ZedBoard: Zynq-7000 AP SoC Concepts, Tools, and Techniques

A Hands-On Guide to Effective Embedded System Design

ZedBoard (Vivado 2013.2)

Notice of Disclaimer

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 the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; 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 Critical Applications: http://www.xilinx.com/warranty.htm#critapps.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
08/07/2013	2013.2	Update for Vivado 2013.2.

Table of Contents

Table of Figures

Chapter 1
Introduction

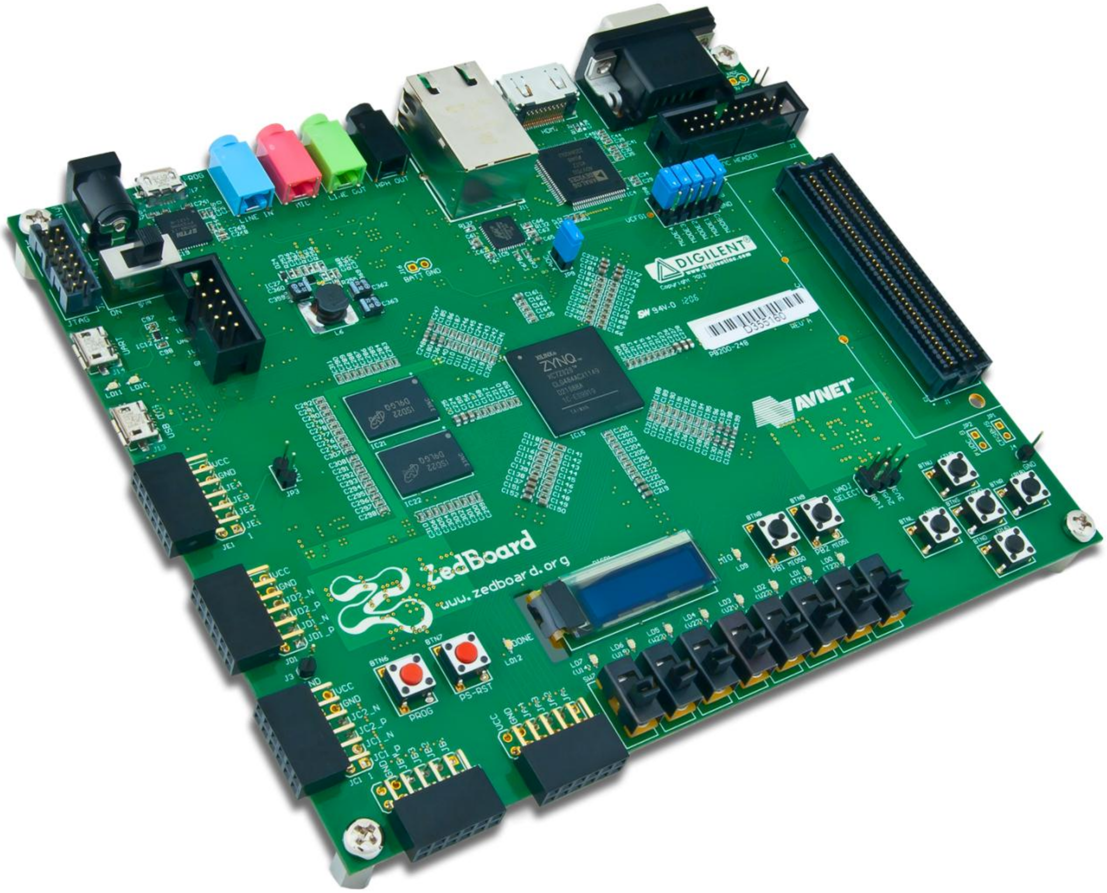


Figure 1-1: The ZedBoard Zynq Evaluation and Development Kit

1.1 About this Guide

This document provides an introduction to using the Xilinx® Vivado® Design Suite to build a Zynq™-7000 All Programmable SoC (AP SoC) design. The examples target the ZedBoard (http://www.zedboard.org) using Vivado version 2013.2.

Note: The Test Drives in this document were created using Microsoft Windows 7 64-bit operating system. Other versions of Windows might provide varied results.

The Zynq-7000 family is the world's first All Programmable SoC. This innovative class of product combines an industry-standard ARM® dual-core Cortex™-A9 MPCore™ processing system with Xilinx 28 nm unified programmable logic architecture. This processor-centric architecture delivers a complete embedded processing platform that offers developers ASIC levels of performance and power consumption, the flexibility of an FPGA, and the ease of programmability of a microprocessor.

This guide describes the design flow for developing a custom Zynq-7000 AP SoC based embedded processing system using the Xilinx Vivado software tools. It contains the following five chapters:

- **Chapter 1,** (this chapter) provides a general overview.
- **Chapter 2**, **"Embedded System Design Using the Zynq Processing System"** describes the tool flow for the Zynq Processing System (PS) to create a simple standalone "Hello World" application.
- **Chapter 3**, **"Embedded System Design Using the Zynq Processing System and Programmable Logic"** describes how to create a system utilizing both the Zynq PS as well as the Programmable Logic (PL).
- **Chapter 4**, **"Debugging with SDK and ChipScope Pro"** provides debugging debugging techniques via software (using SDK Debug) and hardware (using the ChipScope™ software)debugging tools.
- **Chapter 5, "Booting Linux and Application Debugging using SDK"** covers programming of the non-volatile memories such as QSPI Flash and SD Card with the Linux precompiled images, which are used for automatic Linux booting after switching on the board.
- **Chapter 6, "Further "How-to's" and Examples"** links the reader to online resources available to the ZedBoard designer including design projects and further documentation.
- **Appendix A**, **"Application Software"** describes details of the application needed for the example design used in this guide.

1.1.1 **Take a Test Drive!**

The best way to learn a software tool is to use it, so this guide provides opportunities for you to work with the tools under discussion. Procedures for sample projects are given in the Test Drive exercise sections, along with an explanation of what is happening behind the scenes and why you need to do it.

Test Drive exercises are indicated by the car icon, as shown beside the heading above.

1.1.2 **Additional Documentation**

For further information, refer to:

- **Xilinx Zynq-7000 Documentation:**

  http://www.xilinx.com/support/documentation/zynq-7000.htm

- **Vivado Design Suite User Guide: Release Notes, Installation and Licensing Guide (UG973):** http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug973-vivado-release-notes-install-license.pdf

- **Vivado Design Suite User Guide: Getting Started (UG910):** http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug910-vivado-getting-started.pdf

- **Xilinx Glossary:** http://www.xilinx.com/company/terms.htm

- **ZedBoard.org:**

  http://www.zedboard.org

1.2 How Zynq AP SoC and Xilinx software Simplify Embedded Processor Design

The Zynq-7000 All Programmable SoC reduces system complexity by offering a dual core ARM Cortex-A9 processing system and hard peripherals coupled with Xilinx 7-Series 28 nm programmable logic all integrated on a single SoC. It is the first of its kind in the market and has tremendous potential as a tightly integrated system.

The Zynq Processing System (PS) may be used without anything programmed in the Programmable Logic (PL). However, in order to use any soft IP in the PL, or to route PS dedicated peripherals to device pins for the PL, you have to program the PL.

Xilinx offers several sets of tools to simplify the design process. This document focuses on using the Vivado™ Design Suite. The Vivado Design Suite with IP Integrator replaces many functionalities traditionally found in the ISE™ Design Suite's PlanAhead and XPS tools. This includes constructing the PS hardware system while streamlining the process tremendously. The Vivado Logic Analyzer replaces the ChipScope Analyzer from ISE.

With the Xilinx tools required to work with your ZedBoard, it is a good idea to get to know the basic tool names, project file names, and acronyms. You can find Xilinx software-specific terms in the Xilinx Glossary:

http://www.xilinx.com/company/terms.htm

Xilinx Vivado Design Suite

The Vivado Design Suite the free, downloadable, fully featured front-to-back FPGA design solution running under Linux, Windows XP, and Windows 7, supporting the ZedBoard.

The Vivado Design Suite includes Vivado, IP Integrator and the Software Development Kit (SDK), amongst others. The DVD packaged with your ZedBoard may contain ISE or Vivado depending on when the board was manufactured. A description of the license features in the various installation categories of the Vivado Design Suite is available via this hyperlink:

http://www.xilinx.com/products/design_tools/vivado/vivado-webpack.htm

Vivado Software Tools

The Vivado Design Suite provides a central cockpit for design entry in RTL, synthesis, implementation and verification. The software includes integration with IP Integrator for access to the Xilinx IP catalog (including embedded processor design), and the SDK to complete the embedded processor software design. The implementation flow of your design may be centrally launched from the Vivado GUI.

- For more information on the embedded design process as it relates to Vivado, see the "Design Process Overview" in the ***Vivado Design Suite User Guide Design Flows Overview (UG892)*:**

  http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug892-vivado-design-flows-overview.pdf

Note: Direct simulation of the Processing System is not available.

IP Integrator

The IP Integrator feature in the Vivado Design Suite can be used for designing the hardware portion of your embedded processor system. You can specify the ARM Cortex-A9 processor core, IP peripherals, and the interconnection of these components along with their respective detailed configuration.

Software Development Kit

The SDK is an integrated development environment, complementary to Vivado, that is used for C/C++ embedded software application creation and verification. The SDK is built on the Eclipse open-source framework. For more information about the Eclipse development environment, please refer to

http://www.eclipse.org.

Other Components of Vivado Design Suite

Other components include:

- Hardware IP to complement the embedded processors
- Drivers and libraries for the embedded software development
- GNU compiler and debugger for C/C++ software development targeting the ARM Cortex-A9 MPCore in the Zynq Processing System
- Documentation
- Sample projects

1.3 What You Need to Set Up Before Starting

Before discussing the tools in depth, it would be a good idea to make sure they are installed properly and that the environments you set up match those required for the Test Drive sections of this guide.

1.3.1 Software Installation Requirements:

1. Xilinx Vivado Design Suite

This tuorial requires version 2013.2 of the Vivado Design Suite, and version 2013.2 of the SDK. Most new ZedBoards include a Vivado Design Suite 2013.2 DVD and a license entitlement voucher. If your board shipped with a ISE DVD, you should download the single file Vivado 2013.2 installation package via the Download Center on the Xilinx website and choose the Vivado WebPACK + SDK installation option during setup.

2. Software Licensing

Xilinx software uses FLEXnet licensing. A license is required to synthesize, implement and generate bitstreams in the Vivado Design Suite.

If your ZedBoard kit contains a Vivado Design Suite DVD and voucher, then you are entitled to the Vivado Design Suite Design edition license. This license will enable more functionalities than the WebPACK version. Please refer to the earlier Section 1.2 for a link to the webpage comparing WebPACK features against the Design Edition. The Xilinx License Configuration Manager (XLCM) is automatically launched as a final installation step. When XLCM starts, it prompts you to register via the Xilinx Licensing

Center via a web browser. In the licensing center, generate a Vivado Design Suite Design edition node locked license with the voucher code.

The WebPACK license is already available in the Xilinx Licensing Center by default and does not require a voucher code to enable. Only use this if you do not have a voucher.

The generated license file will be e-mailed to you. Save the file to a convenient location on your hard drive. With XLCM open, specify the location of the license file, XLCM then automatically identifies the enabled features and Vivado is ready for use.

Note, the Hardware Analyzer feature is not enabled for the WebPACK version.

3. Serial Terminal Emulation

Certain Test Drive exercises require the use of a serial terminal emulator external to the SDK. The exercises have been tested with PuTTY and Tera Term although other terminal utilities can be used as well. The settings for setting up a session can be found in Section 2.1.2.

1.3.2 Hardware Requirements for this Guide

The ZedBoard is required to complete the tutorial. Two micro USB cables are required to connect both the USB-JTAG and USB-UART on-board. A standard Ethernet cable is also required for exercises later on in this document.

Chapter 2
Embedded System Design Using the Zynq Processing System

Now that you've been introduced to the Xilinx software tools and hardware requirements, you will begin looking at how to use them to develop an embedded system using the Zynq PS.

The Zynq AP SoC consists of an ARM Cortex A9 MPCore PS which includes various dedicated peripherals as well as a configurable PL. This offering can be used in three ways:

1. The Zynq PS can be used independently of the PL.
2. Soft IP may be added in the PL and connected to extend the functionality of the PS. You can use this PS + PL combination to achieve complex and efficient design on the SoC.
3. Logic in the PL can be designed to operate independently of the PS. However the PS or JTAG must be used to program the PL.

The design flow is described in Figure 2-1: Vivado Design Flow for Zynq.

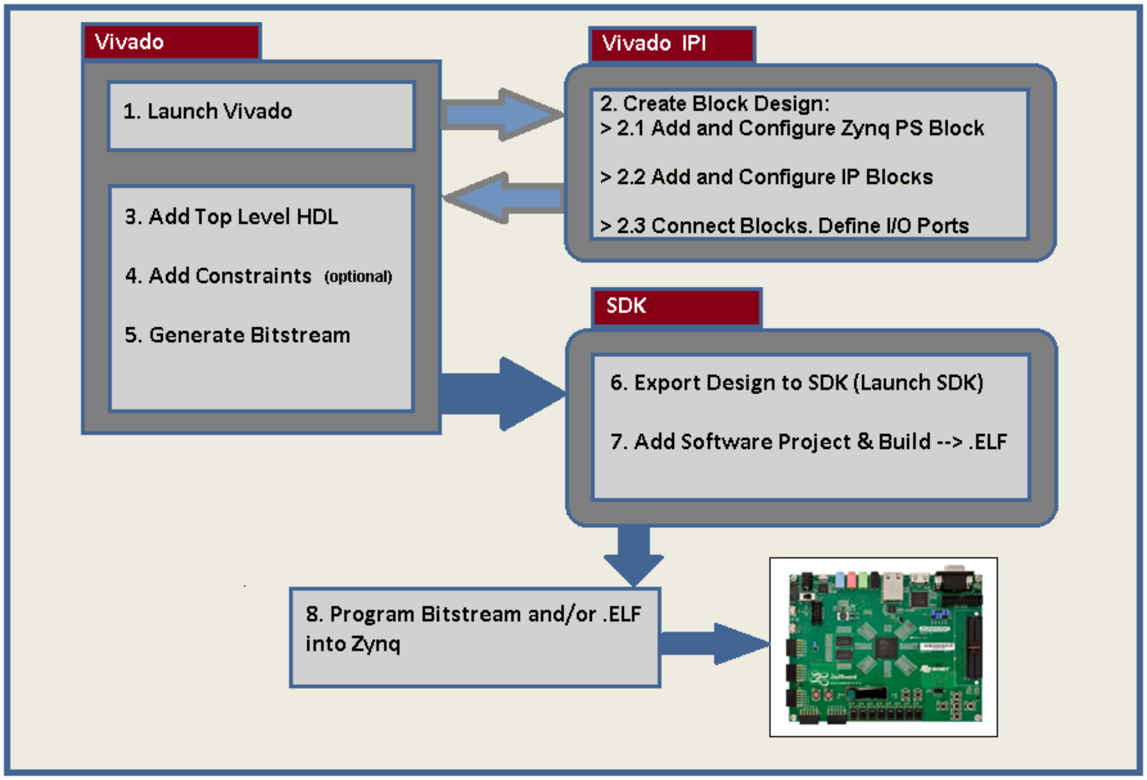

Vivado
1. Launch Vivado
3. Add Top Level HDL
4. Add Constraints (optional)
5. Generate Bitstream
Vivado IPI
2. Create Block Design:
> 2.1 Add and Configure Zynq PS Block
> 2.2 Add and Configure IP Blocks
> 2.3 Connect Blocks. Define I/O Ports
SDK
6. Export Design to SDK (Launch SDK)
7. Add Software Project & Build --> .ELF
8. Program Bitstream and/or .ELF into Zynq

Figure 2-1: Vivado Design Flow for Zynq

1. The recommended design and implementation process begins with launching Vivado, which is the central cockpit from which design entry through bitstream generation is completed.
2. From Vivado, select Create Block Design to launch IP Integrator within the GUI. Add the ZYNQ7 Processing System IP to include the ARM Cortex-A9 PS in the project. Selection of optional addition of PL peripherals occur within IP Integrator.
3. Double click on the ZYNQ7 Processing System block to configure settings to make the appropriate design decisions such as selection/de-selection of dedicated PS I/O peripherals, memory configurations, clock speeds, etc.
4. At this point, you may also optionally add IP from the IP catalog or create your own customized IP. Connect the different blocks together by dragging signals / nets from one port of an IP to another.
5. When finished, generate a top-level HDL wrapper for the system.
6. Ensure that the appropriate PL related design constraints are defined as required by the tools. If any signal coming from the PL section to an I/O pin is not defined then the tools will generate an error during the bitstream generation. Also, do not include pin constraints which are connected to the dedicated pins as the tools will generate the error. These constraints would typically be useful to ensure that signals to general purpose I/O such as the switches, LEDs, and Push Buttons on the ZedBoard are routed appropriately. This is done via the creation/addition of a .XDC constraints file in the Vivado project.
7. Generate the bitstream for configuring the logic in the PL if soft peripherals or other HDL are included in the design, or if hard peripheral IO was routed through the PL. At this stage, the hardware has been defined in <system.xml>, and if necessary a bitstream <system.bit> has been generated. The bitstream could be programmed into the FPGA; or it could be done from within SDK.
8. Now that the hardware portion of the embedded system design has been built, export it to SDK to create the software design. (A convenient method to ensure that the hardware for this design is automatically integrated with the software portion is achieved by Exporting the Hardware from Vivado to SDK). In order to export the design to SDK successfully, the Design Block MUST be open and the implemented design, if exists, MUST be open, otherwise the tools will generate an error.
9. In SDK, add a software project to associate with the hardware design exported from Vivado.
10. Within SDK, for a standalone application (no operating system) create a Board Support Package (BSP) based on the hardware platform and then develop your user application. Once compiled, a <designname.elf> is generated.

11. The combination of the optional bitstream and the .ELF file provides embedded SoC functionality on your ZedBoard.

2.1 Embedded System Construction

Creation of a Zynq system design involves configuring the PS to select appropriate peripherals. As long as the selected PS hard peripherals use Multiplexed IO (MIO) connections, and no additional logic or IP is built or routed through the PL, no bitstream is required. This section guides you through creating one such design, where only the PS is used.

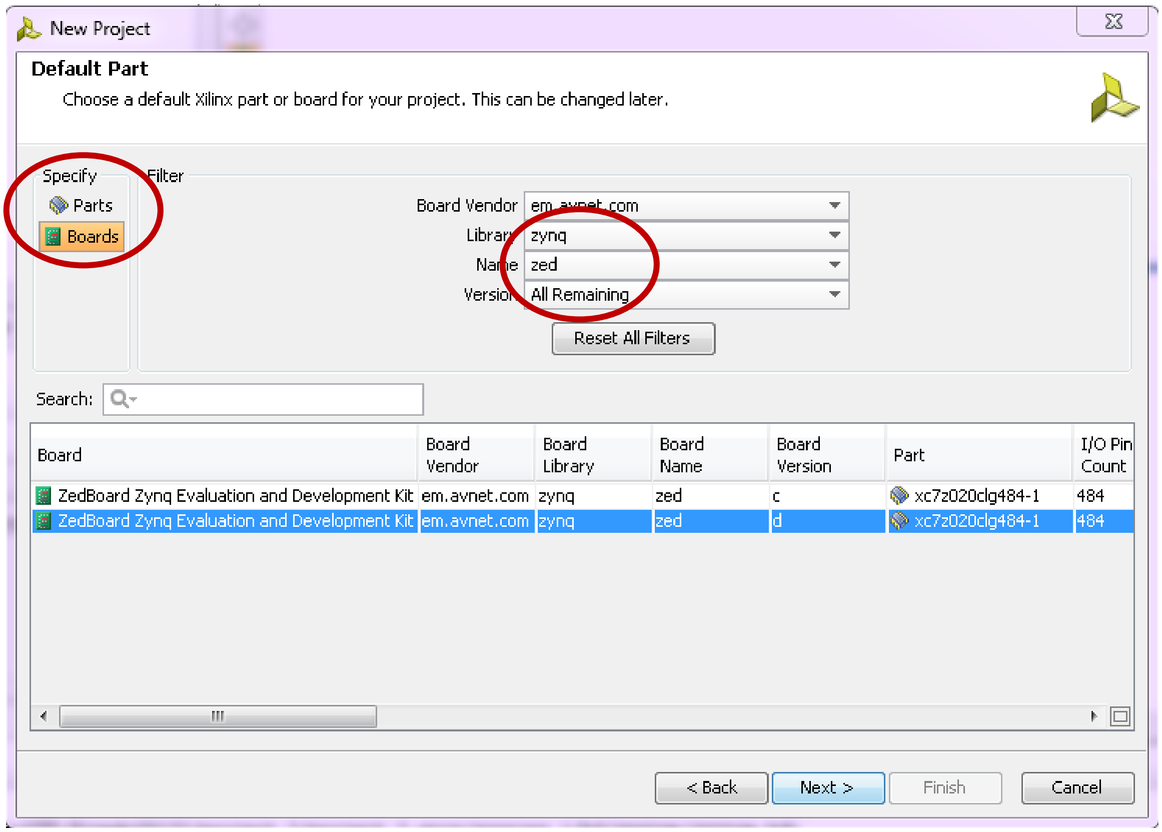
2.1.1 Take a Test Drive! Creating a New Embedded Project With a Zynq Processing System

For this test drive, you start Vivado and create a project with an embedded processor system as the top level.

Launch the Vivado Design Suite.

1. Select **Create New Project** to open the New Project wizard.
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
Project Name	Project name	Specify the project name, use the suggested default.
	Project location	Specify the directory in which to store the project files.
	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	Use the default selection, **RTL Project**.
Add Sources	Do not make any changes on this screen.	
Add Existing IP	Do not make any changes on this screen.	
Add Constraints	Do not make any changes on this screen.	
Default Part	Specify	Select **Boards**.
	Board	Select **ZedBoard Zynq Evaluation and Development Kit** of appropriate board version (C or D)
New Project Summary	Project summary	Review the project summary before clicking **Finish** to create the project.



New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

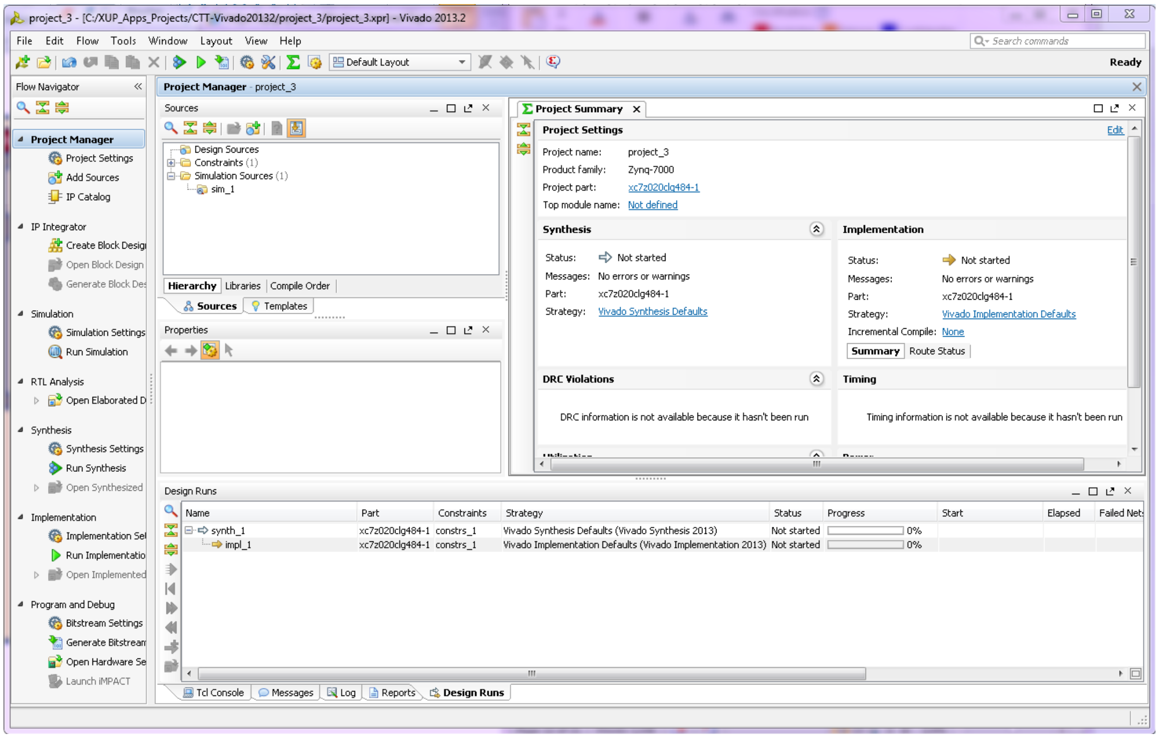
Board Vendor: em.avnet.com
Library: zynq
Name: zed
Version: All Remaining

Search:

Board	Board Vendor	Board Library	Board Name	Board Version	Part	I/O Pin Count
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	c	xc7z020clg484-1	484
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	d	xc7z020clg484-1	484

Figure 2-2: New Project Wizard Part Selection

When you click **Finish**, the New Project wizard closes and the project you just created opens in Vivado. The board you choose in the wizard has a direct impact on how IP Integrator functions. IP Integrator is board aware and will automatically assign dedicated PS ports to physical pin locations mapped to the specific board. It also applies the correct I/O standard, saving the designer time in doing so.



Project Settings

Project name: project_3
Product family: Zynq-7000
Project part: xc7z020clg484-1
Top module name: Not defined

Synthesis

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults

Implementation

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Incremental Compile: None

DRC Violations

DRC information is not available because it hasn't been run

Timing

Timing information is not available because it hasn't been run

Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Failed Nets
synth_1	xc7z020clg484-1	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2013)	Not started	0%			
impl_1	xc7z020clg484-1	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2013)	Not started	0%			

Figure 2-3: Vivado GUI

You'll now use the IP Integrator to create an embedded processor project.

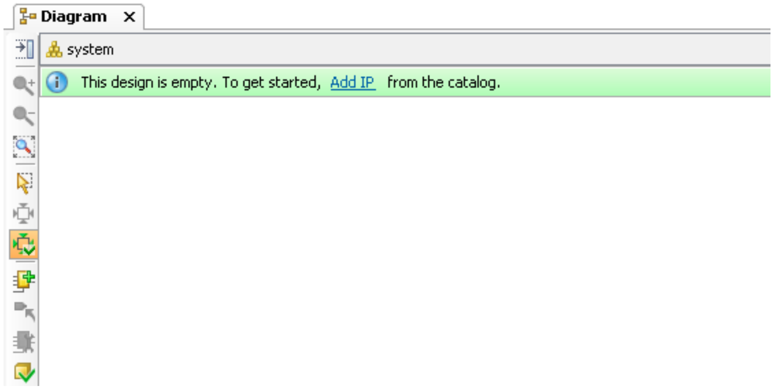
1. Click **Create Block Design** in the Project Manager.
2. Type a name for the module and click **OK**. For this example, use the name: **system**.
3. You will be presented a blank Block Diagram view in the Vivado GUI.

Creating Your Embedded System via IP Integrator

You can design a new embedded system in Vivado using IP Integrator by adding a ZYNQ7 Processing System block. By adding this block, you can configure one of the ARM Cortex-A9 processor cores for your application. You can also place additional IP blocks to further the capabilities of the embedded system.

2.1.1.1 Inserting a New Processing System Block

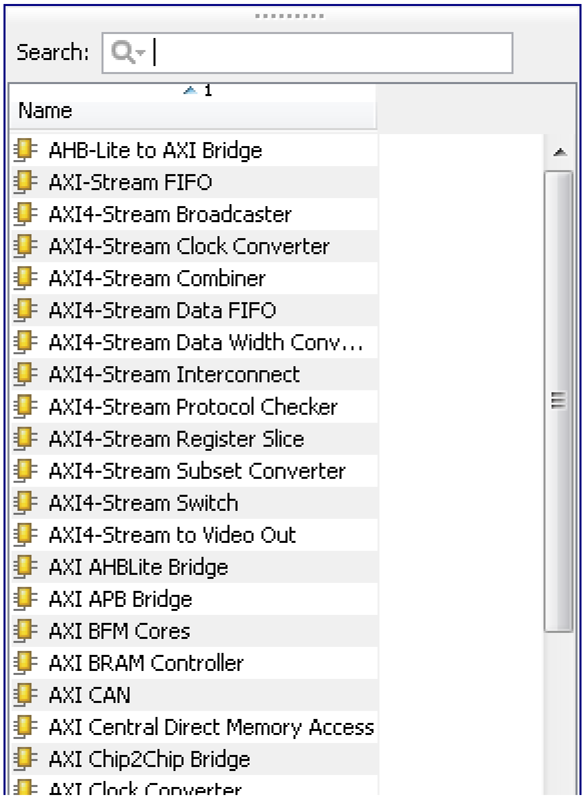
1. After creating a new Block Diagram, you will be presented with a blank Block Diagram view in the Vivado GUI. To add a new IP, you can either click **Add IP** in the green information bar at the top of the view, click on the Add IP icon on the left hand side toolbar or right click on the blank space and select **Add IP**.



