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

A Hands-On Guide to Effective Embedded System Design

ZedBoard (Vivado 2014.2)

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
09/03/2014	2014.2	Update for Vivado 2014.2.
05/20/2014	2013.4	Update for Vivado 2013.4.
08/06/2013	2013.2	Update for Vivado 2013.2.

Table of Contents

Table of Figures

Chapter 1
Introduction

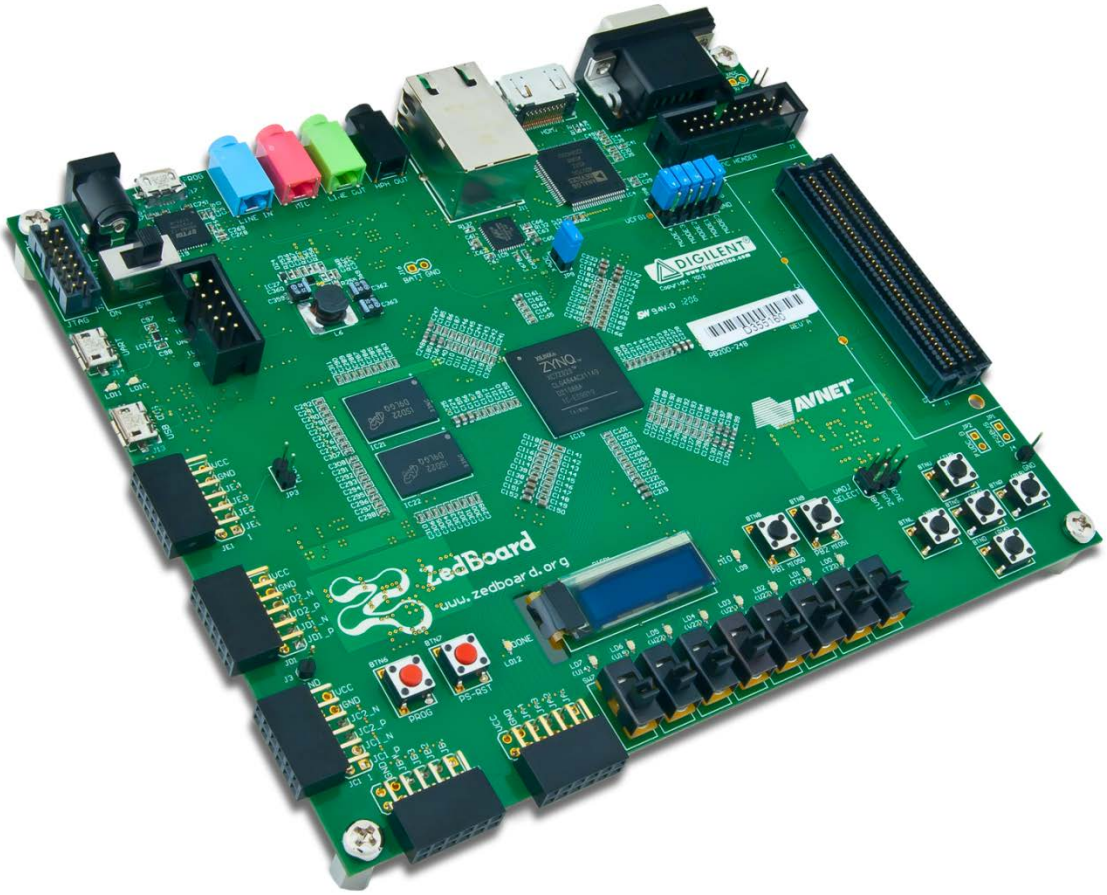


Figure 1-1: The ZedBoard Zynq Evaluation and Development Kit

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 2014.2.

Note: The Test Drives in this document were created using the Microsoft Windows 7 64-bit operating system. Other operating systems may produce 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 the creation of a system utilizing both the Zynq PS as well as the Programmable Logic (PL).
- **Chapter 4**, **"Debugging with SDK and Vivado Analyzer"** provides debugging techniques via software (using the SDK Debug) and hardware (using the Vivado Hardware Analyzer) 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 precompiled Linux images, which are used for booting Linux 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 in the heading above.

