUT
gpio_sw_tri_io (1)	INOUT
gpio_sw_tri_io[0]	INOUT
Scalar ports (0)	
Scalar ports (0)	

Figure 3-9: I/O Ports Site Map

29. Find **GPIO\_0\_0\_tri\_io[0]** and set the following properties, shown in Figure 3-10:

- Package Pin = F19
- I/O Std = LVCMOS25

30. Find **gpio\_sw\_tri\_io[0]** and set the following properties, shown in Figure 3-10:

- Package Pin = G19
- I/O Std = LVCMOS25

the second of the second second second									
직 꽃 ≑ 📧 + 놰									0
Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	
🗸 📮 All ports (132)									
> 🖙 DDR_18048 (71)	INOUT					~	502	(Multiple)*	
> @ FIXED_IO_18048 (59)	INOUT					~	(Multiple)	(Multiple)*	
✓ □ GPIO_0_18048 (1)	INOUT					~	35	LVCMOS25*	8.
GPIO_0_0_tri_io (1)	INOUT					~	35	LVCMOS25*	100
GPIO_0_tri_io[0]	INOUT				F19 🗸 🗸	4	35	LVCMOS25*	
🖾 Scalar ports (0)									
✓ IB gpio_sw_18048 (1)	INOUT					4	35	LVCMOS25*	
gpio_sw_tri_io (1)	INOUT					~	35	LVCMOS25*	10
gpio_sw_tri_io[0]	INOUT				G19 🛩	~	35	LVCMOS25*	•
Scalar ports (0)									

Figure 3-10: I/O Port Properties

**Note:** For additional information about creating other design constraints, refer to the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 3].

31. In the Flow Navigator, under Program and Debug, select Generate Bitstream.



The Save Project dialog box opens. Make sure the **Elaborated Design - constrs\_1** check box is selected and click **Save**.

32. The **Save Constraints** dialog box appears (shown in the following figure). Provide a file name (GPIO\_Constraints) and click **OK**. If the Synthesis is Out-of-date dialog box opens, click **Yes** to rerun synthesis. The Launch Runs dialog box opens. Click **OK** to launch synthesis.

onstraints.	will update that file with	i the new
• <u>C</u> reate a new file		
<u>F</u> ile type:	D XDC	~
File name:	GPIO_Constraints	۵
File location:	<local project="" to=""></local>	~
Select an existing	file	
<select a="" targ<="" td=""><td>get file&gt;</td><td>~</td></select>	get file>	~

Figure 3-11: Save Constraints Dialog Box

A constraints file is created and saved under **Sources > Hierarchy > Constraints**, as shown in the following figure.



Sources × Netlist	Device Constraints	? _ 0 6
Q ₹ \$ +	🖻 🔍 💿 o	0
V Design Sources (1)		
v 🔵 👶 tutorial_bd	wrapper (tutorial_bd_wrappe	r.v) (l)
🗸 🔬 🔳 tutorial	_bd_i: tutorial_bd (tutorial_bd.	bd) (1)
🗸 🔵 tutori	al_bd (tutorial_bd.v) (6)	
> 🖓 🔳	axi_gpio_0 : tutorial_bd_axi_g	pio_0_0 (tutorial_bd_a
> 🖓 🔳	axi_timer_0 : tutorial_bd_axi_t	imer_0_0 (tutorial_bd_
> 🖓 🔳	processing_system7_0 : tutoria	al_bd_processing_syst
> 😐 p	s7_0_axi_periph : tutorial_bd_p	s7_0_axi_periph_0 (tut
% p	s7_0_axi_periph : tutorial_bd_p	s7_0_axi_periph_0
> 🖓 🔳	rst_ps7_0_50M : tutorial_bd_rs	t_ps7_0_50M_0 (tutori
✓ ⊆ Constraints (1)		
~ 🖙 constrs_1 (1)		
C GPIO_Con	straints.xdc (target)	
> Simulation Source	s (1)	

Figure 3-12: Sources Window Showing New Constraints File

33. After Bitstream generation completes, export the hardware and launch the Vitis<sup>™</sup> IDE as described in Exporting Hardware to the Vitis Software Platform, page 22.

**Note:** When exporting hardware, because programmable logic IP is involved in the design, make sure that the **Include bitstream** option is checked. Set the Export to option with the path C:/designs/edt\_tutorial as shown in the following figure.

Export naruw	are platform for software developm	ent tools.
Include I	pitstream	
Export to:	C:/designs/edt_tutorial	⊗ …

Figure 3-13: Export Hardware with Include Bitstream Checked

#### Working with the Vitis Software Platform

Open the Vitis IDE and manually update the exported hardware from Vivado. In Project Explorer, right-click on the **hw\_platform** platform project and click on the **Update Hardware Specification** option as shown in the following figure.





🐮 workspace - hw	plat	tform/dsa/tutorial_bd_wrapper.xsa	- Vitis IDE
	-	oper Xilinx Project Window H	
i 🗗 🗕 🖪 👘   😒 🔻	- 4	- I I I I I I I I I I I I I I I I I I I	.e   🗟 📌 😰 🗖
🏇 Debug ବ Proje	ct E	xplorer 🖾 🖳 Explorer	
		m [ hw_platform ]	
▲ 😂 hw_platform	-	New	Þ
▲		Add Application Project	
⊳ 🖻 ps7_ini		Add Library Project	
⊳ 🖻 ps7_ini		Go Into	
⊳ 🖻 ps7_ini		Open in New Window	
Ini		Show in Local Terminal	•
🔓 ps/_ini		Сору	Ctrl+C
🖻 🗁 export		Paste Delete	Ctrl+V Delete
Iogs		Source	+
<ul> <li>b b ps7_corte</li> <li>b b resources</li> </ul>		Move	
<ul> <li>isources</li> <li>isources</li> <li>isources</li> </ul>		Rename	F2
platform.	è	Import	
	,	Import Sources	
	4	Export	
		Build Project	
	ন্ধ্য	Clean Project Refresh	F5
		Close Project	
		Close Unrelated Projects	
		Build Configurations	•
		Validate	
	0	Run As	•
	脊	Debug As Compare With	
		Restore from Local History	,
		Make Project Portable	
	E.	-	
		Team	+
		Configure	•
		Source	+
		Properties	Alt+Enter

Figure 3-14: Update Hardware Specification



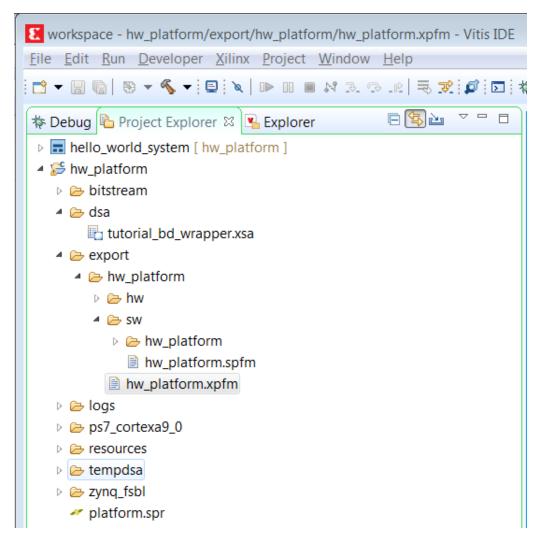
1. In Update Hardware Specification dialog box, browse for the exported XSA file from Vivado and click **OK**. A dialog box opens stating that the hardware specification for the platform project has been updated, as shown in following figure.

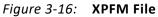
😢 workspace - hw_platform/platform.spr - Vitis IDE		
File Edit Run Developer Xilinx Project Window Help		
	22	
☐ ▼ 🖩 🕲   ® ▼ 🍕 ▼ 🖳 🔌   ▶ 0 = 14 3. 3. 3. 4.   ₹ 🕱 💋   ♦ ▼ (	0 -	
🎋 Debug 🔁 Project Explorer 🛛 🖳 Explorer		
hello_world_system [ hw_platform ]		
▲ 🛃 hw_platform (Out-of-date)		
🖻 🗁 bitstream		
🔺 🗁 dsa		
🖬 tutorial_bd_wrapper.xsa		
🖻 🗁 export		
🕑 🗁 logs		
b 🗁 ps7_cortexa9_0		
Fesources		
🖻 🗁 tempdsa		
Zynq_fsbl		
platform.spr		
E Update Hardware Specification	8	
Hardware specification for platform project 'hw_platform' is updated.		
	OK	

Figure 3-15: Hardware Specification Updated

- 2. Rebuild the out-of-date platform project. Right-click on the **hw\_platform** project. Select **Clean Project** followed by **Build Project**.
- 3. After the hw\_platform project build completes, the hw\_platform.xpfm file is generated, as shown in the following figure.







- 4. Open the helloworld.c file from the hello\_world project created with standalone PS in Chapter 2 and modify the application software code as described in Standalone Application Software for the Design, page 50.
- 5. Save the file and re-build the project.
- 6. Open the serial communication utility with baud rate set to **115200**.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.

7. Connect to the board.

Because you have a bitstream for the PL fabric, you must download the bitstream.

8. Select **Xilinx > Program FPGA**. The Program FPGA dialog box, shown in Figure 3-17, opens. Browse for the bitstream exported from Vivado.



	hello_world		-		
Connection:	Local		•	New	
Device:	Auto Detec	t		Select	
Bitstream:	C\designs\	workspace\hw_platform\bitstream\tutorial_bd_w	rap	Search	Browse
Partial Bitstr	eam				
3MM/MMI File				Search	Browse-

Figure 3-17: Program FPGA

9. Click **Program** to download the bitstream and program the PL fabric. When the FPGA programming is done, progress information pop up opens and shows the status as **FPGA configuration complete**, as shown in the following figure.

E Prog	gress Information	
1	FPGA configuration complete.	
	Car	Details >>

*Figure 3-18:* **FPGA Configuration Complete** 

10. Run the project similar to the steps in Example Project: Running the "Hello World" Application, page 29. If steps 9 and 10 fail, open the **Run Configurations** window, browse for the bitstream file exported by Vivado, and then click the **Run** button as shown in following figure. With this step, the FPGA is programmed and the application runs.



Create, manage, and run configurations Debug a program using Application Debugger					
]∦⊛⊚¥∣∈⇒∙	Name: Debugger_hell	o_world-Default			
type filter text	🐔 Main 🗖 Applicatio	n 🐵 Target Setup 🛛 🐏 Arguments 🗮 Environment 🖡 Symbol Files 🦭 Source	🚨 Path Map 🔲 Com	mon	
<ul> <li>Launch Group</li> <li>OpenCL</li> <li>OpenCL (TCF)</li> </ul>	Hardware Platform:	\$(sdxTcfLaunchFile;project=hello_world;fileType=hw;)	Search	Browse	
Target Communication Framework	Bitstream File:	$C\begin{tabular}{lllllllllllllllllllllllllllllllllll$	Search	Browse	Generate
Xilinx Al Engine systeme simulator	FPGA Device:	Auto Detect	Select		
5 Xilinx Application Debugger	PS Device:	Auto Detect	Select		
Debugger_hello_world-Default Xilinx Application Debugger (GDB)	Use FSBL flow for	initialization			
Zilinx SPM Analysis	FSBL File:	\$(sdxTcfLaunchFile:project=hello_world;fileType=fsbl;}	Search	Browse	
<ul> <li>Xilinx System Debugger</li> <li>Xilinx SystemC-RTL Co-Simulator</li> </ul>	FSBL Exit Function:	FsbiHandoffJtagExit			
The Annu Systeme Kie co-Sindiator		Summary:			
ilter matched 12 of 21 items				Revert	Apply

*Figure 3-19:* **Run Configuration** 

- 11. In the system, the AXI GPIO pin is connected to push button **SW5** on the board, and the PS section GPIO pin is connected to push button **SW7** on the board via an EMIO interface.
- 12. Follow the instructions printed on the serial terminal to run the application. See the following figure for serial output logs.

### **XILINX**<sub>®</sub>

<pre>     Terminal 1      E Serial COM85 (8/29/19, 4:06 PM)      ###### Application Starts ##### </pre>	
<pre> State: The Operation from the Below Menu ####################################</pre>	le Edit Run Developer Xilinx Project Window Help
<pre>© Serial COM85 (8/29/19.406 PM) 32 ##### Application Starts ##### SELECT the Operation from the Below Menu ####################################</pre>	3 ▼ 🖩 🖏   🏵 ▼ 🗞 ▼ 🖳 🔌   🗣 💵 📾 🗱 ೫ ೫. ۞ 🕫   ≅ 🥵 🕼
<pre>##### Application Starts ##### SELECT the Operation from the Below Menu ####################################</pre>	
<pre>SELECT the Operation from the Below Menu #################### Menu Status ####################################</pre>	E Serial COM85 (8/29/19, 4:06 PM) 🛛
SELECT the Operation from the Below Menu ####################################	###### Application Starts #####
Press '1' to use NORMAL GPIO as an input (SW5 switch) Press any other key to Exit ####################################	SELECT the Operation from the Below Menu
Press '2' to use EMIO as an input (SW7 switch) Press any other key to Exit ####################################	
Press any other key to Exit ####################################	
Selection : 1 Press Switch 'SW5' push button on board SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS	
Press Switch 'SW5' push button on board \$ SW5 PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ @@@@@@@@	######################################
<pre>\$ \$\$\$\$\$\$\$ \$\$\$\$\$ \$\$\$\$\$ \$\$</pre>	
SWS PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ eeccecccccccccc	Press Switch 'SW5' push button on board
LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$	\$
<pre>timer start Wait for the Timer interrupt to tigger S\$</pre>	
<pre>Wait for the Timer interrupt to tigger \$</pre>	
<pre>\$ CONCONCONCONCONCONCONCONCONCONCONCONCONC</pre>	
<pre>Performance Provide Provi</pre>	
Inside Timer ISR LED 'DS23' Turned ON Timer ISR Exit @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	
Inside Timer ISR LED 'DS23' Turned ON Timer ISR Exit @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	200000000000000000000000000000000000000
LED 'DS23' Turned ON Timer ISR Exit @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	
<pre>Timer ISR Exit @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</pre>	
<pre>####################################</pre>	
<pre>####################################</pre>	
<pre>Press '0' to go to Main Menu Press any other key to remain in AXI GPIO Test ####################################</pre>	
<pre>Press '0' to go to Main Menu Press any other key to remain in AXI GPIO Test ####################################</pre>	
<pre>Press any other key to remain in AXI GPIO Test ####################################</pre>	
<pre>####################################</pre>	
SELECT the Operation from the Below Menu ####################################	
<pre>####################################</pre>	Select = 0
<pre>####################################</pre>	SELECT the Operation from the Below Manu
Press '1' to use NORMAL GPIO as an input (SW5 switch) Press '2' to use EMIO as an input (SW7 switch) Press any other key to Exit ####################################	
Press any other key to Exit ####################################	The second se
<pre>####################################</pre>	
Selection : 2 Press Switch 'SW7' push button on board \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ SW7 PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$	
Press Switch 'SW7' push button on board \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ SW7 PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ @@@@@@@@	
SW7 PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ @@@@@@@@@@@@@@@@	
SW7 PUSH Button pressed LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ @@@@@@@@@@@@@@@@	
LED 'DS23' Turned OFF timer start Wait for the Timer interrupt to tigger \$ @@@@@@@@@@@@@@@	
timer start Wait for the Timer interrupt to tigger \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ @@@@@@@@@@@@@@@@@@	
Wait for the Timer interrupt to tigger \$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$ @@@@@@@@@@@@@@@@@@	
00000000000000000000000000000000000000	
Inside Timer ISR LED 'DS23' Turned ON Timer ISR Exit	\$
Inside Timer ISR LED 'DS23' Turned ON Timer ISR Exit	
LED 'DS23' Turned ON Timer ISR Exit	
Timer ISR Exit	
<u>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</u>	
	<u>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</u>

Figure 3-20: Serial Output Logs





### Standalone Application Software for the Design

The system you designed in this chapter requires application software for the execution on the board. This section describes the details about the application software.

The main() function in the application software is the entry point for the execution. This function includes initialization and the required settings for all peripherals connected in the system. It also has a selection procedure for the execution of the different use cases, such as AXI GPIO and PS GPIO using EMIO interface. You can select different use cases by following the instruction on the serial terminal.

#### **Application Software Steps**

Application Software is composed of the following steps:

- 1. Initialize the AXI GPIO module.
- Set a direction control for the AXI GPIO pin as an input pin, which is connected with the SW5 push button on the board. The location is fixed via LOC constraint in the user constraint file (XDC) during system creation.
- 3. Initialize the AXI TIMER module with device ID 0.
- 4. Associate a timer callback function with AXI timer ISR.

This function is called every time the timer interrupt happens. This callback switches on the LED **DS23** on the board and sets the interrupt flag.

The main() function uses the interrupt flag to halt execution, waits for timer interrupt to happen, and then restarts the execution.

- 5. Set the reset value of the timer, which is loaded to the timer during reset and timer starts.
- 6. Set timer options such as Interrupt mode and Auto Reload mode.
- 7. Initialize the PS section GPIO.
- 8. Set the PS section GPIO, channel 0, pin number 10 to the output pin, which is mapped to the MIO pin and physically connected to the LED **DS23** on the board.
- Set PS Section GPIO channel number 2, pin number 0, to an input pin, which is mapped to PL side pin via the EMIO interface and physically connected to the SW7 push button switch.
- 10. Initialize Snoop control unit Global Interrupt controller. Also, register Timer interrupt routine to interrupt ID '91', register the exceptional handler, and enable the interrupt.
- 11. Execute a sequence in the loop to select between AXI GPIO or PS GPIO use case via serial terminal.





The software accepts your selection from the serial terminal and executes the procedure accordingly. After the selection of the use case via the serial terminal, you must press a push button on the board as per the instruction on terminal. This action switches off the LED **DS23**, starts the timer, and tells the function to wait infinitely for the Timer interrupt to happen. After the Timer interrupt happens, LED **DS23** switches ON and restarts execution.

### **Application Software Code**

The Application software for the system is included in helloworld.c, which is available in the ZIP file that accompanies this guide. For more details, see Design Files for This Tutorial, page 134.

### Chapter 4



# Debugging with the Vitis Software Platform

This chapter describes debug possibilities with the design flow you have already been working with. The first option is debugging with software using the Xilinx® Vitis<sup>™</sup> unified software platform.

The Vitis software platform debugger provides the following debug capabilities:

- Supports debugging of programs on MicroBlaze<sup>™</sup> and Arm<sup>®</sup> Cortex<sup>®</sup>-A9 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 integrated design environment (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 software platform supports debugging through Xilinx System Debugger and GNU Debugger (GDB).

*Note:* The GDB flow is deprecated and will not be available for future devices.

### **Xilinx System Debugger**

The Xilinx System Debugger uses the Xilinx hw\_server as the underlying debug engine. The Vitis software platform 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.



52



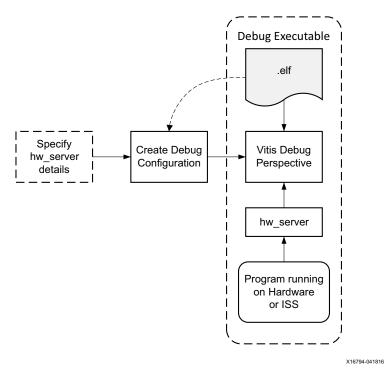


Figure 4-1: System Debugger Flow

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 software platform. 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.
- 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 software platform.

**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 Software Platform

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

If you modified the hello world application in the prior chapter, you will need to create a new hello world application prior to debugging. Follow the steps in Example Project: Running the "Hello World" Application, page 29 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 Software Platform.

 In the C/C++ Perspective, right-click the Hello\_world Project and select Debug As > Debug Configurations.

In Target Setup window, fill the Hardware Platform field with the one exported by the Vivado® Design Suite, and click the **Debug** button.

reate, manage, and run configurations Debug a program using Application Debugger.						Ŕ
8 ið 18 <b>X</b>   8 3 •	Name: Debugger_helk	o_world-D	efault			
type filter text	🗶 Main 🗖 Applicatio	n 💿 Targe	et Setup 🙌 Arguments 👼 Environment 🐺 S	Symbol Files	ource & Path Ma	Common
Claunch Group OpenCL OpenCL OpenCL OpenCL OpenCL C(TCF) Target Communication Framework Callinx Al Engine SystemC Simulator Communication Debugger Communication Debugger Communication Debugger Communication Debugger Communication Debugger Communication C	Bitstream File: FPGA Device: PS Device:	FPGA Device: Auto Detect PS Device: Auto Detect I Use FSBL flow for initialization		Search Search Select Select Search	Browse Browse Browse	Generate
	TODE EXIT FUNCTION.	1,201110110	loff)tagExit Summary:			
	Reset entire system	m	Following operations will be performed bet	fore launching the	e debugger.	E
Iter matched 11 of 23 items					Revert	Apply

The Debug Perspective opens.