This design is empty. To get started, Add IP from the catalog.

Figure 2-4: Blank Block Diagram view

The IP Catalog window will appear showing you all the IP that can be added in this view.



Name
AHB-Lite to AXI Bridge
AXI-Stream FIFO
AXI4-Stream Broadcaster
AXI4-Stream Clock Converter
AXI4-Stream Combiner
AXI4-Stream Data FIFO
AXI4-Stream Data Width Conv...
AXI4-Stream Interconnect
AXI4-Stream Protocol Checker
AXI4-Stream Register Slice
AXI4-Stream Subset Converter
AXI4-Stream Switch
AXI4-Stream to Video Out
AXI AHBLite Bridge
AXI APB Bridge
AXI BFM Cores
AXI BRAM Controller
AXI CAN
AXI Central Direct Memory Access
AXI Chip2Chip Bridge
AXI Clock Converter

Figure 2-5: IP Catalog

4. Either scroll down to the very bottom or search using the keyword *zynq*, double click on **ZYNQ7 Processing System**.
5. The ZYNQ7 Processing System block has been placed in the block diagram view. The ports shown on the block diagram are defined by the default settings for this block as specified by target development board.

6. Double click on the ZYNQ7 Processing System block to edit the settings (or Re-customize the IP).

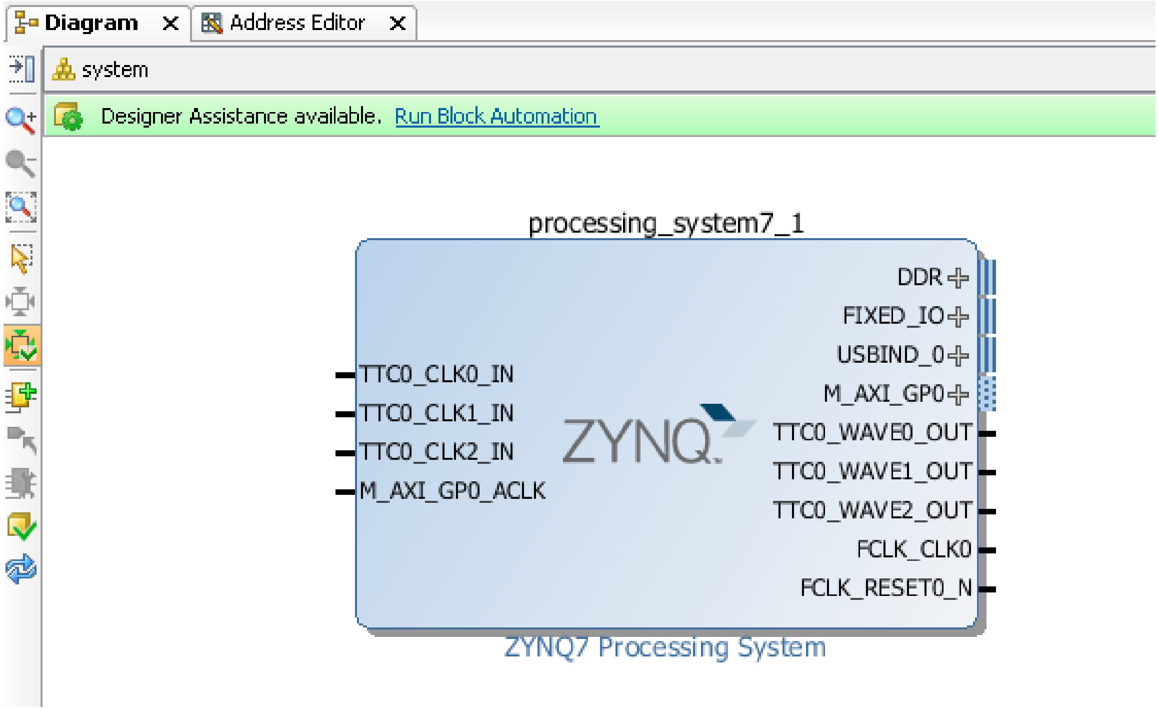


Figure 2-6: Processing System Block in the Block Diagram view

7. The default view of the Re-customize IP window shows the Zynq Processing System block diagram. The window allows you to edit any property of the Zynq PS. Click on each of the Page Navigator options on the left hand side to review the PS properties that can be edited.

   Click on the **Presets** button in the top bar, select the **ZedBoard Development Board Template** and click **Cancel**. You do not need to apply any ZedBoard presets as this was already done for you. This is a good way to reset the Zynq Processing System(PS) to the default options for the ZedBoard.

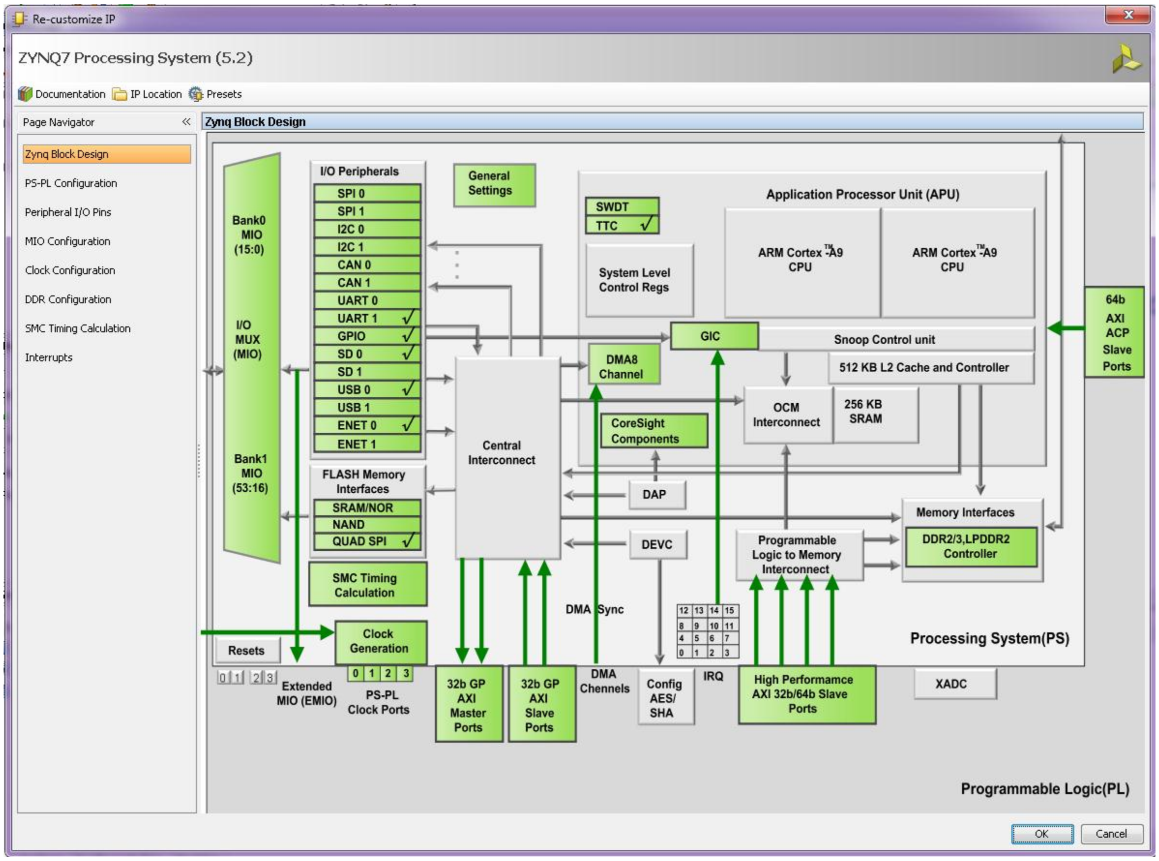


Figure 2-7: Processing System Re-customize IP view

2.1.1.2 Customizing the Processing System settings

We are going to modify the default ZedBoard based PS settings for this example project.

1. For the scope of this exercise, we will not need most of the selected I/O Peripherals. Click on the **MIO Configuration** page under the Page Navigator. Expand **I/O Peripherals** and de-select everything except **UART1**. Expand **GPIO** and de-select **GPIO MIO**.
2. While still being in the MIO Configuration page, expand **Memory Interfaces** and de-select everything. Expand **Application Processor Unit**, de-select everything.
3. Click on the **Clock Configuration** page, expand **PL Fabric Clocks**, de-select **FCLK_CLK0**.
4. Click on the **PS-PL Configuration** page, expand **GP Master AXI Interface**, de-select **M AXI GP0 interface**.
5. Also within the PS-PL Configuration page, expand **General > Enable Clock Resets**, de-select **FCLK_RESET0_N**.

6. Click on **DDR Configuration**, de-select **Enable DDR**. By disabling DDR, you are limited to applications of less than 128 KB. For this exercise, we will de-select the option.
7. Click **OK** to exit the processing system customization window.

Notice the ZYNQ7 Processing System block diagram got simplified by de-selecting all the options mentioned earlier.

8. Click **Run Block Automation** Run Block Automation in the green information bar. Select **processing_system7_1** and select **OK** in the Run Block Automation window. Notice that the FIXED_IO are now connected to output ports.
9. Click the **Save** button to save the current block diagram.

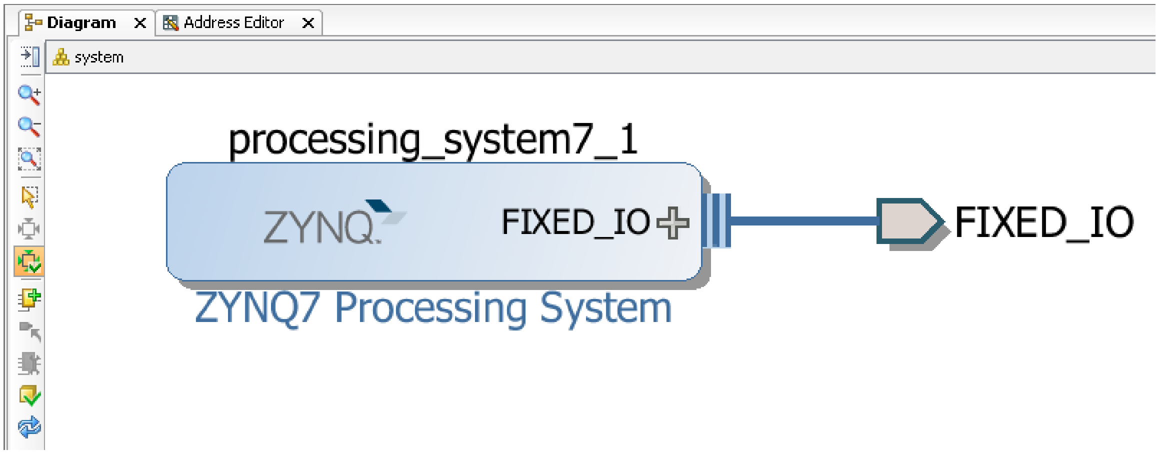
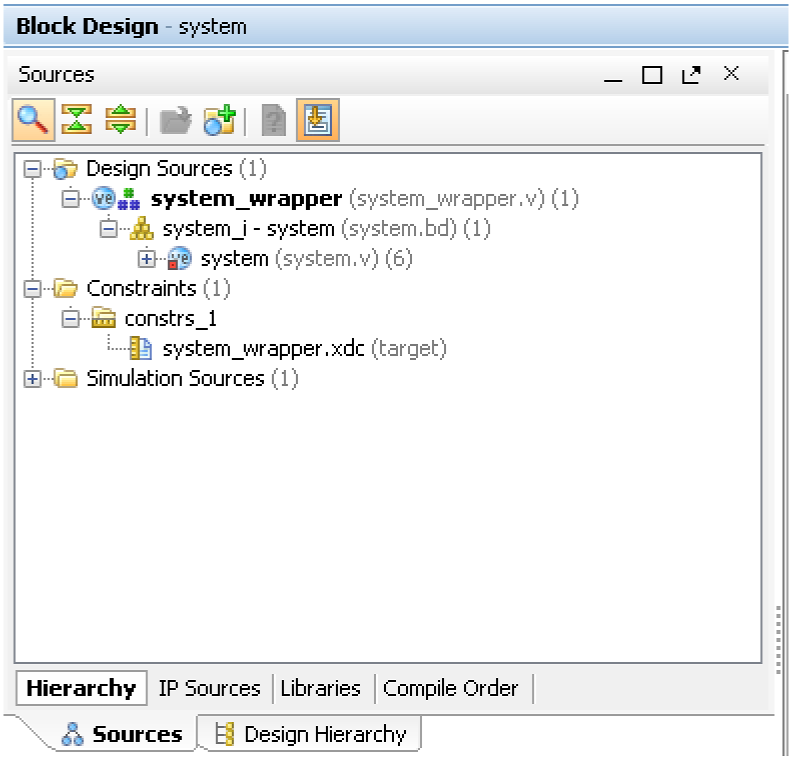


Figure 2-8: Processing System 7

10. In the Sources pane, click on the Sources tab. Right click on **system.bd** and select **Create HDL Wrapper** to create the top level Verilog file from the block diagram. Notice that *system_wrapper.v* got created and placed at the top of the design sources hierarchy.



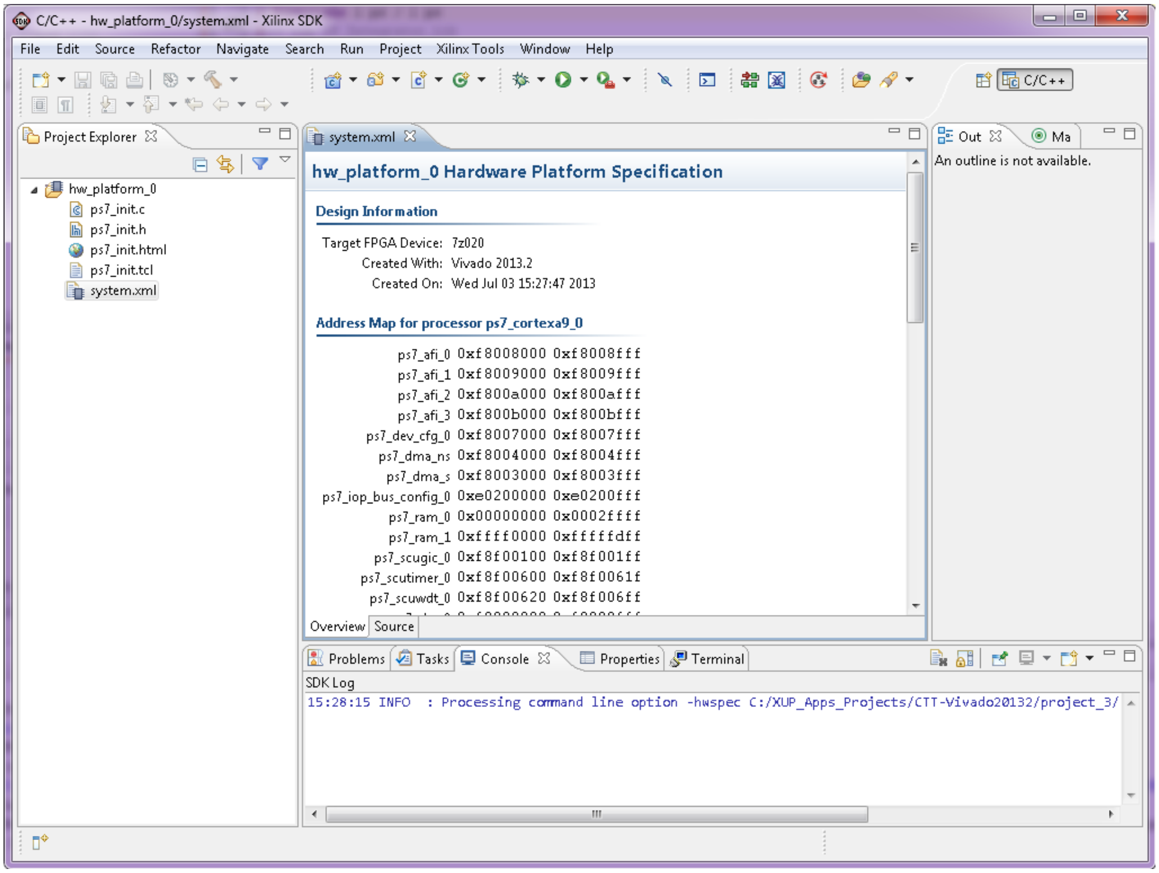
```
Design Sources (1)
    system_wrapper (system_wrapper.v) (1)
        system_i - system (system.bd) (1)
            system (system.v) (6)
Constraints (1)
    constrs_1
        system_wrapper.xdc (target)
Simulation Sources (1)
```

Figure 2-9: Sources pane showing the system.bd file

You have now successfully created a Zynq PS hardware design. Since there are no additional HDL sources to add to the design, the hardware can be exported directly to the SDK.

11. Click **File > Export > Export Hardware for SDK**.
12. The Export Hardware for SDK dialog box opens. Make sure to select **Launch SDK**. Click **OK** to continue.

    If prompted, click **Save** to save the project.


C/C++ - hw_platform_0/system.xml - Xilinx SDK
File Edit Source Refactor Navigate Search Run Project Xilinx Tools Window Help
C/C++
Project Explorer
hw_platform_0
ps7_init.c
ps7_init.h
ps7_init.html
ps7_init.tcl
system.xml
Out
Ma
An outline is not available.
Overview
Source
Problems
Tasks
Console
Properties
Terminal

hw_platform_0 Hardware Platform Specification

Design Information

Target FPGA Device: 7z020
Created With: Vivado 2013.2
Created On: Wed Jul 03 15:27:47 2013

Address Map for processor ps7_cortexa9_0

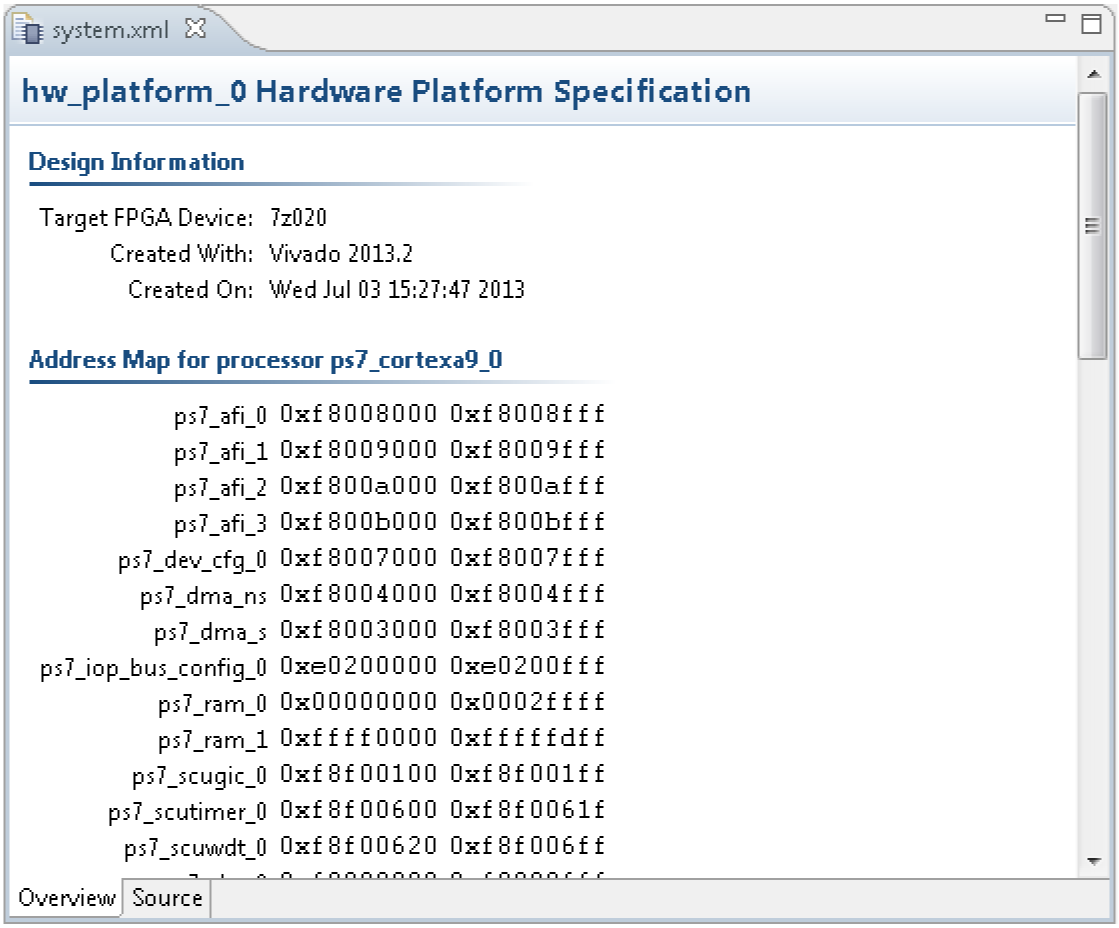
```
          ps7_afi_0 0xf8008000 0xf8008fff
          ps7_afi_1 0xf8009000 0xf8009fff
          ps7_afi_2 0xf800a000 0xf800afff
          ps7_afi_3 0xf800b000 0xf800bfff
      ps7_dev_cfg_0 0xf8007000 0xf8007fff
         ps7_dma_ns 0xf8004000 0xf8004fff
          ps7_dma_s 0xf8003000 0xf8003fff
ps7_iop_bus_config_0 0xe0200000 0xe0200fff
          ps7_ram_0 0x00000000 0x0002ffff
          ps7_ram_1 0xffff0000 0xfffffdff
       ps7_scugic_0 0xf8f00100 0xf8f001ff
     ps7_scutimer_0 0xf8f00600 0xf8f0061f
       ps7_scuwdt_0 0xf8f00620 0xf8f006ff
```

SDK Log

```
15:28:15 INFO  : Processing command line option -hwspec C:/XUP_Apps_Projects/CTT-Vivado20132/project_3/
```

Figure 2-10: The SDK GUI

When SDK launches you may need to close the Welcome Tab which obscures the entire GUI. The hardware description file is automatically read in. The system.xml tab shows the address map for the entire hardware system.



hw_platform_0 Hardware Platform Specification

Design Information

Target FPGA Device:	7z020
Created With:	Vivado 2013.2
Created On:	Wed Jul 03 15:27:47 2013

Address Map for processor ps7_cortexa9_0

ps7_afi_0	0xf8008000	0xf8008fff
ps7_afi_1	0xf8009000	0xf8009fff
ps7_afi_2	0xf800a000	0xf800afff
ps7_afi_3	0xf800b000	0xf800bfff
ps7_dev_cfg_0	0xf8007000	0xf8007fff
ps7_dma_ns	0xf8004000	0xf8004fff
ps7_dma_s	0xf8003000	0xf8003fff
ps7_iop_bus_config_0	0xe0200000	0xe0200fff
ps7_ram_0	0x00000000	0x0002ffff
ps7_ram_1	0xffff0000	0xfffffdff
ps7_scugic_0	0xf8f00100	0xf8f001ff
ps7_scutimer_0	0xf8f00600	0xf8f0061f
ps7_scuwdt_0	0xf8f00620	0xf8f006ff

Figure 2-11: Address Map in SDK system.xml Tab

What Just Happened?

The Vivado design tool exported the Hardware Platform Specification for your design (system.xml in this example) to SDK. In addition to system.xml, there are four more files relevant to SDK got created and exported. They are **ps7_init.c**, **ps7_init.h**, **ps7_init.tcl**, and **ps7_init.html**.

The system.xml file opens by default when SDK is launched. The address map of your system read from this file is shown by default in the SDK window.

The **ps7_init.c** and **ps7_init.h** files contain the initialization code for the Zynq Processing System and initialization settings for DDR, clocks, plls, and MIOs. SDK uses these settings when initializing the processing system so that applications can be run on top of the processing system.

What's Next?

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

2.1.2 Take a Test Drive! Running the "Hello World" Application

1. Connect the 12V AC/DC converter power cable to the ZedBoard barrel jack.
2. Connect a USB micro cable between the Windows Host machine and the ZedBoard JTAG (J17).
3. Connect a USB micro cable to the USB UART connector (J14) on the ZedBoard with the Windows Host machine. This is used for USB to serial transfer.
4. Power on the board using the switch indicated in Figure 2-12: ZedBoard Power switch and Jumper settings.

If this is your first time starting up the ZedBoard with the USB UART connected to your Windows PC, you may need to install the Cypress USB-to-UART device drivers. Please refer to the Cypress USB-to-UART Setup Guide on ZedBoard.org for more information:

http://www.zedboard.org/documentation

Please also make sure that under **Xilinx Tools > Configure JTAG Settings** that the JTAG Cable Type is set to "Digilent USB Cable" instead of "Auto Detect".

IMPORTANT: *Ensure that jumpers JP7 to JP11 are set as shown in the figure for the JTAG configuration mode. They should be set to 0 0 0 0 0.*

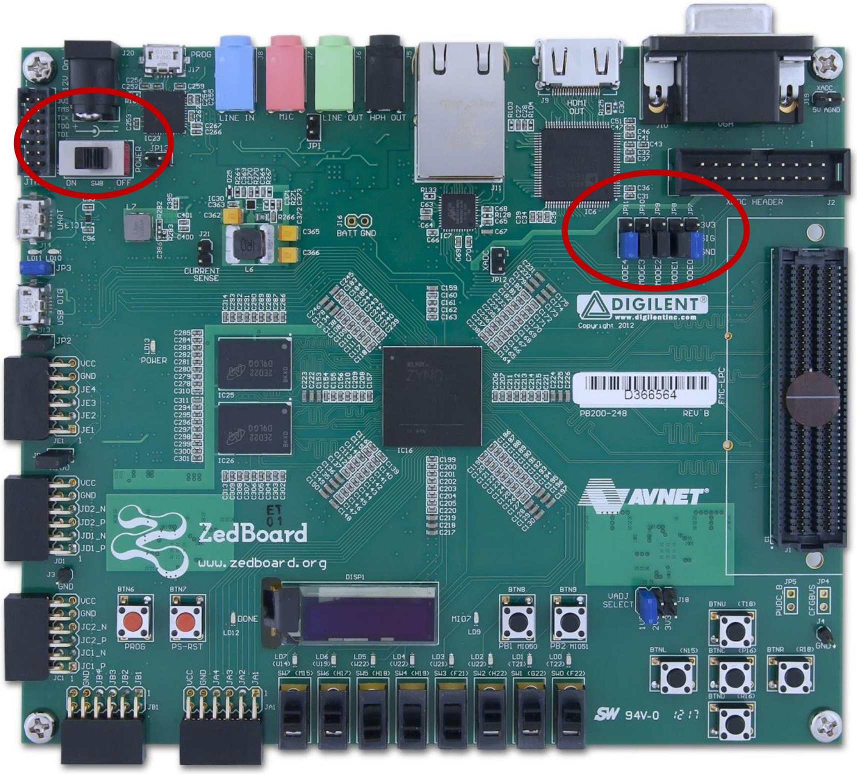


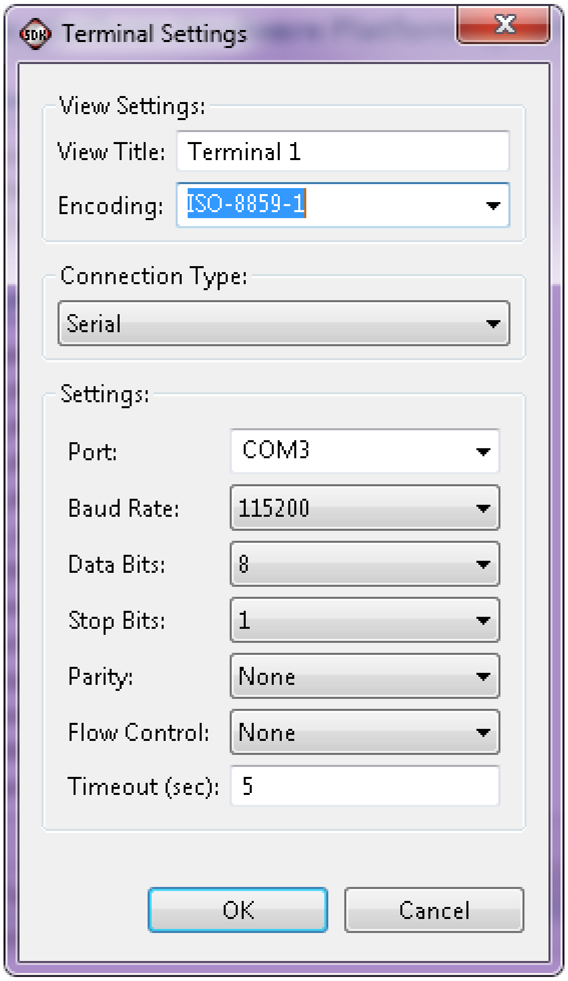
Figure 2-12: ZedBoard Power switch and Jumper settings

5. Open a serial communication utility for the COM port assigned on your system.

Note: The default configuration for Zynq7 Processing System is: **Baud rate 115200; 8 bit; Parity: none; Stop: 1 bit; Flow control: none**. Third party/external serial terminal emulators can be used in place of the SDK terminal and is required for certain test drives.