1.1.2 Additional Documentation

For further information, please refer to:

- **Xilinx Zynq-7000 Documentation:**
  http://www.xilinx.com/support/documentation/zynq-7000.htm

- **Vivado Design Suite User Guide: Getting Started (UG910):**
  http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug910-vivado-getting-started.pdf

- **ZedBoard.org:**
  http://www.zedboard.org

1.2 How the 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 a bitstream loaded into the Programmable Logic (PL). In order to use any soft IP in the PL, or to route PS dedicated peripherals to device pins for the PL, the PL will need to be configured or programmed.

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 is a free, downloadable, fully featured front-to-back FPGA design solution running under Linux, Windows XP, and Windows 7, with full support for 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 licensing 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 IP Integrator for access to the Xilinx IP catalog (including embedded processor cores), 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/xilinx2014_1/ug892-vivado-design-flows-overview.pdf

IP Integrator

Vivado Design Suite features the IP Integrator which 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 Vivado Design Suite components:

The Vivado installation comes with these other components:

- 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 is a good idea to make sure they are installed properly and that the environments 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 2014.2 of the Vivado Design Suite, and version 2014.2 of the SDK. Most new ZedBoards include a Vivado Design Suite 2014.2 DVD and a license entitlement voucher for the Design Edition installation type.

If your board shipped with a ISE DVD, you should download the single file Vivado 2014.2 installation package via the Download Center on the Xilinx website and choose the Vivado WebPACK + SDK installation option during setup. Most exercises in this document can be run with WebPACK. However to use the Vivado Logic Analyzer portion of this document, you will need the Design Edition installed.

After downloading Vivado 2014.2, make sure to check the MD5 SUM value of the package against the MD5 SUM value listed on the Vivado download page. A mismatch would indicate a corrupted package and you should download the package again.

During installation, please make sure the SDK option is selected so that the SDK is installed along with the rest of the Vivado package.

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 direct connection to a host PC in 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 designs 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.