Figure 4-2: Debug Configurations

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



File Edit Run Developer Xilinx Project Window Help								
□ • 🖩 🕲   9 • • • •   0   •   ● 🗉 = 14 3. 9. /¢   ≒ 🖄	Ø 🖬 🛊 • O •					Qu	iick Access 🛛 🔐 I	Design 🔯 Deb
🛊 Debug 🕸 📴 Project Explorer 🔏 Explorer 📲 🙀 😤 🗖	helloworld.c 🕸	-		(*)= Varia	💊 Break 😚 Exp	re 🛋 Modul	IIII Regis 🕮 🚺	Memory 🐡
<ul> <li>         &amp; Debugger_hello_world-Default (Local)              &amp; 2 APU               &amp; ARM Cortex-A9 MPCore #0 (Breakpoint: main)      </li> </ul>	47 48 #include < 49 #include * 50 #include *	'platform.h"	*	Name IIII r12	Hex 00000000	Decimal 0	신 Main Rescription	All Ci Ci Ci Mnemonic
0x0010058c main():/src/helloworld.c, line 55 0x00100834 _start(): xil-crt0.S, line 136 	51 52 53=int main()			IIII sp IIII Ir IIII pc	0010c028 00100834 0010058c	1097768 1050676 1049996		
P arm Cortex-A9 MPCore #1 (Suspended) 2 xc7z020	<pre>&gt;54 { &gt;55 init_p 56</pre>	platform();		<ul> <li>IIII cpsr</li> <li>IIII usr</li> </ul>	6000005f	1610612831		
	58	("Hello World\n\r"); up_platform(); u 0;	-	Hex: 001005 Bin: 0000,00	8c, <b>Dec:</b> 1049996, 900,0001,0000,00 readable, writable	Oct: 04002614, At 000,0101,1000,11		
		□ v B p "; □ □	• EV Disassem	HIV 22	Enter	location here	1 2 4 3 3	1
	Platform Tcl Consol platform read	e (C:\designs\workspa ^ e (hw_platform)	00100582 54 • 00100584 00100584	: nop { main: : push	{r11, lr} r11, sp, #4			
- Assistant 🗵 😑 😟 📽 🐨 🗢 🗆	::scw::get_hw_		55 56		platform();			
<ul> <li>Enclo_world_system [System]</li> <li>Enclo_world [Application]</li> <li>hw_platform [Platform]</li> </ul>	<pre>::scw::get_mss domain active ::scw::get_hw_</pre>	_path {standalone_domain} path ate_psinit C:/desig	<ul> <li>0010058c</li> <li>57</li> <li>58</li> <li>00100590</li> <li>00100594</li> <li>00100598</li> </ul>	print : movw : movt	r0, #2656 r0, #16	dr=0x001005f4 \n\r"); dr=0x00100620		cm

Figure 4-3: Debugging 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  $0 \times 0010058c$ . You can confirm this information in the Disassembly view, which shows the assembly-level program execution also suspended at  $0 \times 0010058c$ .

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

2. 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 0x0010058c.

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

3. Double-click in the margin of the helloworld.c window next to the line of code that reads init\_platform () and print (). This sets the breakpoints at init\_platform () and print (). To confirm the breakpoints, review the Breakpoints window.

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

4. Select **Run > Step Into** to step into the init\_platform () routine.

Program execution suspends at location 0x001005fc. The call stack is now two levels deep.



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

Program execution stops at the line of code that includes the print command. The Disassembly and Debug windows both show program execution stopped at  $0 \times 00100590$ .

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

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

When the program completes, the Terminal window shows the Hello World print and 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.

7. 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 Vitis tool debugging shortcuts for step-into (F5), step-return (F7), step-over (F6), and resume (F8).

8. Exit the Vitis software platform.



### Using the HP Slave Port with AXI CDMA IP

In this chapter, you will instantiate AXI CDMA IP in fabric and integrate it with the processing system high performance (HP) 64-bit slave port. In this system, AXI CDMA acts as master device to copy an array of the data from the source buffer location to the destination buffer location in DDR system memory. The AXI CDMA uses the processing system HP slave port to get read/write access of DDR system memory.

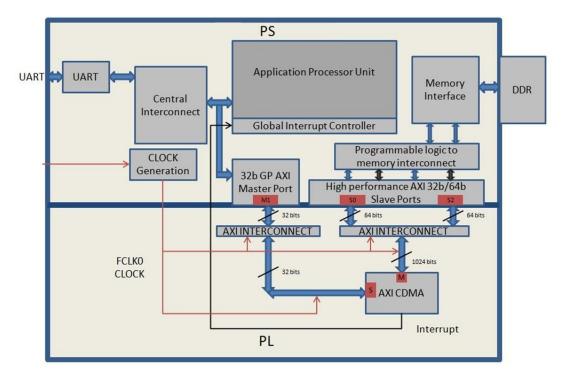
You will write Standalone application software and Linux OS based application software using mmap() for the data transfer using AXI CDMA block. You will also execute both standalone and Linux-based application software separately on the ZC702 board.

## Integrating AXI CDMA with the Zynq SoC PS HP Slave Port

Xilinx<sup>®</sup> Zynq<sup>®</sup>-7000 SoC devices internally provide four high performance (HP) AXI slave interface ports that connect the programmable logic (PL) to asynchronous FIFO interface (AFI) blocks in the processing system (PS). The HP Ports enable a high throughput data path between AXI masters in programmable logic and the processing system's memory system (DDR and on-chip memory). HP slave ports are configurable to 64 bit or 32 bit interfaces.

In this section, you'll create a design using AXI CDMA intellectual property (IP) as master in fabric and integrate it with the PS HP 64 bit slave port. The block diagram for the system is as shown in the following figure.







This system covers the following connections:

- AXI CDMA Slave Port is connected to the PS General Purpose master port 1 (M\_AXI\_GP1). It is used by the PS CPU to configure the AXI CDMA register set for the data transfer and also to check the status.
- AXI CDMA Master Port is connected to the PS High performance Slave Port 0 (S\_AXI\_HP0). It is used by the AXI CDMA to read from the DDR system memory. It acts as the source buffer location for the CDMA during data transfer.
- AXI CDMA Master Port is connected to the PS High performance Slave Port 2 (S\_AXI\_HP2). It is used by the AXI CDMA to write the data to the DDR system memory. It acts as a destination buffer location for the CDMA during the Data transfer.
- AXI CDMA interrupt is connected from fabric to the PS section interrupt controller. After Data Transfer or Error during Data transaction, the AXI CDMA interrupt is triggered.

In this system, you will configure the HP slave port 0 to access a DDR memory location range from 0x20000000 to 0x2fffffff. This DDR system memory location acts as the source buffer location to CDMA for reading the data.

You will also configure HP slave Port 2 to access a DDR memory Location range from 0x3000000 to 0x3fffffff. This DDR system memory location acts as a destination location to CDMA for writing the data.



You will also configure the AXI CDMA IP data width of the Data Transfer channel to 1024 bits with Maximum Burst length set to 32. With this setting, CDMA Maximum transfer size is set to 1024x32 bits in one transaction.

You will write the application software code for the above system. When you execute the code, it first initializes the source buffer memory with the specified data pattern and also clears the destination buffer memory by writing all zeroes to the memory location. Then, it starts configuring the CDMA register for the DMA transfer. It writes the source buffer location, destination buffer location, and number of bytes to be transferred to the CDMA registers and waits for the CDMA interrupt. When the interrupt occurs, it checks the status of the DMA transfers.

If the data transfer status is successful, it compares the source buffer data with the destination buffer data and displays the comparison result on the serial terminal.

If the data transfer status is an error, it displays the error status on the serial terminal and stops execution.

### Example Project: Integrating AXI CDMA with the PS HP Slave Port

- 1. Start with one of the following:
  - Use the system you created in Example Project: Validate Instantiated Fabric IP Functionality, page 36.
  - Create a new project as described in Creating an Embedded Processor Project, page 13.
- 2. Open the Vivado<sup>®</sup> design from Chapter 3 called **edt\_tutorial** and from the IP integrator click **Open Block Design**.
- 3. In the Diagram window, right-click in the blank space and select Add IP.
- 4. In the search box, type **CDMA** and double-click the **AXI Central Direct Memory Access** IP to add it to the Block Design. The AXI Central Direct Memory Access IP block appears in the Diagram view.
- 5. In the Diagram window, right-click in the blank space and select Add IP.
- 6. In the search box type **concat** and double-click the **Concat** IP to add it to the Block Design. The Concat IP block appears in the Diagram view. This block is used to concatenate the two interrupt signals if you are using the prior design with the AXI Timer.
- 7. Right-click the **interrupt** > **IRQ\_F2P[0:0]** net and select delete.
- 8. Click the **IRQ\_F2P[0:0]** port and drag to the **dout[1:0]** output port on the Concat IP core to make a connection between the two ports.



- 9. Click the **interrupt** port on the AXI Timer IP core and drag to the **In0[0:0]** input port on the Concat IP core to make a connection between the two ports.
- 10. Click the **cdma\_introut** port on the AXI CDMA IP core and drag to the **In1[0:0]** input port on the Concat IP core to make a connection between the two ports.
- 11. Right-click the **ZYNQ7 Processing System** core and select **Customize Block**.
- 12. Select **PS-PL Configuration** and expand the **HP Slave AXI Interface**.
- 13. Select the check box for **S AXI HP0 interface** and for **S AXI HP2** interface.
- 14. Click **OK** to accept the changes.
- 15. Right-click the AXI CDMA IP core and select Customize Block.
- 16. Set the block settings in the Re-customize IP wizard screen as follows:

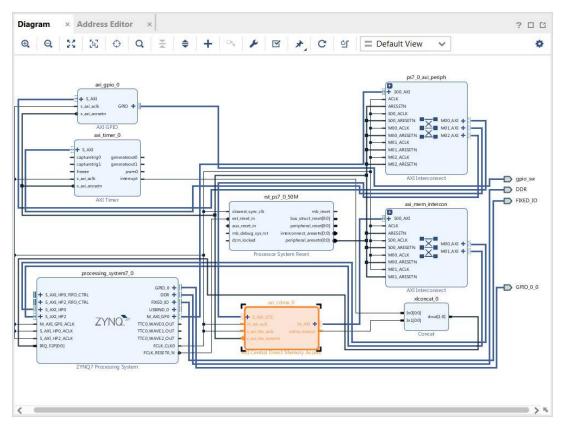
System Property	Setting or Command to Use
Enable Scatter Gather	Unchecked
Disable 4K Boundary Checks	Unchecked
Allow Unaligned Transfers	Unchecked
Write/Read Data Width	1024
Write/Read Burst Size	32
Enable Asynchronous Mode (Auto)	Unchecked
Enable CDMA Store and Forward	Unchecked
Address Width	32

- 17. Click **OK** to accept the changes.
- 18. Click the **Run Connection Automation** link in the Diagram window to automate the remaining connections.
- 19. In the Run Connection Automation dialog box make sure the **All Automation** box is checked, then, click **OK** to accept the default connections. The finished diagram should look like the following figure.

**Note:** You might receive a Critical Message regarding forcibly mapping a net into a conflicting address. You will address the error by manually updating the memory mapped address in the next steps. Click **OK** if you see the error message.









20. Select the Address Editor tab.

Diagram × Address Editor						
Q, ₹ ♦ 📾						•
Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address	
processing_system7_0						
✓ 团 Data (32 address bits : 0x <sup>2</sup>	40000000 [ 1G ])					
🚥 axi_gpio_0	S_AXI	Reg	0x4120_0000	64K -	0x4120_FFFF	
🚥 axi_timer_0	S_AXI	Reg	0x4280_0000	64K *	0x4280_FFFF	
🚥 axi_cdma_0	S_AXI_LITE	Reg	0x7E20_0000	64K ·	0x7E20_FFFF	
🗸 🌻 axi_cdma_0						
🛩 🖪 Data (32 address bits : 4G	)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G •	0x3FFF_FFFF	
processing_system7_0	S AXT HP2	HP2_DDR_LOWOCM	0x0000 0000	1G -	0x3FFF FFFF	

Figure 5-3: Address Editor Tab



- 21. In the Address Editor view, expand **axi\_cdma\_0 > Data**. Right-click on **HP2\_DDR\_LOWOCM** and select **Unmap Segment**.
- 22. In the Range column for **S\_AXI\_HP0**, select **256M**.
- 23. Under Offset Address for **S\_AXI\_HP0**, set a value of **0x2000\_0000**.
- 24. In the Address Editor view, expand **axi\_cdma\_0 > Data > Unmapped Slaves.** Right click on **HP2\_DDR\_LOWOCM** and select Assign Address.
- 25. In the Range column for **S\_AXI\_HP2**, select **256M**.
- 26. Under Offset Address for **S\_AXI\_HP2**, set a value of **0x3000\_0000**.

Diagram × Address Editor	×				? 🗆 [
Q ≚ ♦ 📾					•
Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
v # processing_system7_0					
🗸 🖪 Data (32 address bits : 0x	40000000 [ 1G ])				
🚥 axi_gpio_0	S_AXI	Reg	0x4120_0000	64K •	0x4120_FFFF
🚥 axi_timer_0	S_AXI	Reg	0x4280_0000	64K -	0x4280_FFFF
🚥 axi_cdma_0	S_AXI_LITE	Reg	0x7E20_0000	64K •	0x7e20_ffff
∨ 👎 axi_cdma_0					
🗸 🔣 Data (32 address bits : 4G	)				
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x2000_0000	256M 👻	0x2FFF_FFFF
processing_system7_0	S_AXI_HP2	HP2_DDR_LOWOCM	0x3000_0000	256M 👻	0x3fff_ffff

Figure 5-4: Address Changes to processing\_system7\_0

27. In the Flow Navigator, select Generate Bitstream under Program and Debug.

The Save Project dialog box opens.

- 28. Ensure that the **Block Design tutorial\_bd** check box is selected, then click **Save**.
- 29. A message might appear that states Synthesis is out of date. If it does, click Yes.
- 30. After the Bitstream generation completes, export the hardware and launch the Vitis™ unified software platform as described in Exporting Hardware to the Vitis Software Platform, page 22.

### Standalone Application Software for the Design

The CDMA-based system that you designed in this chapter requires application software to execute on the board. This section describes the details about the CDMA-based Standalone application software.

The main() function in the application software is the entry point for the execution. It initializes the source memory buffer with the specified test pattern and clears the destination memory buffer by writing all zeroes.



The application software then configures the CDMA registers sets by providing information such as source buffer and destination buffer starting locations. To initiate DMA transfer, it writes the number of bytes to be transferred in the CDMA register and waits for the CDMA interrupt to happen. After the interrupt, it checks the status of the DMA transfer and compares the source buffer with the destination buffer. Finally, it prints the comparison result in the serial terminal and stops running.

### **Application Software Flow**

The application software does the following:

1. Initializes the source buffer with the specified test pattern. The source buffer location ranges from 0x20000000 to 0x2fffffff.

Clears the destination buffer by writing all zeroes into the destination address range. The destination buffer location ranges from 0x30000000 to 0x3fffffff.

- 2. Initializes AXI CDMA IP and does the following:
  - a. Associates a CDMA callback function with AXI CDMA ISR and Enable the Interrupt.

This Callback function executes during the CDMA interrupt. It sets the interrupt Done and/or Error flags depending on the DMA transfer status.

Application software waits for the Callback function to populate these flags and executes the software according to the status flag.

- b. Configures the CDMA in Simple mode.
- c. Checks the Status register of the CDMA IP to verify the CDMA idle status.
- d. Sets the source buffer starting location, 0x2000000, to the CDMA register.
- e. Sets the destination buffer starting location, 0x30000000, to the CDMA register.
- f. Sets the number of bytes to transfer to the CDMA register. The application software starts the DMA transfer.
- 3. After the CDMA interrupt is triggered, checks the DMA transfer status.

If the transfer status is successful, the application software compares the source buffer location with the destination buffer location and displays the comparison result on the serial terminal, and then exits from the execution.

If the transfer status displays an error, the software prints the error status in the serial terminal and stops running.



### Running the Standalone CDMA Application Using the Vitis Software Platform

- 1. Open the Vitis software platform.
- Check that the Target Communication Frame (TCF) (hw\_server.exe) agent is running on your Windows machine. If it is not running in the Vitis software platform, select Xilinx > XSCT Console.
- 3. In the XSCT Console window, type **Connect**. A message appears, stating that the hw\_server application started, or if it has started and is running, you see tcfchan#, as shown in the following figure.

	workspace - hello_world/src/helloworld.c - Vitis IDE
le	Edit Bun Developer Xilinx Project Window Help
9	-
	SCT Console 22
	ASCT Frocess
	<pre>***** Xilinx Software Commandline Tool (XSCT) v2019.2.0 **** SW Build 2631120 on Sat Aug 24 09:37:26 MDT 2019 ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.</pre>
	X3DB Server URL: TCF:localhost:53689 xact% X3DB Server Channel: tofchan#0 xact% INFO: [Hsi 55-2053] elapsed time for repository (C:/Xilinx/Vitis/2019.2/data\embeddedsw) loading 1 second xact% connect attempting to launch hw_server
	<pre>***** Xilinx hw_server v2019.2.0 **** Build date : Aug 24 2019 at 10:13:06 ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.</pre>
	INFO: hw_server application started INFO: Use Ctrl-C to exit hw_server application
	xxxxx Xilinx hw_server v2019.2.0
	**** Build date : Aug 24 2019 at 10:13:06
	** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
	INFO: hw_server application started
	INFO: Use Ctrl-C to exit hw_server application
	INFO: To connect to this hw_server instance use url: TCF:127.0.0.1:3121
	tofchan#1 xact%
	4
ľ.	test %

Figure 5-5: Hardware Server Message in XSCT Process Window

- 4. In the Vitis software platform, **Select File > New > Application Project**.
- 5. The New Application Project wizard opens.

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



Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	cdma_app
	Use Default Locations	Select this option
	System project	cdma_app_system
	Platform	hw_platform
	Processor	PS7_cortexa9_0
	Domain	Standalone on ps7_cortexa9_0
	OS	Standalone
	Language	С
Templates	Available Templates	Empty Application

#### 6. Click Finish.

The New Project wizard closes and the Vitis software platform creates the cdma\_app application project under the project explorer.

- 7. In the Project Explorer tab, expand the **cdma\_app** project, right-click the **src** directory, and select **Import** to open the Import dialog box.
- 8. Expand **General** in the Import dialog box and select **File System**.
- 9. Click Next.
- 10. Select Browse.
- 11. Navigate to the design files folder, which you saved earlier (see Design Files for This Tutorial, page 134) and click **OK**.
- 12. Add the cdma\_app.c file and click **Finish**.

As the build succeeds and cdma\_app.elf gets generated. Then build Application project either by clicking the **hammer icon** or by right-clicking on the **cdma\_app project** and selecting Build Project.

**Note:** The Application software file name for the system is cdma\_app.c. It is available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.

13. Open the serial communication utility with baud rate set to 115200.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.

14. Make sure that the hardware board is set up and turned on.

*Note:* Refer to Example Project: Running the "Hello World" Application, page 29 for information about setting up the board.

- 15. Select **Xilinx Tools > Program FPGA** to open the Program FPGA dialog box. The dialog box shows the bitstream path.
- 16. Click **Program** to download the bitstream and program the PL Fabric.



- 17. Run the project similar to the steps in Example Project: Running the "Hello World" Application, page 29.
- 18. Check the Status of the CDMA transfer in the Serial terminal. If the transfer is successful, the message "DMA Transfer is Successful" displays. Otherwise, the serial terminal displays an error message.

# Linux OS Based Application Software for the CDMA System

In this section, you will create a Linux-based application software for CDMA using the mmap() system call provided by Linux and run it on the hardware to check the functionality of the CDMA IP.

The mmap() system call is used to map specified kernel memory area to the User layer, so that you can read or write on it depending on the attribute provided during the memory mapping.

**Note:** Details about the mmap() system call is beyond the scope of this guide.



**CAUTION!** Use of the mmap() call might crash the kernel if it accesses, by mistake, some restricted area or shared resources of the kernel.

The main() function in the application software is the entry point for the execution. It initializes the source array with the specified test pattern and clears the destination array. Then it copies the source array contents to the DDR memory starting at location  $0 \times 20000000$  and makes the DMA register setting to initiate DMA transfer to the destination. After the DMA transfer, the application reads the status of the transfer and displays the result on the serial terminal.

### **Application Software Creation Steps**

Application software creation is composed of the following steps:

- 1. Initialize the whole source array, which is in the User layer with value 0xa5a5a5a5.
- 2. Clear the whole destination buffer, which is in the User layer, by writing all zeroes.
- 3. Map the kernel memory location starting from 0x20000000 to the User layer with writing permission using mmap() system calls.

By doing so, you can write to the specified kernel memory.