To open a serial communication terminal in SDK:

Select **Window** > **Show view** > **Terminal** and click in the console view area**.** Configure it with the parameters as shown below (replacing COM3 with the appropriate COM port number, verify using Control Panel > Device Manager).



Terminal Settings

View Settings:

View Title: Terminal 1

Encoding: ISO-8859-1

Connection Type:

Serial

Settings:

Port: COM3

Baud Rate: 115200

Data Bits: 8

Stop Bits: 1

Parity: None

Flow Control: None

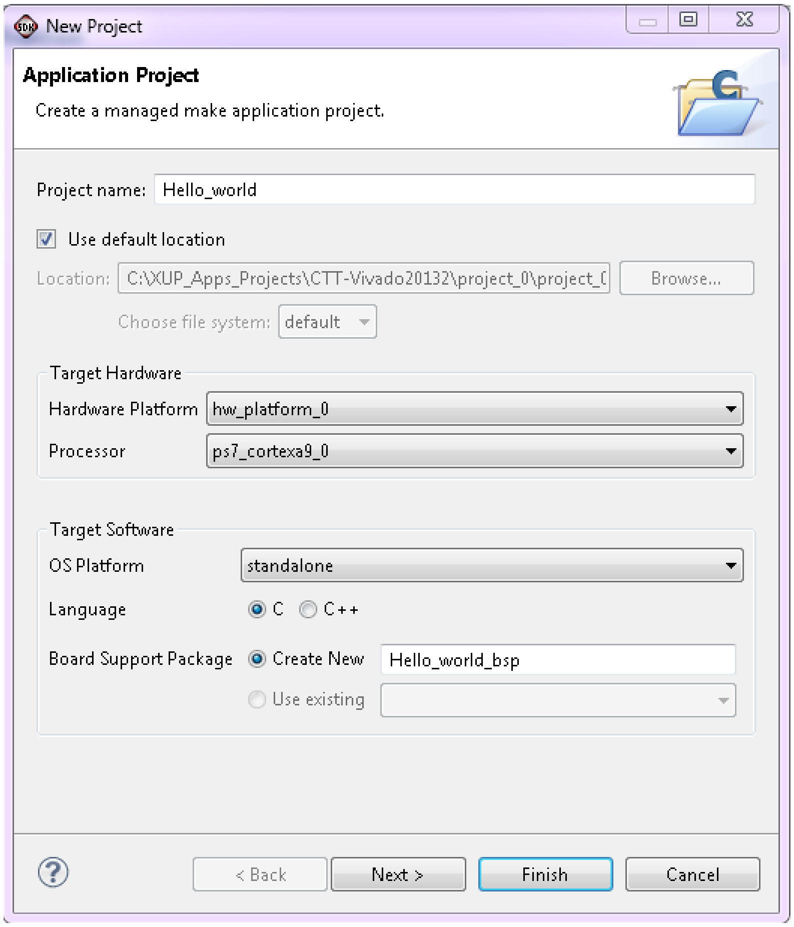
Timeout (sec): 5

Figure 2-13: Serial Terminal Settings

6. In SDK, select **File** > **New** > **Application Project**.

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

Wizard Screen	System Property	Setting or Command to USe
Application Project	Project name	Hello_world
	Use default location	Check this option
	Hardware Platform	system_hw_platform
	Processor	ps7_cortexa9_0
	OS platform	Standalone
	Language	C
	Board Support Package	**Create New** : Hello_world_bsp
Click **Next**		
Templates	Available Templates	Hello World



New Project

Application Project

Create a managed make application project.

Project name: Hello_world

☑ Use default location

Location: C:\XUP_Apps_Projects\CTT-Vivado20132\project_0\project_(

Choose file system: default

Target Hardware

Hardware Platform: hw_platform_0

Processor: ps7_cortexa9_0

Target Software

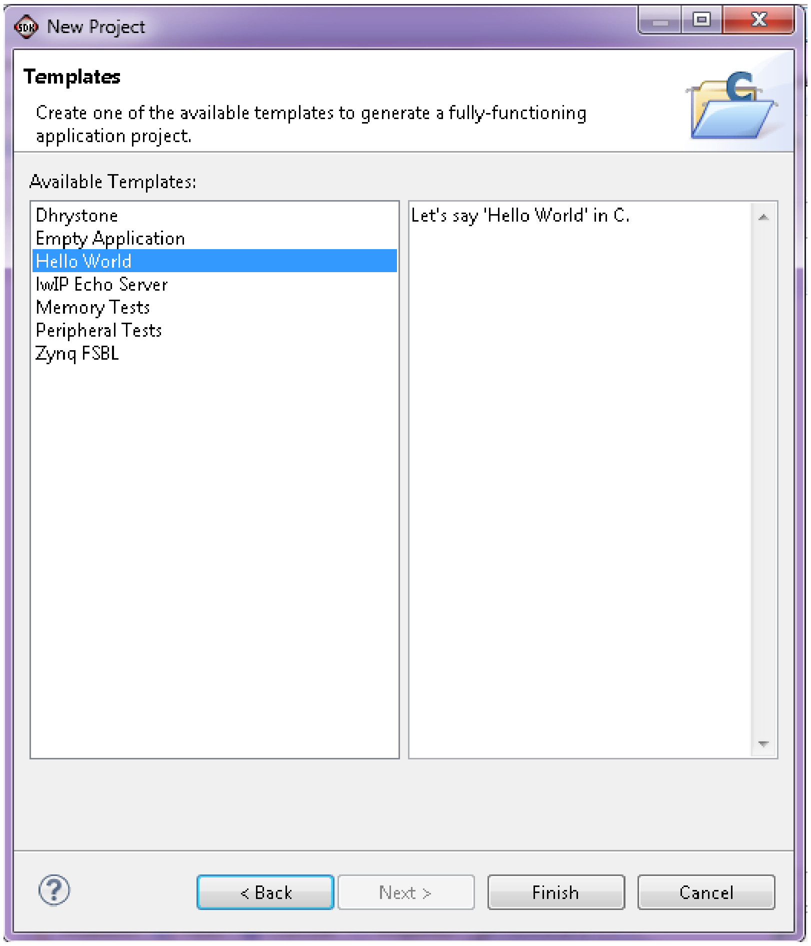
OS Platform: standalone

Language: ● C ○ C++

Board Support Package: ● Create New Hello_world_bsp

○ Use existing

Figure 2-14: New Application Project Wizard



Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

- Dhrystone
- Empty Application
- Hello World
- lwIP Echo Server
- Memory Tests
- Peripheral Tests
- Zynq FSBL

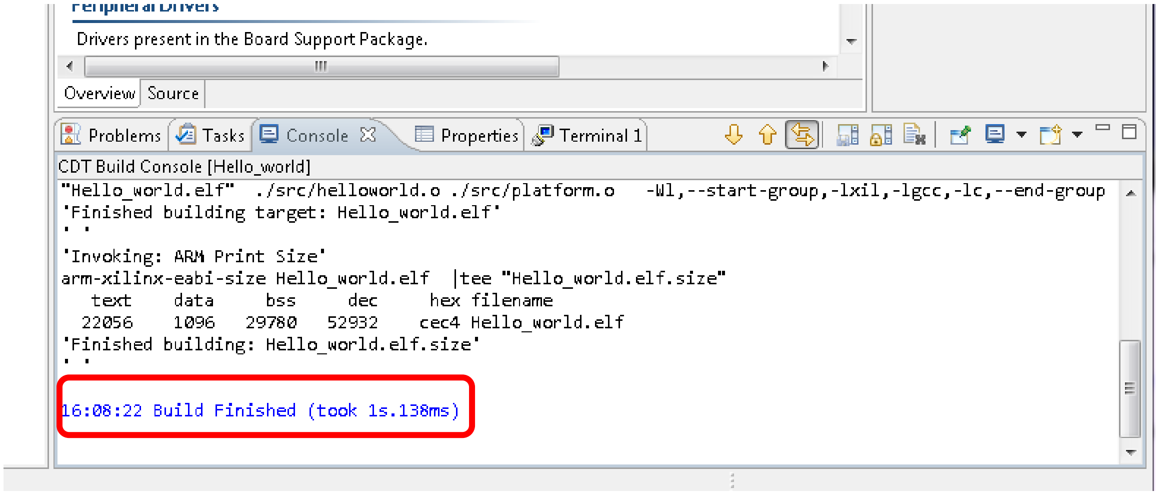
Let's say 'Hello World' in C.

Figure 2-15: Hello World from Available Templates

8. Close the new New Application Project Wizard by clicking **Finish**.

By doing so, the Hello_world application project and Hello_world_bsp BSP project get created under the project explorer. Both the Hello_world application, and its BSP are compiled automatically and the .elf file is generated. You can open the newly generated *helloworld.c* file to view the C code in the Hello_World application under the *src* folder.

9. Watch the messages in the Console window. When the project is successfully built, you will see **Finished building: Hello_world.elf.size**.



Peripheral Drivers
Drivers present in the Board Support Package.
Overview
Source
Problems
Tasks
Console
Properties
Terminal 1
CDT Build Console [Hello_world]

```
"Hello_world.elf"  ./src/helloworld.o ./src/platform.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: Hello_world.elf'
' '
'Invoking: ARM Print Size'
arm-xilinx-eabi-size Hello_world.elf  |tee "Hello_world.elf.size"
   text    data     bss     dec     hex filename
  22056    1096   29780   52932    cec4 Hello_world.elf
'Finished building: Hello_world.elf.size'
' '

16:08:22 Build Finished (took 1s.138ms)
```

Figure 2-16: Successful Build

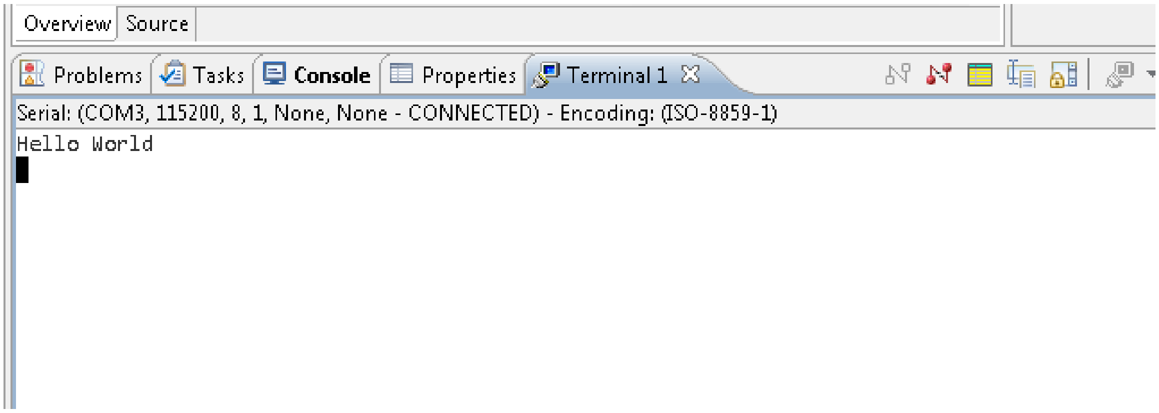
10. The application and its BSP are both compiled and the .elf file is generated.
11. Right-click **Hello_world** and select **Run as** > **Run Configurations**.
12. Right-click **Xilinx C/C++ ELF** and click **New**.
13. The new run configuration is created named **Hello_world Debug**.

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

14. Click the **Device Initialization** tab in the launch configurations and check the settings here.

Notice that there is a configuration path to the initialization TCL file (ps7_init.tcl). This is the file that was generated when you imported your design into SDK; it contains the initialization information for the processing system when using JTAG.

15. The STDIO Connection tab is available in the launch configurations settings. You can use this to have your STDIO connected to the console. Note that both STDIO and Terminal connections are not permitted to use the same COM port. We will not use this now because we have already launched a serial communication utility. There are more options in launch configurations but we will focus on them later.
16. Click **Run**. You may get a message that configuration was not done on the target FPGA, click **OK** to continue and ignore the message.
17. **"Hello World"** appears on the serial communication terminal.



```
Serial: (COM3, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
Hello World
```

Figure 2-17: "Hello World" on the Serial Terminal

18. Close the SDK.

Note: There was no bitstream download required for the above software application on the ZedBoard. The ARM Cortex-A9 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.

2.1.3 Additional Information

Board Support Package

The Board Support Package (BSP) is the support code for a given hardware platform or board that initializes the board at power up for software applications to execute on the platform. It can be specific to some operating systems with bootloader and device drivers.

Standalone OS

Standalone applications do not utilize an Operating System (OS). They are sometimes also referred to as bare-metal applications. Standalone applications have access to basic processor features such as caches, interrupts, exceptions as well as other simple features specific to the processor. These basic features include standard input/output, profiling, abort, and exit. It is a single threaded semi-hosted environment.

The application you ran in this chapter was created on top of a BSP built for the ZedBoard.

Chapter 3
Embedded System Design Using the Zynq Processing System and Programmable Logic

One of the unique features of using the Zynq AP SoC as an embedded design platform is using the available PL in addition to the Zynq PS for its ARM Cortex-A9 MPCore processing system.

In this chapter we will be creating a design with:

- PL-based AXI GPIO and AXI Timer with interrupt from the PL to PS section
- Zynq PS GPIO pin connected through the PL pins routed via the Extended MIO (EMIO) interface

The flow of this chapter is similar to that in **Chapter 2**. If you have skipped that chapter, you might want to look at it because we will refer to it many times in this chapter.

3.1 Adding soft IP in the PL to interface with the Zynq PS

Complex soft peripherals can be added into the PL to be tightly coupled with the Zynq PS. This section covers a simple example with AXI GPIO, AXI Timer with interrupt, PS section GPIO pin connected to a PL side pin via the EMIO interface, and ChipScope instantiation for proof of concept.

The block diagram for the system is as shown in Figure 3-1: System Design Overview.

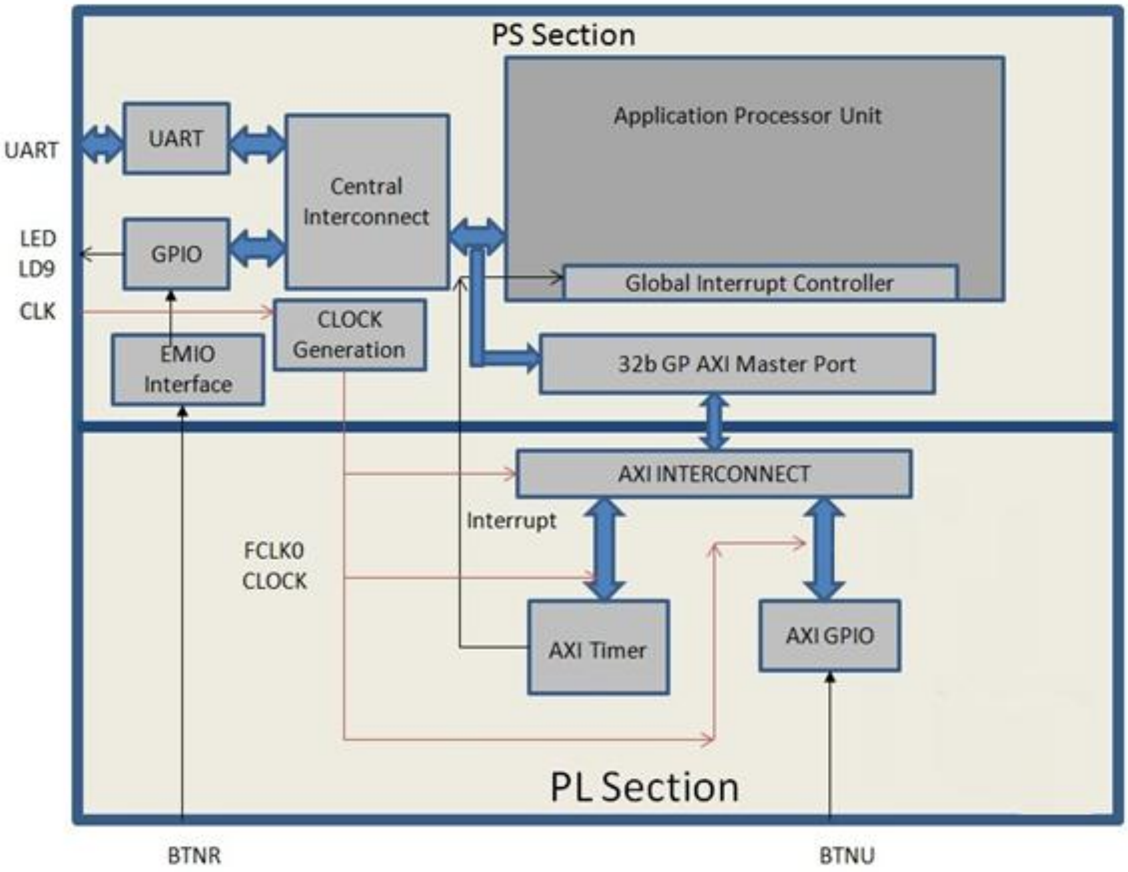


Figure 3-1: System Design Overview

This system covers the following connections:

- The PL-side AXI GPIO has only a 1 bit channel and it is connected to the push-button 'BTNU' on the ZedBoard
- The PS section GPIO also has a 1 bit interface routed to PL pin via the EMIO interface and connected to the push-button 'BTNR' on the board
- In the PS section another 1 bit GPIO is connected to the LED 'LD9' on board which is on the MIO port
- An AXI timer interrupt is connected from PL to PS section interrupt controller. The timer starts when the user presses any of the selected push buttons on board and toggles the LED 'LD9' on board

You will create a software application, taking input from the user to select the push button on the board and waits for the user to press that particular push button. When the push button is pressed, the timer starts automatically, turns OFF the LED and waits for the timer interrupt to happen. After the Timer Interrupts, the LED switches ON and execution starts again, and it waits for a valid selection from the user.

The sections of **Chapter 2** are also valid for this design flow. You'll use the system created in that chapter and pick up the procedure following **2.1.1 Take a Test Drive! Creating a New Embedded Project With a Zynq Processing System.**

3.1.1 Take a Test Drive! Check Functionality of IP instantiated in the PL

In this test drive exercise, you'll check the functionality of the AXI GPIO, AXI Timer with interrupt instantiated in PL and EMIO interface.

1. In the Vivado Sources pane, invoke the Block Diagram / IP Integrator by double-clicking **system_i-system(system.bd)**.

   This is the embedded source you created in section 2.1.1. Click on and highlight both the wire and FIXED_IO port and delete them from the block diagram (confirm the deletion when prompted).

2. Double click on the **ZYNQ7_Processing_System** block diagram to edit the block parameters.
3. Click on the **Presets** button near the top of the window, select the **ZedBoard Development Board Template** and click **OK**. This will reset the Zynq PS to default ZedBoard settings.
4. Click on the **Add IP** button. From the IP catalog, double-click **AXI GPIO** to add it. Notice the block has been automatically added to the Block Diagram.

   Drag the AXI GPIO block to a convenient location to the right of the Processing System block.

5. Click the **Add IP** button again, double click on the **AXI Interconnect** IP. Drag the IP to a convenient location to the right of the Processing System block.Again, click the **Add IP** button and double click on the **AXI Timer** IP to add it to the block diagram. Drag this IP to the right of the Processing System block.

   Now all the IP blocks have been added into the block diagram, the next step is to customize specific blocks and connect all of them together.

6. Double click on the **ZYNQ7 Processing System** block to edit the block parameters.
7. Click on the **Interrupts** page in the Page Navigator. Enable **Fabric Interrupts**. Expand the **PL-PS Interrupt Ports**, enable the **IRQ_F2P[15:0]** port.
8. Next, click on the **MIO Configuration** page. Expand **I/O peripherals** and de-select the following peripherals: **ENET 0** and **USB 0.** Expand **GPIO** and select **EMIO GPIO (Width)**, set the width to **1 bit** wide. Expand **Application Processor Unit** and de-select **Timer 0**.

9. Click on the **PS-PL** page and expand **GP Master AXI** Interface. Enable the **M AXI GP0 Interface**.
10. Click **OK**.
11. Double click on the AXI GPIO block to edit the block parameters. Click on the **IP Configuration** tab and select the **All Inputs** box to set all ports to being input only. Set the **GPIO Width** to **1**. Click **OK** to exit the window.

Making connections between the blocks is very straightforward. All you need to do is to place the mouse cursor at the origin port, click and drag a line to the destination port. The default mouse pointer becomes a pencil during this process and valid destination ports have green check marks indicating they can be connected to. A tool tip will also appear indicating the number of valid connections in the block diagram.

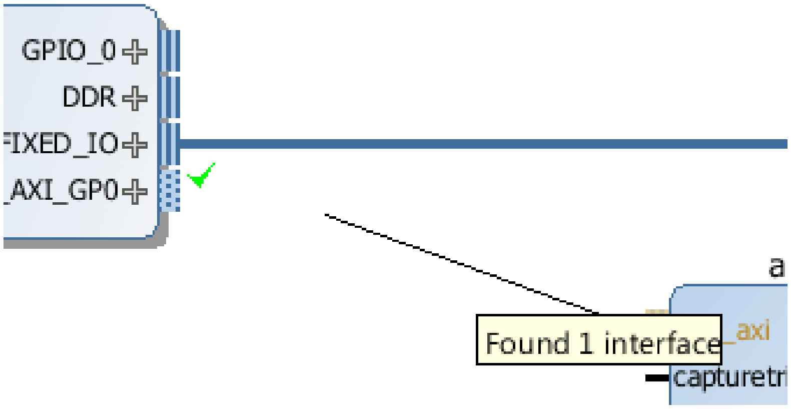


Figure 3-2: Connecting ports on two IP blocks

Note: This applies to both signals and busses. AXI busses will be connected between two valid ports and appear as a thick line. A thin line represents a one bit wide signal. A thick line is used to represent data busses for multiple bit wide ports. Similar signal types can be connected together.

12. Connect all the clock ports together. The clock ports in the block diagram will have **CLK** as part of the name.
13. Connect all the reset signals together. The reset ports in the block diagram will have **RESET** as part of the name.
14. Connect the rest of the signals as per the table below.

IP	Port	Connection
axi_interconnect_1	S00_AXI	processing_system7_1:: M_AXI_GP0
	M00_AXI	axi_gpio_1::S_AXI
	M01_AXI	axi_timer_1::S_AXI
axi_timer_1	Interrupt	processing_system7_1::IRQ_F2P[0:0]

15. Right click on a blank area in the block diagram and select **Create Port**. Specify the **Port Name** to be **BTNR**. Make sure the direction is set for **Input** only. Click **OK** to close the window.
16. Right click on a blank area in the block diagram and select **Create Port**. Specify the **Port Name** to be **BTNU**. Make sure the direction is set for **Input** only an that the width is **1-bit**. Click **OK** to close the window.
17. Connect **BTNR** to the Processing System block’s **GPIO_I[0:0]** port.
18. Connect **BTNU** to the AXI GPIO block’s **gpio_io_i[0:0] port**.
19. Right click on the Processing System block’s **MIO[53:0]** port and select **Make External**. Notice that a bi-directional port and thick line representing a data bus got added to the block diagram.

Your connections should be similar to Figure 3-3:.

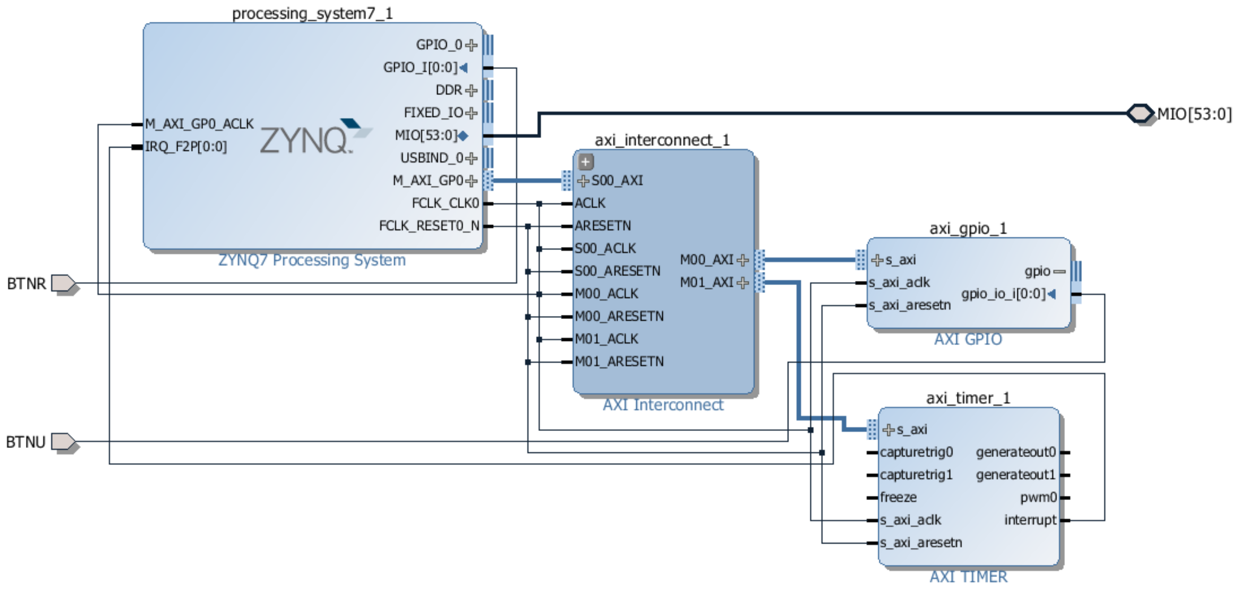
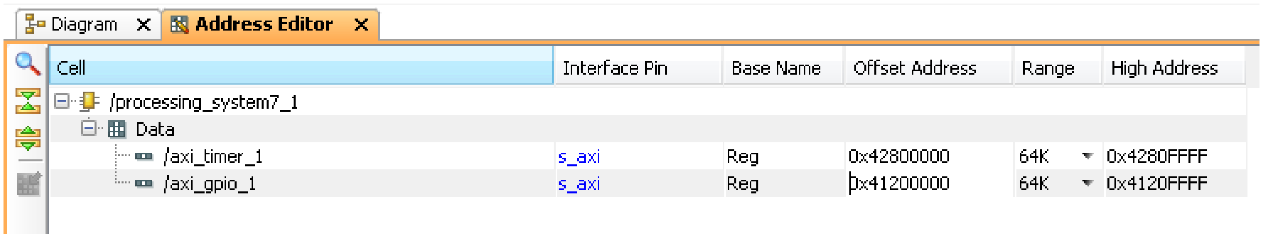


Figure 3-3: Completed Port Connections

20. Click on the Address Editor tab **Address Editor** . Make sure the **axi_timer_1** and **axi_gpio_1** have Offset Addresses assigned to them. If not, click on the **Automatically Assign Addresses** button to populate the empty fields. You should have axi_timer_1 assigned **0x42800000** and axi_gpio_1 assigned **0x41200000**.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
/processing_system7_1					
Data					
/axi_timer_1	s_axi	Reg	0x42800000	64K	0x4280FFFF
/axi_gpio_1	s_axi	Reg	0x41200000	64K	0x4120FFFF

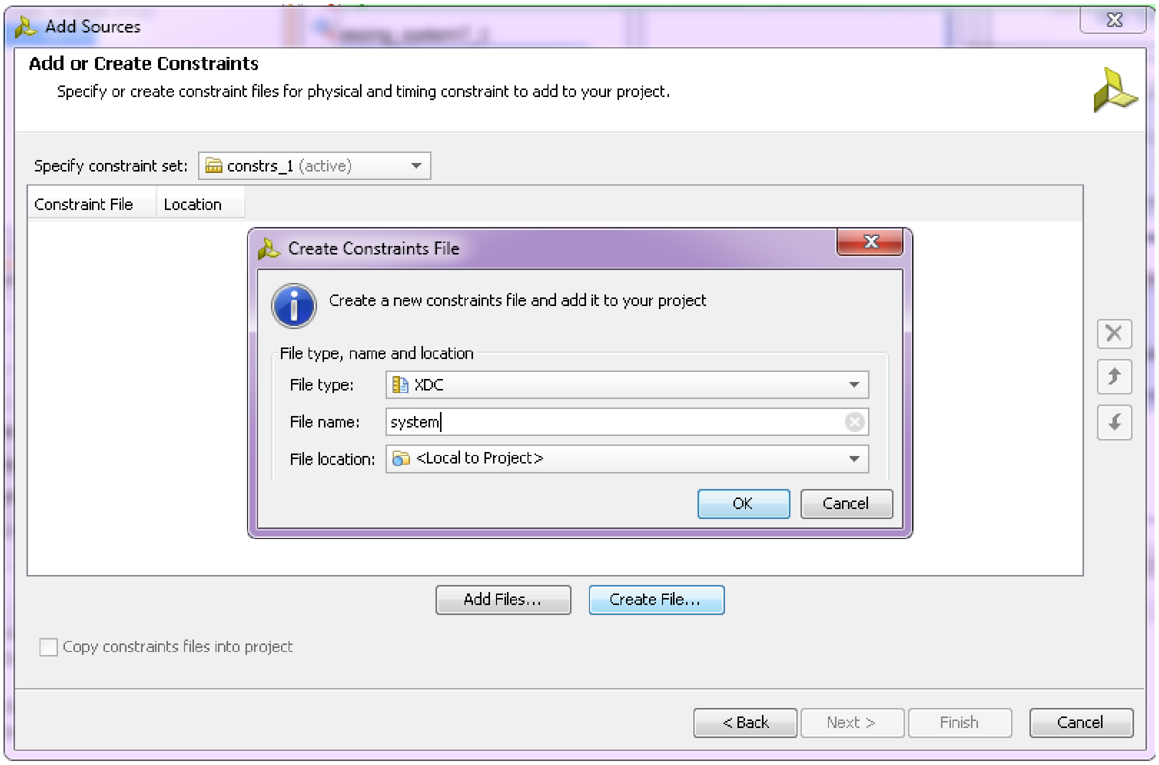
Figure 3-4: Assigned peripheral memory addresses

21. Right click on the **system_i** block diagram (system.bd) in the Sources pane and select **Create HDL Wrapper**. Make sure you select **Copy and Overwrite** the existing top level HDL file when the dialog box appears.