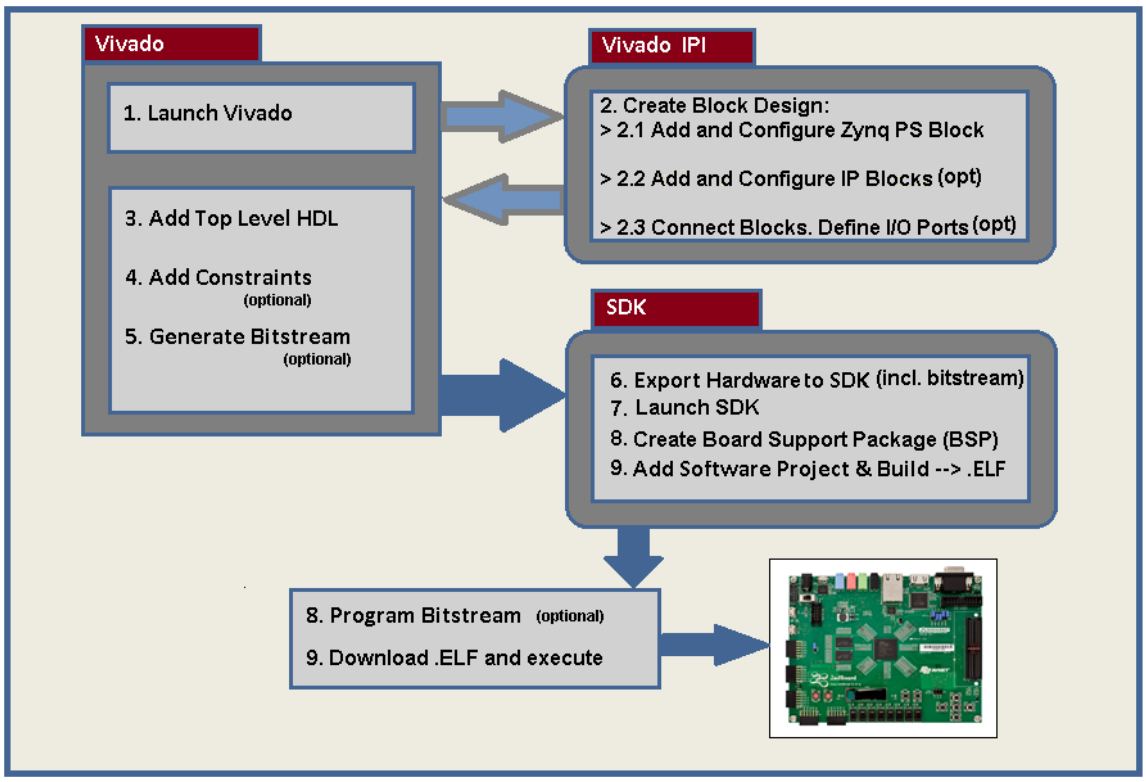


Figure 2-1: Vivado Design Flow for Zynq

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 and add your own customized IP. Connect the different blocks together by dragging signals / nets from one port of an IP to another. You can also use the design automation capability of the IP Integrator to automatically connect blocks together.
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 a supported evaluation board is specified during project creation (i.e.: the ZedBoard) then constraints may not be needed since the software is board aware. 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 error messages. This is done via creation/addition of a Xilinx Design Constraints (XDC) file to 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 any hard peripheral IO were routed through the PL. At this stage, the hardware has been defined in <system>.hdf, and if necessary a bitstream <system>.bit has been generated. The bitstream could be programmed into the FPGA from within Vivado; or it could be done from within SDK.
8. Now that the hardware portion of the embedded system design has been built, export the design to the 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. In order to export the design successfully, the Design Block MUST be open and the implemented design, if exists, MUST be open, otherwise the tools will report an error.Once the hardware has been exported, as a separate step, the SDK can then be launched.

9. Within the 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 *.ELF file is generated.
10. The combination of the optional bitstream and the *.ELF file provides embedded SoC functionality on the 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 generation 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. At the welcome page, click **Next**.
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_1**.
	Project location	Specify the directory in which to store the project files.
	Create Project Subdirectory	Check this option. Click **Next.**
Project Type	Specify the type of sources for your design. You can start with RTL or a synthesized EDIF	Select **RTL Project**. Keep the Do not specify sources at this time box unchecked. Click **Next.**
Add Sources	Do not make any changes on this screen. Click **Next.**	
Add Existing IP	Do not make any changes on this screen. Click **Next.**	
Add Constraints	Do not make any changes on this screen. Click **Next.**	
Default Part	Specify	Select **Boards**.
	Board	Select the **ZedBoard Zynq Evaluation and Development Kit,** make sure the vendor is set to All and Board Rev is set to All. Select the appropriate board revision (C or D). Click **Next.**
New Project Summary	Project summary	Review the project summary before clicking **Finish** to create the project.