- 4. Copy the source array contents to the mapped kernel memory.
- 5. Un-map the kernel memory from the User layer.



- 6. Map the AXI CDMA register memory location to the User layer with reading and writing permission using the mmap() system call. Make the following CDMA register settings from the User layer:
  - a. Reset DMA to stop any previous communication.
  - b. Enable interrupt to get the status of the DMA transfer.
  - c. Set the CDMA in simple mode.
  - d. Verify that the CDMA is idle.
  - e. Set the source buffer starting location, 0x2000000, to the CDMA register.
  - f. Set the destination buffer starting location, 0x3000000, to the CDMA register.
  - g. Set the number of bytes to be transferred in the CDMA register. Writing to this register starts the DMA transfer.
- 7. Continuously read the DMA transfer status until the transfer finishes.
- 8. After CDMA transfer finishes, un-map the CDMA register memory for editing from the User layer using the mmap() system call.
- 9. Map the kernel memory location starting from 0x3000000 to the User layer with reading and writing permissions.
- 10. Copy the kernel memory contents starting from 0x3000000 to the User layer destination array.
- 11. Un-map the kernel memory from the User layer.
- 12. Compare the source array with the destination array.
- 13. Display the comparison result in the serial terminal. If the comparison is successful, the message "DATA Transfer is Successful" displays. Otherwise, the serial terminal displays an error message.

### Running Linux CDMA Application Using the Vitis Software Platform

Detailed steps on running Linux on the target board are outlined in Chapter 6. If you are not comfortable running Linux, run through the Chapter 6 examples prior to running this example. Running a Linux OS based application is composed of the following steps:

- 1. Booting Linux on the Target Board, page 68
- 2. Linux Domain Creation for Linux Applications, page 75
- 3. Building an Application and Running it on the Target Board Using the Vitis Software Platform, page 77

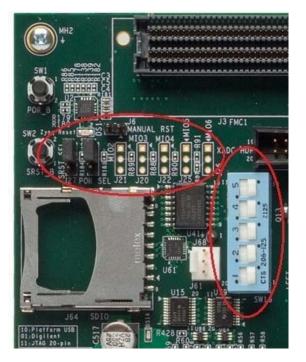


### Booting Linux on the Target Board

You will now boot Linux on the Zynq-7000 SoC ZC702 target board using JTAG mode.

*Note:* Additional boot options will be explained in Chapter 6, Linux Booting and Debug in the Vitis Software Platform.

- 1. Check the following Board Connection and Setting for Linux booting using JTAG mode:
  - a. Ensure that the settings of Jumpers J27 and J28 are set as described in Example Project: Running the "Hello World" Application, page 29.
  - b. Ensure that the SW16 switch is set as shown in the following figure.
  - c. Connect an Ethernet cable from the Zynq SoC board to your network.
  - d. Connect the Windows Host machine to your network.
  - e. Connect the power cable to the board.



*Figure 5-6:* Ensure the SW16 Switch Setting

- 2. Connect a USB Micro cable between the Windows host machine and the target board with the following SW10 switch settings, as shown in Figure 5-7.
  - Bit-1 is 0
  - Bit-2 is 1

**Note:** 0 = switch is open. 1 = switch is closed. The correct JTAG mode has to be selected, according to the user interface. The JTAG mode is controlled by switch SW10 on the ZC702 and SW4 on the ZC706.





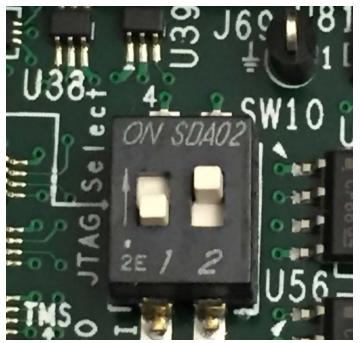
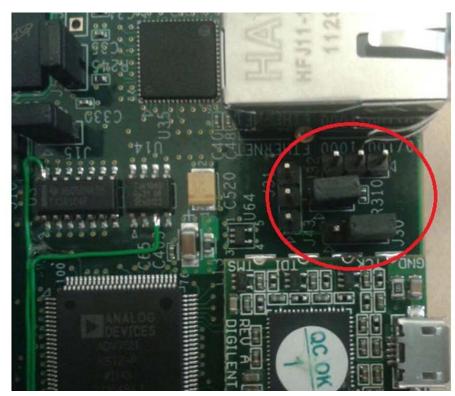


Figure 5-7: SW10 on a ZC702 Set to use Digilent USB JTAG

3. Connect a USB cable to connector J17 on the target board with the Windows Host machine. This is used for USB to serial transfer.



4. Change Ethernet Jumper J30 and J43 as shown in the following figure.

*Figure 5-8:* Change JumpersJ30 and J43



- 5. Power on the target board.
- 6. Launch the Vitis software platform and open the same workspace you used in Chapter 2 and Chapter 3.
- 7. If the serial terminal is not open, connect the serial communication utility with the baud rate set to **115200**.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.

- 8. Select **Xilinx Tools > Program FPGA**, then click **Program** to download the bitstream.
- 9. Open the Xilinx System Debugger (XSCT) tool by selecting **Xilinx Tools > XSCT console**.
- 10. At the XSCT prompt, do the following:
  - a. Type **connect** to connect with the PS section.
  - b. Type targets to get the list of target processors.
  - c. Type targets 2 to select the processor CPU1.

```
xsct% targets
1 APU
2 Arm Cortex-A9 MPCore #0 (Running)
3 Arm Cortex-A9 MPCore #1 (Running)
4 xc7z020
xsct% targets 2
xsct% targets
1 APU
2* Arm Cortex-A9 MPCore #0 (Running)
3 Arm Cortex-A9 MPCore #1 (Running)
4 xc7z020
```

- d. Type dow <tutorial\_download\_path>zynq\_fsbl.elf to download Petalinux FSBL.
- e. Type **con** to start execution of FSBL and then type **stop** to stop it.



<pre>cdma_app_system [ hw_platform ] cdma_test_system [ hw_platfor</pre>	<pre>     Terminal ≥     Serial COM85 (9/11/19, 11-28 AM) ≥      Serial COM85 (9/11/19, 11-28 AM) ≥      XSCT Console ≥      XSCT Process 27% INE 1.7NE/s ??:?? ETA 45% INE 1.7NE/s ??:?? ETA 45% INE 1.7NE/s ??:?? ETA 94% 3ME 1.8NE/s ??:?? ETA 94% 3ME 1.8NE/s ??:?? ETA 10% 3ME 1.8NE/s ??:?? ETA 10% SME 1.8NE/s ??:?? ETA 10% AFE (7.200 Connect 1 APU 2 ARM Cortex-A5 MPCore \$0 (Running) 3 ARM Cortex-A5 MPCore \$1 (Running) 4 * x07x020 xact% targets 2 xact% targets 2 xact% targets 1 1 APU </pre>	© x  I © = □
<pre>cdma_app_system [ hw_platform ] cdma_test_system [ hw_platform ] cdma_test_system [ hw_platform ] hello_world_system [ hw_platform ] hello_world_system [ hw_platform ] b dsa b dsa b dsa b export b dsa b export b logs c ps7_cortexa9_0 b resources b tempdsa c 2 xnq_lsbl c platform.spr linux_cdma_app_system [ hw_platform ] c] linux_cdma_app [ linux_application_domain ] c ** Platform.spr linux_cdma_app [ linux_application_domain ] c** Platform.spr linux_cdma_app [ linux_application_domain ] c** Platform.spr </pre>	Serial COM85 (9/11/19, 11:28 AM) 28          * XSCT Console 32         XSCT Process         27%         1MB         1.7MB/s         27%         1MB         1.7MB/s         21%         1.7MB/s         21%         1.8MB/s         2.8MB         1.8MB/s         2.8MB         2.8MB         1.8MB/s         2.8MB         2.8MB         2.8MB         3.8MB         3.8MB         3.8MB         44         45         45         45         45         5.8MB/s         5.8MB/s         5.8MB/s         5.8MB         5.8MB         5.8MB         5.8MB         5.8MB         5.8MB          6.8MB         6.8MB         7.8         7.8         7.8         7.8         7.8         7.8         7.8         7.8         7.8         7.8         7.8	
<pre>cdma_app_system [ hw_platform ] cdma_test_system [ hw_platform ] cdma_test_system [ hw_platform ] challe_world_system [ hw_platform ]</pre>	XSCT Console 32           XSCT Process           27%         IMB 1.9MB/s ??:?? ETA           45%         IMB 1.7MB/s ??:?? ETA           73%         2MB 1.6MB/s ??:?? ETA           94%         3MB 1.7MB/s ??:?? ETA           100%         3MB 1.6MB/s 00:02           connect         tofkhanf2           xsact% targets         1           1 APU         2           2 ARM Cortex-A9 MPCore \$0 (Running)           3 ARM Cortex-A9 MPCore \$1 (Running)           4* xo7z020           xsact% targets           xsact% targets	
hello_world_system [ hw_platform ] 5 hw_platform 6 bistream 6 dsa 6 export 6 logs 6 ps7_cortexa9_0 7 ersources 8 tempdsa 6 zynq_(sbl 7 platform.spr 9 inux_cdma_app_system [ hw_platform ] 10 inux_cdma_app [ linux_application_domain ] 10 inv interest	XSCT Process 275 IMB 1.9MB/s ??:?? ETA 475 IMB 1.7MB/s ??:?? ETA 735 2MB 1.8MB/s ??:?? ETA 945 3MB 1.7MB/s ??:?? ETA 1005 3MB 1.8MB/s 00:02 connect tofshanf2 xsct% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xsct% targets 2 xsct% targets	
bity_platform bitstream dsa export ps7_cortexa9_0 eresources bitempdsa zynq_fsbl platform.spr inux_cdma_app_system [hw_platform] (*) Inux_cdma_app [inux_application_domain]	XSCT Process 275 IMB 1.9MB/s ??:?? ETA 475 IMB 1.7MB/s ??:?? ETA 735 2MB 1.8MB/s ??:?? ETA 945 3MB 1.7MB/s ??:?? ETA 1005 3MB 1.8MB/s 00:02 connect tofshanf2 xsct% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xsct% targets 2 xsct% targets	
bitstream bitst	XSCT Process 275 IMB 1.9MB/s ??:?? ETA 475 IMB 1.7MB/s ??:?? ETA 735 2MB 1.8MB/s ??:?? ETA 945 3MB 1.7MB/s ??:?? ETA 1005 3MB 1.8MB/s 00:02 connect tofshanf2 xsct% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xsct% targets 2 xsct% targets	
bistream     bistream     bistream     bistream     bistream     bistream     bistream     logs     bigs     ps7_cortexa9_0     bistream     tempdsa     bistream     platform.spr     linux_cdma_app_system [hw_platform]     w     linux_cdma_app [linux_application_domain]     w	XSCT Process 275 IMB 1.9MB/s ??:?? ETA 475 IMB 1.7MB/s ??:?? ETA 735 2MB 1.8MB/s ??:?? ETA 945 3MB 1.7MB/s ??:?? ETA 1005 3MB 1.8MB/s 00:02 connect tofshanf2 xsct% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xsct% targets 2 xsct% targets	
b dsa     b dsa     b export     b logs     b ps7_cortexa9_0     b resources     b tempdsa     b zynq_fsbl     v platform.spr     linux_cdma_app_system [hw_platform]     d) linux_cdma_app [linux_application_domain]     b **	27% 1MB 1.9MB/s ??:?? ETA 49% 1MB 1.7MB/s ??:?? ETA 73% 2MB 1.8MB/s ??:?? ETA 94% 3MB 1.7MB/s ??:?? ETA 100% 3MB 1.8MB/s 00:02 connect tcEchan#2 xact% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7z020 xact% targets 2 xact% targets 2 xact% targets 2	
bogs     bogs     bryz_cortexa9_0     bresources     be tempdaa     cortexa1(sbl     platform.spr     linux_cdma_app_system [hw_platform]     fillux_cdma_app[_linux_application_domain]     break	734 2MB 1.8MB/s ??:?? ETA 944 3MB 1.7MB/s ??:?? ETA 1004 3MB 1.8MB/s 00:02 connect tcfchan#2 xact% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xact% targets 2 xact% targets 2 xact% targets	
ps7_cortexa9_0     resources     tempdsa     zynq_fsbl     platform.spr     linux_cdma_app_system [ hw_platform ]     linux_cdma_app [ linux_application_domain ]     *** Planaere	944 3MB 1.7MB/s ??:?? ETA 1004 3MB 1.6MB/s 00:02 connect tcEchan#2 xsct% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7z020 xsct% targets 2 xsct% targets	87
resources  tempdsa  zynq_fsbl  platform.spr  linux_cdma_app_system [ hw_platform ]  linux_cdma_app [ linux_application_domain ]   " Planacetma	<pre>100% 3MB 1.0MB/s 00:02 connect toEchan#2 xact% targets 1 APU 2 ARM Cortex-A9 MPCore #0 (Running) 3 ARM Cortex-A9 MPCore #1 (Running) %* xc7s020 xact% targets 2 xact% targets</pre>	
<ul> <li>tempdsa</li> <li>zynq_fsbl</li> <li>platform.spr</li> <li>inux_cdma_app_system [ hw_platform ]</li> <li>linux_cdma_app [ linux_application_domain ]</li> <li>the platform [ hw_platform ]</li> </ul>	<pre>tefchan#2 xact5 targets 1 APU 2 ARM Cortex-A9 MPCore #0 (Running) 3 ARM Cortex-A9 MPCore #1 (Running) 4* xc7s020 xact5 targets 2 xact5 targets</pre>	
> zynq_fsbl r platform.spr nux_cdma_app_system [ hw_platform ] ) linux_cdma_app [ linux_application_domain ] # Planate	<pre>xact% targets 1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xact% targets 2 xact% targets 2 xact% targets</pre>	
zynic_sol platform.spr ux_cdma_app_system [ hw_platform ] linux_cdma_app [ linux_application_domain ] ** Dinacide	1 APU 2 ARM Cortex-A9 MPCore \$0 (Running) 3 ARM Cortex-A9 MPCore \$1 (Running) 4* xc7x020 xsct% targets 2 xsct% targets	
ux_cdma_app_system   <u>hw_platform</u> ]  inux_cdma_app [ linux_application_domain ] # <sup>®</sup> Pinater	3 ARM Cortex-A5 MPCore \$1 (Running) 4* xc7z020 xact5 targets 2 xact6 targets	
linux_cdma_app [ linux_application_domain ]	4 x07x020 xactt targeta 2 xactt targeta	
Pinarias	xact% targets 2 xact% targets	
1 Includes		
🔁 ide	2* ARM Cortex-A9 MPCore #0 (Running)	
B Debug	3 ARM Cortex-A5 MPCore #1 (Running)	
	4 xc7z020	
project.sdx	xact% dow C:/designa/boot/zynq_fabl.elf	
	Downloading Program C:/designs/boot/zyng_fsbl.elf	
Market States	section, .text: 0x00000000 - 0x0001137f	
	section, .handoff: 0x00011380 - 0x000113cb section, .init: 0x000113cc - 0x000113d7	
	section, fini: 0x000113d9 - 0x000113e3	
	section, .rodata: 0x000113e4 - 0x00011a2b	
	<pre>section, .data: 0x00011a30 - 0x00014b7f section, .mmu_tbl: 0x00018000 - 0x0001bfff</pre>	
	section, .imit_array: 0x0001c000 - 0x0001c003	
tant 🛙 📄 🕀 🧶 🖉 🕸 🐨 🗖 🗖	section, .fini_array: 0x0001c004 - 0x0001c007	-
dma_app_system [System]	<pre>section, .rsa_ac: 0x0001c008 - 0x0001d03f section, .bss: 0x0001d040 - 0x0001f271</pre>	
dma_app_system [System] dma_test_system [System]	section, .heap: 0x00012272 - 0x0002127f	
lo_world_system [System]	section, .stack: 0xffff0000 - 0xffffd3ff	
platform [Platform]	0% 0HB 0.0HB/s ??:?? ETA	
cdma_app_system [System]	100% OMB 0.4MB/s 00:00	
us ofma ann lApplication!		
COLUMN SAN TANKA COLUMN SAN SAN SAN SAN SAN SAN SAN SAN SAN SA	Setting PC to Program Start Address 0x00000000 Successfully downloaded C:/designs/boot/zyng fsbl.elf	
	xsct% Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffff28 (Suspended)	
	xact% targets	
	1 APU 2 <sup>+</sup> ABM Cortex-A9 MPCore #0 (Suspended)	
	3 ARM Cortex-A9 MPCore \$1 (Suspended)	
	4 xc7z020	
	xsct% con Info: ARM Cortex-A9 MPCore #0 (target 2) Running	
	Info: ARM Cortex-A9 MPCore #0 (target 2) Running xsct% stop	
	Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x113a0 (Suspended)	
	xsct% targets	
	1 APU 2* ARN Cortex-A9 MPCore #0 (Sumpended)	
	3 ARM Cortex-A9 MPCore \$1 (Running)	
	4 xc7z020	
	xsct%	

*Figure 5-9:* **XSCT Console** 

- f. Type dow <tutorial\_download\_path>/u-boot.elf to download PetaLinux U-Boot.elf.
- g. Type **con** to start execution of U-Boot.

On the serial terminal, the autoboot countdown message appears: Hit any key to stop autoboot: 3

h. Press Enter.



Automatic booting from U-Boot stops and a command prompt appears on the serial terminal.

Edit Run Developer Xilinx Project Window Help		and a second reasons
• 🖩 🕼 🖲 • 🔦 • 10 💽 🌬 🏾 🖷 🕫 2. 👁 2. 🗟 🗮 3		ess 🛛 🖉 Design 🔯 D
🕸 Debug 🔂 Project Explorer 😂 🅦 Explorer 👘 🗖	🖉 Terminal 🛙 🖳 🕅 🖏	REEST
<ul> <li>Richma_app_system [hw_platform]</li> <li>Richma_test_system [hw_platform]</li> <li>Richma_test_statistics</li> <li>Richma_test_statistics</li></ul>	U-Boot 2019.01 (Sep 10 2019 - 12:53:15 +0000) Xilinx Zynq ZC702 CPU: Zynq 7z020 Silicon: v3.1 Model: Zynq ZC702 Development Board DRAM: ECC disabled 1 GiB MMC: mmc@e01000000: 0 Loading Environment from SPI Flash SF: Detected n25q128a with page size 256 Bytes, erass otal 16 MiB *** Warning - bad CRC, using default environment In: serial@e0001000 Out: serial@e0001000 Err: serial@e0001000 Ford: Zynq ZC702 Development Board Net: ZYNQ GEM: e000b000 kaiting for PHY auto negotiation to complete done BOOTP broadcast 1 BOOTP broadcast 2 Abort Hit any key to stop autoboot: 0 Zynq3 Zynq3 Zynq3	e size 64 KiB, t
	-	
dma_app_system (System)		
<ul> <li>Brown (System)</li> <li>Coma, test, system (System)</li> <li>Brown (System)</li> <li>Brown (System)</li> <li>Invx_codma_app_system (System)</li> <li>Invx_codma_app (Application)</li> </ul>	XSCT Console XX         XSCT Freess         dew C:/designs/boot/u-boot.elf         section, .data: No00400000 - 0x0048e736         04 0MB 0.0MB/s ??:?? TTA         364 0MB 0.0MB/s ??:?? TTA         365 0MB 0.0MB/s ??:?? TTA         1004 0MB 0.4MB/s 00:01         Successfully downloaded C:/designs/boot/u-boot.elf         xsect*	Console R Console Program FPGA
<ul> <li></li></ul>	<pre>XSCT Process dow C:/designs/boot/u-boot.elf Downloeding Program C:/designs/boot/u-boot.elf section, .data: 0x00400000 - 0x0048e736 04 0MB 0.0MB/s ??:?? TTA 364 0MB 0.0MB/s ??:?? TTA 785 0MB 0.4MB/s ??:?? TTA 785 0MB 0.4MB/s ??:?? TTA 100% 0MB 0.4MB/s ??:??? TTA 100% 0MB 0.4MB/s ??:?? TTA 100% 0MB 0.4MB/s ??:???? TTA 100% 0MB 0.4MB/s ??:??? TTA 100% 0MB 0.4MB/s ??:??? TTA 100% 0MB 0.4MB/s ??:??? TTA 100% 0MB 0.4MB/s ??:???? TTA 100% 0MB 0.4MB/s ??!???? TTA 100% 0MB 0.4MB/s ?????? TTA 100% 0MB 0.4MB/s ????????? TTA 100% 0MB 0.4MB/s ?????? TTA 100% 0MB 0.4MB/s ?????? TTA 100% 0MB 0.4MB/s ?????? TTA 100% 0MB 0.4MB/s ?????? TTA 100% 0MB 0.4MB/s ????? TTA 100% 0MB 0.4MB/s ????? 100% 0MB 0.4MB/s ????? 100% 0MB 0.4MB/s ????? 100% 0MB 0.4MB/s ????? 100% 0MB 0.4MB/s ???? 100% 0MB 0.4MB/s ???? 100% 0MB 0.4MB/s ???? 100% 0MB 0.4MB/s ??? 100% 0MB 0.4MB/s ??? 100% 0MB 0.4MB/s ?? 100% 0MB 0.4MB/s ??</pre>	

Figure 5-10: Serial Terminal

i. At the XSCT Prompt, type **stop**.