22. Click the **Add Sources** button in the Flow Navigator. Select **Add or Create Constraints**. Click **Next**.

23. The Add or Create constraints window will appear. Click **Create File** and the file creation sub-window opens. Make sure the file extension is **.XDC** and

name the constraints file **system**. Click **OK** to exit the sub-window and click **Finish** to complete the process.



Add Sources

Add or Create Constraints

Specify or create constraint files for physical and timing constraint to add to your project.

Specify constraint set: constrs_1 (active)

Constraint File	Location

Create Constraints File

Create a new constraints file and add it to your project

File type, name and location

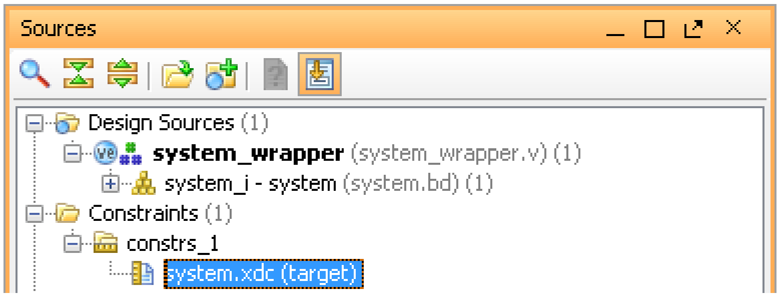
File type: XDC

File name: system

File location: <Local to Project>

Figure 3-5: Adding a constraints file

Notice the Sources pane now has **system.xdc** added under the constraints category.



Sources

```
Design Sources (1)
    system_wrapper (system_wrapper.v) (1)
        system_i - system (system.bd) (1)
Constraints (1)
    constrs_1
        system.xdc (target)
```

Figure 3-6: Constraints file added into project

24. Double click on **system.xdc** and add the following location constraints to the empty file:

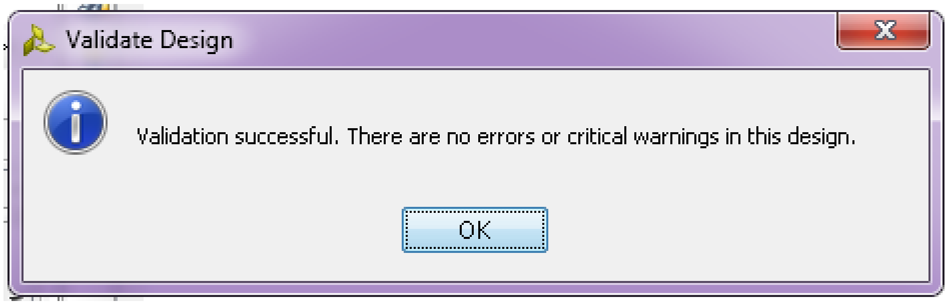
```
set_property PACKAGE_PIN T18 [get_ports BTNU]
set_property PACKAGE_PIN R18 [get_ports BTNR]

set_property IOSTANDARD LVCMOS25 [get_ports BTNR]
set_property IOSTANDARD LVCMOS25 [get_ports BTNU]
```

The following settings are made:

- The PACKAGE PIN T18 constraint connects the AXI GPIO pin to the T18 pin of the PL section and physically connects it to the BTNU push button on the board.
- The PACKAGE PIN R18 constraint connects the PS section GPIO to the R18 pin of the PL section and physically connects it to the BTNR push button on the board.
- The IOSTANDARD LVCMOS25 constraint sets both pins to the LVCMOS 2.5V I/O standard.

25. **Save** the edited constraints file.

26. Run **Tools** > **Validate Design**. The tool will report that design validation was successful. Click **OK** to close the dialogue box.



Validate Design

Validation successful. There are no errors or critical warnings in this design.

Figure 3-7: Design validation successful

27. In the Program and Debug list in the Flow Navigator pane, click **Generate Bitstream**. This step is necessary since PL resources are added into the project and will need to be implemented in hardware before launching the SDK.

    A dialog box will appear to warn you that synthesis is not run on the updated files, click **Yes** to run synthesis. Generating the bitstream may invoke the entire implementation process after synthesis, click **Yes** to run implementation as well when prompted. Synthesis may take a while to complete.

    You may get dialogue boxes notifying you of critical warnings during implementation, click **OK** to ignore them. Do not stop the runs.

28. After the Bitstream generation completes select **Open Implemented Design** in the dialog box and click **OK.** This allows you to get a graphical overview of the PL resource usage and routing.
29. Export the hardware (make sure that you enable the “Include Bitstream” option) and Launch SDK as described in **Chapter 2**. For this design, since there is a bitstream generated for the PL, the bitstream will also be exported to SDK.

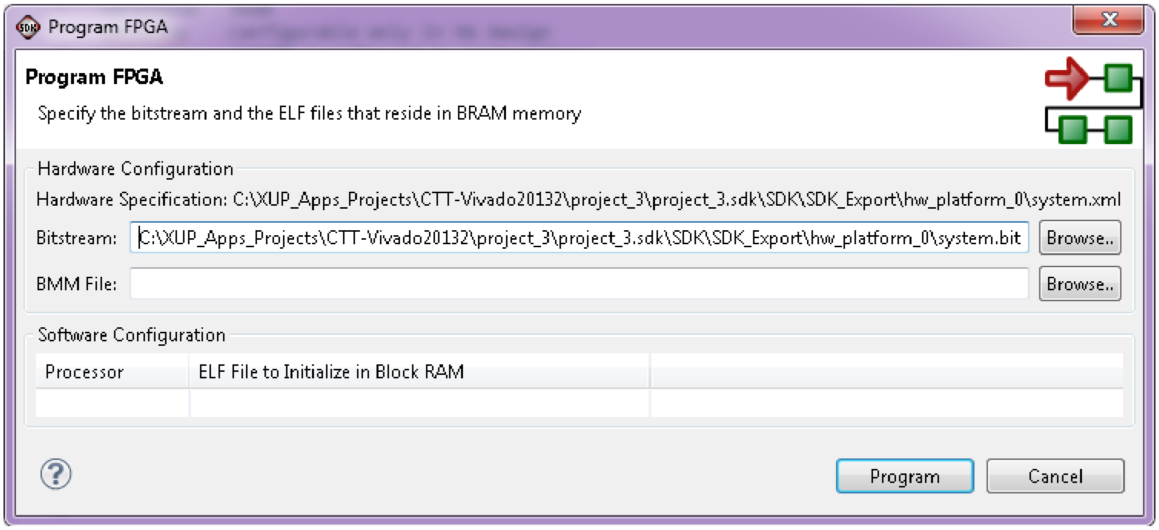
A dialog box will appear indicacting that there exisits an exported module, click YES to overwrite.

3.1.2 Take a Test Drive! Working with SDK

SDK launches with the "Hello World" project you created with the Standalone PS in **Chapter 2**.

Note: You should use an external terminal emulator program (PuTTY or Tera Term) in place of the SDK Terminal utility due to a compatibility issue between the ZedBoard and the SDK terminal. Please make sure that the terminal emulator program uses the recommended connection settings from Figure 2-13.

1. Select **Project > Clean Project** to clean and build the project again.
2. Open the **helloworld.c** file and modify the application software code. Refer to **Appendix A, Application Software** for the application software details.
3. Connect and power-on the board.
4. Open the serial communication utility with baud rate set to **115200**.
5. Because you have a bitstream for the PL, you must download the bitstream. To do this, select **Xilinx Tools > Program FPGA**. The Program FPGA dialog box, shown below, opens. It displays the bitstream exported from Vivado. Please make sure the bitstream path points to your current project.



Program FPGA

Program FPGA

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

Hardware Configuration

Hardware Specification: C:\XUP_Apps_Projects\CTT-Vivado20132\project_3\project_3.sdk\SDK\SDK_Export\hw_platform_0\system.xml

Bitstream: C:\XUP_Apps_Projects\CTT-Vivado20132\project_3\project_3.sdk\SDK\SDK_Export\hw_platform_0\system.bit

BMM File:

Software Configuration

Processor	ELF File to Initialize in Block RAM	

Figure 3-8: Program FPGA Dialog Box

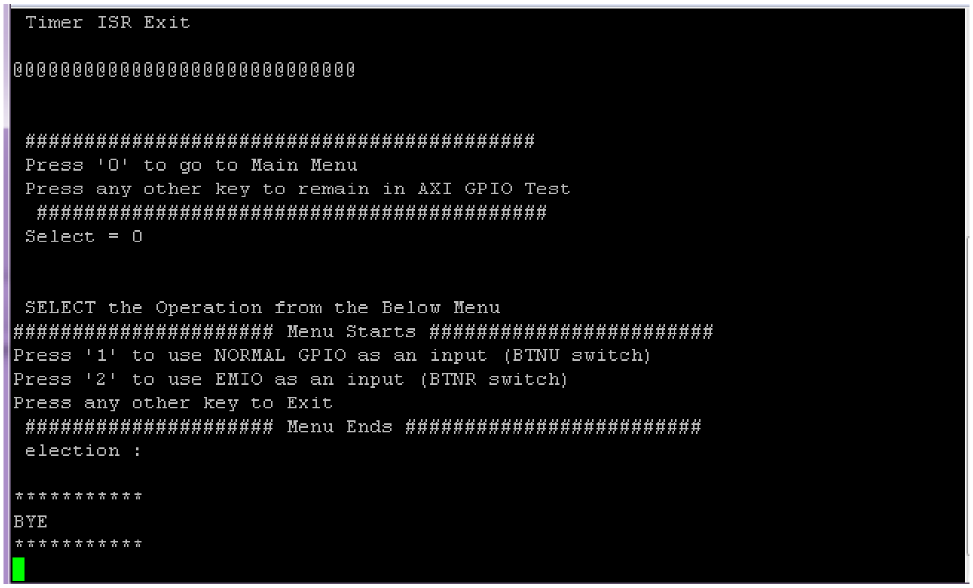
6. Click **Program** to configure the PL with the bitstream. The Blue DONE LED (LD12) will light up.

7. Run the application similar to the steps in **Take a Test Drive! Running the "Hello World" Application.**

8. In the system, the AXI GPIO pin is connected to push button BTNU on the board, and the PS section GPIO pin is connected to push button BTNR on the board via an EMIO interface.

9. Follow the instructions shown on the serial terminal to run the application.

10. When the run is complete, do not close the SDK.



```
 Timer ISR Exit

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


 ###########################################
 Press '0' to go to Main Menu
 Press any other key to remain in AXI GPIO Test
  ###########################################
 Select = 0


 SELECT the Operation from the Below Menu
###################### Menu Starts ########################
Press '1' to use NORMAL GPIO as an input (BTNU switch)
Press '2' to use EMIO as an input (BTNR switch)
Press any other key to Exit
 ##################### Menu Ends #########################
 election :

***********
BYE
***********
```

Figure 3-9: Terminal showing application output

Chapter 4 Debugging with SDK and Vivado Logic Analyzer

This chapter describes two types of debug possibilities with the design flow you've already been working with. The first option is debugging with software using SDK. The second option is hardware debug supported by the Vivado Logic Analyzer.

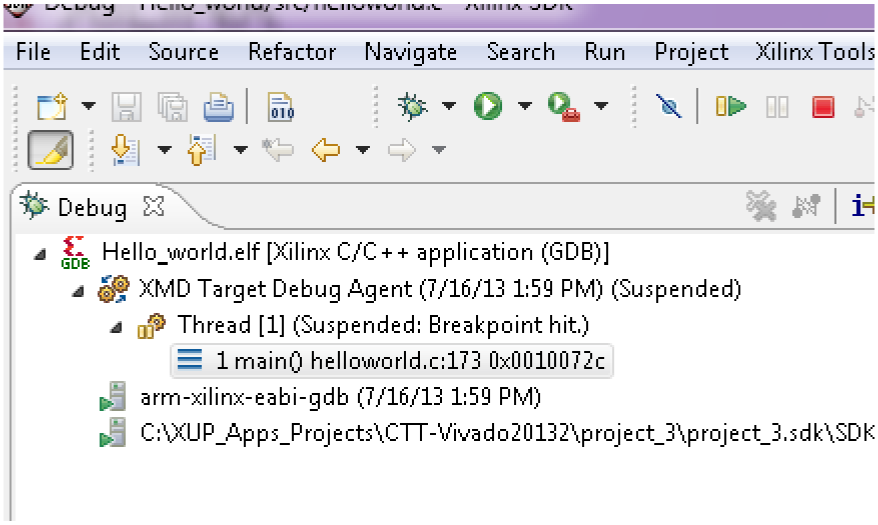
4.1 Take a Test Drive! Debugging with Software, Using SDK

First you will try debugging with software using SDK. This step assumes the PL is still configured from the previous chapter.

1. In the C/C++ Perspective, right-click on the **Hello_world** Project and select **Debug As > Debug Configurations**. Check that settings are correct for your debug operation.
2. Click **Debug**.

A dialog box appears to notify you that this kind of launch is configured to open the Debug perspective when it suspends.

3. Click **Yes**. The Debug Perspective opens, executing the ps7_init and then suspending at the main() entry point.



Hello_world.elf [Xilinx C/C++ application (GDB)]
XMD Target Debug Agent (7/16/13 1:59 PM) (Suspended)
Thread [1] (Suspended: Breakpoint hit.)
1 main() helloworld.c:173 0x0010072c
arm-xilinx-eabi-gdb (7/16/13 1:59 PM)
C:\XUP_Apps_Projects\CTT-Vivado20132\project_3\project_3.sdk\SDK

Figure 4-1: Debug Perspective Suspended

Note: The address shown on this page might be different from the address shown on your system.

The processor is currently sitting at the beginning of main() with program execution suspended at address 0x0010072c. You can confirm this information with the Disassembly view, which shows the assembly-level program execution also suspended at 0x0010072c.

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

The helloworld.c window also shows execution suspended at the first executable line of C code.

4. Select the Registers view to confirm that the program counter, pc register, contains **0x0010072c**.

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

5. Double-click in the margin of the helloworld.c window next to the line of code that reads init_platform (). This sets a breakpoint at init_platform (). To confirm the breakpoint, review the Breakpoints window.

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

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

Program execution stops at the line of code that includes init_platform (). The Disassembly and Debug windows both show program execution stopped at **0x00100754**.

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

Program execution suspends at location **0x00100d68**. The call stack is now two levels deep.

8. Select **Run** > **Resume** again to run the program to conclusion by hitting any key to end the application in the terminal window.

When the program completes running, 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.

9. Re-run your code several times. Experiment with single-stepping, examining memory, changing breakpoints, modifying code, and adding print statements. Try adding and moving views.
10. After you are satisfied with the design behavior, power cycle the ZedBoard.

4.2 Take a Test Drive! Debugging Hardware Using Vivado Logic Analyzer tool

This section is only applicable if you have the Logic Analyzer licensed feature enabled via the ZedBoard license voucher code. If you are using the WebPACK install, please skip this section.

Now you will debug the hardware using the Vivado Logic Analyzer tool using the same application you created in **3.1.2 Take a Test Drive! Working with SDK**.

The signals can be probed via the Vivado Logic Analyzer during a Hardware Session that is opened in Vivado. Remember to keep the SDK open while going back to Vivado to specify signals to probe and while the Hardware Session is open.

Go back to the Vivado window (it should still be running). Recall that Synthesis and Implementation have ran successfully to generate the bitstream.

1. Under Synthesis, click on **Open Synthesized Design**. You will be prompted to close the implemented design, click **Yes**.

   The synthesized design schematics will appear, notice the Sources pane is now showing the synthesized netlist. We will use the synthesized netlist to define signals to debug. Expand the different branches and explore what is available in the synthesized netlist pane.

2. Expand the **system_i > Nets** branch. This shows you a list of nets in the top most level of the design.

3. Find **axi_interconnect_1_m01_axi_ARVALID**, right click and select **Mark Debug**. When prompted, confirm by clicking **OK** to debug the net and create the MARK_DEBUG constraints in the XDC constraints file.

4. Repeat the above steps for the following signals:

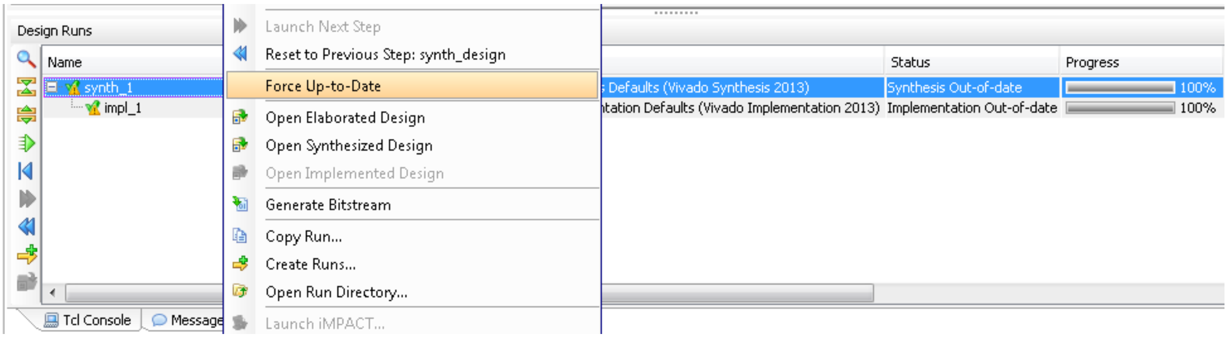
   - **axi_interconnect_1_m01_axi_ARREADY**
   - **axi_interconnect_1_m01_axi_AWADDR (5-bits wide bus)**
   - **axi_interconnect_1_m01_axi_WDATA (32-bits wide bus)**

5. Click the **Save** button at the top tool bar.

   If you already have system.xdc open, notice a yellow notification bar at the top of the text editor asking reload since the file has changed. Reload the file. If system.xdc is not open, click on the Sources tab, expand **Constraints > constrs_1 > system.xdc** to open the file in Vivado. Notice Vivado added all the nets that were marked for debug earlier.

6. Click on the **Design Runs** tab. Notice Vivado flagged the synthesis run as being out of date. This is due to the constraints file being modified. Right click on the **synth_1** row and select **Force-Up-to-Date**.

   This forces Vivado to keep the synthesized results up to date and is an important step in adding signals to debug.

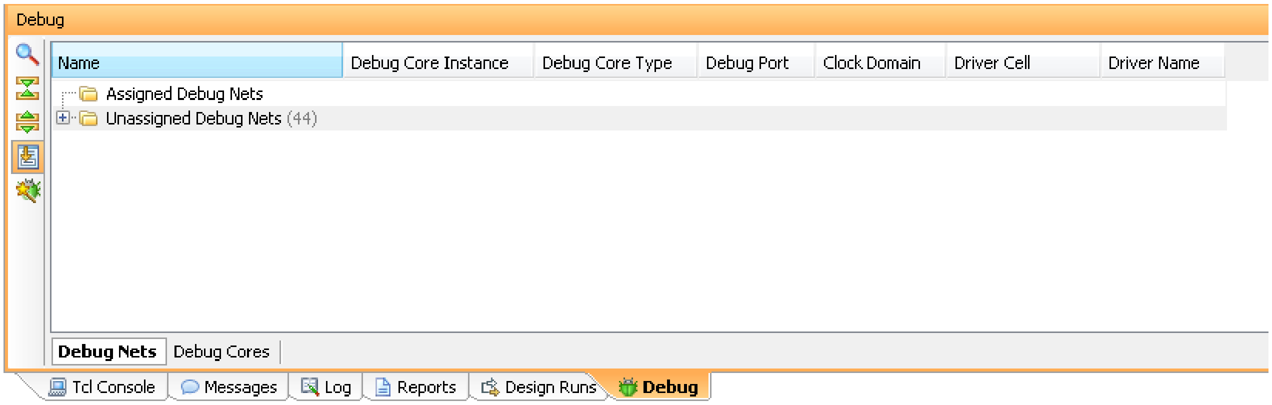


Name		Status	Progress
synth_1	Defaults (Vivado Synthesis 2013)	Synthesis Out-of-date	100%
impl_1	ntation Defaults (Vivado Implementation 2013)	Implementation Out-of-date	100%

Figure 4-2: Forcing synthesis up-to-date

7. At the Vivado tool bar, click on the drop down menu to select the **Debug Perspective**.

   Click on the **Debug** tab at the bottom of the Vivado GUI. Notice there are two branches of nets defined, **Assigned Debug Nets** and **Unassigned Debug Nets**. Expand Assigned Debug Nets and notice it is currently empty. This is because previously marked debug nets have not been assigned to it.



Name	Debug Core Instance	Debug Core Type	Debug Port	Clock Domain	Driver Cell	Driver Name
Assigned Debug Nets						
Unassigned Debug Nets (44)						

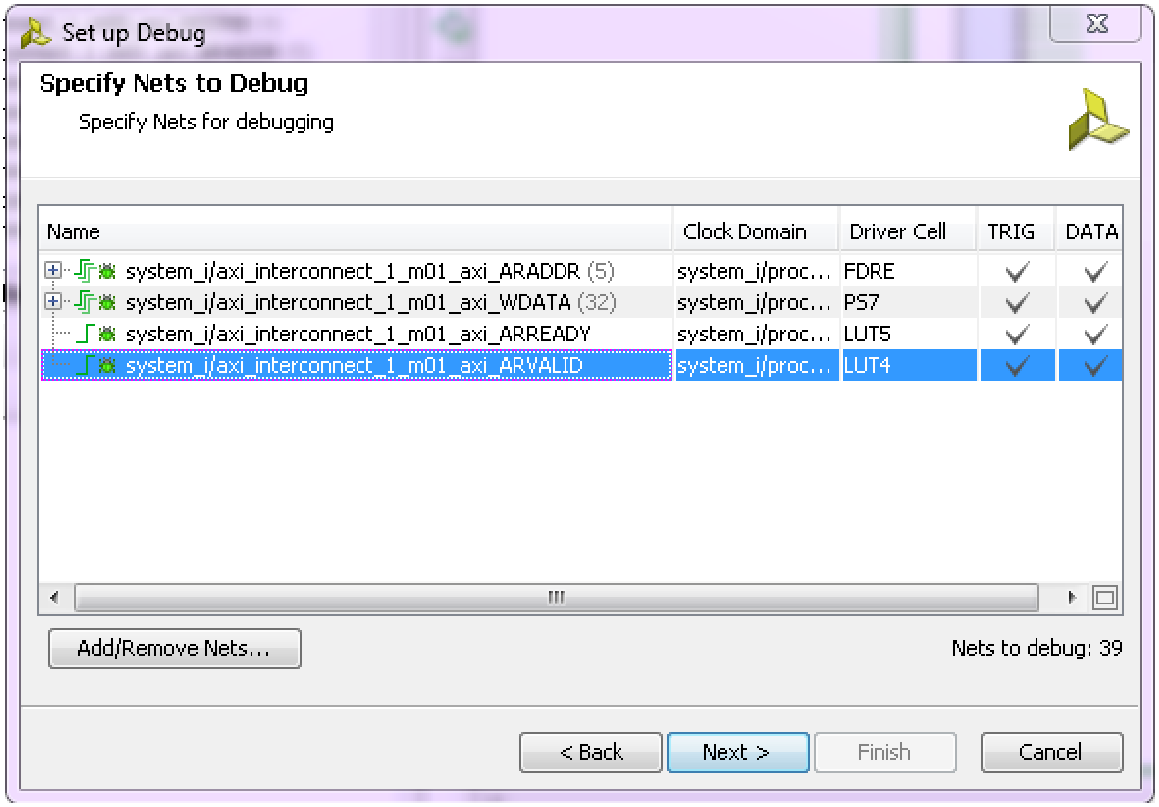
Figure 4-3: Debug nets in the Debug Perspective

8. Right click on the **Unassigned Debug Nets branch**. Select **Set up Debug**.

   The Set up Debug Wizard opens.

9. Click **Next** to exit the welcome screen. The Specify Nets to Debug screen shows the nets to assign to debug, click **Next**. Review the Set up Debug Summary. Notice the wizard will add a debug core. Click **Finish**.

10. Now expand the **Assign Debug Nets** branch in the Debug tab, notice the signals previously specified for debug appearing in there.