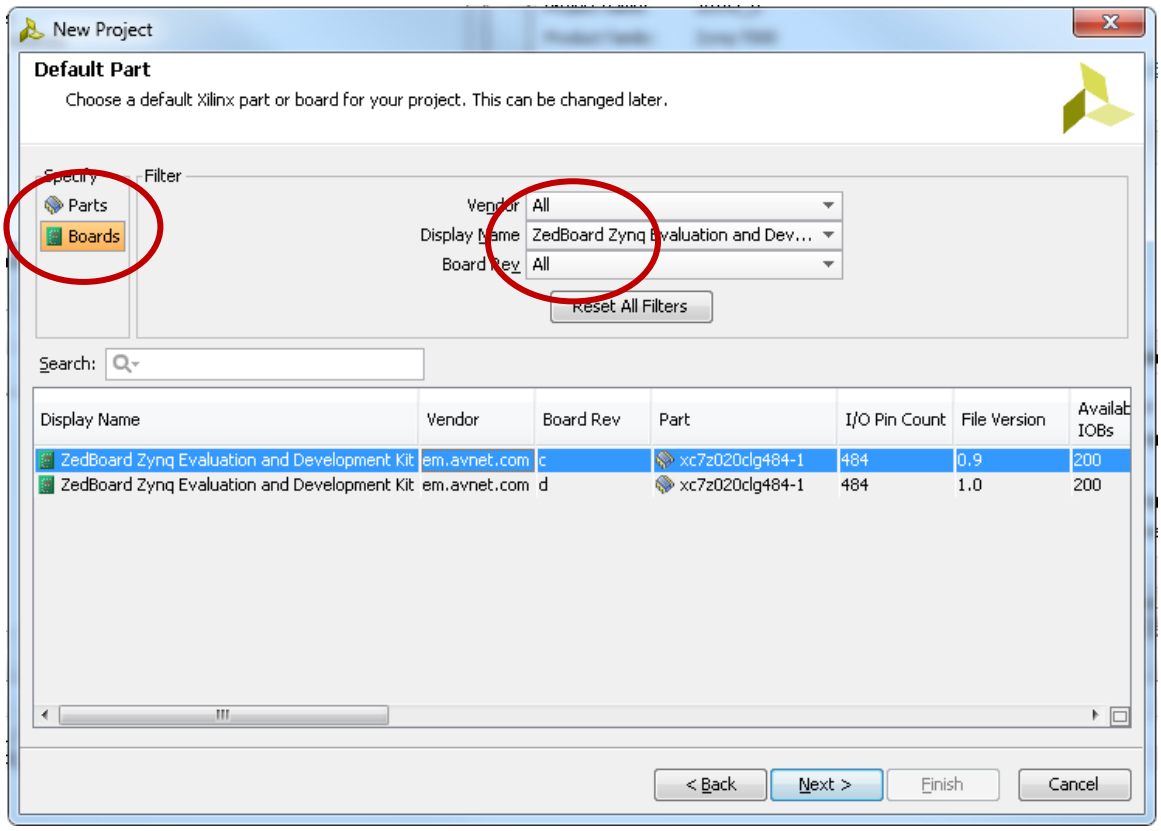


Figure 2-2: New Project Wizard Part Selection

When **Finish** is clicked, the New Project wizard closes and the project you just created opens in Vivado. The board you chose 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 if Run Connection wizard is used. It also applies the correct I/O standard, saving the designer time in doing so. In such case the XDC file related to the pre-defined IO locations are not required from user.