The U-Boot execution stops.

- j. Type **dow -data <tutorial\_download\_path>/image.ub 0x3000000** to download the Linux Kernel image at location 0x3000000.
- k. Type **con** to start executing U-Boot.
- 11. At the command prompt of the serial terminal, type **bootm 0x30000000**.

The Linux OS boots.



12. If required, provide the Zynq login as **root** and the password as **root** on the serial terminal to complete booting the processor.

After booting completes, # prompt appears on the serial terminal.

- 13. At the root@Xilinx-ZC702-2019\_2:~# prompt, make sure that the board Ethernet connection is configured:
  - a. Check the IP address of the board by typing the following command at the Zynq> prompt: **ifconfig eth0**.

This command displays all the details of the currently active interface. In the message that displays, the inet addr value denotes the IP address that is assigned to the Zynq SoC board.

b. If inet addr and netmask values do not exist, you can assign them using the following commands:

root@Xilinx-ZC702-2019\_2:~# ifconfig eth0 inet 192.168.1.10 root@Xilinx-ZC702-2019\_2:~# ifconfig eth0 netmask 255.255.255.0

- 14. Confirm that the IP address settings on the Windows machine are set up to match the board settings. Adjust the local area connection properties by opening your network connections.
  - a. Right-click the local area connection that is linked to the XC702 board and select **Properties**.
  - b. In the Local Area Connection Properties dialog box, select **Internet Protocol Version 4 (TCP/IPv4)** from the item list and select **Properties**.
  - c. Select Use the following IP address and set the following values:

IP address: 192.168.1.11 Subnet mask: 255.255.255.0

d. Click **OK** to accept the values.



15. In the Windows machine command prompt, check the connection with the board by typing **ping** followed by the board IP address. The ping response displays in a loop.

This response means that the connection between the Windows host machine and the target board is established.

16. Press **Ctrl+C** to stop displaying the ping response on the Windows host machine command prompt.

Cobleg & Poject Explore:     Constraint:	<ul> <li>Debug Project Explorer S Debug Project Explorer Discrete State State</li></ul>	t) (gcc version 8.2.0 (GCC)) #1 SMP PREEMPT Tue Sep 10 E:53 cope:Link Metric:1 :0 frame:2 :0 carrier:0 (5.6 KiB)
<pre>     dima.gop.ystem   Nuc_stations]     fe dima.gop.ystem   Nuc_stations]     fe dima.got.ystem   Nuc_s</pre>	<ul> <li>Gema, app.system [hw.platform]</li> <li>Gemalest.system [hw.platform]</li> <li>Gemalest.system [hw.platform]</li> <li>Shup_lattorm</li> <li>Shup_</li></ul>	t) (gcc version 8.2.0 (GCC)) #1 SMP PREEMPT Tue Sep 10 E:53 cope:Link Metric:1 :0 frame:2 :0 carrier:0 (5.6 KiB)
<pre># dima_scg.spaten [hw_glation] # Gima_scg.spaten [hw_glat</pre>	<pre>bit cdma_app_system [hw_platform] bit cdmm_tex_system [hw_platform] bit hellow codd system [hw_platform] bit hellow c</pre>	E:53 cope:Link Metric:1 s:0 frame:2 :0 carrier:0 (5.6 KiB)
<pre>Assistant %</pre>	RX bytes:12137 (11.8 KiR) TX bytes:9894 (	55.255.255.0 E:53 55 Mask:255.255.255.0 cope:Link Metric:1 s:0 frame:2 :0 carrier:0
X3CT Process       Symp 00103 ErA       Image: Symp 00103 ErA         S904 9080 0.3MB/s 00102 ETA       Symp 00102 ETA         524 9080 0.3MB/s 00101 ETA       Symp 00101 ETA         964 10MB 0.3MB/s 00100 ETA       Symp 00100 ETA	Assistant ⊠	1 ms 5 ms
100% 10MB 0.3MB/s 00:32 Successfully downloaded C://seigns/boot/image.ub wact% con Info: ABM Cortex-A9 MPCore #0 (target 2) Running xact%	XBCT Process 90% 90H 0.3ME/s 00103 FIX 90% 90H 0.3ME/s 00102 FTA 92% 90H 0.3ME/s 00102 FTA 94% 90H 0.3ME/s 00101 FTA 96% 10MB 0.3ME/s 00100 FTA 97% 10MB 0.3ME/s 00100 FTA 100% 10ME 0.3ME/s 00100 FTA 100% 10ME 0.3ME/s 00132 Successfully downloaded C:/designs/boot/image.ub Mact% con Info: RAM Cortex-A9 MPCore #0 (target 2) Running xact%	Program FPGA

Figure 5-11: Windows Host Machine Command Prompt

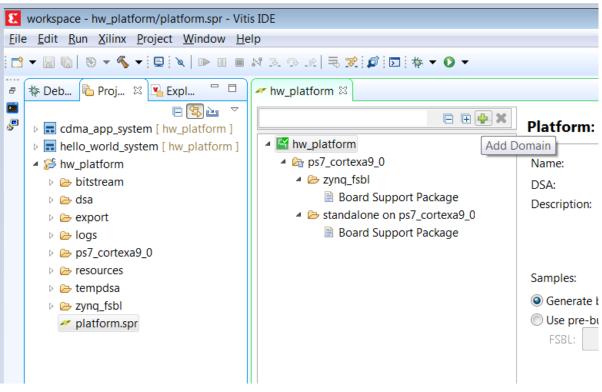
Linux booting completes on the target board and the connection between the host machine and the target board is complete.



# **Linux Domain Creation for Linux Applications**

Now that Linux is running on the board, you can create a Linux domain followed by a Linux application. The steps to create a Linux domain are given below:

- 1. Go to the explorer in the Vitis software platform and expand the hw\_platform platform project.
- 2. Open the hardware by double clicking **platform.spr**.
- 3. The platform explorer opens. Click the + button in the right corner to add a domain, as shown in the following figure.



*Figure 5-12:* Add a Domain

- 4. When the new domain window opens, enter the details as given below
  - a. Name > linux\_domain
  - b. Display Name > linux\_application\_domain
  - c. OS > Linux
  - d. Processor > ps7\_cortexa9
  - e. Supported Runtimes > C/C++
  - f. Select Use pre-built software components, then under this:



- Create one boot directory in the C:\designs folder, then copy the boot components into it: (FSBL, PMUFW from the Vitis software platform, ATF, u-boot.elf and image.ub from PetaLinux.
- Create one BIF file as below.

pe filter text	8 8 <b>4</b> X	Platform:	hw_platform		
<ul> <li>hw_platform</li> <li>ps7_cortexa9_0</li> <li>ynq_fsbi</li> </ul>		Name: DSA:	hw_platform tutorial bd wrapper.xsa		
📄 Board Su 🔺 😂 standalone d	pport Package on ps7_cortexa9_0 pport Package	DSA: Description:			1
E New Domain in 'hw	_platform'			[3]	Q
Domain Create a new domain	n.				. Q 1%
Name:	linux_domain				
Display Name:	linux_application_domain	n			
OS:	linux	•			
Processor:	ps7_cortexa9	•			
Supported Runtimes:	C/C++	•			
Use pre-built softw	vare components				
Boot Directory:	C:\designs\boot			Browse	
Bif File:	C:\designs\boot\cdma_tes	st.bif		Browse	
Generate software	components				
(?)			ОК	Cancel	

*Figure 5-13:* **Create BIF File** 

g. Click **OK** to finish and observe that the Linux domain has been added to the hw\_platform as shown below.

8 E 🔶 🗙	Domain: linux d	lomain		
	OS: Processor Supported Runtimes: Display Name Description	Inax pd7_contexa@ <u>CPC++: *</u> Inax_application_domain		
Board Support Package	Boot Directory: Bit File:	C/designs/boot C/designs/boot/cdma_test.bif	Bjowse. Bjowse.	
	Image: Swroot		Becavie	
	QEMU Data QEMU Arguments		Bjowse.,	
	<b>x</b> • <b>x</b>	OS Processor Supported Runtimes Display Name Description Boot Directory Boot Directory Bit File Image Sysroct QEMU Data	Officer     Instruction       OS     Instruction       Processor     p1_contexa@       Supported Runtimes     C/C++ *       Display Name     Insu_application_domain       Description     Insu_application_domain       Boot Directory:     C/designs/boot       Bit File     C/designs/boot/colma_bet/bit       Image:     Systect       QSMU Data     Image	Ook         Inux           OS:         Inux           Processor:         p0_centes@           Supported Runtimes:         GC++-           Display Name:         Inux_application_domain           Description:         Bioxide           Boot Directory:         C/designu/boot           Bit File:         C/designu/boot           Image:         Systect:           Systect:         Bioxee.           QEMU Data         Bioxee.



h. Add **image.ub** file path in Image file path.



Now you are ready with Linux domain to create Linux applications.

## Building an Application and Running it on the Target Board Using the Vitis Software Platform

- 1. Now that Linux is running on the board, we will create a linux application to utilize the CDMA. Select **File > New > Application Project**.
- 2. Use the information in the table below to make your selections in the wizard screens.

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	linux_cdma_app
	Syatem project	linux_cdma_app_system
	Platform	<hw_platform></hw_platform>
	CPU	cortex-a9
	Domain	linux_application_domain
	OS	linux
	Language	С
Templates	Available Templates	Linux Empty Application

#### 3. Click Finish.

The New Project wizard closes and the Vitis software platform creates the linux\_cdma\_app project under the project explorer.

- 4. In the Project Explorer tab, expand linux\_cdma\_app project, right-click the **src** directory, and select **Import** to open the Import dialog box.
- 5. Expand **General** in the Import dialog box and select **File System**.
- 6. Click Next.
- 7. Add the linux\_cdma\_app.c file and click **Finish**.

Build Application project either by clicking the **hammer icon** or by right-clicking on the **linux\_cdma\_app project** and selecting Build Project. Binary file linux\_cdma\_app.elf gets generated.

**Note:** The example application software file for the system is linux\_cdma\_app.c. This file is available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.

- 8. Right-click linux\_cdma\_app and select **Run As > Run Configurations** to open the Run Configurations wizard, shown in the following figure.
- 9. Right-click Xilinx C/C++ application (Application Debugger) and select New.



Run Configurations		
Create, manage, and run configurations Debug a program using Application Debugger.		
○ 2 3 3 3 × 8 3 -	Name: AXI DMA Debug	
type filter text	🔀 Main 🔲 Application 🐵 Target Setup 🍄 Arguments 👼 Environment 🐺 Symbol Files 🕏 Source 🕹 Path Map 🖾 Common	
CopenCL (TCF)     OpenCL (TCF)     Target Communication Framework     Xilinx AI Engine SystemC Simulator     Xilinx AI Engine S48 Simulator	Debug Type: Linux Application Debug  Connection: Linux Agent  New Note: TCF agent port should be used as port in the target connection (Default TCF agent port: 1534).	
<ul> <li>Š<sub>0</sub> Xilinx Application Debugger</li> <li>Š<sub>0</sub> AXI DMA Debug</li> </ul>	Project linux_cdma_app Configuration: Debug	Browse
Xilinx Application Debugger (GDB) Xilinx SPM Analysis Xilinx System Debugger Xilinx SystemC-RTL Co-Simulator	Configuration: Debug     Enduation     Performance Analysis	

Figure 5-15: Run Configurations Setup

- 10. In the **Connection** tab, click **New** to open the New Connection wizard.
- 11. In the New Target Connection screen, apply the settings below:
  - a. Specify a name in the **Target Name** field. For the purposes of this exercise, use CDMALinux.
  - b. In the **Host** field, enter the target board IP address.

To determine the target board IP address, type ifconfig eth0 at the Zynq> prompt in the serial terminal. The terminal displays the target IP address that is assigned to the board.

- c. In the **Port** field, type 1534.
- 12. Click **OK** to create the connection.
- 13. As shown in the following figure, from the **Application** tab, enter application data settings for the following:
  - a. Project Name: linux\_cdma\_app
  - b. Local File Path: Debug/linux\_cdma\_app.elf
  - c. Remote File Path: /tmp/cdma.elf



reate, manage, and run configurations		F
Debug a program using Application Debugger.		<b>W</b>
8008888	Name: AXI DMA Debug	
type filter text Launch Group OpenCL OpenCL (TCF)	Main C Application     Target Setup     Arguments     Environment     Symbol Files     Symbol Files	& Path Map 🖾 Common
Target Communication Framework     Xilinx Al Engine System C Simulator     Xilinx A Engine & Simulator     Xilinx Application Debugger     Xilinx Application Debugger     Xilinx Syntamic Debugger     Xilinx Syntam Debugger     Xilinx System C-RTL Co-Simulator	Debug/linux.cdma_app.elf      Remote File Path:     /tmp/cdma.elf      Working directory      Use default      Auto-attach process children     Stop at program entry     Stop at main'     Disconnect when last debug context exits     Vue pseudo-terminal for process standard VO     Hide debug contexts started by other debug sessions      4	Search Browse
ilter matched 12 of 21 items		Revert Apply

Figure 5-16: Debug Configuration Settings in the Application Tab

14. Click **Run**. The application executes, and the message **DATA Transfer is Successful** appears in the console window, as shown in the following figure.

Image: Second system       Image: Second system         Image: Secon	ጲ,⊉;⊡; ‡ ▾ Ο ▾
Image: Second state     Image: Second s	Console      TCF Debug Process Terminal - P1257     /dev/mem opened.     Memory mapped at address 0xa6e0f000.     /dev/mem opened.     Transfer Completed     /dev/mem opened. Memory mapped at address 0xa6e0f000. DATA Transfer is Successful

Figure 5-17: Data Transfer Message



# Chapter 6

# Linux Booting and Debug in the Vitis Software Platform

This chapter describes the steps to configure and build the Linux OS for Zynq®-7000 SoC board with PetaLinux Tools. It also provides information about downloading images precompiled by Linux on the target memory using a JTAG interface.

The later part of this chapter covers programming the following non-volatile memory with the precompiled images, which are used for automatic Linux booting after switching on the board:

- On-board QSPI Flash
- SD card

This chapter also describes using the remote debugging feature in the Xilinx® Vitis<sup>™</sup> unified software platform to debug Linux applications running on the target board. The Vitis software platform runs on the Windows host machine. For application debugging, the platform establishes an Ethernet connection to the target board that is already running the Linux OS.

For more information, see the Embedded Design Tools web page [Ref 12].

# Requirements

In this chapter, the target platform refers to a Zynq SoC board. The host platform refers to a Windows machine that is running the Vivado® tools and PetaLinux installed on a Linux machine (either physical or virtual).

*Note:* The Das U-Boot universal bootloader is required for the tutorials in this chapter. It is included in the precompiled images that you will download next.

From the Xilinx documentation website, download the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134. It includes the following files:

• BOOT.bin: Binary image containing the FSBL and U-Boot images produced by bootgen.



- cdma\_app.c: Standalone Application software for the system you will create in this chapter.
- helloworld.c: Standalone Application software for the system you created in Chapter 3.
- linux\_cdma\_app: Linux OS based Application software for the system you will create in this chapter.
- README.txt: Copyright and release information pertaining to the ZIP file.
- u-boot.elf: U-Boot file used to create the BOOT.BIN image.
- Image.ub: PetaLinux build Image (which have kernel image, ramdisk and dtb)
- fsbl.elf: FSBL image used to create BOOT.BIN image.

# **Booting Linux on a Zynq SoC Board**

This section covers the flow for booting Linux on the target board using the precompiled images that you downloaded in Requirements, page 80.

*Note:* The compilations of the different images like Kernel image, U-Boot, Device tree, and root file system is beyond the scope of this guide.

# **Boot Methods**

The following boot methods are available:

- Master Boot Method
- Slave Boot Method

#### Master Boot Method

In the master boot method, different kinds of non-volatile memories such as QSPI, NAND, NOR flash, and SD cards are used to store boot images. In this method, the CPU loads and executes the external boot images from non-volatile memory into the Processor System (PS). The master boot method is further divided into Secure and Non Secure modes. Refer to the *Zynq-7000 SoC Technical Reference Manual* (UG585) [Ref 1] for more detail.

The boot process is initiated by one of the Arm Cortex-A9 CPUs in the processing system (PS) and it executes on-chip ROM code. The on-chip ROM code is responsible for loading the first stage boot loader (FSBL). The FSBL does the following:

- Configures the FPGA with the hardware bitstream (if it exists)
- Configures the MIO interface



- Initializes the DDR controller
- Initializes the clock PLL
- Loads and executes the Linux U-Boot image from non-volatile memory to DDR

The U-Boot loads and starts the execution of the Kernel image, the root file system, and the device tree from non-volatile RAM to DDR. It finishes booting Linux on the target platform.

#### Slave Boot Method

JTAG can only be used in slave boot mode. An external host computer acts as the master to load the boot image into the OCM using a JTAG connection.

*Note:* The PS CPU remains in idle mode while the boot image loads. The slave boot method is always a non-secure mode of booting.

In JTAG boot mode, the CPU enters halt mode immediately after it disables access to all security related items and enables the JTAG port. You must download the boot images into the DDR memory before restarting the CPU for execution.

# **Booting Linux from JTAG**

The flow chart in the following figure describes the process used to boot Linux on the target platform.

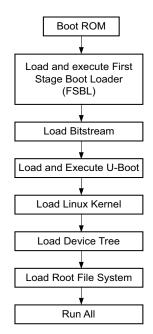


Figure 6-1: Linux Boot Process on the Target Platform



# Preparing the PetaLinux Build for Debugging

To debug Linux applications (using tcf-agent), you must manually enable tcf-agent in PetaLinux RootFS.

Ensure that dropbear-openssh-sftp server is disabled in PetaLinux RootFS.

**Note:** The Vitis debugger supports Linux Application Debug using tcf-agent (TCF - Target Communication Framework). TCF agent is provided as a part of PetaLinux roofts packages, but needs to be enabled when required.

Detailed information on enabling these components in the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 8], section "Debugging Applications with TCF Agent."



# **Booting Linux Using JTAG Mode**

- 1. Check the following board connections and settings for Linux booting using JTAG mode:
  - a. Ensure that the settings of Jumpers J27 and J28 are set as described in Example Project: Running the "Hello World" Application, page 29.
  - b. Ensure that the SW16 switch is set as shown in the following figure.
  - c. Connect an Ethernet cable from the Zynq®-7000 SoC board to your network or directly to your host machine.
  - d. Connect the Windows Host machine to your network.
  - e. Connect the power cable to the board.



Figure 6-2: Ensure the SW16 Switch Setting

- 2. Connect a USB Micro cable between the Windows host machine and the target board with the following SW10 switch settings, as shown in the following figure.
  - Bit-1 **is** 0
  - Bit-2 is 1

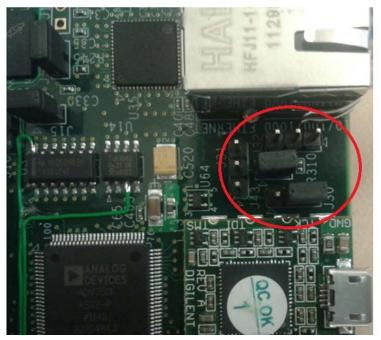
**Note:** 0 = switch is open. 1 = switch is closed. The correct JTAG mode has to be selected, according to the user interface. The JTAG mode is controlled by switch SW10 on the zc702 and SW4 on the zc706.





Figure 6-3: SW4 on a ZC706 Set to use Digilent USB JTAG

- 3. Connect a USB cable to connector J17 on the target board with the Windows Host machine. This is used for USB to serial transfer.
- 4. Change Ethernet Jumper J30 and J43 as shown in the following figure.



*Figure 6-4:* Change Jumpers J30 and J43

- 5. Power on the target board.
- 6. Launch the Vitis software platform and open same workspace you used in Chapter 2 and Chapter 3.
- 7. If the serial terminal is not open, connect the serial communication utility with the baud rate set to **115200**.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.