Set up Debug

Specify Nets to Debug

Specify Nets for debugging

Name	Clock Domain	Driver Cell	TRIG	DATA
system_i/axi_interconnect_1_m01_axi_ARADDR (5)	system_i/proc...	FDRE	✓	✓
system_i/axi_interconnect_1_m01_axi_WDATA (32)	system_i/proc...	PS7	✓	✓
system_i/axi_interconnect_1_m01_axi_ARREADY	system_i/proc...	LUT5	✓	✓
system_i/axi_interconnect_1_m01_axi_ARVALID	system_i/proc...	LUT4	✓	✓

Nets to debug: 39

Figure 4-4: Set up Debug wizard specifying nets to debug

11. Run implementation and re-generate the bitstream.

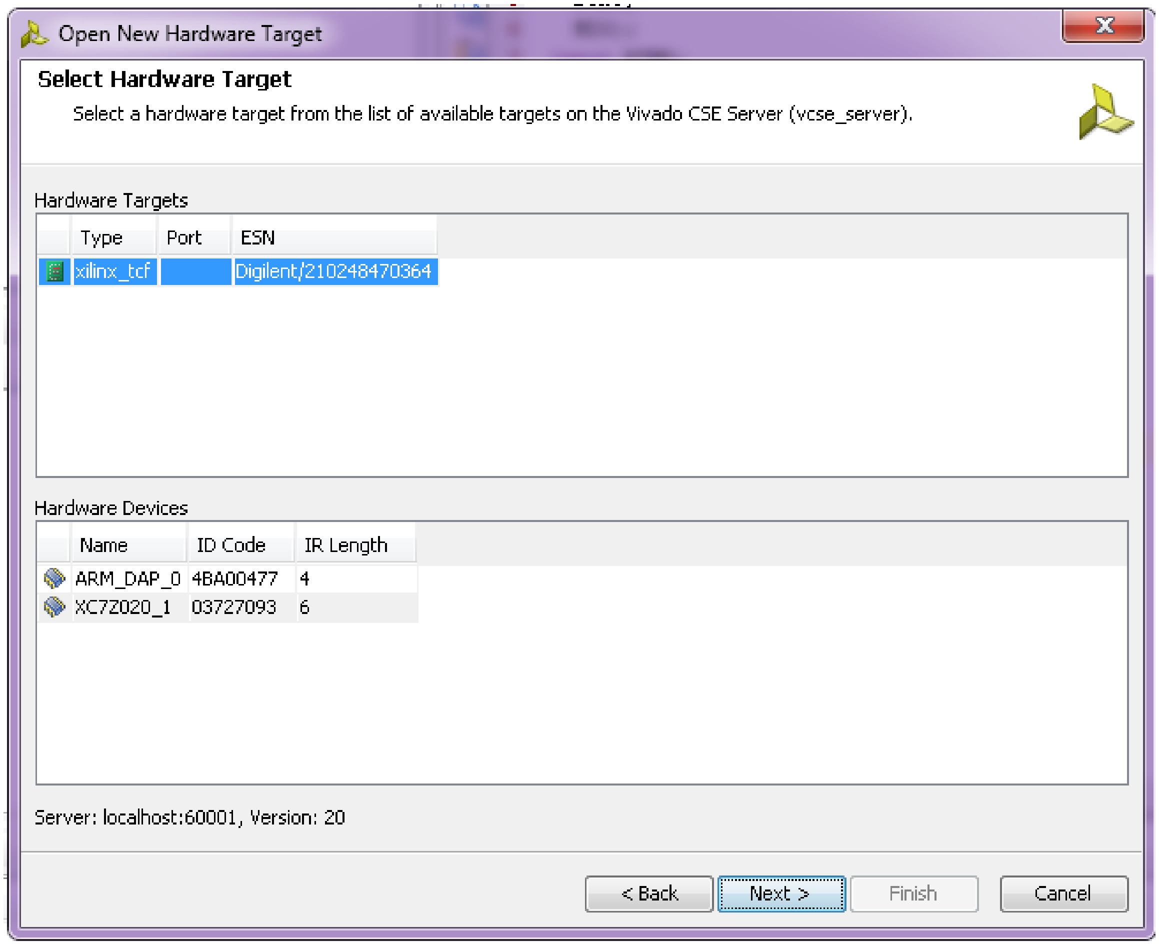
12. Once the bitstream has been generated, click on Open Implemented Design in order to port the bitstream to SDK later on.

13. Click on **Open Hardware Session** under Program and Debug.

Vivado will prompt you to either Open recent target or Open a new hardware target.

No hardware target is open. Open recent target Open a new hardware target

Click on **Open a new hardware target**. The Open Hardware Target wizard will open.



Select Hardware Target

Select a hardware target from the list of available targets on the Vivado CSE Server (vcse_server).

Hardware Targets

Type	Port	ESN
xilinx_tcf		Digilent/210248470364

Hardware Devices

Name	ID Code	IR Length
ARM_DAP_0	4BA00477	4
XC7Z020_1	03727093	6

Server: localhost:60001, Version: 20

Figure 4-5: Trigger Condition Dialog Box

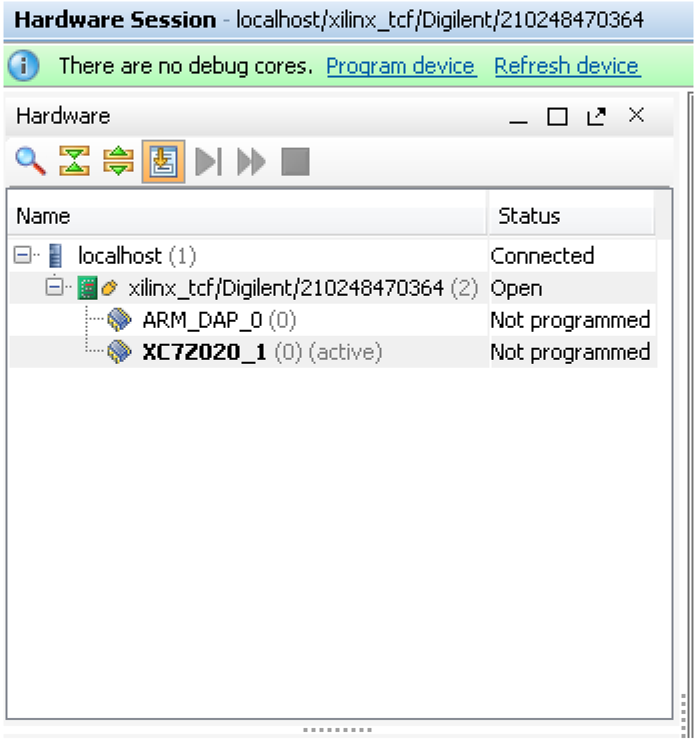
14. Click **Next** to exit the welcome screen. Go with the default Vivado CSE Server Name options, click **Next** again.

15. Click **Next** again at the Select Hardware Target screen.

16. Click **Next** again to go with the default frequency of 1.50 MHz in the Set Hardware Target Properties screen.

17. Review the summary and click **Finish**.

    The Hardware pane shows the XC7Z020_1 device is Not Programmed. Also notice the green information bar providing the option to Program device. Click on **Program device** and select the target as XC7Z020_1.



Hardware Session - localhost/xilinx_tcf/Digilent/210248470364

There are no debug cores. Program device Refresh device

Hardware

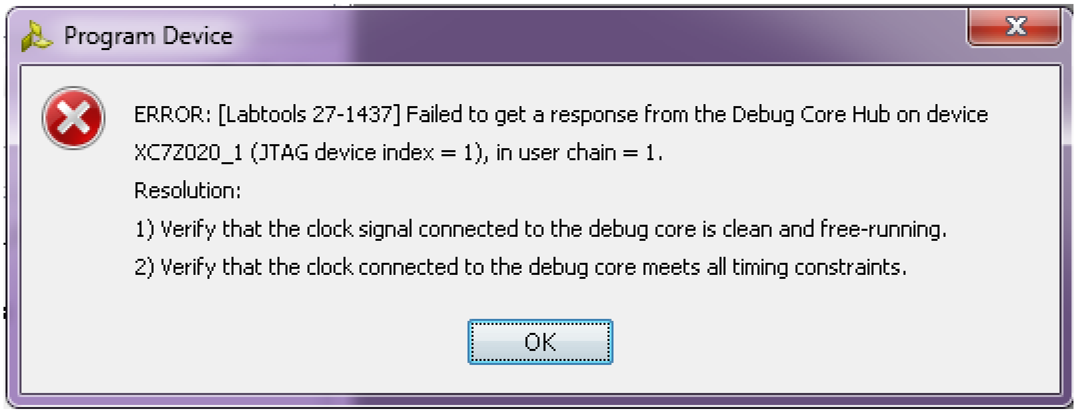
Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210248470364 (2)	Open
ARM_DAP_0 (0)	Not programmed
XC7Z020_1 (0) (active)	Not programmed

Figure 4-6: Identified devices during Hardware Session

18. Go with the default path to the system_wrapper.bit file in the Program Device window and click **Program**.

    After the device is programmed, notice the blue DONE LED lighting up on the ZedBoard. However, also notice that Vivado is complaining that the clock cannot be found in the debug core. This is not a problem since the software design is not yet running. The next step is to run the design in SDK (which should not have been closed since the previous exercise.)

    Click **OK** to continue.



Program Device

ERROR: [Labtools 27-1437] Failed to get a response from the Debug Core Hub on device XC7Z020_1 (JTAG device index = 1), in user chain = 1.

Resolution:

1) Verify that the clock signal connected to the debug core is clean and free-running.

2) Verify that the clock connected to the debug core meets all timing constraints.

Figure 4-7: Warning about the debug core not receiving a clock signal

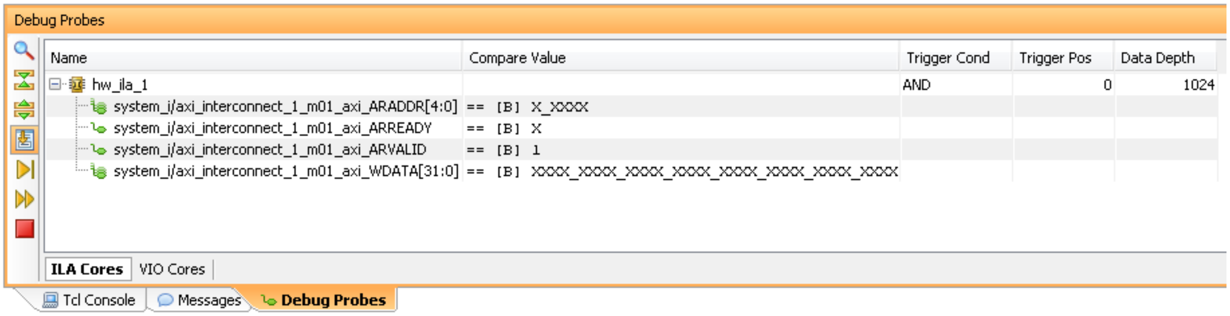
19. If the serial terminal from the previous exercise has been closed, reopen a new session.

20. Go to the SDK window, run the Hello_world design again. Right click on the Hello_world application project, click **Run As** > **Run Configurations**. Double click on **Hello_world.elf** and click **Run**.

21. The terminal screen should show the application running. Press **'1'** as prompted but do not push BTNU on the ZedBoard.

22. Go back to Vivado which should be showing the Hardware Session screen. Click **Refresh Device** in the green information bar at the top. Refresh device Again selecting the **XC7Z020_1** target device.

    Notice in the Debug Probes tab at the bottom of the GUI, you now see the signals that were marked for debug earlier.



Debug Probes

Name	Compare Value	Trigger Cond	Trigger Pos	Data Depth
hw_ila_1		AND	0	1024
system_i/axi_interconnect_1_m01_axi_ARADDR[4:0]	== [B] X_XXXX			
system_i/axi_interconnect_1_m01_axi_ARREADY	== [B] X			
system_i/axi_interconnect_1_m01_axi_ARVALID	== [B] 1			
system_i/axi_interconnect_1_m01_axi_WDATA[31:0]	== [B] XXXX_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX			

Figure 4-8: Debug Probes available for the waveform view

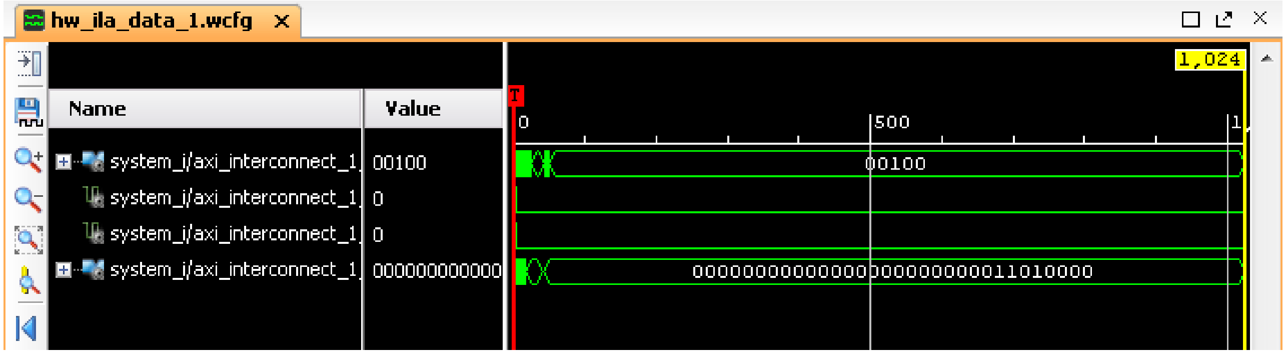
23. Right click on **hw_ila_1** in the Debug Probes tab and **Add All Probes to Waveform**. This will add all the signals to the waveform window (which you cannot yet see.)

24. Set the **Compare Value** (trigger value) of **axi_interconnect_1_m01_axi_ARVALID** to **1**. This means the waveform will only be captured when ARVALID is triggered. Click on the Run Trigger button. ▷|

    Notice the Hardware pane now shows the hw_ila_1 core to be Armed with an animated sphere. ○ Armed

25. Now press BTNU on the ZedBoard.

26. The waveforms will be captured in the hw_ila_data_1.wcfg window.



Name	Value
system_i/axi_interconnect_1	00100
system_i/axi_interconnect_1	0
system_i/axi_interconnect_1	0
system_i/axi_interconnect_1	000000000000

Figure 4-9: Captured waveforms from the triggered run

27. Zoom in to the waveforms. Scroll the window to the extreme left to see the signals toggling. Expand the multi-bit signals to see individual signals toggling.

    To see more of the waveforms, click on the float button and maximize the floating waveform window.Play around with the design and set the trigger conditions to understand the triggering and debug capabilities better. Once you are satisfied, exit the application program, exit the SDK and exit the Hardware Session to complete this exercise.

Chapter 5 Booting Linux and Application Debugging Using SDK

This chapter describes the steps to boot the Linux OS on the Zynq-7000 AP SoC ZedBoard. It covers programming of the following non-volatile memories with the Linux 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 SDK remote debugging feature to debug Linux applications running on the ZedBoard. The SDK tool software runs on the Windows host machine. For application debugging, SDK establishes an Ethernet connection to the target board that is already running the Linux OS.

5.1 Requirements

The target hardware platform is the ZedBoard. The host platform is a Windows machine running Vivado.

Note: The U-Boot universal bootloader is required for the tutorials in this chapter. This is included in the precompiled images supplied with this document.

The zipfile includes these files (in addition to others used in other sections):

- BOOT.bin: Binary image containing the FSBL and U-Boot images produced by bootgen
- bootimage.bif: The file to control bootgen during the creation of BOOT.BIN
- devicetree.dtb: Device tree binary large object (blob) used by Linux, loaded into memory by U-Boot. Note, the devicetree.dtb will not work if the hardware design has different peripherals specified
- ramdisk8M.image.gz: Ramdisk image used by Linux, loaded into memory by U-Boot
- README.txt: Description of the release
- u-boot.elf: U-Boot file used to create the BOOT.BIN image
- zImage: Linux kernel image, loaded into memory by U-Boot
- zynq_fsbl_0.elf: FSBL image used to create BOOT.BIN image

- hello_world_linux.c: sample ‘hello world’ c file used
- stub.tcl: script file specific to the ZedBoard rev C

5.2 Booting Linux on a ZedBoard

This section covers the flow for booting Linux on the target board using the provided precompiled images.

5.2.1 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 like 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 All Programmable SoC Technical Reference Manual (UG585)** for more detail.

http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

The boot process is initiated by the ARM Cortex-A9 CPU0 in the 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 memory 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.

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 the 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.

5.2.2 Booting Linux from JTAG

The flowchart illustrates the process used to boot Linux on the ZedBoard.

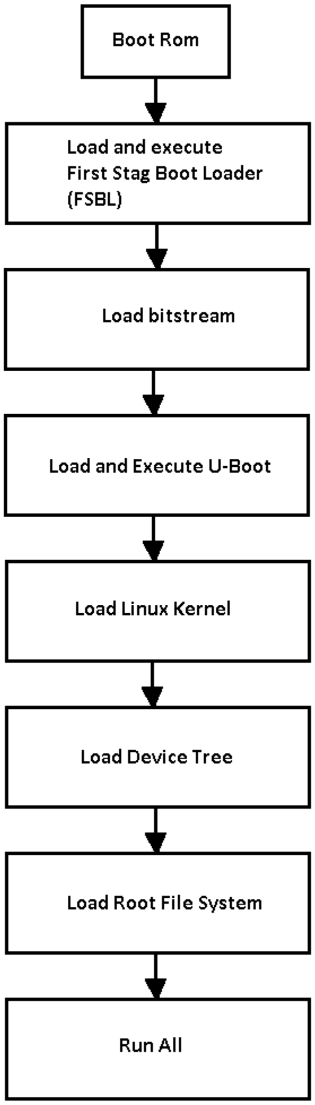


Figure 5-1: Linux Boot Process on the ZedBoard

5.2.3 Take a Test Drive! Booting Linux in JTAG Mode

1. Check the board connections and settings:

   a. Ensure that the jumpers JP7-JP11 are set as shown in Figure 5-2: Jumper Settings to boot in JTAG mode.

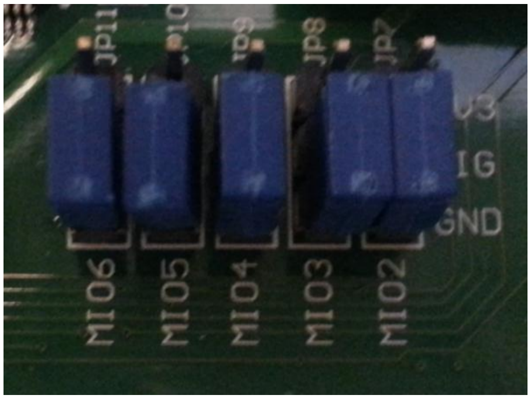


Figure 5-2: Jumper Settings to boot in JTAG mode

   b. Connect an Ethernet cable from the Zynq board to your Windows host machine.

   c. Connect the power cable to the board.

   d. Connect the USB programming micro cable between the Windows Host machine and Prog USB port on the Target board.

   e. Connect a USB micro cable to the USB UART connector on the ZedBoard with the Windows Host machine. This is used for USB to serial transfer.

2. Power on the ZedBoard.

3. Launch the SDK standalone and open the same workspace that you used in Chapter 2 and Chapter 3. The workspace directory is found at the following location: **<project_path>/<project_name>.sdk/SDK/SDK_Export/**

4. If the serial terminal is not open, connect the serial communication utility with the baud rate set to **115200**.

5. Open the XMD tool by selecting **Xilinx Tools > XMD console**

6. At the XMD prompt, do following:

   a. Type **connect arm hw** to connect with the PS section CPU.

   b. Type **source <project_path>/project_name>.sdk/SDK/SDK_Export/hw/ps7_init.tcl** and then type **ps7_init** at the command prompt to initialize the PS section (such as Clock PLL, MIO, and DDR initialization).

   Note: If you are using a rev C or D Zedboard, follow steps c and d. Otherwise, skip to step e.

   c. At the command prompt type **source <directory>/stub.tcl**

Note: stub.tcl is located in the location where you unzipped the contents of the downloaded CTT zip file.

d. Type **target 64** to provide execution control to CPU0.

e. Type **dow <directory>/u-boot.elf** to download Linux U-Boot.

f. Type **con** to start execution of U-Boot. Immediately switch to the serial terminal.

On the serial terminal, the autoboot countdown message appears:

Hit any key to stop autoboot: 3

g. Press any key.

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

h. At the XMD Prompt, type **stop**.

The U-Boot execution is stopped.

i. Make a copy of zImage.bin and rename it to **zimage**. Type **dow -data <directory>/zimage 0x8000** to download the Linux Kernel image (zImage) at location 0x8000.

j. Type **dow -data <directory>/ramdisk8M.image.gz 0x800000** to download the Linux root file system image at location 0x800000.

k. Type **dow -data <directory>/devicetree.dtb 0x1000000** to download the Linux device tree at location 0x1000000.

l. Type **con** to start executing U-Boot.

7. At the command prompt of the serial terminal, type **go 0x8000**.

   The Linux OS boots. After booting completes, the **zynq>** prompt appears on the serial terminal

8. At the **zynq>** prompt, perform the following steps:

a. Set the IP address of the board by typing the following command at the **zynq>** prompt: **ifconfig eth0 192.168.1.10 netmask 255.255.255.0**

This command sets the board IP address to 192.168.1.10.

b. Check the connection with the board by typing **ping 192.168.1.10**. The following ping response displays in a continuous loop:

```
64 bytes from 192.168.1.10: seq=0 ttl=64 time=0.185 ms
```

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

c. Press **Ctrl+C** to stop displaying the ping response.

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

5.2.4 Booting Linux from QSPI Flash

5.2.5 Take a Test Drive! Booting Linux from QSPI Flash

This Test Drive covers the following steps:

1. Create the First Stage Boot Loader Executable File.
2. Make a Linux Bootable Image for QSPI Flash.
3. Program QSPI Flash with the Boot Image using JTAG.
4. Booting Linux from QSPI Flash.

1. Step 1: Create the First Stage Boot Loader Executable File

Note: You can skip this step by using the zynq_fsbl_0.elf provided.

1. In SDK, select **File** > **New** > **Application Project**.

   The New Project wizard opens; for **Project Name**, type in **zynq_fsbl_0** and click **Next**.

2. Select **Zynq FSBL** in the Template list and keep the remaining default options. The Location of your project, the hardware platform used, and the processor are visible in this window. The processor is ps7_cortexa9_0.
3. Click **Finish** to generate the FSBL.

   The Zynq FSBL compiles and .ELF file is generated.

2. Step 2: Make a Linux Bootable Image for QSPI Flash

1. In SDK, select **Xilinx Tools** > **Create Zynq Boot Image**.

   The Create Zynq Boot Image Wizard opens.

2. Provide the path to zynq_fsbl_0.elf in the FSBL ELF field.
3. Add the provided u-boot.elf image.

4. Add the Linux Kernel image, zImage.bin, and provide the offset **0x100000**. If you cannot see the files in the menu, simply select show all extensions (*.*) in the file type selection drop down menu.

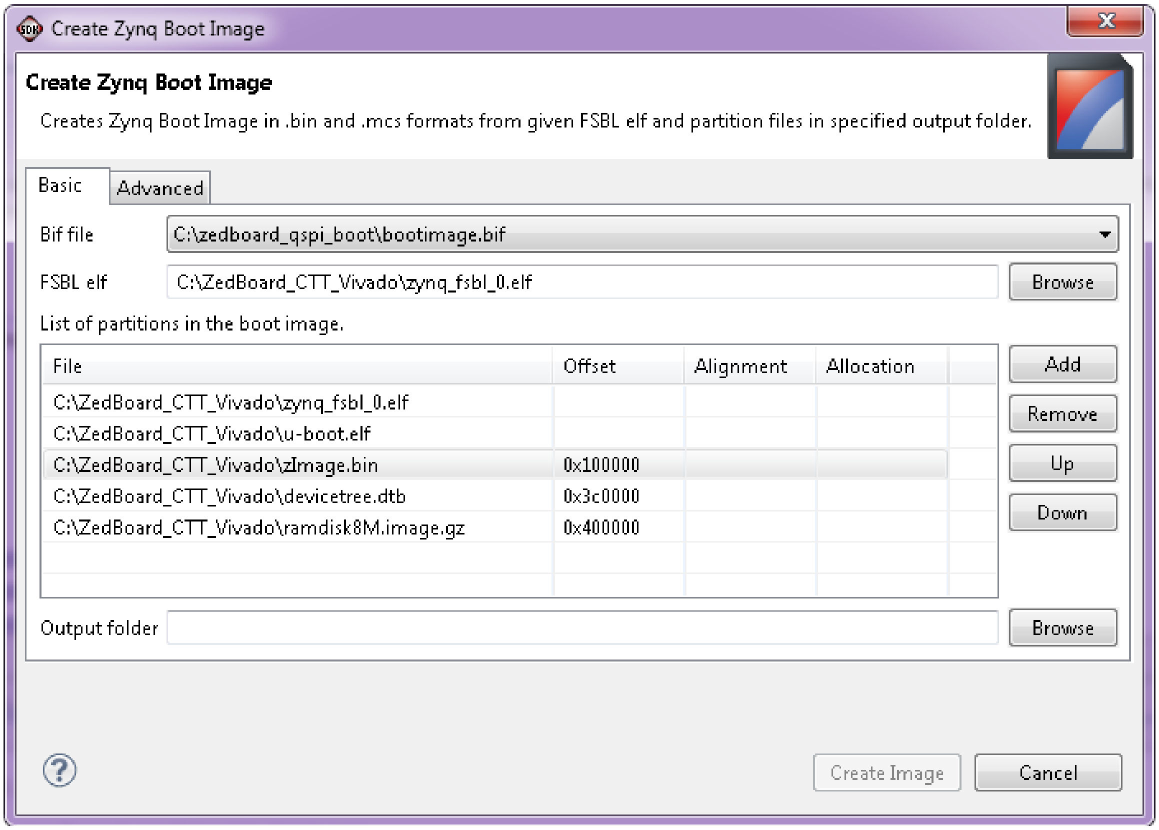
IMPORTANT: *There is a Known Issue with the Bootgen command: it does not accept a file without a file extension. To work around this issue, change the* `zImage` *downloaded file to* `zImage.bin`*.*

5. Add the device tree image (devicetree.dtb) and provide offset - **0x3c0000.**

6. Add the root file system image (ramdisk8M.image.gz) and provide offset **0x400000**.

The provided offsets are predefined in the U-Boot. U-Boot expects those addresses when booting from QSPI, therefore you must not change the offset without modifying and re-building the U-Boot image.

7. Provide the absolute path to the output folder name in the Output folder text box. In this example, we have used "qspi-boot" as the folder to store the output files.



Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

Bif file: C:\zedboard_qspi_boot\bootimage.bif

FSBL elf: C:\ZedBoard_CTT_Vivado\zynq_fsbl_0.elf

List of partitions in the boot image.

File	Offset	Alignment	Allocation
C:\ZedBoard_CTT_Vivado\zynq_fsbl_0.elf			
C:\ZedBoard_CTT_Vivado\u-boot.elf			
C:\ZedBoard_CTT_Vivado\zImage.bin	0x100000		
C:\ZedBoard_CTT_Vivado\devicetree.dtb	0x3c0000		
C:\ZedBoard_CTT_Vivado\ramdisk8M.image.gz	0x400000		

Output folder:

Figure 5-3: Creating a Zynq QSPI Boot Image

8. Click **Create Image.**

The Create Zynq Boot Image window creates following files in the specified output folder:

bootimage.bif

u-boot.bin

u-boot.mcs

3. Step 3: Program QSPI Flash with Boot Image using JTAG & UBoot

1. Power on the ZedBoard.
2. Set the Jumpers JP7-11 to the JTAG boot mode:

   MI06: 0

   MI05: 0

   MI04: 0

   MI03: 0

   MI02: 0
3. If a serial terminal is not open, connect the serial terminal with the baud rate set to **115200**.
4. Select **Xilinx Tools > XMD Console** to open the XMD tool.
5. From the XMD prompt, do the following:
   a. Type **connect arm hw** to connect with the PS section CPU.

   b. Type **source <project_path></project_name>.sdk/SDK/SDK_Export/hw/ps7_init.tcl** and then **ps7_init** to initialize the PS section (such as Clock PLL, MIO, and DDR initialization).

   c. Type **dow <directory>/u-boot.elf** to download the Linux U-Boot to the QSPI Flash.

   Note: The <directory> is the output directory you previously provided.

   d. Type **dow -data <boot_directory>/u-boot.bin 0x08000000** to download the Linux bootable image to the target memory at location 0x08000000.

You just downloaded the binary executable to DDR memory. You can download the binary executable to any address in DDR memory, but make sure that you do not change the U-Boot executable, which is loaded at 0x04000000. You run this file after loading the u-boot.bin data file.

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

On the serial terminal, the autoboot countdown message appears:

```
Hit any key to stop autoboot: 3
```

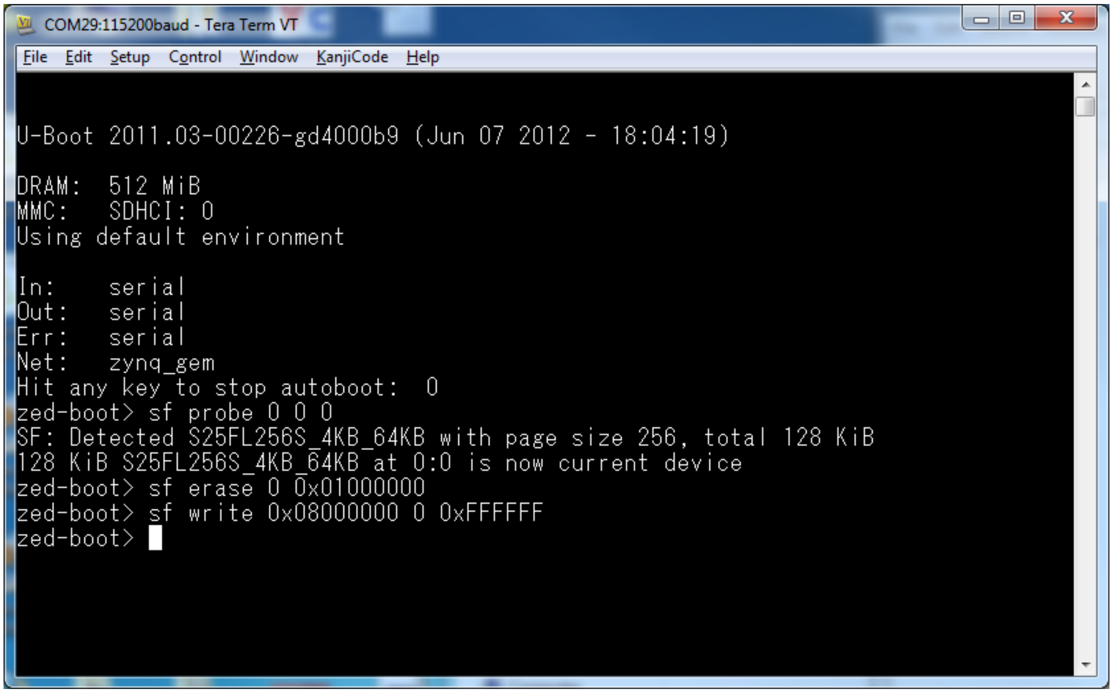
6. Press any key to stop the boot process.