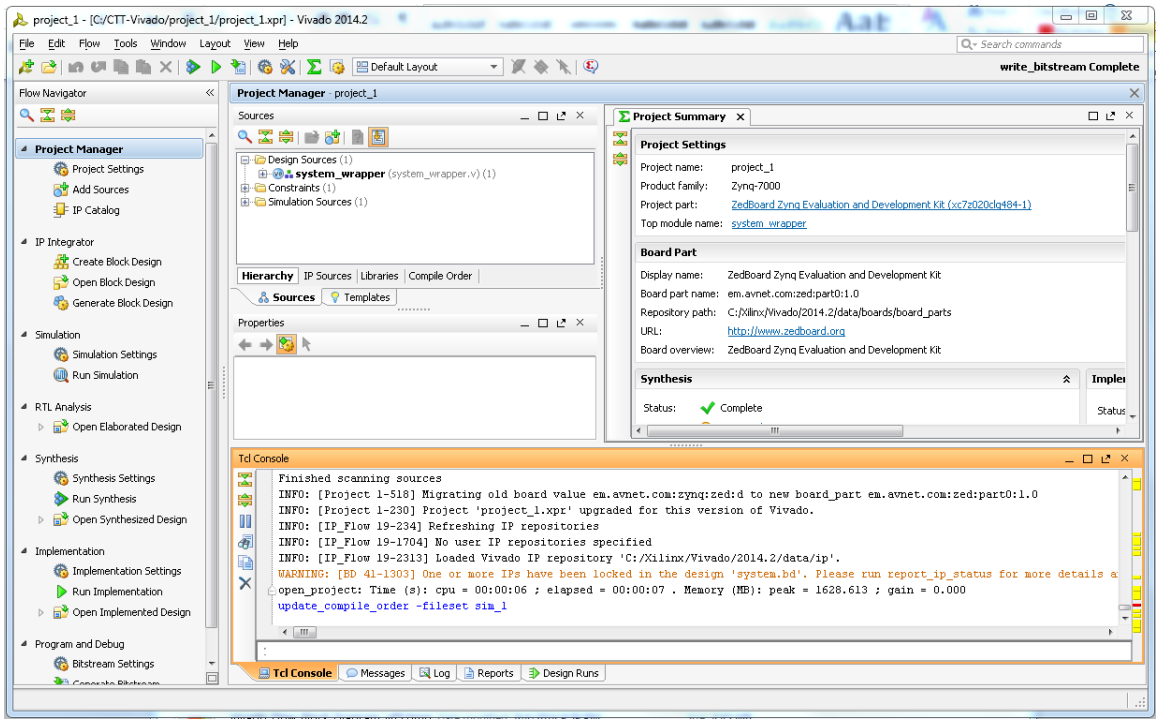


Figure 2-3: The Vivado GUI

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

1. Click **Create Block Design** in the Flow Navigator under IP Integrator.
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, press Ctrl+I keys, or right click on the blank space and select **Add IP**.

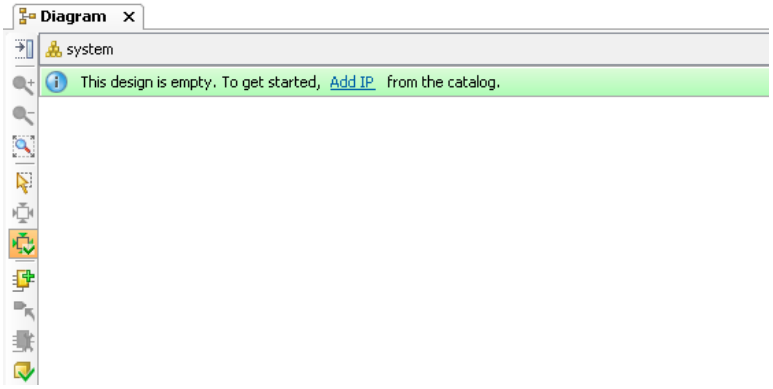


Figure 2-4: Blank Block Diagram view

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

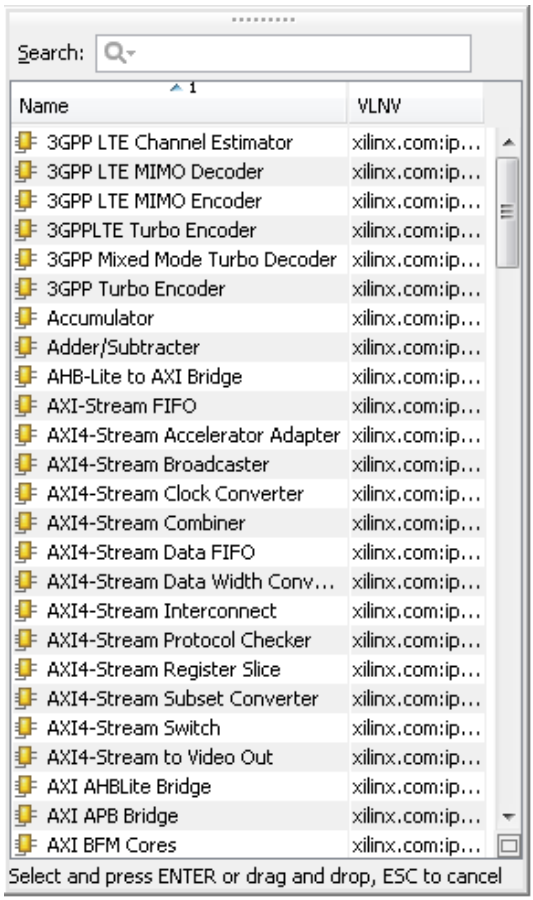


Figure 2-5: IP Catalog

4. Either scroll down to the very bottom or search using the keyword *zynq*, then 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 the ZYNQ7 Processing System block to edit the settings and re-customize the IP.