- 8. Download the bitstream by selecting **Xilinx Tools > Program FPGA**, then clicking **Program**.
- 9. Open the Xilinx System Debugger (XSCT) tool by selecting **Xilinx Tools > XSCT console**.
- 10. At the XSCT prompt, do the following:
  - a. Type **connect** to connect with the PS section.
  - b. Type targets to get the list of target processors.
  - c. Type ta 2 to select the processor CPU1.

```
xsct% targets
1 APU
2 Arm Cortex-A9 MPCore #0 (Running)
3 Arm Cortex-A9 MPCore #1 (Running)
4 xc7z020
xsct% ta 2
xsct% targets
1 APU
2* Arm Cortex-A9 MPCore #0 (Running)
3 Arm Cortex-A9 MPCore #1 (Running)
4 xc7z020
```

- d. Type dow <tutorial\_download\_path>zynq\_fsbl.elf to download Petalinux FSBL.
- e. Type **con** to start execution of FSBL and then type **stop** to stop it.
- f. Type dow <tutorial\_download\_path>/u-boot.elf to download PetaLinux U-Boot.elf.
- g. Type **con** to start execution of U-Boot.

On the serial terminal, the autoboot countdown message appears:

Hit any key to stop autoboot: 3

h. Press Enter.

Automatic booting from U-Boot stops and a command prompt appears on the serial terminal.

i. At the XSCT Prompt, type **stop**.

The U-Boot execution stops.

- j. Type **dow -data image.ub 0x30000000** to download the Linux Kernel image at location <tutorial\_download\_path>/image.ub.
- k. Type **con** to start executing U-Boot.
- 11. At the command prompt of the serial terminal, type **bootm 0x30000000**.

The Linux OS boots.



12. If required, provide the Zynq login as **root** and the password as **root** on the serial terminal to complete booting the processor.

After booting completes, # prompt appears on the serial terminal.

- 13. At the root@xilinx-zc702-2019\_2:~# prompt, make sure that the board Ethernet connection is configured:
  - a. Check the IP address of the board by typing the following command at the Zynq> prompt: **ifconfig eth0**.

This command displays all the details of the currently active interface. In the message that displays, the inet addr value denotes the IP address that is assigned to the Zynq SoC board.

b. If inet addr and netmask values do not exist, you can assign them using the following commands:

```
root@xilinx-zc702-2019_2:~# ifconfig eth0 inet 192.168.1.10
root@xilinx-zc702-2019_2:~# ifconfig eth0 netmask 255.255.255.0
```



**IMPORTANT:** If the target and host are connected back-to-back, you must set up the IP address. If the target and host are connected over a LAN, DHCP will get the IP address for the target; use the ifconfig eth0 to display the IP address.

Next, confirm that the IP address settings on the Windows machine match the board settings. Adjust the local area connection properties by opening your network connections.

- i Right click the local area connection that is linked to the XC702 board and select **Properties**.
- ii With the Local Area Connection properties window open, select **Internet Protocol Version 4 (TCP/IPv4)** from the item list and select **Properties**.
- iii Select **Use the following IP address** and set the values as follows (also shown in the following figure):

IP address: 192.168.1.11 (target and host must be in the same subnet if connected back- to-back)

Subnet mask : 255.255.255.0



eneral		
You can get IP settings assigne this capability. Otherwise, you r for the appropriate IP settings.	need to ask your network	
🔘 Obtain an IP address auto	matically	
Output Se the following IP address	ss:	
IP address:	192 . 168 . 1	. 11
Subnet mask:	255 . 255 . 25	5.0
Default gateway:	• •	•
Obtain DNS server addres	s automatically	
Ose the following DNS served	ver addresses:	
Preferred DNS server:		
Alternate DNS server:		
	it (	Advanced

Figure 6-5: IP Address Settings

- c. Click **OK** to accept the values and close the window.
- 14. In the Windows machine command prompt, check the connection with the board by typing **ping** followed by the board IP address. The ping response displays in a loop.

This response means that the connection between the Windows host machine and the target board is established.

15. Press **Ctrl+C** to stop displaying the ping response on windows host machine command prompt.

Linux booting completes on the target board and the connection between the host machine and the target board is complete. The next Example Design describes using the Vitis software platform to debug the Linux application.



# Example Design: Debugging the Linux Application Using the Vitis Software Platform

In this section, you will create a default Linux hello world application and practice the steps for debugging the Linux application from the Windows host machine.

1. Open the Vitis software platform.

#### 2. Select File > New > Application Project.

The New Applications Project wizard opens.

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

Wizard Screen System Property Setting or Command to Use **Application Project** HelloLinux Project Name Use Default Location Select this option System project HelloLinux\_system Platform <platform> CPU cortex-a9 OS linux С Language Sysroot path Leave it unchecked Templates **Available Templates** Linux Hello World

Table 6-1: New Project Wizard Selections for Debugging in the Vitis Software Platform

#### 4. Click Finish.

The New Project wizard closes and the Vitis software platform creates the HelloLinux project under the project explorer. Build Application project either by clicking the **hammer icon** or by right-clicking on the **linux\_cdma\_app project** and selecting Build Project. Binary file linux\_cdma\_app.elf gets generated.

- 5. Right-click **HelloLinux** and select **Debug as > Debug Configurations** to open the Debug Configurations wizard.
- 6. Select Linux Application Debug as the Debug Type, as shown in the following figure.



	- 0
	Ŕ
Name: HelloLinux Debug	
Debug Type: Linux Application Debug   Connection: Linux Agent   Note: TCF agent port should be used as port in the target connection (Default TCF)	
Configuration: Debug  Enulation  Performance Analysis	
	K Main      Application      Target Setup)      Arguments      Environment      Sym     Debug Type: Linux Application Debug     Connection: Linux Agent     New     Note: TCF agent port should be used as port in the target connection (Default TCF     Project: HelloLinux     Configuration: Debug     Emulation

*Figure 6-6:* **Debug Type Selection** 

- 7. In the **Target Setup** tab, **Connection** field, click **New**.
- 8. In the Target Connection Details dialog box (shown in the following figure):
  - a. Specify the Target Name of your choice.
  - b. In the **Host** field, use the target IP address.
  - c. In the **Port** field, specify 1534.

C 26 /2 18 18 18 1 8 1 8 1 8 1 1 1 1 1 1 1 1 1	Name: HelloLinux Debug
type filter text	Image: State in the state
	New Target Connection         Creates new configuration for connecting to a target.         Target Name       Linux_Demo         Set as default target         Specify the connection type and properties         Type       Linux_TCF Agent
	Host 192.168.1.10 Port 1534 Advanced >>   Test Connection OK
Filter matched 4 of 4 items	Revert

Figure 6-7: Debug Configuration Target Connection Settings

9. Set the Application configuration details, as described below (and shown in the following figure).



- a. Select the **Application** tab.
- b. Set the **Remote File** path, for example **/tmp/hellolinux.elf** and click **Apply**.

Create, manage, and run configuration	ns					
Debug a program using Application Debugg	er.					
0 2 9 18 × 0 > -	Name: HelloLinux Debug					
type filter text	🗱 Main 🗖 Application 🔍 🐵 Target Setup 🎂 Arguments 🖏 Environment 🕼 Symbol Files 🖖 Source 💩 Path Map 🗖 Common					
<ul> <li>Š. Xilinx Application Debugger</li> <li>S. HelloLinux Debug</li> <li>Xilinx Application Debugger (GDB)</li> </ul>	Application Local File Path:					
Xilinx SPM Analysis	Debug/HelloLinux.elf Search					
	Remote File Path:					
	/tmp/hellolinux.elf					
	Working directory					
	Use default					
	Auto-attach process children					
	Stop at program entry					
	Stop at 'main'					
	Disconnect when last debug context exits     V Use pseudo-terminal for process standard I/O					
	Hide debug contexts started by other debug sessions					
	е [					
Filter matched 4 of 4 items	Revert Apply					
(?)						

Figure 6-8: Debug Configuration Target Options

#### 10. Click Debug.

The Debug Perspective opens (see Figure 6-9). From this screen you can:

- Observe that execution stopped at the main() function.
- See disassembly points to the address.
- Setup break points by right clicking the function on the left side of the editor pane (showing the helloworld.c).
- Once a breakpoint is set, it appears in the break point list. You can observe and modify register contents. Notice that the PC register address in the **Registers** tab and the address shown in the **Disassembly** tab are the same (see the following figure).
- Use **step-into** (F5), **step-return** (F7), **step-over** (F6), **Resume** (F8) and **continue debugging** outlined in green in the following figure.



 $\bigcirc$ 

ile Edit Run Developer Xilinx Project Window Help							
	0 -					Quick Access	ign 🔯 Del
P i helloworld.c ∞	Breakpoints	: 51	1		11 场		- Annotation
<pre>19/* 2 * Copyright (c) 2012 Xilinx, Inc. All rights reserved 3 * 4 * Xilinx, Inc. 5 * Xilinx is PROVODING THIS DESIGN, CODE, OR INFORMATIO</pre>		world.c [line: 23]					
6 * COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR	4						
7 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLI							
8 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT TH 9 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE	IIII Registers 2						
10 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IM	Name	Hex	Decimal	Description	Mnemonic		1
11 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER W	1111 r6	00000000	0				
12 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NO		0000000	0				
13 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMEN	100 r8	00000000	0				
14 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF M 15 * AND FITNESS FOR A PARTICULAR PURPOSE.	10 r9	b6f9ffa4	3069837220				
16 *	10 IIII fp	bef2acc4	3203574980				
17 */	W ip	bef2ad40	3203575104				
18	- Will sp	bef2acc0	3283574976				
<pre>% #include <stdio.h> 20</stdio.h></pre>	W Ir	b6e9c4a5	3068773541				
21@int main()	WI pc	0001040c	66572				
22 {	IIII COST	40080010	1074266128				
P 23 printf("Hello World\n");	0	ffffffff	4294967295				
25 return 0;	▶ ₩₽ vfp						
・ III ・ F 参 Debug 22 合 Project Explorer S Explorer 演 は マーロ	Disassembl	y 22]		En	ter location here	- 10 10 SE 13 13	, , ,
& HelloLinux Debug (Linux_Demo)     & @/tmg/hellolinux.elf     & @ \$1381.1381 (Breakpoint: helloworld.c.23; Signal: Trace/breakpoint trace)	00010404: 00010408:	push {r1 add r11	1,lr) ., sp, #4 ⊨llo World\n")			,	
0x0001040c [hellolinux.elf] main():/src/helloworld.c, line 23	24	ALCONSOL ALCO					1.
0xb6e9c4a4 (libc-2.28.so)libc_start_main()	0001040c: 00010410;		#1172 #1				- 11
0xb6e9c4a4 (libc-2.28.so)libc_start_main()	00010410.						
	25	bl -31 return 0;		000102e4; pu	tsepit		
	25 00010418: 26	return 0; mov r3, }	#0	000102e4; pu	tsepit		
	25 00010418; 26 0001041c: 00010420:	return 0; mov r3, ) mov r0, pop (r1 _libc_csu_in	#0 r3 1,pc) iit:		tsepit		ш
	25 00010418: 26 0001041c: 00010420: 00010424: 00010428:	return 0; mov r3, } mov r0, pop {r1 _libc_csu_in push {r4 mov r7,	#0 r3 1,pc) iit: ,r5,r6,r7,r8,r r0		tsepit		III
	25 00010418; 26 0001041c: 00010420; 00010424;	return 0; mov r3, } mov r0, pop {r1 _libc_csu_in push {r4 mov r7, ldr r6,	#0 r3 1,pc) iit: 1,r5,r6,r7,r8,r		tsepit		- M
	25 00010418; 26 0001041c: 00010420; 00010424; 00010424; 00010424; 00010426; 00010430; 00010430;	return 0; mov r3, ) mov r0, pop (r1 libc_csu_in push (r4 mov r7, ldr r6, mov r8, mov r9,	#0 r3 (1,pc) it: ,r5,r6,r7,r8,r r0 [pc, #+72] r1 r2		tsepit		- Mi
	25 00010418: 26 0001041c: 00010420: 00010424: 00010428: 00010428: 00010426: 00010438: 00010438:	return 0; mov r3, ) mov r0, pop (r1 _libc_csu_in push (r4 mov r7, ldr r6, mov r8, mov r9, ldr r5,	#0 r3 1,pc) it: r0,r5,r6,r7,r8,r r0 [pc, #+72] r1 r2 [pc, #+64]		itsepit		10
	25 00010418: 26 0001041c: 00010420: 00010424: 00010424: 00010426: 00010438: 00010434: 00010433: 00010433:	return 0; mov r3, ) mov r0, pop {r1- 	#0 r3 1.pc) it: .r5,r6,r7,r8,r r0 [pc, #+72] r1 r2 [pc, #+64] pc, r6	9,r10,lr}			- III
	25 00010418: 26 0001041c: 00010420: 00010424: 00010428: 00010428: 00010426: 00010438: 00010438:	return 0; mov r3, ) mov r0, pop {r1 libc_csu_in mov r7, ldr r6, mov r9, ldr r5, add r6, bl -38	#0 r3 1,pc) it: r0 [pc, #+72] r1 r2 [pc, #+64] pc, r6 is ; addr=0x				111
	25 00010418: 26 0001041c: 00010428: 00010428: 00010428: 00010428: 00010438: 00010438: 00010438: 00010438: 00010448: 00010448:	return 0; mov r3, } mov r6, pop {r1 _libc_csu_in mov r6, mov r6, mov r8, ldr r6, mov r8, ldr r6, bl -38 add r5, bl -38	#0 r3 1,pc) it: r0 [pc, #+72] r1 r1 [pc, #+64] pc, r5 r6, r5	9,r10,lr}			- HI
	25 00010418: 26 0001041c: 00010428: 00010428: 00010428: 00010438: 00010438: 00010438: 00010438: 00010448: 00010448: 00010446: 00010446:	return 0; mov r3, ) pop (r0, pop (r1, push {r4 mov r7, ldr r6, mov r8, mov r8, mov r8, dd r6, bl -38 add r5, sub r6,	#0 r3 1,pc) ii: ,r5,r6,r7,r8,r r0 [pc, #+72] r1 r2 [pc, r6 8; ; addr=8x pc, r5 r6, r5 r6, r5 r6, r5	9,r10,lr) 800102c4: _i			- III
	25 00010418: 26 00010426: 00010424: 00010424: 00010422: 00010432: 00010432: 00010432: 00010434: 00010434: 00010448: 00010448: 00010448:	return 0; mov r3, } mov r6, pop (r1 libc_csu_in mov r7, ldr r6, mov r8, mov r9, ldr r5, sub r6, asrs r6, popeq (r4	#0 r3 1,pc) it: r0 [pc, #+72] r1 r2 [pc, #+64] pc, r6 r6, r5 r6, r5 r6, #2, r, r8, r, r8, r	9,r10,lr) 800102c4: _i			ш
	25 00010418: 26 0001041c: 00010428: 00010428: 00010428: 00010438: 00010438: 00010438: 00010438: 00010448: 00010448: 00010446: 00010446:	return 0; mov r0; ) mov r0; pop {r1 <u>libc_esu in</u> push {r4 mov r7, ldr r6, mov r8, mov r8, mov r9, ldr r5, add r5, sub r6, sub r6, popeq {r4	#0 r3 1,pc) ii: ,r5,r6,r7,r8,r r0 [pc, #+72] r1 r2 [pc, r6 8; ; addr=8x pc, r5 r6, r5 r6, r5 r6, r5	9,r10,lr) 800102c4: _i			H

*Figure 6-9:* **Debug Perspective Launched with HelloLinux Application** 

**TIP:** The Linux application output displays in the Vitis software platform console, not the Terminal window used for running Linux.

11. After you finish debugging the Linux application, close the Vitis IDE.





# **Example Project: Booting Linux from QSPI Flash**

This Example Project covers the following steps:

- 1. Create the First Stage Boot Loader Executable File.
- 2. Make a Linux-bootable image for QSPI flash.

PetaLinux must be configured for QSPI flash boot mode and rebuilt. By default, the Boot option is SD boot.



**TIP:** The ZIP file that accompanies this document contains the prebuilt images. If you prefer, you can use these and skip to either Booting Linux from QSPI Flash, page 100 or Booting Linux from the SD Card, page 101, as appropriate to your design.

- 3. Run the following steps on a Linux machine to change the boot mode to QSPI flash.
  - a. Change to the root directory of your PetaLinux project: \$ cd <plnx-proj-root>
  - b. Launch the top level system configuration menu: \$ petalinux-config
  - c. Select Subsystem AUTO Hardware Settings.
  - d. Select Advanced Bootable Images Storage Settings.
    - Select boot image settings.
    - Select Image Storage Media.
    - Select boot device as primary flash.
  - e. Under the Advanced Bootable Images Storage Settings sub-menu:
    - Select kernel image settings.
    - Select Image Storage Media.
    - Select the storage device as **primary flash**.
  - f. Save the configuration settings and exit the configuration wizard.
  - g. Rebuild using the Petalinux-build command.

**Note:** For more information, refer to the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 8].

- 4. Program QSPI flash with the Boot Image Using JTAG and U-Boot Command.
- 5. Boot Linux from QSPI flash.



#### Create the First Stage Boot Loader Executable File

- 1. Open the Vitis software platform.
- Check that the Target Communication Frame (TCF) (hw\_server.exe) agent is running on your Windows machine. If it is not, in the Vitis software platform, select Xilinx Tools > XSCT Console.
- 3. In the XSCT Console window, type **Connect**. A message appears, stating that the hw\_server application started, or, if it is already running, you will see tcfchan#, as shown in the following figure.

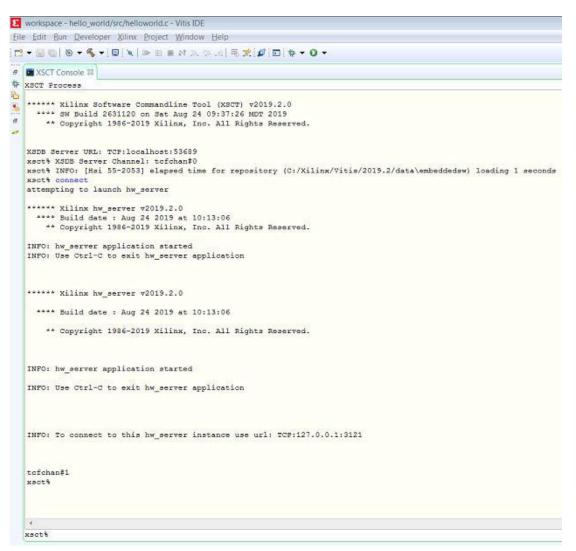


Figure 6-10: XSCT Console: hw\_server Application Started Message



- 4. In the Vitis software platform, select **File > New > Application Project**. The New Application Project wizard opens.
- 5. Use the information the following table to make your selections in the wizard screens.

 Table 6-2:
 New Project Wizard Selections for Booting Linux Project

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	fsbl
	Use Default Location	Select this option
	System Project	Select Create New
	Platform	hw_platform
	CPU	ps7_cortexa9_0
	OS Platform	standalone
	Language	С
	Domain	Standalone on ps7_cortexa9_0
Templates	Available Templates	Zynq FSBL

6. Click **Finish**. If a pop up message comes up that "This application required xilffs library in Board Support Package." Do the same and repeat the above steps to create FSBL standalone application.

The New Project wizard closes. The Vitis software platform creates the fsbl application project under the project explorer. For generating fsbl.elf build the project by right-clicking on the **fsbl project** and selecting Build Project.





### Make a Linux Bootable Image for QSPI Flash

1. In the Vitis software platform, select **Xilinx > Create Boot Image** to open the Create Boot Image wizard.

	ed output folder.	files in specifie	format from given FSBL elf and partition		Create Boot Ima Creates Zyng Boo
			from existing BIF file		rchitecture: Zyno
Browse.			\edt_qspi_images\output.bif		Basic Security Output BIF file par
Browse.					UDF data:
			mat: BIN 💌		Split
Browse.	Output path: C\designs\edt_qspi_images\800T.bin				
Add	Authentic	Encrypted	ace\fsbl\Debug\fsbl.elf		Boot image partiti File path (bootloader) C\d
Dele	none	none	orm\bitstream\tutorial_bd_wrapper.bit		
Edit	none	none		0.000	C:\designs\edt_q: C:\designs\edt_q:
Dow					3
	none none	none none	orm\bitstream\tutorial_bd_wrapper.bit woot.elf	orkspace dt_qspi_i	C:\designs\works C:\designs\edt_q:

Figure 6-11: Creating a Zynq Device Boot Image

- 2. From the Architecture drop-down list, select Zynq.
- 3. Click **Browse** next to the **Output BIF file path** field, and navigate to your output.bif file.
- 4. Click **Browse** next to the **Output path** field, and navigate to your BOOT.bin file.

**Note:** The QSPI Boot file, BOOT.bin, is available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.



- 5. Click **Add** to add the following boot image partitions:
  - fsbl.elf (bootloader).

**Note:** You can find fsbl.elf in <project dir>/fsbl/Debug. Alternately, you can use fsbl.elf from the file you downloaded in Requirements, page 80.

- Add Bitstream file tutorial\_bd\_wrapper.bit.
- Add U-Boot image u-boot.elf.
- Add the PetaLinux output image, image.ub, and provide the offset 0x520000 (image.ub: PetaLinux image consists of kernel image, device tree blob and minimal rootfs).
- 6. Click **Create Image** to create the BOOT.bin file in the specified output path folder.