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

7. 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. (Note that this step can take about 8 minutes to complete.)
   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.

You can change the argument to adjust the bootable image size.


COM29:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

```
U-Boot 2011.03-00226-gd4000b9 (Jun 07 2012 - 18:04:19)

DRAM:  512 MiB
MMC:   SDHCI: 0
Using default environment

In:    serial
Out:   serial
Err:   serial
Net:   zynq_gem
Hit any key to stop autoboot:  0
zed-boot> sf probe 0 0 0
SF: Detected S25FL256S_4KB_64KB with page size 256, total 128 KiB
128 KiB S25FL256S_4KB_64KB at 0:0 is now current device
zed-boot> sf erase 0 0x01000000
zed-boot> sf write 0x08000000 0 0xFFFFFF
zed-boot> █
```

Figure 5-4: Serial Terminal Window showing QSPI programming

8. Power off the ZedBoard.

4. Booting Linux from the QSPI Flash

1. After you program the QSPI Flash, set the jumper settings (JP7-11) on the ZedBoard. Jumper settings for QSPI:

MI06: 0

MI05: 1

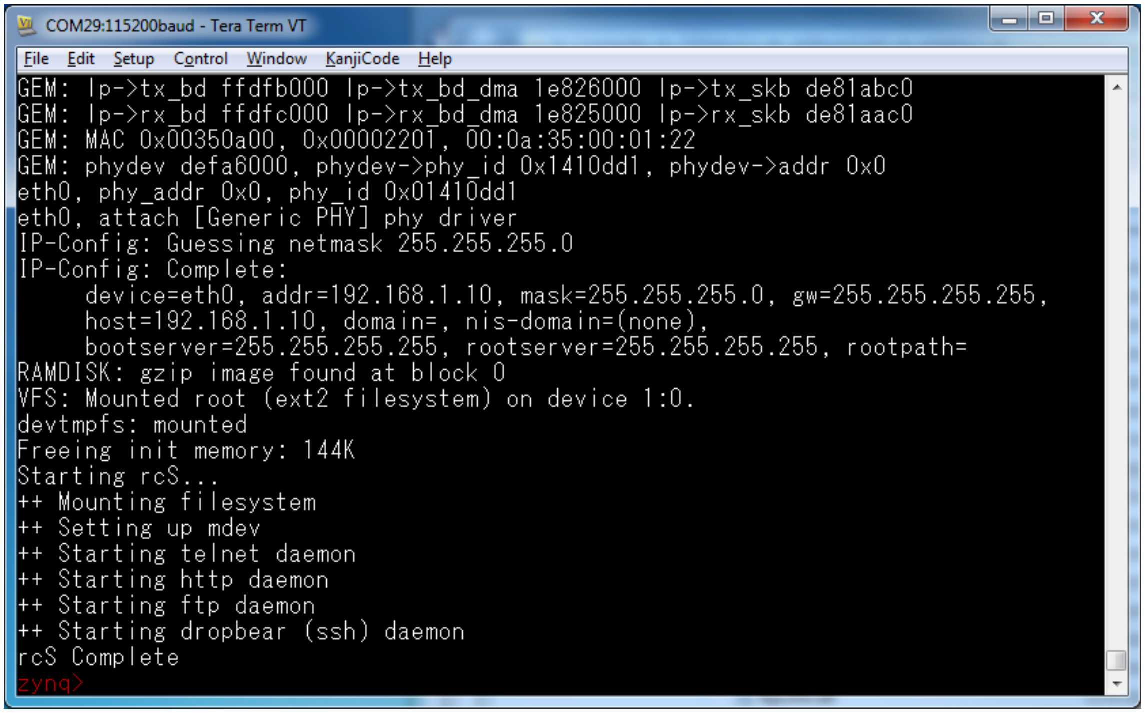
MI04: 0

MI03: 0

MI02: 0

2. Connect the Serial terminal with a 115200 baud rate setting.
3. Switch on the board power.

A Linux booting message appears on the serial terminal. After booting finishes, the **zynq>** prompt appears.



COM29:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

```
GEM: lp->tx_bd ffdfb000 lp->tx_bd_dma 1e826000 lp->tx_skb de81abc0
GEM: lp->rx_bd ffdfc000 lp->rx_bd_dma 1e825000 lp->rx_skb de81aac0
GEM: MAC 0x00350a00, 0x00002201, 00:0a:35:00:01:22
GEM: phydev defa6000, phydev->phy_id 0x1410dd1, phydev->addr 0x0
eth0, phy_addr 0x0, phy_id 0x01410dd1
eth0, attach [Generic PHY] phy driver
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
     device=eth0, addr=192.168.1.10, mask=255.255.255.0, gw=255.255.255.255,
     host=192.168.1.10, domain=, nis-domain=(none),
     bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
RAMDISK: gzip image found at block 0
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing init memory: 144K
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
rcS Complete
zynq>
```

Figure 5-5: Serial Terminal Window showing Linux Booting

5.2.6 Booting Linux from the SD Card

5.2.7 Take a Test Drive! Booting Linux from the SD Card

Ensure that the jumper settings (JP7-11) are set to boot from SD card as shown in the figure.

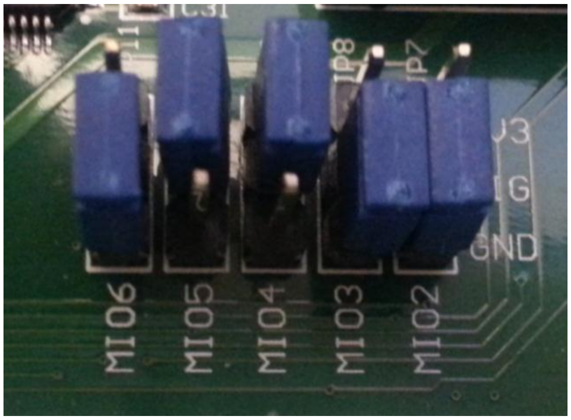


Figure 5-6: Jumper Settings to boot from SD Card

1. Create an FSBL for your design as described in “Step 1: Create the First Stage Boot Loader Executable File” . Alternatively, you can use the zync_fsbl_0.elf file that you downloaded previously.

2. In SDK, select **Xilinx Tools > Create Zynq Boot Image** to open the “Create Zynq Boot Image” wizard. Alternatively, you can use the u-boot.bin file that you downloaded previously, and skip to step 6.

3. Add **zynq_fsbl_0.elf** and **u-boot.elf**.

4. Provide the location to store all generated files in the Output Folder field. Make sure to rename

5. Click **Create Image**. SDK generates the u-boot.bin file in the specified output folder.

6. Rename u-boot.bin to BOOT.bin. Copy **BOOT.bin, zImage, devicetree.dtb** and **ramdisk8M.image.gz** to the SD card. Rename zImage.bin to just zimage (no file extension) in the SD card. Make sure the SD card is FAT32 formatted before copying the files into it.

7. Turn on the power to the board and check the messages on the Serial terminal. The **zynq>** prompt appears after Linux booting is complete on the target board.

5.3 Hello World Example

This example shows you how to create a simple Linux application that prints “Hello World” on a serial terminal window.

5.3.1 Take a Test Drive! Running a “Hello World” Application

1. Setup your ZedBoard connections

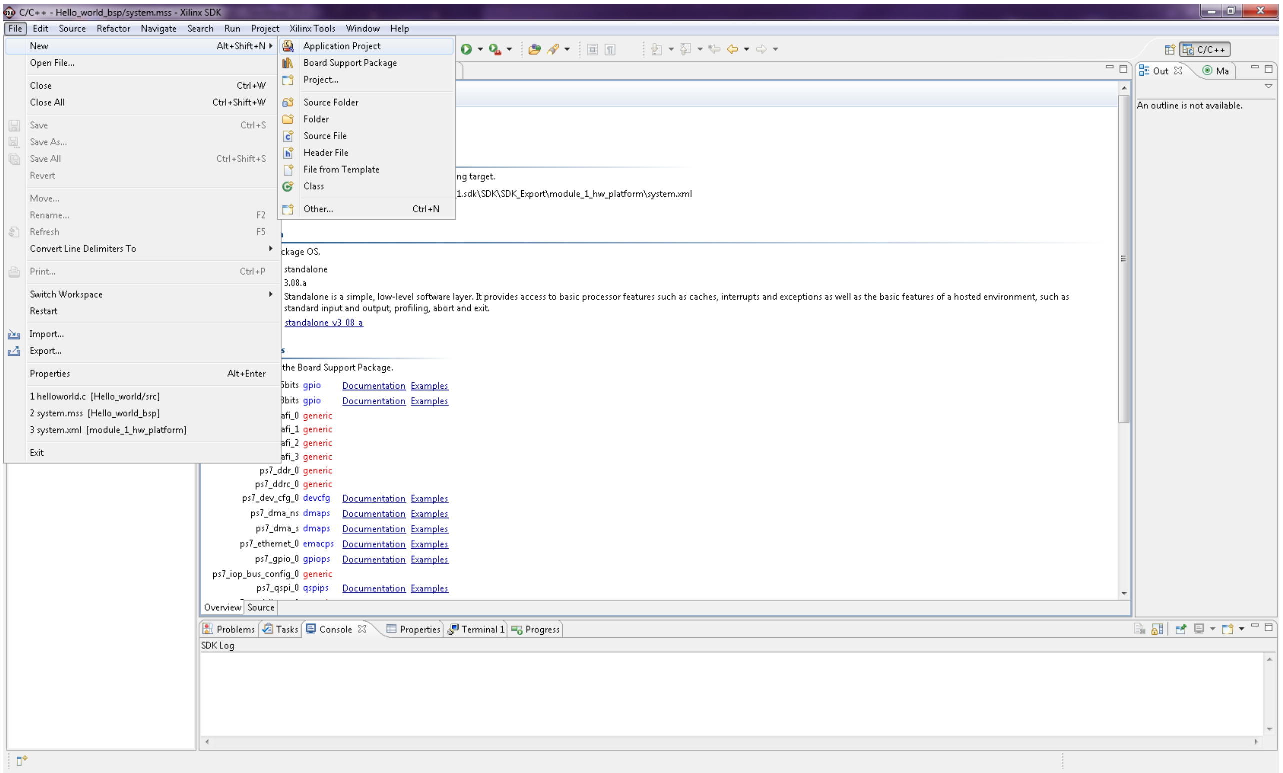
a. Connect the power cable to the ZedBoard.

b. Connect a USB micro cable to the USB UART connector on the ZedBoard with the Windows Host machine. This is used for USB to serial transfer.

c. Make sure the SD card with the Linux image is inserted into the ZedBoard.

2. Launch SDK, and navigate to the same project directory that you used earlier in this chapter to create an FSBL. In this section, the directory used for illustration is: **C:\<project_path>\<project_name>\<project_name>.sdk\SDK\SDK_Export**. You can define your own project directory.

3. In SDK, select **File > New > Application Project.**



ng target.
1.sdk\SDK\SDK_Export\module_1_hw_platform\system.xml

ckage 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 features of a hosted environment, such as standard input and output, profiling, abort and exit.

the Board Support Package.

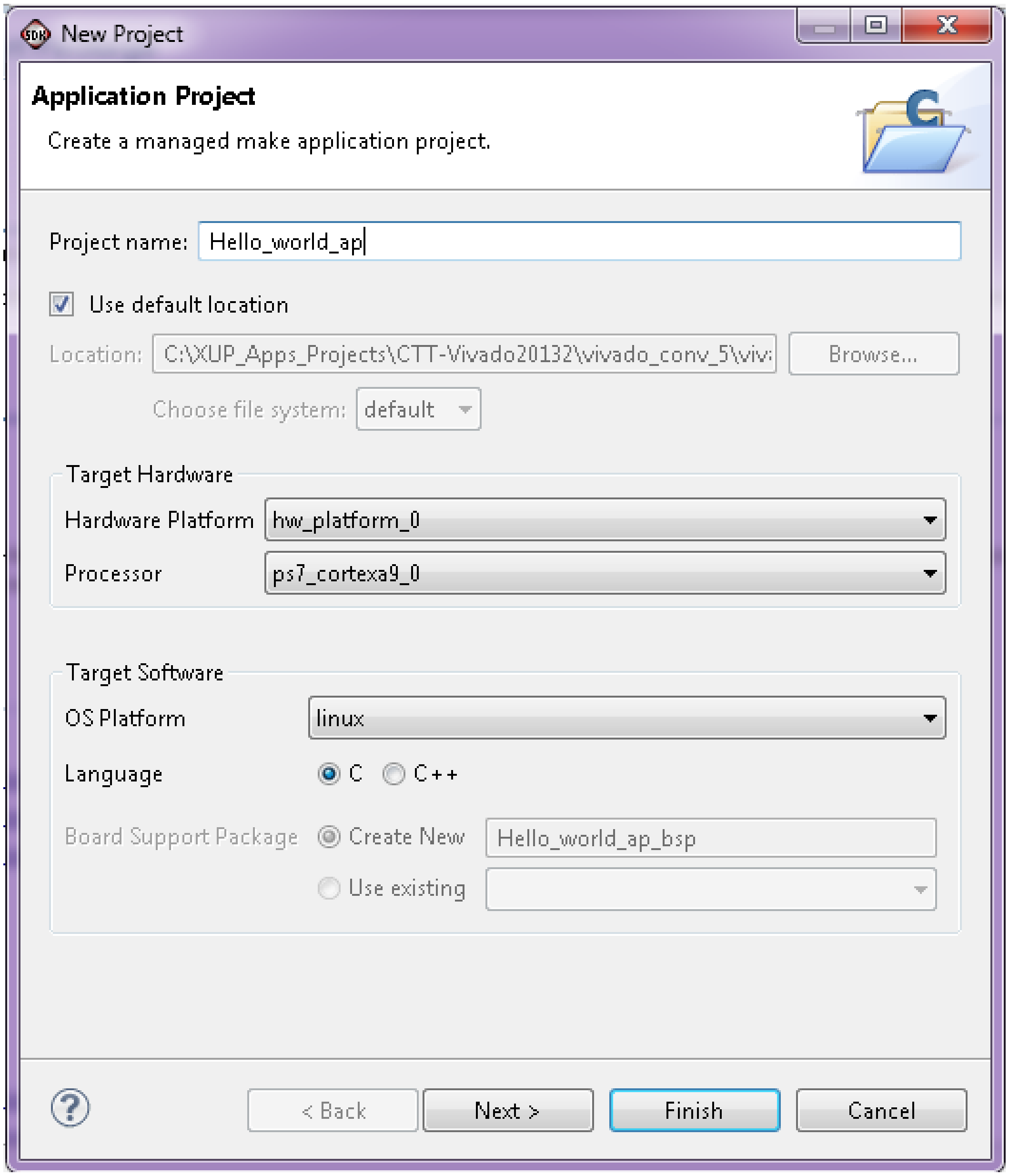
Figure 5-7: New Project Selection

4. Enter **hello_world_ap** in the **Project name** field

5. Select **Linux** as the OS Platform in the Target Software and select **Finish.**

6. Select **C** as the Language.

7. Click **Next.**



New Project

Application Project

Create a managed make application project.

Project name: Hello_world_ap

☑ Use default location

Location: C:\XUP_Apps_Projects\CTT-Vivado20132\vivado_conv_5\viv:

Choose file system: default

Target Hardware

Hardware Platform: hw_platform_0

Processor: ps7_cortexa9_0

Target Software

OS Platform: linux

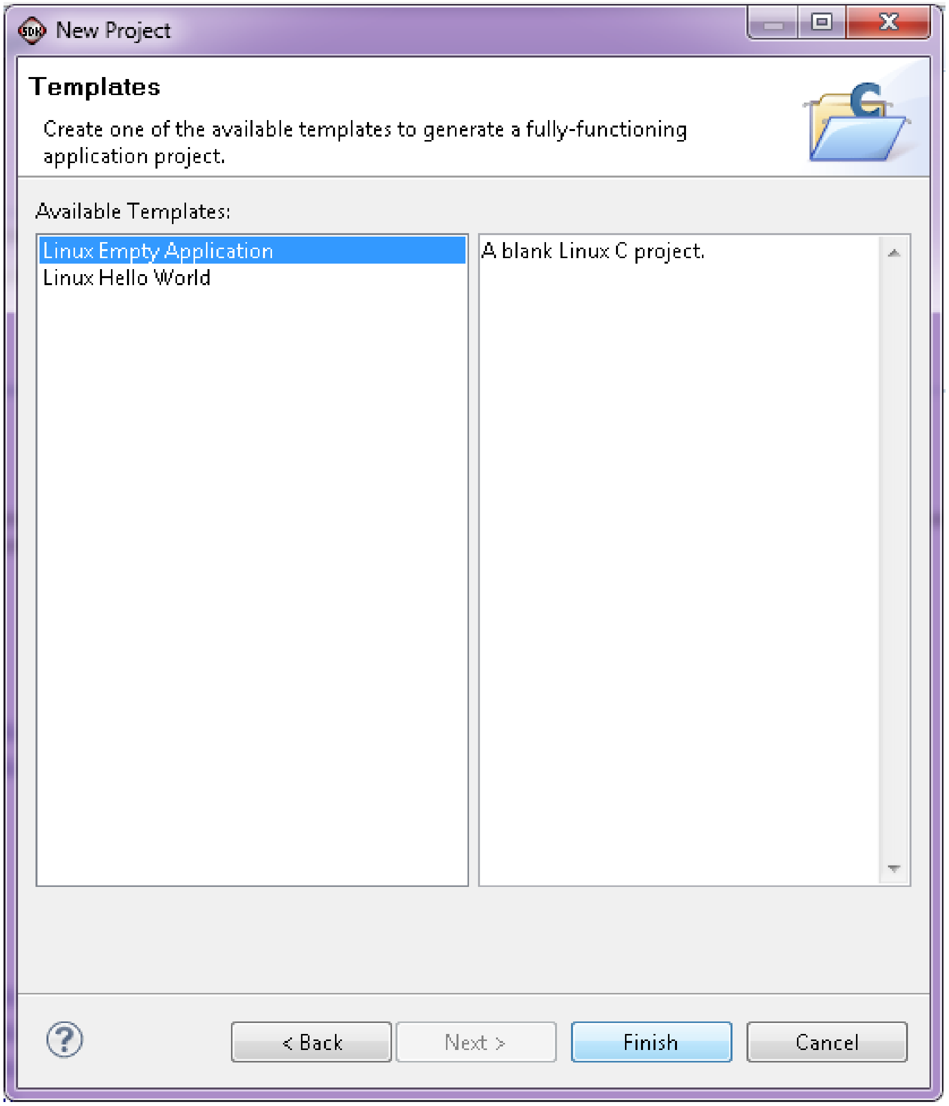
Language: ● C ○ C++

Board Support Package: ● Create New Hello_world_ap_bsp

○ Use existing

Figure 5-8: Application Project

8. Select **Linux Empty Application** and click **Finish.**



Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

- Linux Empty Application
- Linux Hello World

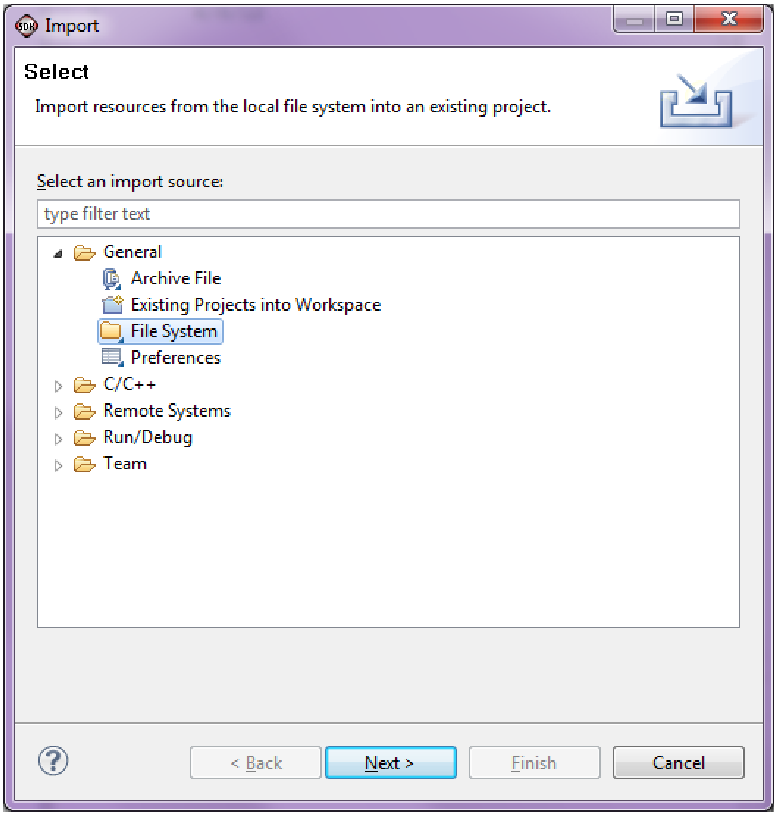
A blank Linux C project.

Figure 5-9: Add An Empty Application

9. Add a Software Application. At this point, you will create a software platform and an empty software project for the hardware. You will then import the hello_world_linux.c into the project, and SDK will automatically build and produce an elf (Executable and Load Format) file.

10. Right Click **hello_world_ap** and select **Import.**

11. In the Import dialog box, select **General** > **File System** and select **Next**.


Import

Select

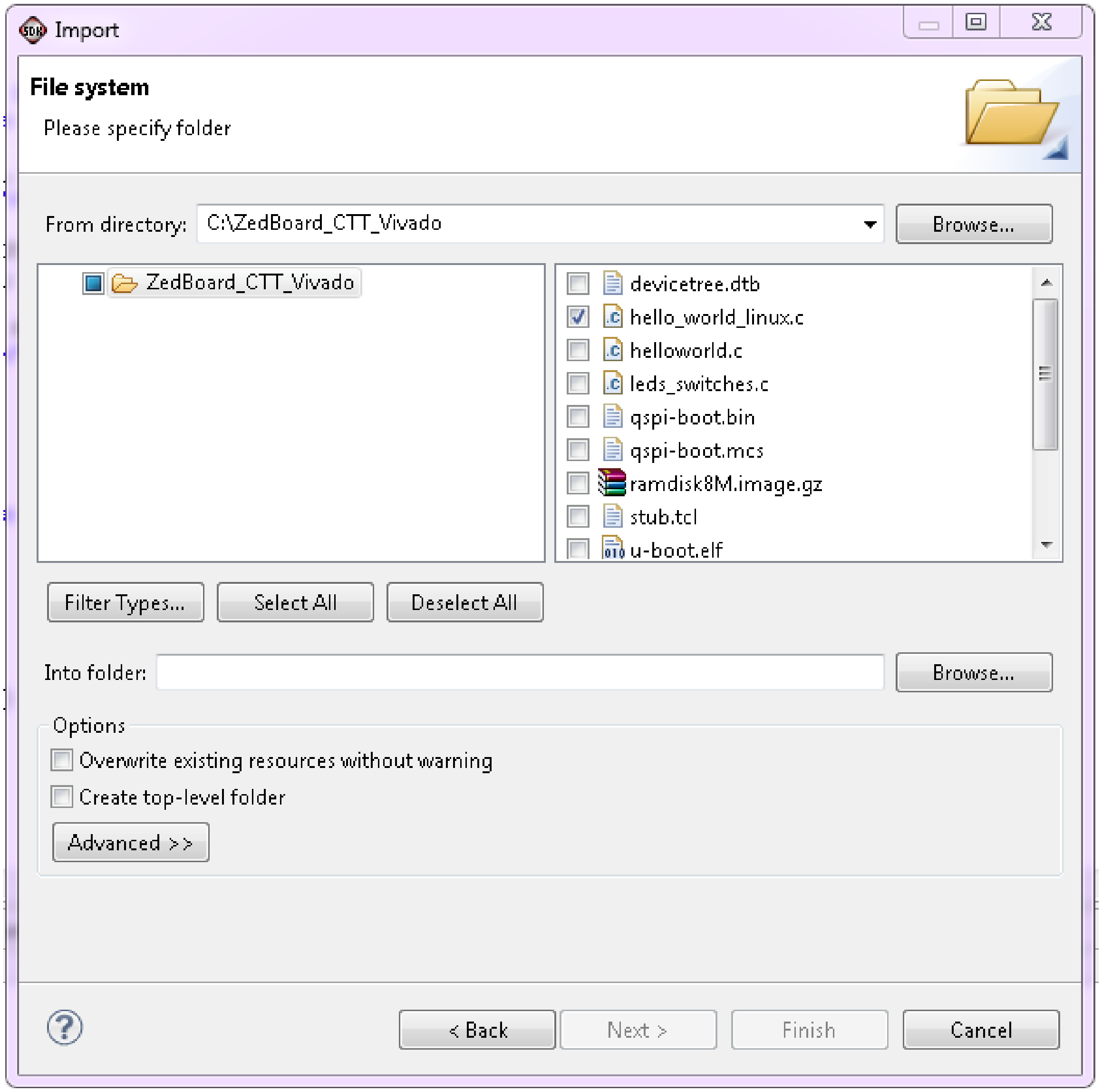
Import resources from the local file system into an existing project.

Select an import source:

type filter text

Figure 5-10: Import .C file

12. Browse to the directory in which you saved the ZedBoard CTT files that you downloaded. Select **hello_world_linux.c** and select **Finish** .



Import

File system

Please specify folder

From directory: C:\ZedBoard_CTT_Vivado

■ ZedBoard_CTT_Vivado

- ☐ devicetree.dtb
- ☑ hello_world_linux.c
- ☐ helloworld.c
- ☐ leds_switches.c
- ☐ qspi-boot.bin
- ☐ qspi-boot.mcs
- ☐ ramdisk8M.image.gz
- ☐ stub.tcl
- ☐ u-boot.elf

Into folder:

Options

- ☐ Overwrite existing resources without warning
- ☐ Create top-level folder

Figure 5-11: Select hello_world_linux.c

Check that the application is built without errors. Check the message log in the Console window. You will see text similar to:

```
Invoking: ARM Linux Print Size
arm-xilinx-linux-gnueabi-size hello_world_ap.elf  |tee
"hello_world_ap.elf.size"
   text        data        bss        dec        hex     filename
   1440         292          4       1736        6c8
      hello_world_ap.elf
Finished building: hello_world_ap.elf.size
   ' '
```

1. In your project directory, you will see that the compiled file, hello_world_ap.elf has been created. In this example, hello_world_ap.elf is located in the directory:

   C:\<project_path>\<project_name>\<project_name>.sdk\SDK\SDK_Export\hello_world_ap\Debug

2. Copy **hello_world_ap.elf** to the SD card containing the Linux boot files.

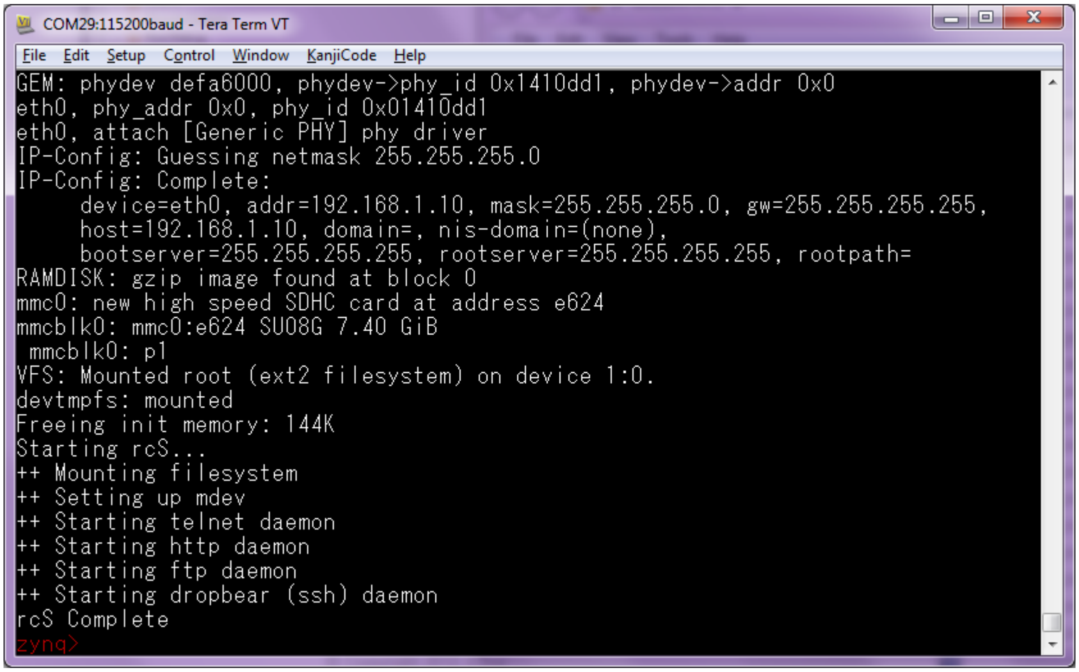
3. Insert the SD card back into the ZedBoard.