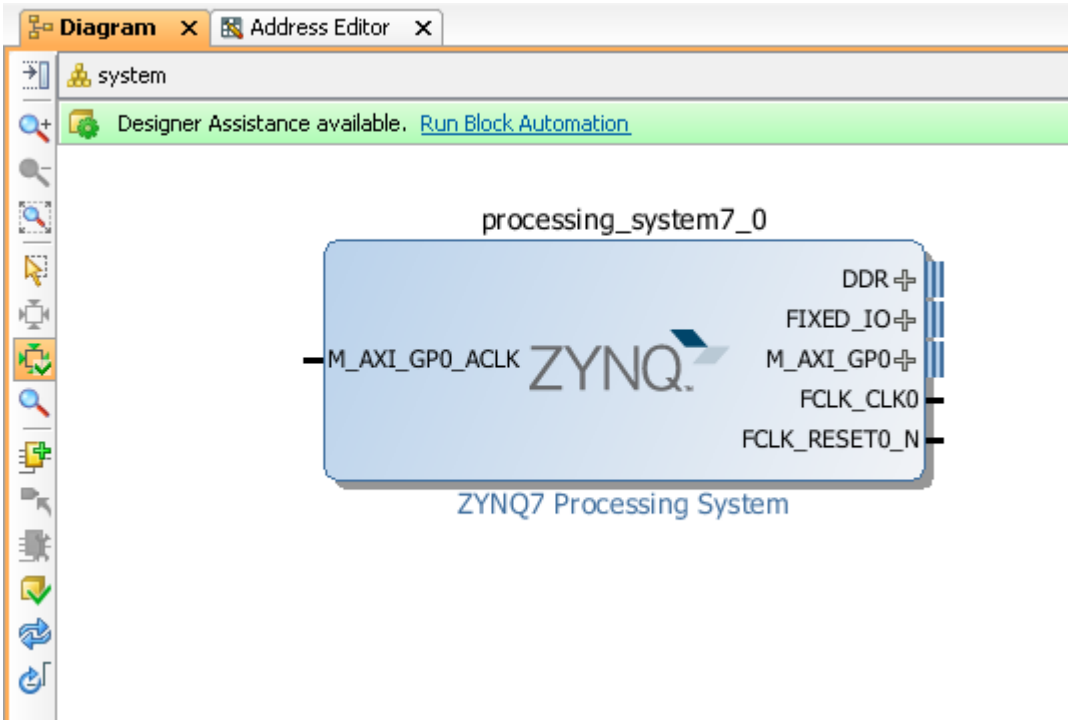


Figure 2-6: Processing System Block in the Block Diagram view

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.

   Notice the presets for other Zynq based boards in the drop down menu. This is a good way to reset the Zynq Processing System (PS) to the default options for the ZedBoard. Do not apply any presets to the block, click elsewhere to exit the drop down menu.