#### Program QSPI Flash with the Boot Image Using JTAG

You can program QSPI Flash with the boot image using JTAG.

- 1. Power on the ZC702 Board.
- 2. If a serial terminal is not already open, connect the serial terminal with the baud rate set to **115200**.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.

- 3. Select Xilinx > XSCT Console to open the XSCT tool.
- 4. From the XSCT prompt, do the following:
  - a. Type **connect** to connect with the PS section.
  - b. Type **targets** to get the list of target processors.
  - c. Type ta 2 to select the processor CPU1.
  - d. Type **dow fsbl.elf** to download the FSBL image.
  - e. Type **con** and then **stop**.
  - f. Type **dow u-boot.elf** to download the Linux U-Boot.
  - g. Type **dow -data BOOT.bin 0x08000000** to download the Linux bootable image to the target memory at location 0x08000000.

*Note:* You just downloaded the binary executable to DDR memory. You can download the binary executable to any address in DDR memory.

h. Type **con** to start execution of U-Boot.

U-Boot begins booting. On the serial terminal, the autoboot countdown message appears:

Hit any key to stop autoboot: 3





#### 5. Press Enter.

Automatic booting from U-Boot stops and the U-Boot command prompt appears on the serial terminal.

- 6. Do the following steps to program U-Boot with the bootable image:
  - a. At the prompt, type **sf probe 0 0 0** to select the QSPI Flash.
  - b. Type **sf erase 0 0x01000000** to erase the Flash data.

This command completely erases 16 MB of on-board QSPI Flash memory.

c. Type **sf write 0x08000000 0 0xffffff** to write the boot image on the QSPI Flash.

Note that you already copied the bootable image at DDR location 0x08000000. This command copied the data, of the size equivalent to the bootable image size, from DDR to QSPI location 0x0.

For this example, because you have 16 MB of Flash memory, you copied 16 MB of data. You can change the argument to adjust the bootable image size.

7. Power off the board and follow the booting steps described in the following section.

#### Program QSPI Flash with the Flash Programming Tool

Following the steps below, you can program QSPI Flash with the flash programming tool in the Vitis software platform:

- 1. Power on the ZC702 Board.
- 2. If a serial terminal is not open, connect the serial terminal with the baud rate set to **115200**.

*Note:* This is the baud rate to which the UART is programmed on Zynq devices.

- 3. Select Xilinx > Program Flash.
- 4. Select the BOOT.bin file to flash and select **Program** (see the following figure).



2			8
Program F	lash Memory		
Program FI	ash Memory via In-system Programmer.		
Project Type:	System O Application		
Project:	fsbl		
Connection:	Local	New	
Device:	Auto Detect	Select	
Image File:	C:\designs\edt_qspi_images\BOOT.bin	Search	Browse
Offset:	0x0		
Flash Type	qspi-x4-single		
FSBL File:	C:\designs\workspace\fsbl\Debug\fsbl.elf	Browse	
	LF to bootloadable SREC format and program		
a second second second	ck after erase		
Verify afte	er flash		
	[	Program	Cancel

*Figure 6-12:* **Programming the BOOT.bin file Using the Flash Tool** 

On successful programming, a message appears in the console window saying **Flash Operation Successful**.

5. Power off the board and follow the booting steps in Booting Linux from QSPI Flash, page 100 or Booting Linux from the SD Card, page 101, as appropriate to your design.



#### Booting Linux from QSPI Flash

1. After you program the QSPI Flash, set the SW16 switch on your board as shown in the following figure.

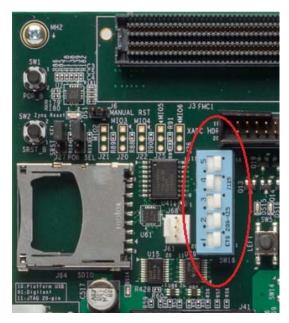


Figure 6-13: Jumper Settings for Booting Linux from QSPI Flash

2. Connect the Serial terminal using an **115200** baud rate setting.

*Note:* This is the baud rate that the UART is programmed to on Zynq devices.

3. Switch on the board power.

A Linux booting message appears on the serial terminal. After booting finishes, the root@xilinx-zc702-2019\_2:~# prompt appears. Enter the login and password as root when prompted.

4. Check the Board IP address connectivity as described in Booting Linux Using JTAG Mode, page 84.

For Linux Application creation and debugging, refer to Example Design: Debugging the Linux Application Using the Vitis Software Platform, page 89.



### Booting Linux from the SD Card

1. Change the SW16 switch setting as shown in the following figure.

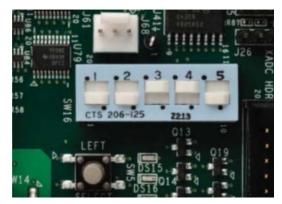


Figure 6-14: Jumper Settings for Booting Linux from SD Card

- 2. Make the board settings as described in Booting Linux Using JTAG Mode, page 84.
- 3. Create a first stage bootloader (FSBL) for your design as described in Create the First Stage Boot Loader Executable File, page 94.

**Note:** If you do not need to change the default FSBL image, you can use the fsbl.elf file that you downloaded as part of the ZIP file for this guide. See Design Files for This Tutorial, page 134.

- In the Vitis IDE, select Xilinx Tools > Create Boot Image to open the Create Boot Image wizard.
- 5. Add fsbl.elf, bit file (if any), and u-boot.elf.
- 6. Provide the output folder path in the Output folder field.
- 7. Click **Create Image**. The Vitis software platform generates the BOOT.bin file in the specified folder.



😢 Create Boot Image					×
Create Boot Image	e				
Creates Zynq Boot Ir	nage in .bin format from given FSBL elf and partition	files in specified	d output folde	r.	- ioi
Architecture: Zynq	•				
Oreate new BIF file	◎ Import from existing BIF file				
Basic Security					
Output BIF file path:	C:\designs\edt_sdboot_images\output.bif				Browse
UDF data:					Browse
Split	Output format: BIN 🔻				
Output path:	C:\designs\edt_sdboot_images\BOOT.bin				Browse
Boot image partitions	5			1	
File path		Encrypted	Authentic		Add
	gns\edt_sdboot_images\fsbl.elf	none	none		
	e\hw_platform\bitstream\tutorial_bd_wrapper.bit	none	none		Delete
C:\designs\edt_sdbo	ot_images\u-boot.elf	none	none		Edit
					Up
					Down
(?)		Preview BI	Changes	Create Image	Cancel

Figure 6-15: Creating the Zynq Device Boot Image

#### 8. Copy **BOOT.bin** and **image.ub** to the SD card.



**IMPORTANT:** Do not change the file names. U-Boot searches for these file names in the SD card while booting the system.

- 9. Turn on the power to the board and check the messages on the Serial terminal. The root@plnx\_arm:~# prompt appears after Linux booting is complete on the target board.
- 10. Set the board IP address and check the connectivity as described in Booting Linux Using JTAG Mode, page 84.

For Linux application creation and debugging, see Example Design: Debugging the Linux Application Using the Vitis Software Platform, page 89.



# Chapter 7

# Creating Custom IP and Device Driver for Linux

In this chapter, you will create an Intellectual Property (IP) using the Create and Package New IP wizard. You will also design a system to include the new IP created for the Xilinx® Zynq®-7000 SoC device.

For the IP, you will develop a Linux-based device driver as a module that can be dynamically loaded onto the running kernel.

You will also develop Linux-based application software for the system to execute on the Zynq SoC ZC702 board.

# Requirements

In this chapter, the target platform points to a ZC702 board. The host platform points a Windows machine that is running the Vivado® Design Suite tools.

The requirements for Linux-based device driver development and kernel compilation are as follows:

- Linux-based workstation. The workstation is used to build the kernel and the device driver for the IP.
- An Eclipse-based Integrated Development Environment (IDE) that incorporates the GNU Toolchain for cross development for target architectures. For Tool related information and installation, refer to the Xilinx Zynq Tools Wiki Page [Ref 13].
- Kernel source code and build environment. Refer to the Xilinx Zynq Linux Wiki Page [Ref 14], which provides details about the Linux kernel specific to Zynq SoC FPGAs. You can download the Kernel Source files and also get the information for building a Linux kernel for the Zynq SoC FPGA.

*Note:* You can download kernel source files and u-boot source files from the Xilinx GitHub website [Ref 18].

• Device driver software file (blink.c) and the corresponding header file (blink.h). These files are available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.



- Application software (linux\_blinkled\_apps.c) and corresponding header file (blink.h). These files are available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.
- If you want to skip the Kernel and device driver compilation, use the already compiled images that are required for this section. These images are available in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.



**CAUTION!** You must build Peripheral IP loadable kernel module (LKM) as part of the same kernel build process that generates the base kernel image. If you want to skip kernel or LKM Build process, use the precompiled images for both kernel and LKM module for this section provided in the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.

# **Creating Peripheral IP**

In this section, you will create an AXI4-Lite compliant slave peripheral IP framework using the Create and Package New IP wizard. You will also add functionality and port assignments to the peripheral IP framework.

The Peripheral IP you will create is an AXI4-Lite compliant Slave IP. It includes a 28-bit counter. The 4 MSB bits of the counter drive the 4 output ports of the peripheral IP. The Block Diagram is shown in the following figure.

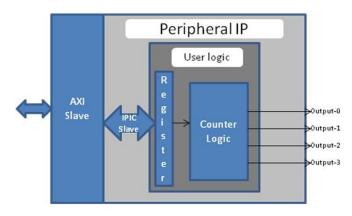


Figure 7-1: Block Diagram for Peripheral IP



The block diagram includes the following configuration register:

Register Name	Control Register
Relative Address	0x0000_0000
Width	1 bit
Access Type	Read/Write
Description	Start/Stop the Counter

Field Name	Bits	Туре	Reset Value	Description
Control Bit	0	R/W		1 : Start Counter 2 : Stop Counter

## **Example Project: Creating Peripheral IP**

In this section, you will create an AXI4-Lite compliant slave peripheral IP.

- 1. Create a new project as described in Example Project: Creating a New Embedded Project with Zynq SoC, page 13.
- With the Vivado design open, select Tools > Create and Package New IP. Click Next to continue.
- 3. Select Create a new AXI4 peripheral and then click Next.
- 4. Fill in the peripheral details as follows:

Wizard Screen	System Property	Setting or Comment to Use
Peripheral Details	Name	Blink
	Version	1.0
	Display Name	Blink_v1.0
	Description	My new AXI IP
	IP Location	C:/designs/ip_repro
	Overwrite existing	unchecked

#### 5. Click Next.

- 6. In the Add Interfaces page, accept the default settings and click **Next**.
- 7. In the Create Peripheral page, select Edit IP and then click Finish. Upon completion of the new IP generation process, the Package IP window opens (see the following figure).



ackaging Steps	Identification		
/ Identification	Vendor.	xilinx.com	0
Compatibility	Library:	user	٢
File Groups	Name:	Blink	0
Customization Parameters	Version:	1.0	0
	Display name:	Blink_v1.0	٢
Ports and Interfaces	Description:	My new AXI IP	0
Addressing and Memory	Vendor display name:		
Customization GUI	Company uri:		
Review and Package	Root directory: Xml file name:	c:/designs/ip_repo/Blink_1.0 c:/designs/ip_repo/Blink_1.0/component.xml	
	Categories		
	+   -   *	\$	
	AXLPeripheral		

Figure 7-2: Package IP Window

8. In the Hierarchy tab of the Sources window, right-click blink\_v1\_0 under the Design Sources folder and select Open File. We will need to add Verilog code that creates output ports to map to the external LEDs on the ZC702 board. Navigate to the line //Users to add ports here and add the following code below this line:

```
//Users to add ports here
output wire [3:0] leds,
//User ports ends
```

9. Find the instance instantiation to the AXI bus interface and add the following code to map the port connections:

```
.S_AXI_RREADY(s00_axi_rready),
.leds(leds)
);
```

- 10. Save and close blink\_v1\_0.v.
- 11. Under Sources > Hierarchy > Design Sources > blink\_v1\_0, right-click blink\_v1\_0\_S00\_AXI\_inst - blink\_v1\_0\_S00\_AXI and select Open File.

Next, you will need to add Verilog code that creates output ports to map to the external LEDs on the ZC702 board and also create the logic code to blink the LEDs when Register 0 is written to.

12. Navigate to the line //Users to add ports here and add the following code below this line.

```
//Users to add ports here
output wire [3:0] leds,
//User ports ends
```



13. Find the AXI4Lite signals section and add a custom register, which you will use as a counter. The added code is highlighted in red:

```
// AXI4LITE signals
reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_awaddr;
reg axi_awready;
reg axi_wready;
reg [1 : 0] axi_bresp;
reg axi_bvalid;
reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_araddr;
reg axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0] axi_rdata;
reg [1 : 0] axi_rresp;
reg axi_rvalid;
// add 28-bit register to use as counter
reg [27:0] count;
```

14. Find the I/O connections assignments section. This is where you assign the last four bits of the counter to the LEDs. The added code is highlighted in red:

```
// I/O Connections assignments
assign S_AXI_AWREADY= axi_awready;
assign S_AXI_WREADY= axi_wready;
assign S_AXI_BRESP= axi_bresp;
assign S_AXI_BVALID= axi_bvalid;
assign S_AXI_ARREADY= axi_arready;
assign S_AXI_RDATA= axi_rdata;
assign S_AXI_RRESP= axi_rresp;
assign S_AXI_RVALID= axi_rvalid;
// assign MSB of count to LEDs
assign leds = count[27:24];
```

15. Toward the bottom of the file, find the section that states add user logic here. Add the following code, which will increment count while the slv\_reg0 is set to 0x1. If the register is not set, the counter does not increment. The added code is highlighted in red:

```
// Add user logic here
 // on positive edge of input clock
 always @( posedge S_AXI_ACLK )
   begin
     //if reset is set, set count = 0x0
     if ( S_AXI_ARESETN == 1'b0 )
      begin
        count <= 28'b0;
      end
     else
      begin
        //when slv_reg_0 is set to 0x1, increment count
        if (slv_reg0 == 2'h01)
          begin
            count <= count+1;</pre>
          end
        else
          begin
           count <= count;</pre>
          end
      end
end
// User logic ends
```



- 16. Save and close blink\_v1\_0\_S00\_AXI.v.
- 17. Open the Package IP blink tab. Under Packaging Steps, select Ports and Interfaces.

18. Click the Merge Changes from Ports and Interfaces Wizard link.

Packaging Steps	Ports and Interfaces					
<ul> <li>Identification</li> </ul>	Merge changes from Ports a	and Interface	s Wizard			
<ul> <li>Compatibility</li> </ul>	Q ≚ ≑ + ⊕	C				Ę
File Groups	Name	Interface Mode	Enablement Dependency	Is Declaration	Access Handle	Access Type
Ø Customization Parameters	> - S00_AXI	slave				
	> 😑 Clock and Reset Signals					
Ports and Interfaces						
<ul> <li>Addressing and Memory</li> </ul>						
Customization GUI						
Review and Package						



19. Make sure that the window is updated and includes the LEDs output ports.

Packaging Steps	Ports and Interfaces					
<ul> <li>Identification</li> </ul>	Q ≩ ≑ + ⊕	C				
<ul> <li>Compatibility</li> </ul>	Name	Interface Mode	Enablement Dependency	Is Declaration	Access Handle	Access Type
File Groups	> - SOO_AXI	slave				
	> 😑 Clock and Reset Signals					
<ul> <li>Customization Parameters</li> </ul>	- leds					ref
<ul> <li>Ports and Interfaces</li> </ul>						
<ul> <li>Addressing and Memory</li> </ul>						
<ul> <li>Customization GUI</li> </ul>						
Review and Package						

Figure 7-4: Package IP Tab: Ports and Interfaces Page

20. Under Packaging Steps, select **Review and Package**. At the bottom of the Review and Package window, click **Re-Package IP**.

The dialog box that opens states that packaging is complete and asks if you would like to close the project.

21. Click Yes.

**Note:** The custom core creation process that we have worked through is very simple with the example Verilog included in the IP creation process. For more information, refer to the *GitHub* Zynq Cookbook: How to Run BFM Simulation web page [Ref 19].



## **Integrating Peripheral IP with PS GP Master Port**

Now, you will create a system for the ZC702 board by instantiating the peripheral IP as a slave in the Zynq SoC processing logic (PL) section. You will then connect it with the PS processor through the processing system (PS) general purpose (GP) master port. The block diagram for the system is shown in the following figure.

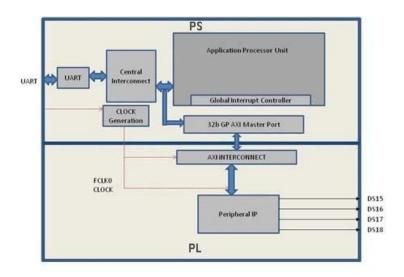


Figure 7-5: Block Diagram

This system covers the following connections:

- Peripheral IP connected to PS General Purpose master port 0 (M\_AXI\_GP0). This connection is used by the PS CPU to configure Peripheral IP register configurations.
- Four output ports of Peripheral IP connected to DS15, DS16, DS17, and DS18 on-board LEDs.

In this system, when you run application code, a message appears on the serial terminal and asks you to choose the option to make the LEDs start or stop blinking.

- When you select the start option on the serial terminal, all four LEDs start blinking.
- When you select the stop option, all four LEDs stop blinking and retain the previous state.

In this section, you will connect an AXI4-lite compliant custom slave peripheral IP that you created in Example Project: Creating Peripheral IP, page 105.

- 1. Open the Vivado project you previously created in Example Project: Creating a New Embedded Project with Zynq SoC, page 13.
- 2. Add the custom IP to the existing design. Right-click the Diagram view and select **Add IP**.



- 3. Type **blink** into the search view. Blink\_v1.0 appears. Double-click the IP to add it to the design.
- 4. Click Run Connection Automation to make automatic port connections.
- 5. With the **All Automation** box checked by default, click **OK** to make the connections.

Your new IP is automatically connected but the leds output port is unconnected.

6. Right-click the **leds** port and select **Make External**.

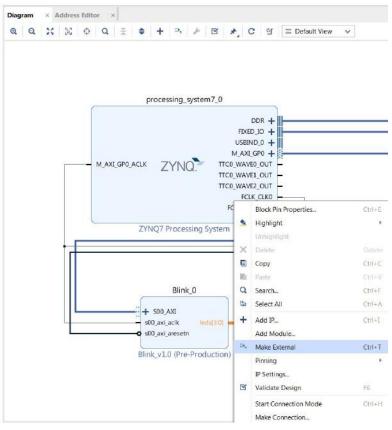


Figure 7-6: Make the leds Port External

- 7. In the Flow Navigator view, navigate to **RTL Analysis** and select **Open Elaborated Design**.
- 8. Click **OK**.
- 9. After the elaborated design opens, click the I/O Ports tab and expand All ports > led\_0.



Tcl Console Messages I	Log Reports	Design Runs	Packag	le Pins	I/O Ports	×
Q ∄ ♦ K +	ы					
Name	Direction	Board	Part Pin	Board	Part Interface	Neg Diff Pair
<ul> <li>All ports (134)</li> </ul>						
> 1% DDR_18048 (71)	INOUT					
> @ FIXED_IO_18048 (59)	INOUT					
✓ 1 leds_0 (4)	OUT					
- leds_0(3)	OUT					
Ieds_0(2)	OUT					
leds_0[1]	OUT					
@ leds_0[0]	OUT					
Scalar ports (0)						

*Figure 7-7:* **I/O Ports** 

Port NameI/O StdPackage PinLeds[3]LVCMOS25P17Leds[2]LVCMOS25P18Leds[1]LVCMOS25W10Leds[0]LVCMOS25V7

The following figure shows the completed led port settings in the I/O Ports window.

Tcl Console Messages L	og Report	5 Design Runs	Packa	ge Pins	I/O Ports	×
Q ≚ ≑ + +	ы					
Name	Direction	I/O Std		Package	Pin	
✓ ➡ All ports (134)						
> 🕞 DDR_18048 (71)	INOUT	(Multiple)*				
> 🕞 FIXED_IO_18048 (59)	INOUT	(Multiple)*				
<ul> <li>V de leds_0 (4)</li> </ul>	OUT	LVCMOS25*				
Ieds_0[3]	OUT	LVCMOS25*	*	P17	×	/
Ieds_0[2]	OUT	LVCMOS25*	*	P18	×	/
Ieds_0[1]	OUT	LVCMOS25*	*	W10	×	/
Ieds_0[0]	OUT	LVCMOS25*	*	V7	×	/

Figure 7-8: LED Port Settings

11. Select Generate Bitstream.

10. Edit the led port settings as follows:

- 12. The Save Project dialog box opens. Ensure that the check box is selected and then click **Save**.
- 13. If a message appears stating that Synthesis is Out-of-date, click Yes.
- 14. After the Bitstream generation completes, export the hardware and launch the Vitis<sup>™</sup> unified software platform as described in Exporting Hardware to the Vitis Software Platform, page 22.