4. Ensure that the Jumpers JP7-11 are set in SD card boot mode.

5. Power on the ZedBoard, and open a serial terminal window.

6. Boot Linux on the ZedBoard from the SD card with the pre-built image.

7. Linux has been successfully booted when you see the **zynq>** prompt in your serial teriminal window.


COM29:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

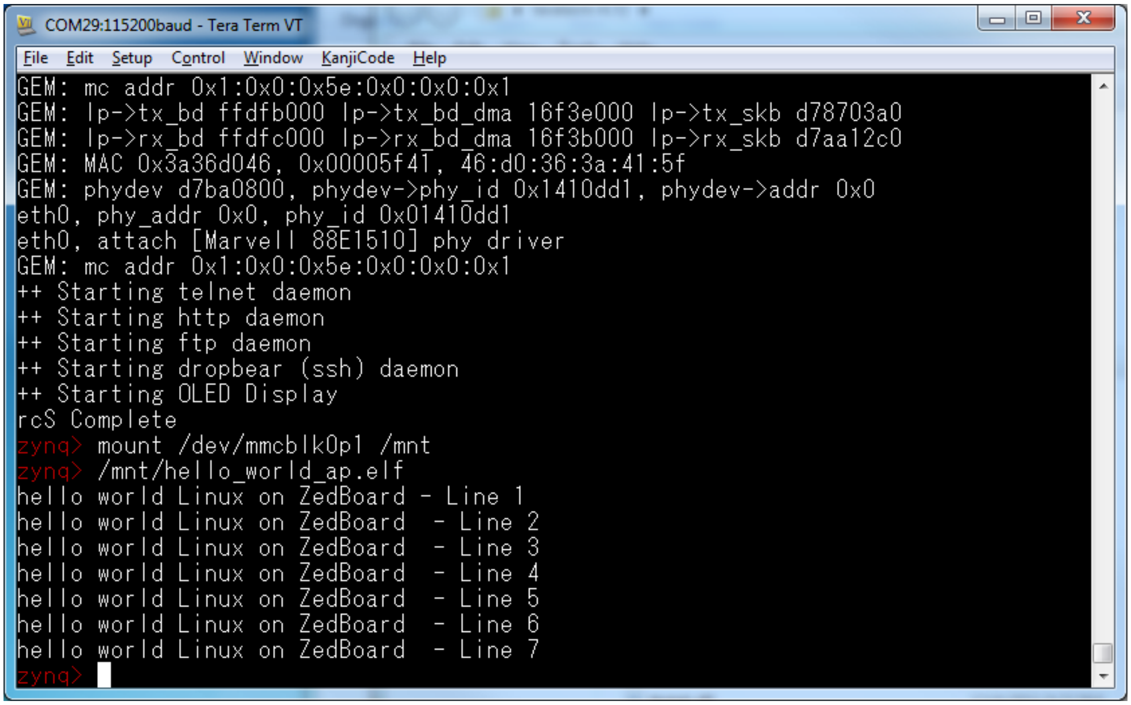
```
GEM: phydev defa6000, phydev->phy_id 0x1410dd1, phydev->addr 0x0
eth0, phy_addr 0x0, phy_id 0x01410dd1
eth0, attach [Generic PHY] phy driver
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
     device=eth0, addr=192.168.1.10, mask=255.255.255.0, gw=255.255.255.255,
     host=192.168.1.10, domain=, nis-domain=(none),
     bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address e624
mmcblk0: mmc0:e624 SU08G 7.40 GiB
 mmcblk0: p1
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing init memory: 144K
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
rcS Complete
zynq>
```

Figure 5-12: Serial Teriminal Window showing Linux Booting

8. In the serial terminal window, at the **zynq>** prompt type:

```
zynq> mount /dev/mmcblk0p1 /mnt

zynq> /mnt/hello_world_ap.elf
```

This executes the hello_world_ap program and you see the display on the terminal.



COM29:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

```
GEM: mc addr 0x1:0x0:0x5e:0x0:0x0:0x1
GEM: lp->tx_bd ffdfb000 lp->tx_bd_dma 16f3e000 lp->tx_skb d78703a0
GEM: lp->rx_bd ffdfc000 lp->rx_bd_dma 16f3b000 lp->rx_skb d7aa12c0
GEM: MAC 0x3a36d046, 0x00005f41, 46:d0:36:3a:41:5f
GEM: phydev d7ba0800, phydev->phy_id 0x1410dd1, phydev->addr 0x0
eth0, phy_addr 0x0, phy_id 0x01410dd1
eth0, attach [Marvell 88E1510] phy driver
GEM: mc addr 0x1:0x0:0x5e:0x0:0x0:0x1
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
++ Starting OLED Display
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt
zynq> /mnt/hello_world_ap.elf
hello world Linux on ZedBoard - Line 1
hello world Linux on ZedBoard  - Line 2
hello world Linux on ZedBoard  - Line 3
hello world Linux on ZedBoard  - Line 4
hello world Linux on ZedBoard  - Line 5
hello world Linux on ZedBoard  - Line 6
hello world Linux on ZedBoard  - Line 7
zynq> 
```

Figure 5-13: Serial Terminal Window with hello_world_linux running

5.4 Controlling LEDs and Switches in Linux Example

This example shows you how to create a simple Linux application that controls the status of the LEDs and prints the value of the switch settings, then prints “Hello World” on a serial terminal window. In this example, the default ZedBoard settings in Vivado are used; a bitstream is generated in and then the entire design is exported to SDK.

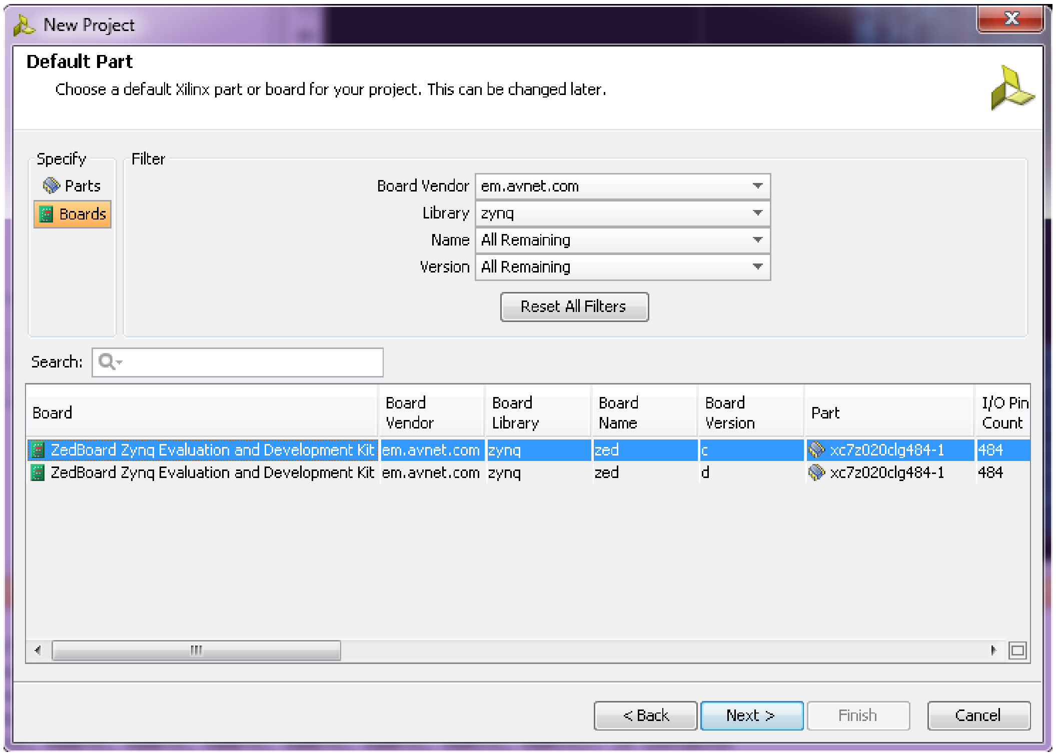
5.4.1 Take a Test Drive! Controlling LEDs and Switches in a Linux Application

For this test drive, just as you did in Chapter 2, you start the Vivado design and analysis tool and create a project with an embedded processor system as the top level.

Launch a new instance of the Vivado Design Suite.

1. Select **Create New Project** to open the New Project wizard.
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
Project Name	Project name	Specify the project name.
	Project location	Specify the directory in which to store the project files.
	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	Use the default selection, **RTL Project**.
Add Sources	Do not make any changes on this screen.	
Add Existing IP	Do not make any changes on this screen.	
Add Constraints	Do not make any changes on this screen.	
Default Part	Specify	Select **Boards**.
	Board	Select **ZedBoard Zynq Evaluation and Development Kit**
New Project Summary	Project summary	Review the project summary before clicking **Finish** to create the project.



New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Specify: Parts / Boards

Filter

Board Vendor: em.avnet.com

Library: zynq

Name: All Remaining

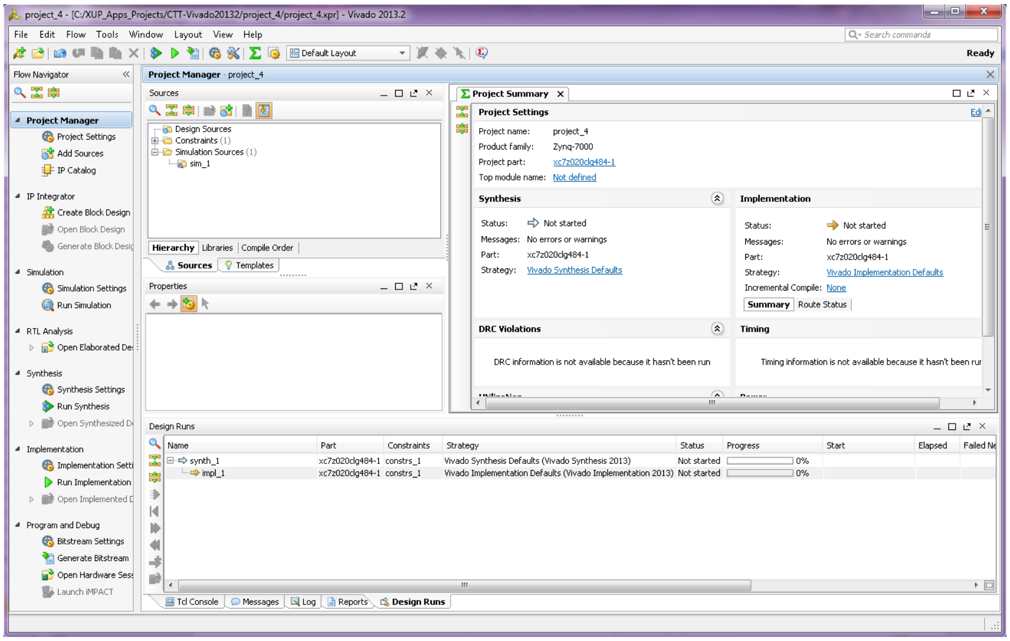
Version: All Remaining

Search:

Board	Board Vendor	Board Library	Board Name	Board Version	Part	I/O Pin Count
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	c	xc7z020clg484-1	484
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	d	xc7z020clg484-1	484

Figure 5-14: New Project Wizard Part Selection

When you click **Finish**, the New Project wizard closes and the project you just created opens in the Vivado GUI.



project_4 - [C:/XUP_Apps_Projects/CTT-Vivado20132/project_4/project_4.xpr] - Vivado 2013.2

Project Settings

Project name: project_4

Product family: Zynq-7000

Project part: xc7z020clg484-1

Top module name: Not defined

Synthesis

Status: Not started

Messages: No errors or warnings

Part: xc7z020clg484-1

Strategy: Vivado Synthesis Defaults

Implementation

Status: Not started

Messages: No errors or warnings

Part: xc7z020clg484-1

Strategy: Vivado Implementation Defaults

Incremental Compile: None

DRC Violations

DRC information is not available because it hasn't been run

Timing

Timing information is not available because it hasn't been rur

Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Failed Ne
synth_1	xc7z020clg484-1	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2013)	Not started	0%			
impl_1	xc7z020clg484-1	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2013)	Not started	0%			

Figure 5-15: Vivado GUI

You'll now use the IP Integrator to create an embedded processor block diagram.

3. Click **Create Block Design** in the Project Manager.

You will be asked to specify the design name, use the name **system**.

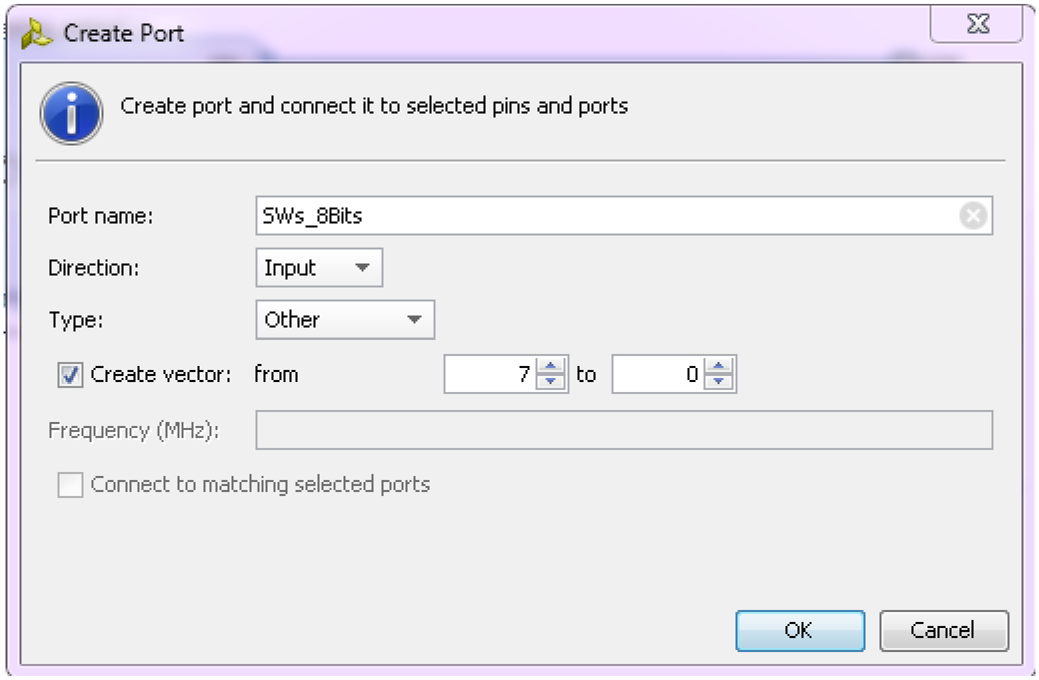
4. Click the Add IP button and double click on the **ZYNQ7 Processing System** to add it to the block diagram.
5. Double click on the **ZYNQ7 Processing System** block to edit the block parameters.
6. Click on the Presets button and select **ZedBoard Development Board Template** and click **OK**. Click **OK** again to exit the ZYNQ7 Processing Systems editing window.
7. Click on the **Add IP** button. From the IP catalog, double-click **AXI GPIO** to add it. Notice the block has been automatically added to the Block Diagram.

Drag the AXI GPIO block to a convenient location to the right of the Processing System block.

8. Repeat the step above to add two more AXI GPIO blocks to the block diagram.
9. Click the **Add IP** button again, double click on the **AXI Interconnect** IP. Drag the IP to a convenient location to the right of the Processing System block.
10. Connect **FCLK_CLK0** on the **ZYNQ7 Processing System** block to all the clock ports in the block diagram.
11. Connect **FCLK_RESET0_N** on the **ZYNQ7 Processing System** block to all the reset ports in the block diagram.
12. Connect **M_AXI_GP0** on the **ZYNQ7 Processing System** block to **S00_AXI** of the **AXI Interconnect** block.
13. Right click on **DDR** on the **ZYNQ7 Processing System** block and select **Make External**. Notice a DDR port is automatically created for it.
14. Right click on **FIXED_IO** on the **ZYNQ7 Processing System** block and select **Make External**. Notice a FIXED_IO port is automatically created for it.
15. Connect **M00_AXI** on the **AXI Interconnect** block to **s_axi** of a **AXI GPIO** block.
16. Connect **M01_AXI** on the **AXI Interconnect** block to **s_axi** of a **AXI GPIO** block.
17. Connect **M02_AXI** on the **AXI Interconnect** block to **s_axi** of the last remaining **AXI GPIO** block.
18. Click on the AXI GPIO block connected to M00_AXI (on the AXI Interconnect) and name it as **SWs_8bits** in the Block Properties pane. Double click on the

sws_8bits block to edit its properties. Click on the **IP Configuration** tab and select the **GPIO Width** as **8**. Click **OK** to close the window.

19. Click on the AXI GPIO block connected to M01_AXI (on the AXI Interconnect) and name it as **LEDs_8bits** in the Block Properties pane. Double click on the leds_8bits block to edit its properties. Click on the **IP Configuration** tab and select the **GPIO Width** as **8**. Click **OK** to close the window.
20. Click on the AXI GPIO block connected to M02_AXI (on the AXI Interconnect) and name it as **BTNs_5bits** in the Block Properties pane. Double click on the btns_5bits block to edit its properties. Click on the **IP Configuration** tab and select the **GPIO Width** as **5**. Click **OK** to close the window.



Create Port

Create port and connect it to selected pins and ports

Port name: SWs_8Bits

Direction: Input

Type: Other

☑ Create vector: from 7 to 0

Frequency (MHz):

☐ Connect to matching selected ports

Figure 5-16: Vivado GUI

21. **Run Connection Automation** (in the green Designer Assistance bar) and select **/LEDs_8bits/gpio**. A green circle appears at the gpio port of the LEDs_8bits block to indicate a valid connection from that port. This is part of the board awareness offered by Vivado.

Notice that LEDs_8bits now has an external port associated with it.

22. Repeat the same procedure as above but with the **SWs_8bits** block. Notice the external port is shown as an output when it should be bi-directional. Do not worry as the HDL wrapper will reflect the actual direction of the port.
23. Repeat the same procedure as above but with the **BTNs_5bits** block. The comment about the port directionality applies to this block as well.

The block diagram should now look like this.

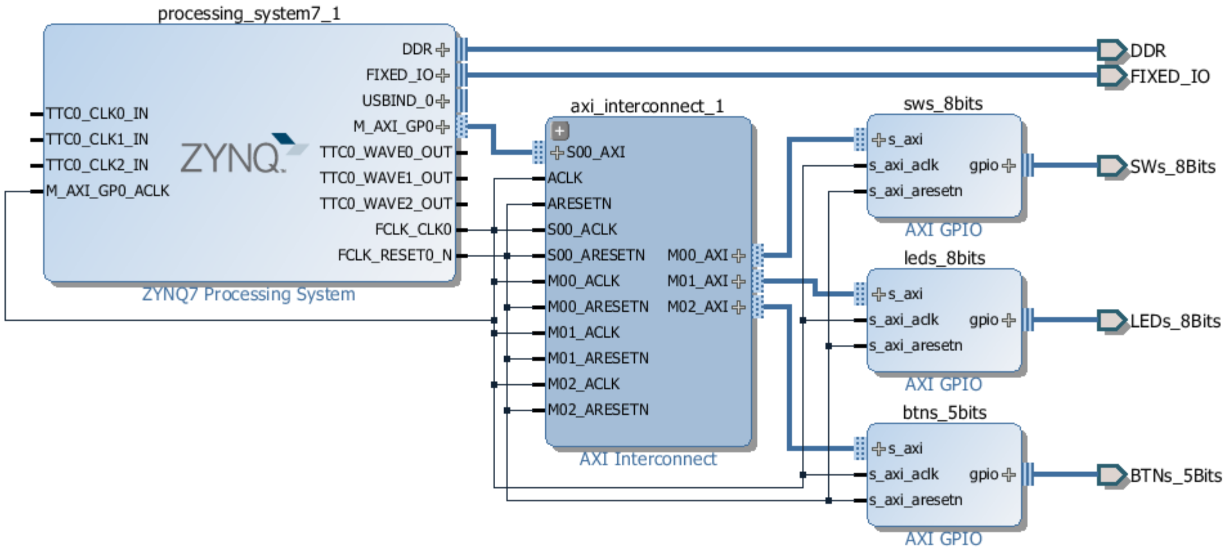
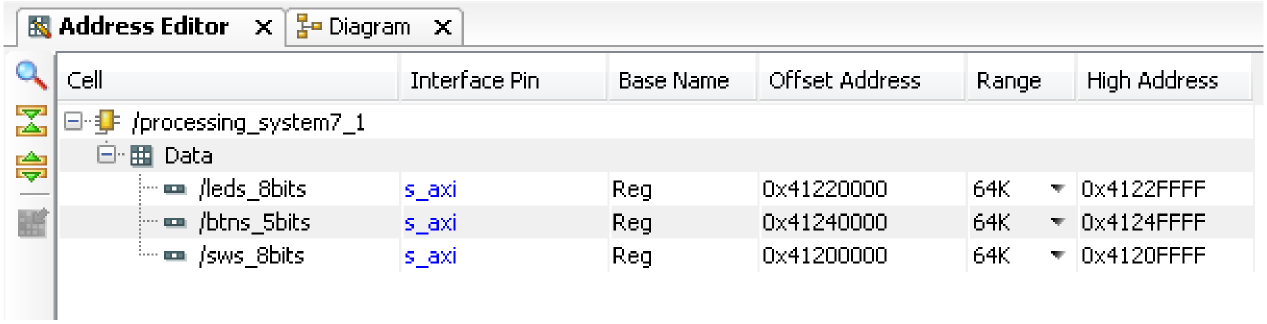


Figure 5-17: System block diagram

1. Unlike the previous exercises, you need to specify addresses for each of the GPIO peripherals. Do not automatically generate addresses.Click on the **Address Editor** tab. If the tab is not available, you can access it via **Window > Address Editor**.
2. Set the **Offset Address** for **leds_8bits** to be **0x41220000**.
3. Set the **Offset Address** for **btns_5bits** to be **0x41240000**.
4. Set the **Offset Address** for **sws_8bits** to be **0x41200000**.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
/processing_system7_1					
Data					
/leds_8bits	s_axi	Reg	0x41220000	64K	0x4122FFFF
/btns_5bits	s_axi	Reg	0x41240000	64K	0x4124FFFF
/sws_8bits	s_axi	Reg	0x41200000	64K	0x4120FFFF

Figure 5-18: Address Editor Offset Addresses for GPIO peripherals

5. **Generate a new HDL wrapper** for the block diagram.
6. Synthesize, Implement and Generate Bitstream for the project.
7. When bitstream generation is complete, make sure to open the implemented design before exporting the design to SDK.

8. Select **File** > **Export** > **Export Hardware for SDK**.

   The Export Hardware dialog box opens.

9. Check the **Include Bitstream** check-box By default, the Export Hardware check-box is checked.

10. Check the **Launch SDK** check-box.

11. Click **OK**; SDK opens.

Continuing Your Design in SDK

1. Connect the12V AC/DC converter power cable to the ZedBoard barrel jack.

2. Connect a USB micro cable between the Windows Host machine and the ZedBoard JTAG (J17).

3. Connect a USB micro cable to the USB UART connector (J14) on the ZedBoard with the Windows Host machine. This is used for USB to serial transfer.

4. Connect an Ethernet cable between the ZedBoard and the Windows Host machine.

5. Create a new BOOT.bin (u-boot.bin) under the **Tools** > **Create Zynq Boot Image** utility. Include the system.bit that was exported to SDK under the **…\project_name>.sdk\SDK_Export\hw_platform_0\** directory.

   The exact order of the *.ELF file is important, make sure it appears in the following order:

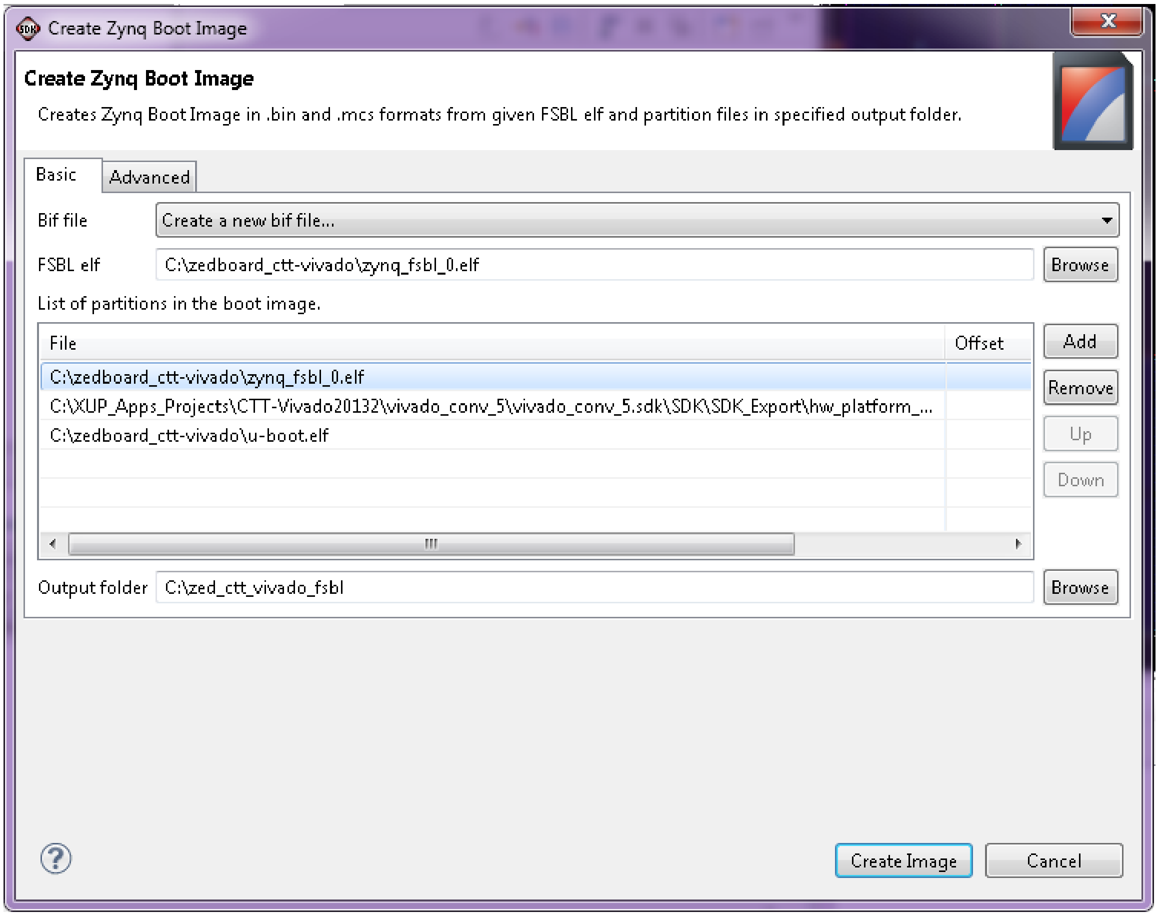
   zynq_fsbl_0.elf

   system.bit

   u-boot.elf

   No offset is needed for the system.bit file. Rename the resultant u-boot.bin to BOOT.bin and copy it to the SD card (assuming the files from Section 5.3 is already in it). Overwrite the previous BOOT.bin on the card if necessary.

   Insert the SD card into the unpowered ZedBoard's card slot once this is done.



Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

Bif file: Create a new bif file...

FSBL elf: C:\zedboard_ctt-vivado\zynq_fsbl_0.elf

List of partitions in the boot image.