## **Linux-Based Device Driver Development**

Modules in Linux are pieces of code that can be loaded and unloaded into the kernel on demand. A piece of code that you add in this way is called a loadable kernel module (LKM). These modules extend the functionality of the kernel without the need to reboot the system. Without modules, you would need to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring you to rebuild and reboot the kernel every time you want new functionality.

LKMs typically are one of the following things:

- Device drivers. A device driver is designed for a specific piece of hardware. The kernel uses it to communicate with that piece of hardware without having to know any details of how the hardware works.
- Filesystem drivers. A filesystem driver interprets the contents of a file system as files and directories.
- System calls. User space programs use system calls to get services from the kernel.

On Linux, each piece of hardware is represented by a file named as a device file, which provides the means to communicate with the hardware. Most hardware devices are used for output as well as input, so device files provide input/output control (ioctl) to send and receive data to and from hardware. Each device can have its own ioctl commands, which can be of the following types:

- read ioct1. These send information from a process to the kernel.
- write ioct1. These return information to a process.
- Both read and write ioctl.
- Neither read nor write ioctl.

For more details about LKM, refer to The Linux Kernel Module Programming Guide [Ref 20].

In this section you are going to develop a Peripheral IP Device driver as a LKM, which is dynamically loadable onto the running Kernel. You must build Peripheral IP LKM as part of the same kernel build process that generates the base kernel image.

**Note:** If you do not want to compile the device driver, you can skip the example of this section and jump to Loading Module into Running Kernel and Application Execution, page 114. In that section, you can use the kernel image, which contains blink.ko (image.ub in the shared ZIP files). See Design Files for This Tutorial, page 134.



For kernel compilation and device driver development, you must use the Linux workstation. Before you start developing the device driver, the following steps are required:

- 1. Set the toolchain path in your Linux Workstation.
- 2. Download kernel source code and compile it. For downloading and compilation, refer to the steps mentioned in Xilinx Zynq Linux Wiki Page [Ref 14].

## **Example Project: Device Driver Development**

You will use a Linux workstation for this example project. The device driver software is provided in the LKM folder of the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134.

1. Under the PetaLinux project directory, use the command below to create your module:

petalinux-create -t modules --name mymodule --enable

PetaLinux creates the module under

<plnx-project>/project-spec/meta-user/recipes-modules/.

For this exercise, create the "blink" module:

petalinux-create -t modules --name blink --enable

The default driver creation includes a Make file, C-file, and Readme files. In our exercise, PetaLinux creates blink.c, Makefile, and README files. It also contains bit bake recipe blink.bb.

- 2. Change the C-file (driver file) and the make file as per your driver.
- 3. Take the LKM folder (reference files) and copy blink.c and blink.h into this directory.
- 4. Open blink.bb recipe and add blink.h entry in SRC\_URI.
- 5. Run the command:

petalinux-build

After successful compilation the .ko file is created in the following location:

<petalinux-build\_directory>/build/tmp/sysroots-components/zc702\_zynq7/blink/lib/mod
ules/4.19.0-xilinx-v2019.2/extra/blink.ko

6. You can install the driver using the modprobe command, which will be explained in further detail in the next section.



# Loading Module into Running Kernel and Application Execution

In this section you will boot Linux onto the Zynq SoC Board and load the peripheral IP as a LKM onto it. You will develop the application for the system and execute it onto the hardware

## Loading Module into Kernel Memory

The basic programs for inserting LKMs are modprobe. The modprobe command makes an init\_module system call to load the LKM into kernel memory. The init\_module system call invokes the LKM initialization routine immediately after it loads the LKM. As part of its initialization routine, insmod passes to the address of the subroutine to init\_module.

In the peripheral IP device driver, you already set up init\_module to call a kernel function that registers the subroutines. It calls the kernel's register\_chrdev subroutine, passing the major and minor number of the devices it intends to drive and the address of its own "open" routine among the arguments. The subroutine register\_chrdev specifies in base kernel tables that when the kernel wants to open that particular device, it should call the open routine in your LKM.

## **Application Software**

The main() function in the application software is the entry point for the execution. It opens the device file for the peripheral IP and then waits for the user selection on the serial terminal.

If you select the start option on the serial terminal, all four LEDs start blinking. If you select the stop option, all four LEDs stop blinking and retain the previous state.

# Example Project: Loading a Module into Kernel and Executing the Application

#### Booting Linux on the Target Board

Boot Linux on the Zynq SoC ZC702 target board, as described in Booting Linux on a Zynq SoC Board, page 81.



#### Loading Modules and Executing Applications

In this section, you will use the Vitis software platform installed on a Windows machine.

1. Open the Vitis software platform.

You must run the Target Communication Frame (TCF) agent on the host machine.

- Select XSCT and then **connect** to connect to the Xilinx Software Command-Line Tool (XSCT).
- 3. In the Vitis software platform, select **File > New > Application Project** to open the New Application Project wizard.
- 4. Use the information in the table below to make your selections in the wizard screens.

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	linux_blinkled_app
	Use Default Location	Select this option
	System project	<linux_blinked_app_system></linux_blinked_app_system>
	Domain	linux on ps7_cortexa9
	CPU	cortexa-a9
	OS	linux
	Language	С
	Sysroot path	Leave it unchecked
Templates	Available Templates	Linux Empty Application

#### 5. Click Finish.

The New Project wizard closes and the Vitis software platform creates the linux\_blinkled\_app project under the project explorer.

6. In the Project Explorer tab, expand the **linux\_blinkled\_app** project, right-click the **src** directory, and select **Import**.

The Import Sources dialog box opens.

7. Browse for LKM\_App folder and select linux\_blinkled\_app.c and blink.h files.

**Note:** The Application software file name for the system is linux\_blinkled\_app.c and the header file name is blink.h. These files are available in the LKM folder of the ZIP file that accompanies this guide. See Design Files for This Tutorial, page 134. Add the linux\_blinkled\_app.c and blink.h files.

8. Click Finish.



Right click on linux\_blinkled\_app project and select Build Project to generate linux\_blinkled\_app.elf file in binary folders. Check the console window for the status of this action.

- 9. Connect the board.
- 10. Because you have a bitstream for the PL Fabric, you must download the bitstream. Select **Xilinx Tools > Program FPGA**.

The Program FPGA dialog box opens. It displays the bitstream exported from Vivado.

- 11. Click **Program** to download the bitstream and program the PL Fabric.
- 12. Follow the steps described in Chapter 6 to load the Linux image and start it.

After the Kernel boots successfully, in a serial terminal, navigate to /lib/modules/<kernel-version>/extra and run the command:

modprobe blink.ko

You will see the following message:

```
<1>Hello module world.
<1>Module parameters were (0xdeadbeef) and "default"
blink_init: Registers mapped to mmio = 0xf09f4000
Registration is a success the major device number is 244.
```

If you want to talk to the device driver, create a device file by running the following command:

mknod /dev/blink\_Dev c 244 0

The device file name is important, because the ioctl program assumes that is the file you will use

#### 13. Create a device node:

Run the **mknod** command and select the the string from the printed message.

For example, the command **mknod /dev/blink\_Dev c 244 0** creates the /dev/blink\_Dev node.

#### 14. Select **Window > Open Perspective > Remote System Explorer** and click **Open**.

The Vitis software platform opens the Remote Systems Explorer.

15. In the Remote Systems Explorer, do the following:

- a. Right-click and select **New > Connection** to open the New Connection wizard.
- b. Click the SSH only tab and click Next.
- c. In the Host Name tab, type the target board IP.



*Note:* To determine the target IP, type **ifconfig eth0** at the Zynq> prompt in the serial terminal. The target IP assigned to the board displays.

- d. Set the connection name as **blink** and type a description.
- e. Click **Finish** to create the connection.
- f. Expand **blink > sftp Files > Root**. The Enter Password wizard opens.
- g. Provide the user ID and Password (**root/root**); select the **Save ID** and **Save Password** options.
- h. Click **OK**.

The window displays the root directory content, because you previously established the connection between the Windows host machine and the target board.

- i. Right-click the "/" in the path name and create a new directory; name it Apps.
- j. Using the Remote Systems Perspective explorer, copy the linux\_blinkled\_app.elf file from the <project-dir> linux\_blinkled\_app/Debug folder and paste it into the /Apps directory under blink connection.
- 16. In the Serial terminal, type **cd Apps** at the Zyng> prompt to open the /Apps directory.
- 17. Go to the **Apps** directory at the root@xilinx-zc702-2019\_2: Linux prompt, and type **chmod 777 linux\_blinkled\_app.elf** to change the linux\_blinkled\_app.elf file mode to executable mode.
- 18. At the root@xilinx-zc702-2019\_2: prompt, type **./Linux\_blinkled\_app.elf** to execute the application.
- 19. Follow the instruction printed on the serial terminal to run the application. The Application asks you to enter 1 or 0 as input.
  - Type 1, and observe the LEDs DS15, DS16, DS17, and DS18. They start glowing.
  - Type 0, and observe that LEDS stop at their state. No more blinking changes.

You can repeat your inputs and observe the LEDs

20. After you finish debugging the Linux application, close the Vitis software platform.





# **E** XILINX.

# Software Profiling Using the Vitis Software Platform

In this chapter, you will enable profiling features for the standalone domain or board support package (BSP) and the application related to AXI CDMA, which you created in Chapter 6.

## Profiling an Application in the Vitis Software Platform with System Debugger

Profiling is a method by which the software execution time of each routine is determined. You can use this information to determine critical pieces of code and optimal code placement in a design. Routines that are frequently called are best suited for placement in fast memories, such as cache memory. You can also use profiling information to determine whether a piece of code can be placed in hardware, thereby improving overall performance.

You can use the system debugger in the Vitis<sup>™</sup> unified software platform to profile your application.

- 1. Select the application you want to profile.
- 2. Right click on the application and **Select > Debug As > Launch on Hardware** (Application Debugger).

If the Confirm Perspective Switch popup window appears, click Yes.

The Debug Perspective opens.

3. When the application stops at main, open the Target Communication Frame (TCF) profiler view by selecting **Window > Show View > Debug > TCF Profiler**.



塗 TCF Prot	filer 🛛			P 🖻 🗖
Idle. Press '	Start' button to :	start profiling		
Address	% Exclusive	% Inclusive	Function	File
Called Fror	n			

Figure 8-1: TCF Profiler View

4. Click the **Start** button to begin profiling. Alternately, you can select the **Aggregate Per Function** option in the Profiler Configuration dialog box. Adjust the **View Update Interval** according to your required profile sample time. The minimum time is 100 msec.

Aggregate per function	
Enable stack tracing	
Max stack frames count:	5
View update interval (msec):	4000

Figure 8-2: Profiler Configuration Dialog Box

5. Click the **Resume** button **ID** to continue running the application.

To view the profile data in the TCF Profiler tab (shown in the following figure), you must add an exit breakpoint for the application to stop.



workspace - Neglections/pid_contend/_0/itendatore_stomes/bip/pid_contenties_fide_stomes/bip/pide_stomes/bip/pid_contenties_f	national distance where	datone_vT_1/wc/_emile	Vita IDE					hat 2
0-008-5-0X - 0X								Quick Access 2 Design (9 Debu
		coma,app.c M Dis	assembly					- Variables  % Breakpoints = 11 Expressions = Modules = Registers () Memory # Discountby
<ul> <li>4. [a AD Cash, Davy di cicali</li> <li>21. AU</li> <li>4. [b AM Cashe, 24 MPCre #1 (Inseption: _ent)</li> <li>11. 6. 00010457(e ent)</li> <li>12. 6. 00010457(e ent)</li> <li>13. 6. 00010457(e ent)</li> <li>14. 6. 00010457(e ent)</li> <li>15. 6. 00010457(e ent)</li> <li>15.</li></ul>	3D         • Carst:           31         • The           32         • The           33         • all           34         •           35         • THE           36         • THE           37         • FITM           38         • THE           30         • LAB           30         • CAT           31         • THE           32         • LAB           33         •           34         •           35         • THE           36         • THE           37         • THE           38         • CAT           39         • LAB           30         • CAT           36         • THE           37         • THE           38         • THE           39         • CAT           30         • CAT           31         • THE           32         • THE           33         • THE           36         • THE           37         • STHE           38         • THE           39         • THE      <	table to de to, to down copyright so copies or subtact to subtact to section to section to section to section where a constant or on to constant so to to constant of the constant section to section	bjert to the following tice and this pression tal pertions of the Sof co "As iss", without we not courte to the we not courte to the work with a scraph of courtains, an across of courtains, a work the software on	e motice shall be i bases. SANITY OF ANY KING, MADIES OF REFOUND MEDIENT. IN NO VER A ANY CLAIM, NOVAGE THEF ON CONTRACT, THE SAE OR CINEN (6	EXPRESS DI ADJUTY, O' SANLE S OR OTHEN ALSING FROM, ALSING TH			R %(B) ⊗ k (B B (B) A " ⊘ # (Index net) ⊘ # consupective 300 ⊗ x consupective 400
	71 */ 12*_ettr: 13 { 14 (vas	ibute_((weak)) vo	entation. Boes not reh id _exit (sint)2 statur					Ho more contried.
	1 TCF Profiles	#]				م.	0.00	
	Potler running		- 10 1023 -					Entering main()
	Address 0030c708 0020c000	% Exclusive	N Indusive 200	Function partite_f write	File writes	tine 50		Start Transfer
	00103ea4	100	300	XUartPs_SendByte	suartps, huse	20		Start Transfer
Assisted III III III III III III III III III I	Called from							Tronsfer Don Malcide_Lifesti Passed 090. Tronsfer is "accessful
	Child Calls							

Figure 8-3: TCF Profiler Tab

## **Additional Design Support Options**

To assist in your design goals, you might want to learn about the System Performance Analysis (SPA) toolbox.

## The System Performance Analysis (SPA) Toolbox

To address the need for performance analysis and benchmarking, the Vitis software platform has a System Performance Analysis (SPA) toolbox to provide early exploration of hardware and software systems. You can use this common toolbox for performance validation to ensure consistent and expected performance throughout the design process.

For more information on exploring and exercising the SPA toolbox using the Vitis software platform, refer to the following documentation:

- Vitis Embedded Software Development Flow Documentation (UG1400) [Ref 10]
- System Performance Analysis of an SoC (XAPP1219) [Ref 9]





Chapter 9

## Linux OS Aware Debugging Using the Vitis Software Platform

OS-aware debugging helps you to visualize OS-specific information such as task lists, processes/threads that are currently running, process/thread-specific stack trace, registers, and variables view.

To support this, the debugger needs to be aware of the operating system used on the target and know about the intrinsic nature of the OS.

With OS-aware debugging, you can debug the OS running on the processor cores and the processes/threads running on the OS simultaneously.

The Vitis<sup>™</sup> unified software platform supports the OS Aware Debug feature for Linux OS running on Zynq<sup>®</sup>-7000 SoC devices.

## Setting Up Linux OS Aware Debugging

This section describes setting up OS aware debug for a Zynq board running Linux OS.

## **Configure the Linux Kernel**

To be able to read the process list or to allow process or module debugging, the Linux awareness accesses the internal kernel structures using the kernel symbols. Therefore the kernel symbols must be available; otherwise Linux aware debugging is not possible. The vmlinux file must be compiled with debugging information enabled as shown in Figure 9-1.

*Note:* The vmlinux file is a statically linked executable file that contains the Linux kernel along with corresponding debug information.

In PetaLinux, enable the below configuration options before compiling the Linux Kernel using the PetaLinux Tools build configuration command.

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





Follow the below steps to configure the Linux kernel to build with the debug information.

- 1. In the Linux machine terminal window, go to the directory of your PetaLinux project.
  - \$ cd <plnx-proj-root>
- 2. Launch the configuration menu to configure the Linux kernel.

\$ petalinux-config -c kernel

- 3. Select Kernel hacking.
  - Select Compile-time checks and compiler options.
  - Select Compile the kernel with debug info.

[*] Compile the kernel with debug info
[] Reduce debugging information (NEW)
[ ] Produce split debuginfo in .dwo files (NEW)
[ ] Generate dwarf4 debuginfo (NEW)
[ ] Provide GDB scripts for kernel debugging (NEW)
[*] Enable must check logic
(1024) Warn for stack frames larger than (needs gcc 4.4)
[ ] Strip assembler-generated symbols during link
[ ] Generate readable assembler code
[ ] Enable unused/obsolete exported symbols
[ ] Track page owner
[ ] Debug Filesystem
[ ] Run 'make headers check' when building vmlinux
[ ] Enable full Section mismatch analysis
[*] Make section mismatch errors non-fatal
[ ] Force weak per-cpu definitions

#### Figure 9-1: Enabling Debug Info Configuration Options in Linux Kernel

4. Save configuration.

This sets the Linux Kernel configuration file options to the following settings. You can verify that these options are enabled by looking in the configuration file:

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

5. Launch the configuration menu to configure the system-level options:

```
$ petalinux-config
```

#### Select Image Packaging Configuration.

Select INITRD for **Root filesystem type**.

Save configuration.

- 6. Build the PetaLinux using the PetaLinux build command petalinux-build.
- 7. After PetaLinux builds successfully, copy the vmlinux file to your host machine.



This file is needed for the debugger to refer all Linux kernel symbols. Vmlinux generates under <petalinux project file>/images/linux/vmlinux.

- 8. Copy Vmlinux to the host machine to use with the Vitis software platform for debugging the Linux Kernel.
- Copy the Linux kernel source code to the host machine for debugging. The Linux kernel is present in <petalinux-project>/build/tmp/work-shared/zc702-zynq7/ kernel-source.

*Note:* This document is composed and exercised using the Windows host machine, so it needs to copy the Linux source code to a location that is accessible for the Vitis tool running locally on Windows host machine.

### Creating the Hello World Linux Application to Exercise the OS Aware Debugging Feature

- 1. Open the Vitis software platform.
- 2. Select File > New > Application Project.

The New Project wizard opens.

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	linux_hello
	Use Default Location	Select this option
	System project	linux_hello_system
	Platform	<hw_platform></hw_platform>
	Domain	linux on ps7_cortexa9
	CPU	cortexa9
	OS	linux
	Language	С
	Sysroot path	Leave uncheaked
Templates	Available Templates	Linux Empty Application

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

#### 4. Click Finish.

- 5. In the Project Explorer tab, expand the linux\_hello project, 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.
- 9. Navigate to your design files folder and select the OSA folder and click **OK**.





*Note:* For more information about downloading the design files for this tutorial, see Design Files for This Tutorial, page 134.

10. Add the linux\_hello.c file and click **Finish**.

The Vitis software platform automatically builds the application and displays the status in the console window.

11. Copy linux\_hello.elf to an SD card.

## Debugging Linux Processes and Threads Using OS Aware Debug

- 1. Boot Linux as described in Booting Linux from the SD Card, page 101.
- 2. Create a Debug configuration.
- 3. Right-click **linux\_hello** and select **Debug as > Debug Configurations**.

The Debug Configuration wizard opens, as shown in the following figure.

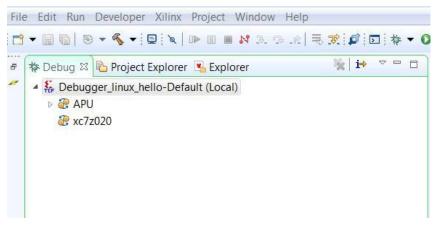
Name: Debugger, linux, hello-Default
Kain      Application      Target Setup      How Arguments     Symbol Files     Symbol
Revert Apply

Figure 9-2: Debug Configuration Wizard

- 4. In Main window, from the **Debug Type** drop-down list, select **Attach to running target**.
- 5. From the **Connection** drop-down list, select **Local**.
- 6. Click Debug.
- 7. If the Confirm Perspective Switch dialog box appears, click Yes.

Debugger\_linux\_hello-Debug opens in the Debug Perspective, as shown in the following figure.





*Figure 9-3:* **Debug Perspective** 

8. Set up the Linux kernel symbol file and enable the Linux OS awareness in the debug configuration.

There are multiple options provided by the Vitis software platform to enable Linux OS awareness feature enablement and debugging the applications. The following options are listed in the Symbol File dialog box.

Enable Linux OS Awareness

This option enables the OS Awareness

• Auto refresh On exec

When this option is selected, all running processes are refreshed and displayed in the Debug view.

When this option is disabled, the new processes are not displayed in the Debug view.

• Auto refresh on suspend

When this option is selected, all processes will be re-synced whenever the processor suspends.

When this option is disabled, only the current process will be re-synced.