File	Offset
C:\zedboard_ctt-vivado\zynq_fsbl_0.elf	
C:\XUP_Apps_Projects\CTT-Vivado20132\vivado_conv_5\vivado_conv_5.sdk\SDK\SDK_Export\hw_platform_...	
C:\zedboard_ctt-vivado\u-boot.elf	

Output folder: C:\zed_ctt_vivado_fsbl

Figure 5-19: Creating a new BOOT.bin (u-boot.bin) for the SD card

6. Power on the board using the jumper settings to boot from SD card.

   MIO6: 0

   MI05: 1

   MIO4: 1

   MIO3: 0

   MIO2: 0

7. Open a serial communication utility for the COM port assigned on your system.

The default configuration for Zynq Processing System is: **Baud rate 115200; 8 bit; Parity: none; Stop: 1 bit; Flow control: none**

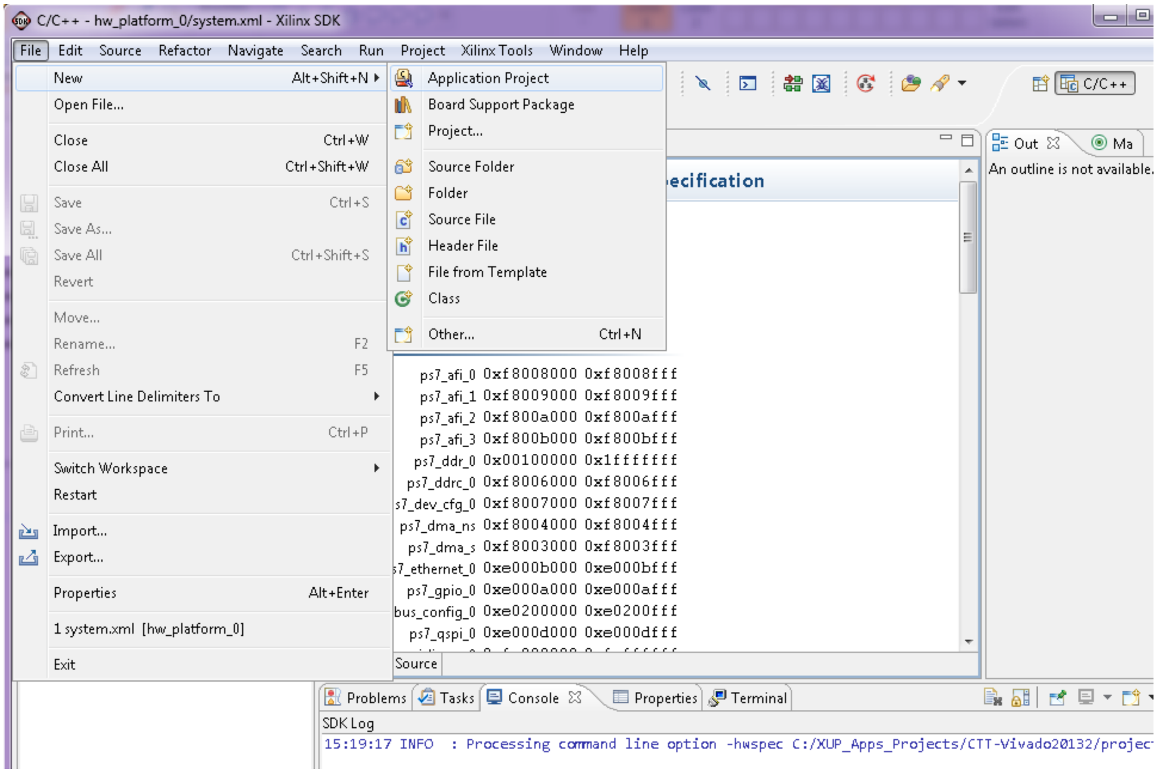
8. Linux boots up, and you will see the promt **zynq**> in the serial terminal window.

9. It may be necessary to double check the IP address of the ZedBoard for the subsequent steps. With the serial terminal open, at the **zynq**> prompt, type **ifconfig eth0** to verify that the address is set for 192.168.1.10. This should be the default IP

address of the ZedBoard. If the IP address is not the same as above, then type: **ifconfig eth0 192.168.1.10 netmask 255.255.255.0** to set the correct board IP address.

Add the software application. At this point, you will create a software platform and an empty software project for the hardware. You will then import the hello_world_linux.c into the project, and SDK will automatically build and produce an .ELF (Executable and Load Format) file.

10. In SDK, select **File > New > Application Project**



```
    ps7_afi_0 0xf8008000 0xf8008fff
    ps7_afi_1 0xf8009000 0xf8009fff
    ps7_afi_2 0xf800a000 0xf800afff
    ps7_afi_3 0xf800b000 0xf800bfff
    ps7_ddr_0 0x00100000 0x1fffffff
   ps7_ddrc_0 0xf8006000 0xf8006fff
s7_dev_cfg_0 0xf8007000 0xf8007fff
  ps7_dma_ns 0xf8004000 0xf8004fff
   ps7_dma_s 0xf8003000 0xf8003fff
s7_ethernet_0 0xe000b000 0xe000bfff
   ps7_gpio_0 0xe000a000 0xe000afff
bus_config_0 0xe0200000 0xe0200fff
   ps7_qspi_0 0xe000d000 0xe000dfff
```

```
15:19:17 INFO  : Processing command line option -hwspec C:/XUP_Apps_Projects/CTT-Vivado20132/projec
```

Figure 5-20: New Project Selection

11. Enter **leds_switches** in the **Project** name field

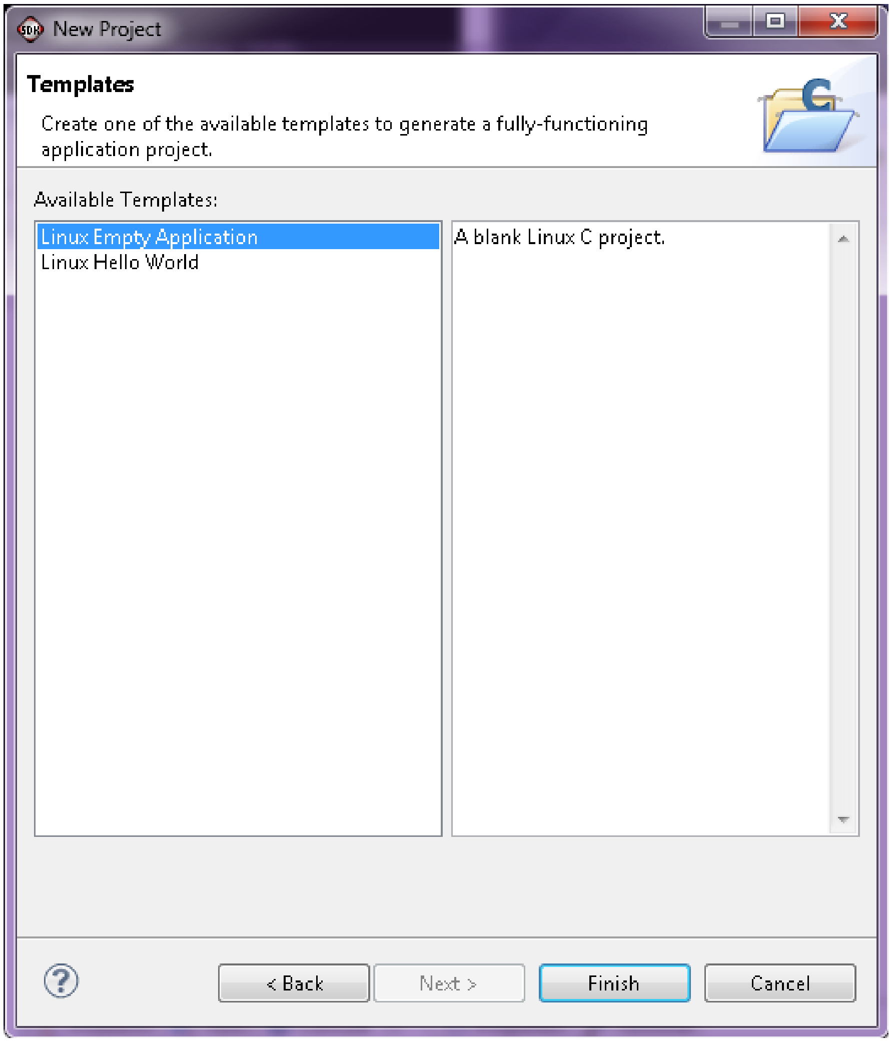
12. Select the Hardware Platform as **zed_hw_platform(pre-defined).**

13. Select **Linux** as the OS Platform in the Target Software and select **Finish.**

14. Select **C** as the Language.

15. Click **Next**.

16. Select **Linux Empty Application** and click **Finish.**



Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

Linux Empty Application

Linux Hello World

A blank Linux C project.

Figure 5-21: Add An Empty Application

17. Back in the SDK GUI. Right click on the **leds_switches** project and select **Import.**

18. In the Import dialog box, select **General** > **File System** and select **Next.**

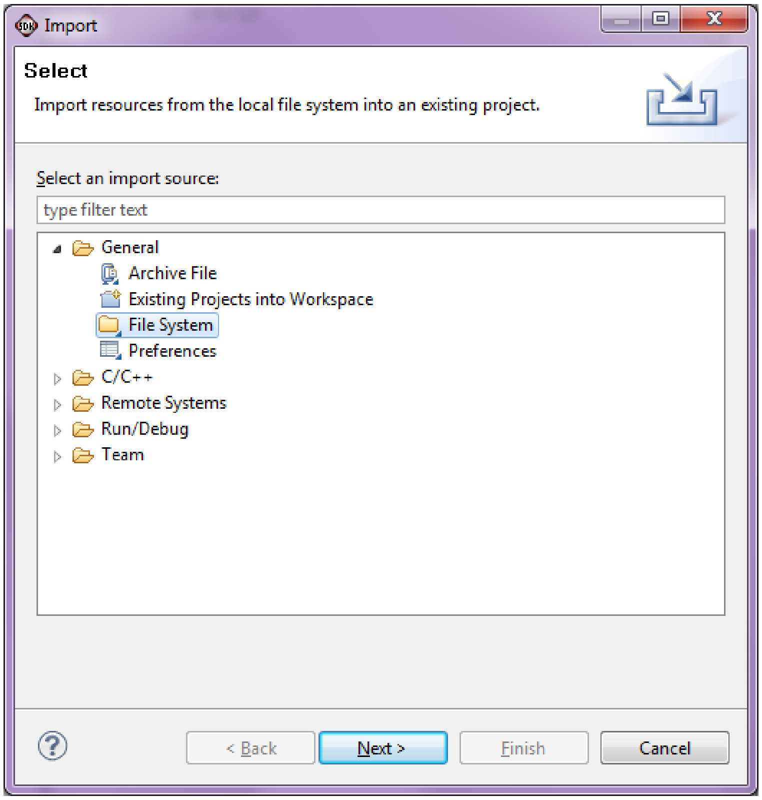


Figure 5-22: Import .C file

19. Browse to the directory in which you saved the files that you downloaded. Select **leds_switches.c** and select **Finish** .

Check that the application is built without errors. Check the message log in the Console window.

Debugging the Linux Application: Using SDK Remote Debugging

We will now launch a debug session to step through the C code. The application will read from the switches and light the corresponding LEDs in a while loop that can be terminated by pressing any button on the ZedBoard directional pad.

1. Right-click leds_switches and select **Debug as** > **Debug Configurations**..
2. In the Debug Configuration wizard, right-click **Remote ARM Linux Application** and click **New**.
3. In the Connection drop-down list, click **New**.
4. The New Connection wizard opens.
5. Click the **SSH Only** tab and click **Next**.
6. In the **Host Name** tab, type the target board IP (it should be 192.168.1.10)
7. Set the connection name and description in the respective tabs.

8. Click **Finish** to create the connection.
9. In the Debug Configuration wizard, under Remote "Absolute File Path for C/C++ Application," click the **Browse** button ... . The Select Remote C/C++ Application File wizard opens.
10. Do the following steps:
    a. Expand the root directory. It opens the Enter Password wizard.
    b. Provide the user ID and Password as **root** and **root** respectively; select the **Save ID** and **Save Password** options.
    c. Click **OK**.

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

    d. Right-click on the "/" in the path name and create a new directory; name it **Apps**.
    e. In the Apps directory, create a new file titled **leds_switches_0.elf**.
    f. Provide an application absolute path, such as **/Apps/leds_switches_0.elf**.
11. Click **Apply**.
12. Click **Debug**. SDK may notify you that it will open the Debug Perspective, click **OK**. The Debug Perspective opens.
13. Turn off the Verbose console mode in the console window.
14. Step through the code or run the code, and watch the messages in the console window. AT the same time, you will notice the values of the Variables in the window on the top left hand side, show the status of the switches and LEDs.
15. The Console window displays the values of the LEDs and Switches.
16. Change the switch settings, notice the LEDs reflect the value of the switches. Stop the program by pressing and holding any of the ZedBoard's directional buttons or the **Terminate** button 🟥 in the SDK GUI.
17. Exit the SDK.

Chapter 6 Further "How-to's" and Examples

Further examples on a variety of ZedBoard topics are explored and explained on ZedBoard.org. In particular, Zynqgeek Blog (http://zedboard.org/zynqgeek) contains helpful step-by-step instructions on several topics in particular:

1. Creating a Custom Peripheral

   This webpage describes how to create a custom peripheral in the Programmable Logic portion of the Zynq-7000 device on the ZedBoard, and to communicate with it via the ARM Processing Subsystem. This design example follows the same steps as described in Chapter 2, starting with creating a PlanAhead project and exporting it to SDK.

2. Talking to a Custom Peripheral

   Once you've created a Custom Peripheral, this webpage explains how to use the Software Template created via the same custom peripheral wizard.

In addition to Zynqgeek's Blog, there are other useful links for the registered ZedBoard user on ZedBoard.org. Here is a sampling of the resources available to the registered members of the website:

3. Building a Zynq Video Design from Scratch

   Leverage the processing and hardware acceleration capabilities of the Zynq SoC in building a HDMI pass-through video design. The latest version is linked above and may require additional mezzanine based hardware to the ZedBoard.

4. Community Projects

   Follow the latest ZedBoard community projects on ZedBoard.org. These projects range from software defined radios to further tutorials to widen your knowledge of Zynq, the Zedboard and Xilinx design tools.

5. Support and Troubleshooting

   There is a very active and vibrant Zynq and ZedBoard community on ZedBoard.org. For help in using the ZedBoard, the Support Forums provide an invaluable community based resource that can be leveraged.

Also helpful are Zynq specific documentation published on the Xilinx website. In particular, these two user guides expand on concepts covered in this document:

6. Zynq-7000 All Programmable SoC Software Developers Guide

   Summarizes the software-centric information required for designing with the Xilinx Zynq-7000 Extensible Processing Platform (EPP) devices.

7. Zynq-7000 All Programmable SoC Technical Reference Manual
   This user guide serves as a technical reference manual for the Zynq-7000 All Programmable SoC (AP SoC).

8. Building the Device Tree Blob (.dtb)
   This link describes the process of compiling a Device Tree Blob.

Appendix A

Application Software

A.1 About the Application Software

The system you designed in this guide requires application software for the execution on the board. This appendix 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.

A.2 Application Software Steps

Application Software comprises the following steps:

Initialize the AXI GPIO module.

1. Set a direction control for the AXI GPIO pin as an input pin, which is connected with BTNU push button on the board. The location is fixed via LOC constraint in the user constraint file (UCF) during system creation.
2. Initialize the AXI TIMER module with device ID 0.
3. Associate a timer callback function with AXI timer ISR.
4. This function is called every time the timer interrupt happens. This callback switches on the LED ‘LD9’ on the board and sets the interrupt flag.
5. The main() function uses the interrupt flag to halt execution, wait for timer interrupt to happen, and then restarts the execution.
6. Set the reset value of the timer, which is loaded to the timer during reset and timer starts.
7. Set timer options such as Interrupt mode and Auto Reload mode.
8. Initialize the PS section GPIO.

9. 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 ‘LD9’ on the board.
10. Set PS Section GPIO channel number 2 pin number 0 to input pin, which is mapped to PL side pin via the EMIO interface and physically connected to the BTNR push button switch.
11. Initialize Snoop control unit Global Interrupt controller. Also, register Timer interrupt routine to interrupt ID '91', register the exceptional handler, and enable the interrupt.
12. 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 ‘LD9’, starts the timer, and tells the function to wait for the Timer interrupt to happen. After the Timer interrupt happens, LED 'LD9" switches ON and restarts execution.

    For more details about the API related to device drivers, refer to the **Zynq-7000 Software Developers Guide** (UG821) linked to in the previous chapter.

A.3 Application Software Code

Below is the source code for **helloworld.c**:

```
/*
 * Copyright (c) 2009 Xilinx, Inc. All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 * helloworld.c: simple test application
 */
#include <stdio.h>
#include "platform.h"
#include "xil_types.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xparameters.h"
#include "xgpiops.h"
```

```
#include "xil_io.h"
#include "xil_exception.h"
#include "xscugic.h"
static XGpioPs psGpioInstancePtr;
extern XGpioPs_Config XGpioPs_ConfigTable[XPAR_XGPIOPS_NUM_INSTANCES];
static int iPinNumber = 7; /*Led LD9 is connected to MIO pin 7*/
XScuGic InterruptController; /* Instance of the Interrupt Controller */
static XScuGic_Config *GicConfig;/* The configuration parameters of the
            controller */
static int InterruptFlag;
extern char inbyte(void);

void Timer_InterruptHandler(void *data, u8 TmrCtrNumber)
{
  print("\r\n");
  print("\r\n");
  print("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\r\n");
  print(" Inside Timer ISR \n \r ");
  XTmrCtr_Stop(data,TmrCtrNumber);
  // PS GPIO Writing
  print("LED 'LD9' Turned ON \r\n");
  XGpioPs_WritePin(&psGpioInstancePtr,iPinNumber,1);
  XTmrCtr_Reset(data,TmrCtrNumber);
  print(" Timer ISR Exit\n \n \r");
  print("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\r\n");
  print("\r\n");
  print("\r\n");
  InterruptFlag = 1;
}

int SetUpInterruptSystem(XScuGic *XScuGicInstancePtr)
{
  /*
   * Connect the interrupt controller interrupt handler to the hardware
   * interrupt handling logic in the ARM processor.
   */
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
      (Xil_ExceptionHandler) XScuGic_InterruptHandler,
      XScuGicInstancePtr);
  /*
   * Enable interrupts in the ARM
   */
  Xil_ExceptionEnable();
  return XST_SUCCESS;
}

int ScuGicInterrupt_Init(u16 DeviceId,XTmrCtr *TimerInstancePtr)
{
  int Status;
  /*
   * Initialize the interrupt controller driver so that it is ready to
   * use.
   */
  GicConfig = XScuGic_LookupConfig(DeviceId);
  if (NULL == GicConfig) {
    return XST_FAILURE;
  }
  Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
      GicConfig->CpuBaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }
  /*
   * Setup the Interrupt System
   */
  Status = SetUpInterruptSystem(&InterruptController);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }
  /*
   * Connect a device driver handler that will be called when an
   * interrupt for the device occurs, the device driver handler performs
   * the specific interrupt processing for the device
   */
  Status = XScuGic_Connect(&InterruptController,
      XPAR_FABRIC_AXI_TIMER_1_INTERRUPT_INTR,
      (Xil_ExceptionHandler)XTmrCtr_InterruptHandler,
      (void *)TimerInstancePtr);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }
  /*
   * Enable the interrupt for the device and then cause (simulate) an
   * interrupt so the handlers will be called
   */
  XScuGic_Enable(&InterruptController, XPAR_FABRIC_AXI_TIMER_1_INTERRUPT_INTR);
  return XST_SUCCESS;
}

int main()
{
  static XGpio GPIOInstance_Ptr;
  XGpioPs_Config*GpioConfigPtr;
  XTmrCtr TimerInstancePtr;
  int xStatus;
  u32 Readstatus=0,OldReadStatus=0;
  //u32 EffectiveAdress = 0xE000A000;
  int iPinNumberEMIO = 54;
  u32 uPinDirectionEMIO = 0x0;
  // Input Pin
  // Pin direction
  u32 uPinDirection = 0x1;
  int exit_flag,choice,internal_choice;
  init_platform();
  /* data = *(u32 *)(0x42800004);
     print("OK \n");
     data = *(u32 *)(0x41200004);
     print("OK-1 \n");
   */
  print("##### Application Starts #####\n\r");
  print("\r\n");
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-1 :AXI GPIO Initialization
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  xStatus = XGpio_Initialize(&GPIOInstance_Ptr,XPAR_AXI_GPIO_1_DEVICE_ID);
  if(XST_SUCCESS != xStatus)
    print("GPIO INIT FAILED\n\r");
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-2 :AXI GPIO Set the Direction
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  XGpio_SetDataDirection(&GPIOInstance_Ptr, 1,1);
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-3 :AXI Timer Initialization
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  xStatus = XTmrCtr_Initialize(&TimerInstancePtr,XPAR_AXI_TIMER_1_DEVICE_ID);
  if(XST_SUCCESS != xStatus)
    print("TIMER INIT FAILED \n\r");
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-4 :Set Timer Handler
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  XTmrCtr_SetHandler(&TimerInstancePtr,
      Timer_InterruptHandler,
      &TimerInstancePtr);
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-5 :Setting timer Reset Value
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  XTmrCtr_SetResetValue(&TimerInstancePtr,
      0, //Change with generic value
      0xf0000000);
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //Step-6 :Setting timer Option (Interrupt Mode And Auto Reload )
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
XTmrCtr_SetOptions(&TimerInstancePtr,
    XPAR_AXI_TIMER_1_DEVICE_ID,
    (XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION ));
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-7 :PS GPIO Intialization
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_1_DEVICE_ID);
if(GpioConfigPtr == NULL)
  return XST_FAILURE;
xStatus = XGpioPs_CfgInitialize(&psGpioInstancePtr,
    GpioConfigPtr,
    GpioConfigPtr->BaseAddr);
if(XST_SUCCESS != xStatus)
  print(" PS GPIO INIT FAILED \n\r");
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-8 :PS GPIO pin setting to Output
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XGpioPs_SetDirectionPin(&psGpioInstancePtr, iPinNumber,uPinDirection);
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, iPinNumber,1);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-9 :EMIO PIN Setting to Input port
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XGpioPs_SetDirectionPin(&psGpioInstancePtr,
    iPinNumberEMIO,uPinDirectionEMIO);
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, iPinNumberEMIO,0);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-10 : SCUGIC interrupt controller Initialization
//Registration of the Timer ISR
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
xStatus=
    ScuGicInterrupt_Init(XPAR_PS7_SCUGIC_1_DEVICE_ID,&TimerInstancePtr);
if(XST_SUCCESS != xStatus)
  print(" :( SCUGIC INIT FAILED \n\r");
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-11 :User selection procedure to select and execute tests
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
exit_flag = 0;
while(exit_flag != 1)
{
  print(" SELECT the Operation from the Below Menu \r\n");
  print("###################### Menu Starts ########################\r\n");
  print("Press '1' to use NORMAL GPIO as an input (BTNU switch)\r\n");
  print("Press '2' to use EMIO as an input (BTNR switch)\r\n");
  print("Press any other key to Exit\r\n");
  print(" ##################### Menu Ends #########################\r\n");
  choice = inbyte();
  printf("Selection : %c \r\n",choice);
  internal_choice = 1;
  switch(choice)
  {
  //~~~~~~~~~~~~~~~~~~~~~~~
  // Use case for AXI GPIO
  //~~~~~~~~~~~~~~~~~~~~~~~~
  case '1':
    exit_flag = 0;
    print("Press Switch 'BTNU' push button on board \r\n");
    print(" \r\n");
    while(internal_choice != '0')
    {
      Readstatus = XGpio_DiscreteRead(&GPIOInstance_Ptr, 1);
      if(1== Readstatus && 0 == OldReadStatus )
      {
        print("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\r\n");
        print("BTNU PUSH Button pressed \n\r");
        print("LED 'LD9' Turned OFF \r\n");
        XGpioPs_WritePin(&psGpioInstancePtr,iPinNumber,0);
        //Start Timer
        XTmrCtr_Start(&TimerInstancePtr,0);
        print("timer start \n\r");
        //Wait For interrupt;
        print("Wait for the Timer interrupt to tigger \r\n");
```

```
        print("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\r\n");
        print(" \r\n");
        while(InterruptFlag != 1);
        InterruptFlag = 0;
        print(" ############################################\r\n ");
        print("Press '0' to go to Main Menu \n\r ");
        print("Press any other key to remain in AXI GPIO Test \n\r ");
        print(" ############################################\r\n ");
        internal_choice = inbyte();
        printf("Select = %c \r\n",internal_choice);
        if(internal_choice != '0')
        {
          print("Press Switch 'BTNU' push button on board \r\n");
        }
      }
      OldReadStatus = Readstatus;
    }
    print(" \r\n");
    print(" \r\n");
    break;
  case '2' :
    //~~~~~~~~~~~~~~~~~~~~~~~
    //Usecase for PS GPIO
    //~~~~~~~~~~~~~~~~~~~~~~~~
    exit_flag = 0;
    print("Press Switch 'BTNR' push button on board \r\n");
    print(" \r\n");
    while(internal_choice != '0')
    {
      Readstatus = XGpioPs_ReadPin(&psGpioInstancePtr,
          iPinNumberEMIO);
      if(1== Readstatus && 0 == OldReadStatus )
      {
        print("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\r\n");
        print("BTNR PUSH Button pressed \n\r");
        print("LED 'LD9' Turned OFF \r\n");
        XGpioPs_WritePin(&psGpioInstancePtr,iPinNumber,0);
        //Start Timer
        XTmrCtr_Start(&TimerInstancePtr,0);
        print("timer start \n\r");
        //Wait For interrupt;
        print("Wait for the Timer interrupt to tigger \r\n");
        print("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\r\n");
        print(" \r\n");
        while(InterruptFlag != 1);
        InterruptFlag = 0;
        print(" ############################################\r\n ");
        print("Press '0' to go to Main Menu \n\r ");
        print("Press any other key to remain in EMIO Test \n\r ");
        print(" ############################################\r\n ");
        internal_choice = inbyte();
        printf("Select = %c \r\n",internal_choice);
        if(internal_choice != '0')
        {
          print("Press Switch 'BTNR' push button on board \r\n");
        }
      }
      OldReadStatus = Readstatus;
    }
    print(" \r\n");
    print(" \r\n");
    break;
  default :
    exit_flag = 1;
    break;
  }
}
print("\r\n");
print("***********\r\n");
print("BYE \r\n");
print("***********\r\n");
```

```
  cleanup_platform();
  return 0;
}
```

Below is the source code for **hello_world_linux.c**:

```
/******************************************************************************
*
*  ZYNQ Linux: Hello world linux example on ZedBoard
*        6/8/12
*
* hello_world_linux.c
*
******************************************************************************/

#include <stdlib.h>
#include <stdio.h>

/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */


int main(void)
{ // main()
   printf("hello world Linux on ZedBoard - Line 1\n");
   printf("hello world Linux on ZedBoard  - Line 2\n");
   printf("hello world Linux on ZedBoard  - Line 3\n");
   printf("hello world Linux on ZedBoard  - Line 4\n");
   printf("hello world Linux on ZedBoard  - Line 5\n");
   printf("hello world Linux on ZedBoard  - Line 6\n");
   printf("hello world Linux on ZedBoard  - Line 7\n");
    return 0;
} // main()
```

Below is the source code for **leds_switches.c**:

```
/* THIS IS THE FILE THAT writes to LEDs and reads from switches..linux
 * Copyright (c) 2012 Xilinx, Inc.  All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE   IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>


int main()
{
        volatile unsigned int *leds;
        volatile unsigned int *switches;
        volatile unsigned int *buttons;

        unsigned int leds_val = 0;
        unsigned int switches_val = 0;
        unsigned int buttons_val = 0;

        int i = 0;

        int fd = open("/dev/mem", O_RDWR|O_SYNC);

        if (fd < 0) {
                printf("Error...");
                exit (1);
        }

    printf("Hello World! \n");
    printf("This program will run an infinite loop. \r\n");
    printf("To end the program, press any of the five directional buttons. \r\n");

        switches = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 
0x41200000);
        printf("Switches mmap Good... \r\n");

        leds = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x41220000);
        printf("LEDs mmap Good... \r\n");

        buttons = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 
0x41240000);
        printf("Buttons mmap Good... \r\n");

        while(1){
                switches_val = *switches;
                buttons_val = *buttons;

                printf("Readback value of the switches is %x \r\n", switches_val);
                *leds = switches_val;

                if (buttons_val != 0){
                        break;
                }

                for (i = 0; i < 99999; i++){}
        }

        printf("BYE! \r\n");
    return 0;
}
```