- 9. In the Debug view, right-click **Debugger\_linux\_hello-Debug(Local)** and select **Edit Debugger\_linux\_hello-Default (Local)**.
- 10. Click the Symbol Files tab.
- 11. Select **/APU/Arm\_Cortex\_A9MPCore #0** from the Debug Context drop-down menu and click **Add**.

The Symbol File dialog box opens.

12. Click the **Browse** button \_\_\_\_.





13. Provide the path of the vmlinux file that you saved locally on the Windows host machine in the previous section, and check the box for **Enable OS awareness- the file** is an OS kernel, Auto refresh on exec and Auto refresh on suspend as shown in the following figure.

ame: Debugger_linux_hell							14
	lo-Default						
	Target Setup	ents 🚾 Envi	ronment	Symbol F	iles 4 Source &	Path Map 🔲 Common	
ebug context: /APU/ARM			Lawren				
File C:\designs\os_aware_de		Address	Size	Flags	File offset/section	Context query	Add
e.(ee.eg.e.(ee_arrais_ar		1.000	1		E.e.		Edit
	🔽 🖉 Symbo	ol File			2		Remove
	File nar	ne: C:\desi	ans\os_awa	re debug	sdboot\vmlinux 🛄		Locate File.
	Addre Size: File of Contex	fset		OK	lags Data read Data write Instructions read Cancel		
						Revert	Apply

Figure 9-4: Enable OS Awareness

- 14. You can also enable the **Auto refresh on exec** and **Auto refresh on suspend** options to get the refreshed process data while debugging the current application.
- 15. Click **OK**.

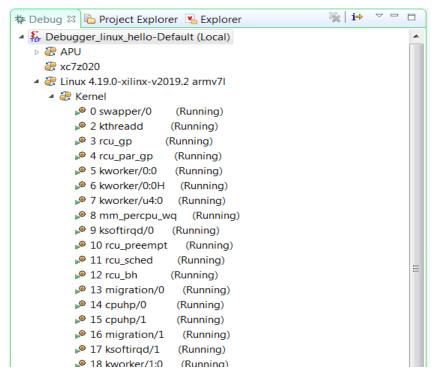
The Symbol File window closes.

16. Click **Continue** and then click **Save** for saving the configuration changes.

The Debug view opens, as shown in the following figure.







*Figure 9-5:* **Debug Perspective** 

You can see the Linux Kernel and list of processes running on the target.

**Note:** Because the Linux Kernel is built on a different system (Linux Machine) than the host machine (Windows Machine) on which we are exercising the Linux OS aware application debug, symbol files path mapping information should be added.

Path mapping will enable you to get source-level debugging and see stack trace, variables, setting up source level breakpoints, and so on.

The debugger uses the Path Map setting to search and load symbols files for all executable files and shared libraries in the system.



reate, manage, and run configurations Debug a program using Application Debugger				Ŕ				
2 A B X B > •	Name: Debugger_linux_hello-Default	t						
type filter text	K Main C Application I Target Se	etup 🕪 Arguments 👼 Environment 🔤 Symbol Files 🦭 Source	A Path Map					
4 Launch Group	File Path Map Rules.							
OpenCL     OpenCL (TCF)	Source Destination							
Target Communication Framework	<pre>&gt;</pre>	C\ <local-directory>\linux-xlnx</local-directory>	Edit.					
& Xilinx Al Engine SystemC Simulator								
Xilinx Application Debugger	& File Path Map F	5.42°	8	Remove				
& Debugger_linux_helio-Default & Xilinx Application Debugger (GDB)			243	Up				
Xilinx SPM Analysis	Add or edit so		Down					
K Xilinx System Debugger	Source and dest							
SystemDebugger_fsbl_system SystemC-RTL Co-Simulator	The rule is applied							
	Source: /·	<pre><petalinux-project>/xilinx-zc702-2019.2/build/tmp/work-shared</petalinux-project></pre>	and the second					
	Destination: C	<local-directory>\linux-xlnx</local-directory>						
	Context Query:							
				_				
		ОК	Cancel					
	×[	বা		Þ				
Iter matched 12 of 24 items			Revert	Apply				

Figure 9-6: Path Mapping Rule Configuration

- 17. Set up the Path Map.
  - a. Click the Path Map tab.
  - b. Click Add.
  - c. The source path for the kernel is the compilation directory path from the Linux
    machine as shown in the previous figure. For example,
     cpetalinux-project>/build/tmp/work-shared/zc702-zynq7/kernel-s
     ource

The destination path is the host location where you copied kernel in the earlier step. For example, <local directory>/linux-xlnx.

- d. Click Apply to apply the changes.
- e. Press Continue to Debug



Debug a program using Application Debugger								
1266 🕷 🖊 日 Þ 🕈	Name: Debugger_linux_hello-Default							
type filter text	💴 🐮 Main 🛅 Application 🧕 Target Setup 🕪 Arguments 👼 Environment 🛼 Symbol Files 💱 Source	& Path Map						
R Launch Group OpenCL	File Path Map Rules:							
- OpenCL - OpenCL (TCF)	Source	Destination	Add					
Target Communication Framework	/ <petalinux-project>/xilinx-zc702-2019.2/build/tmp/work-shared/zc702-zynq7/kernel-source</petalinux-project>	C:\ <local-directory>\linux-xlnx</local-directory>	Edit					
Xilinx AI Engine SystemC Simulator Xilinx Application Debugger			Remove					
🔓 Debugger_linux_hello-Default			Up					
Silinx Application Debugger (GDB) Xilinx SPM Analysis			Down					
ξ, Xilinx SystemC-RTL Co-Simulator	10							
ilter matched 12 of 24 items		Revert	Apply					

Figure 9-7: Path Mapping in Debug Configurations

18. Debug a Linux Process or thread.

As shown in Figure 9-5, the list of processes running on the target is displayed. You can right-click any process and click **Suspend**. Using this method, you can exercise debugging features such as watch stack trace, registers, adding break points, and so on.

In the following figure, the suspended process is named 1 init.





000000-5-00000000000000000000000000000	1.001-001-001-0	0 -			Quick Access IP Design 19-Debug 18	Remote System Exp
Debug # 0 Project Explorer	~	[] [] IM Requiters 25 ]				a) 173 ml
I & APU		Neme	Hex	Decimal	Description	Mnemonic
18 xc7z020			00000000	11	arouge ground	
<ul> <li>W Linux 4.19.0-xilinx-v2019.2 armv7l</li> </ul>		1112 rl	bee7896c	3202812268		
4 @ Kernel		111 12	00000000	0		
P 0 swapper/0 (Running)		0.9 63	00000000			
P 2 kthreadd (Running)		017 rd	bee78858	3202811992		
# 3 rou_gp (Running)		0# r5	8000006c	199		
# 4 rcu_par_gp (Running)		B17 05	88422688	4374536		
# 6 kworker/0.0H (Running)		114.67	8080808e	142		
8 mm percou wa (Running)		/// r8	bee78858	3202811992		
# 9 ksoftirgd/0 (Bunning)		817 (9	83891969	58927977		
I0 rcu preempt (Running)		817 +10	bee7096c	3202812268		
P 11 rcu sched (Running)		/// r11	bee78858	5202811992		
# 12 rcu_bh (Running)		BV +1.2	8000008e	142		
# 15 migration/0 (Running)		/// MD	bee78888	5202811904		
# 14 cpuhp/0 (Running)		JUP 10	b6ec7813	3868958547		
# 15 cpubp/1 (Running)		412 pc	b6e4d6c6	3868458582		
st 16 migration/1 (Running)	-	o fig ense	86818636	65584		
J# 17 kaoftingd/1 (Running)		-0 -817 LBF				
13 kworker/1:0 (Running)		+ All fig				
J9 kworker/1:0H (Running)		892 ing				
20 kdevtmpfs (Running)		u JUZ abt				
"P 21 kawarker/u4:1 (Running)		> 317 Land				
P 22 rcu_tasks_kthre (Running)		u JUF ave				
P 24 kwarker/u4:2 (Running)		v JW man				
s <sup>10</sup> 32 kavarker/1:1 (Running)		u Bif yfp				
30 kworker/0:1 (Running)		<ul> <li>.87 cp15</li> </ul>				
278 com_reaper (Running)		= JUZ Jazelle				
279 writeback (Running)		<ul> <li>ли дру_qos301_сри</li> </ul>			AMBA Quality of Service for CPU-to-DDR	
P 281 crypto (Running)		# 712 gpv_qos301_dmac			AMBA Quality of Service for DMAC	
P 283 kblockd (Running)		# //P gpv_gos301_iou			AMBA Quality of Service for IOU	
# 324 edac-poller (Running)		III gpv_trustzone			AMBA NIC301 TrustZone	
P 340 watchdogd (Running)		# JHF 12cache			L2 cache PL310	
JP 425 rpciod (Running)		+ /11/ mpcore			Application Processing Unit	
JP 426 kworker/u5t0 (Running)						
uP 427 xprtiod (Running)						
P 443 Issvapd0 (Running)     P 444 nfsicd (Running)						÷.
444 infsied (Running)     9 541 api0 (Running)		Disassembly 11				100 C
# 541 spi0 (Running) # 717 irg/30-mmc0 (Running)		- Disappenday			California Internet Alexan	
# 748 ipy6 addreamf (Running)		b6e4d6b8; andeq	r8, r8, r8		Chter location ne	The transferred to the second
JP 749 kworker/0:2 (Running)			r0, r0, r0			
uli 839 kworker/0:2H (Running)		b6e4d6c0; strbtmi	r11, [c7].	-r0, 1s1 #11		
JP 1059 kworker/1:2H (Running)		b6e4d6c4: stclt	p15, c13, [			
JP 1061 (bd2/mmcblk0p2- (Running)			54 d4, d13,			
1062 ext4-nv-conver (Running)		b6e4d6cc: Idrbtmi b6e4d6d0: Idrmi	r8, [r11], r6, [r8], #	#-3952		14
4 ST 1 init		b6e4d6d4: svclt	8x00004770	-2075		
- d 1 (Suspended)			F7, F8, [50	110		
Dob6e4d6c6 (libc-2.28.so)		b6e4d6dc: movwcs	r11, #1288			
Oxb6ec7812 (libc-2.28.so)		b6e4d6e0: b1x	-30687198;	addr=8xb51097	0a	
Dx0D419eb0 (init.sysvinit)			p8, 4, c15,	c12, c7		
<ul> <li>Oxb6ec7812 (libc-2.28.sp)</li> </ul>			54 d4, d13,			
a constant inter-stantabl		b6e4d6ec: Idrbtmi	r0. [r11].			

*Figure 9-8:* **Process/Thread Level Debugging** 

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

## Debugging the linux\_hello Application with OS Aware Debug

- 1. Mount an SD card using mount /dev/mmcblk0p1 /mnt.
- 2. Run the /mnt/linux\_hello.elf application from the terminal as shown in the following figure.

💷 Seri	al COM85	9/14/19, 10:	06 AM)	23			
root@	cilinx-z	702-2019	_2:/#	mount	/dev/mm	cblk0p1	/mnt
root@	cilinx-z	702-2019	2:/#	/mnt/]	inux_he	llo.elf	
Hello	World						
Hello	World						
Hello	World						
Hello	World						
Hello	World						

#### Figure 9-9: Serial Terminal: Running the Linux\_hello Application

3. To debug the linux\_hello application you created in the previous section using OS aware debug, follow the steps described in Debugging Linux Processes and Threads Using OS Aware Debug, page 124, and in addition, add the path mappings for the linux\_hello application as given in the following figure.



The source path is /linux\_hello.elf. The destination path is <vitis-workspace>linux\_hello/Debug/linux\_hello.elf.

Debug Configurations			-					
Debug a program using Application Debugger.			X					
1 R 9 R <b>X</b>   9 9 •	Name Debugger_linus_hello-Default							
type filter text	😰 Main 🛄 Application 🐵 Target Setup 🦇 Arguments 👼 Environment 🗊 Symbol Files 💱 Source	& Path Map						
Launch Group	File Path Map Rules:							
<ul> <li>OpenCL</li> <li>OpenCL (FCF)</li> </ul>	Source	Destination Ac						
Target Communication Framework	📝 /linuc/helloxelf	Chdesigns/customip_workspace\linux_hello\Debug\linux_hello.elf	Edit					
& Xilinx AL Engine SystemC Simulator & Xilinx Application Debugger	/ <petalinux-project>/xilirx-zc702-2019.2/build/tmp/work-shared/zc702-zyrq7/kernel-source</petalinux-project>	C/ <iocal-directory>\linux-xinx</iocal-directory>	Remove					
<ul> <li>4 &amp; Xintx Application Debugger</li> <li>4 Debugger_linur_hello-Default</li> </ul>			Up					
Xilinx Application Debugger (GDB) Xilinx SPM Analysis			Down					
	1 ×	· · · ·						
liter matched 12 of 24 items		Revert	Apply					
(2)		Debug	Close					

Figure 9-10: Path Mapping Information in Debug Configurations

- 4. The destination path is in the Debug view. Right-click on **linux\_hello Debug (Local)** and select **Relaunch**.
- 5. In the Vitis debugger, do the following:
  - a. Observe the running application as one of the processes/threads in kernel.
  - b. Right-click on the linux\_hello.elf thread and click **Suspend** to suspend application.
  - c. Add a breakpoint.

These actions are shown in the following figure.



(0) (0) • • • • (0) × (> (0) = N > (> (0) = N (0) (0) + • (0)	o -				Quick A	Access	ion (1) Det	aug 🕼 Remote System Ex	
Debug 21 Co Project Explorer Sc Explorer		Terminal 12	C 27   D 1	alin n 🔊					
1250 (Running)		Serial COM85 (9/17/19.2						Rex BERN	
4 🐷 1261 tcf-agent		Fingerprint: shall! c4:44:c4:c7:4e:36:9a:2d:d9:46 +				2 > lines			
🔎 1261 (Running)		opbear,							
🔎 1262 (Running)	St	arting internet sup arting syslogd/klog	erserver: in	etd.					
1263 (Running)		arting syslogd/klog arting tcf-agent: C							
1264 (Running)									
<ul> <li>I 266 start_getty</li> <li>1266 (Running)</li> </ul>	Pe	talinux 2019.2 xili	nx-zc702-201	9_2 /dev/ttyl	950				
✓ 200 (Running) ✓ 2017 start getty	~ 1	linx-zc702-2019 2 ]	onin: nost						
1267 (Running)		ssword:	login: root						
* IP 1268 getty	r o	ot@wilinx-zc702-201							
P 1268 (Running)		ot@xilinx-zc702-201							
4 🛞 1269 login		ot@xilinx-zc702-203 ot@xilinx-zc702-201		1.1					
1269 (Running)	no	ot@xilinx-zc702-201	9 2:-# cd /m	nt	pr.				
4 🔐 1272 sh	nd	root@wiliw.zc702.2019_2:/mnt# ./linux_hello.elf Hello Norld Hello Norld Hello Norld Hello Norld							
🔎 1272 (Running)									
4 ₩ 1282 linux_helta.eff						No scope specified.			
<ul> <li>         1282 (Breakpoint: linux_hello.c:25), ARM Cortex-A9 MPCore #1      </li> </ul>	He								
0x0001043c (linux_hello.elf] main(): ./src/linux_hello.c, line 25	II. He	llo World							
Oxb6e604a4 [libc-2.28.so] Oxb6e604a4 [libc-2.28.so]	He	llo World			1				
Okosebolari (IIDC-2.28.50)					-				
	-	10			+				
linux_helio.c 22	н.   ИN	Registers 8	(in 199 in	121007	- 0	III Disassembi	y 23	-	
1=/*		ame	Hex	Decimal	De *		Enter loca	ation here 🔹 🐑 🛱 🕮	
2 * Copyright (c) 2012 Xilinx, Inc. All rights reserved		107 r8	00000000	0		22			
4 * Xilinx, Inc.		382.48	08000000	0		23	{ whill	(1)	
5 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATIC		/// r10	berearad	3869591468 3197328532			main:		
6 COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR 7 MORE POSSTBLE THE EMERITATION OF THIS FEATURE APRIL		///( r11	be935c94 080000a2	3197328532		00010434:	push	{r11, 1r}	
7 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLI 8 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THE		107 ND	be935c90	3197328528		00010438: 25	hhe	r11, ap, #4 printf("Hello World	
9 * 15 FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE		202 Ir	b6ebb243	3063899907	13	> 0001043c :		r0, #1220	
10 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IM		JIV DC	0801843c	66628		00010440:	movt	r0, #1	
11 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER W 12 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NO	1.0	(IIIf opsr	60070010	1611071504		00010444: 26	bl	-312 ; addr=0 sleep(5);	
13 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENT		INF use				00010448:	nov	r0, 45	
4 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF F		//it? Fig				0001044c :	bl	-332 ; addr=9	
15 * AND FITNESS FOR A PARTICULAR PURPOSE.		102 irg				24	6		
16 * 17 */		/// abt				00010450:	ь	-28 ; addr=0	
18		/// und				00010454:	c	r4,r5,r6,r7,r8,	
19 #include (stdio.h>		IIV svc			1	00010458:	mov	r7, r0	
20		10 mon			- 1	0001045c:	1dr	r6, [pc, #+72]	
21=int main() 22 {		307 vrp 307 cp15				00010450:	BOV	r8, r1	
22 L 23 while(1)		III Jazelie				00010464:	ldr	r9, r2 r5, [pc, #+64]	
24 (		III gpv gos301 cpu			A5	0001046c:	ndd	r6, pc, r6	
<pre>printf("Hello World\n");</pre>		107 gpv_qos301_dmac			AA.	00010470:	61	-480 ; addr=0	
26 sleep(5); 27 1		107 gpv_gos301_lou			AA.	00010474:	add	r5, pc, r5	
27 j 28 return 0;		IIII gpv_trustzone			Að.	00010478: 0001047c:	asrs	r6, r6, r5 r6, r6, #2	
29 )		/III? I2cache			1.2	00010488:	popeq	{r4,r5,r6,r7,r8,	
30		IIII mpcore			As -	00010484:	MOV	rd, 40	
	- 20					00010488: 0001048r	add	rd, r4, #1	

Figure 9-11: Debugging a Process from main ()

When the control hits the breakpoint, the Debug view updates with the information of the linux\_hello.elf process.

The Debug View also shows the file, function, and the line information of the breakpoint hit. A thread label includes the name of a CPU core, if the thread is currently running on a core.

You can perform source level debugging, such as stepping in, stepping out, watching variables, stack trace, and registers.

You can perform process/thread level debugging, including insert breakpoints, step in, step out, watch variables, stack trace, and so on.

Some additional information about this process:

- One limitation with this process is that the target side path for a binary file does not include a mount point path. For example, when the linux\_hello process is located on an SD card, which is mounted at /mnt, the debugger shows the file as /linux\_hello.elf instead of /mnt/linux\_hello.elf.
- There is an additional way to Enable Linux OS Awareness in the Vitis software platform using an XSCT command line command. For information about this command, refer to the osa command help in XSCT [Ref 11].



Appendix A

## 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.
- 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.

Send Feedback



## **Related Design Hubs**

The following design hubs are applicable to embedded development and the methods described in this guide:

• PetaLinux Tools Design Hub

## **Design Files for This Tutorial**

The ZIP file associated with this document contains the design files for the tutorial. You can download this file from this link.

Design files contain the HDF files for each section, and the source code and pre-built images for all the sections.

## **Xilinx Resources**

The following Xilinx Vivado Design Suite and Zynq®-7000 SoC guides are referenced in this document.

- 1. Zynq-7000 SoC Technical Reference Manual (UG585)
- 2. Zynq-7000 SoC Software Developers Guide (UG821)
- 3. Vivado Design Suite User Guide: Using Constraints (UG903)
- 4. Vivado Design Suite User Guide: Getting Started (UG910)
- 5. Vivado Design Suite Tutorial: Embedded Processor Hardware Design (UG940)
- 6. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)
- 7. UltraFast Embedded Design Methodology Guide (UG1046)
- 8. PetaLinux Tools Documentation: Reference Guide (UG1144)
- 9. System Performance Analysis of an SoC (XAPP1219)
- 10. Vitis Embedded Software Development Flow Documentation (UG1400)
- 11. Xilinx Software Command-Line Tool Reference Guide (UG1208)

## **Support Resources**

- 12. Embedded Design Tools Web page
- 13. Xilinx Zynq® Tools Wiki Page



- 14. Xilinx Zynq Linux Wiki page
- 15. The Software Zone

## **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. Arm Cortex-A9 MPCore Technical Reference Manual, section 2.4, Accelerator Coherency Port (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407e/CACGGBCF.html)
- 18. Xilinx GitHub website: https://github.com/xilinx
- 19. GitHub ZYNQ Cookbook: How to Run BFM Simulation: https://github.com/imrickysu/ZYNQ-Cookbook/wiki/How-to-run-BFM-simulation
- 20. The Linux Kernel Module Programming Guide: http://tldp.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:

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 2015-2019 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries.