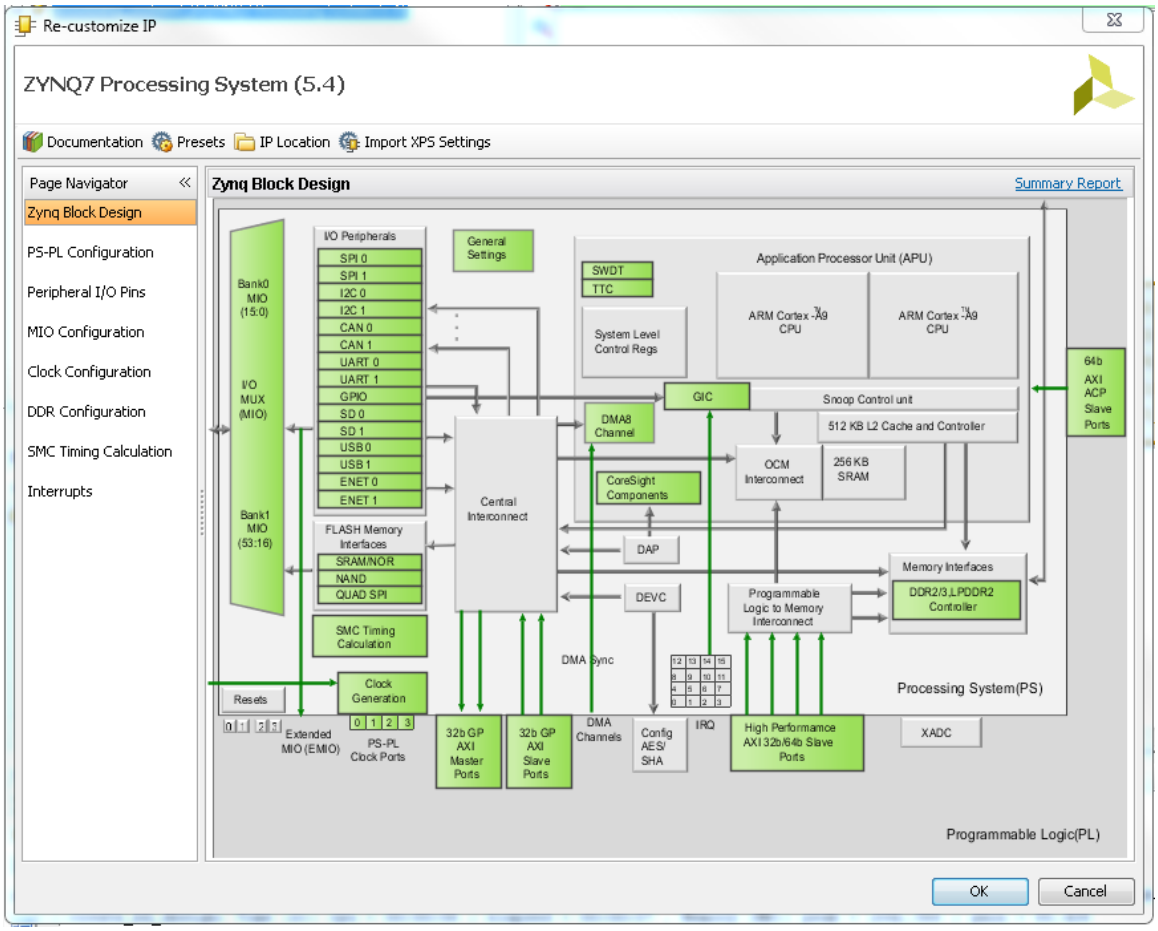


Figure 2-7: Processing System Re-customize IP view

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** select **UART1**.
2. Click on the **Clock Configuration** page, expand **PL Fabric Clocks**, de-select **FCLK_CLK0**.
3. Click on the **PS-PL Configuration** page, expand **GP Master AXI Interface**, de-select **M AXI GP0 interface**.
4. Also within the PS-PL Configuration page, expand **General > Enable Clock Resets**, de-select **FCLK_RESET0_N**.
5. Click on **DDR Configuration**, de-select **Enable DDR**. By disabling DDR, you are limited to applications of less than 128 KB which can be run from the On-Chip Memory (OCM).

6. 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.

7. Click **Run Block Automation** Run Block Automation in the green information bar. Select **/processing_system7_0** make sure **Apply Board Presets** is unchecked and select **OK** in the Run Block Automation window (leave everything else as default). Notice that the FIXED_IO are now connected to output ports.
8. Click the **Save** button to save the current block diagram.

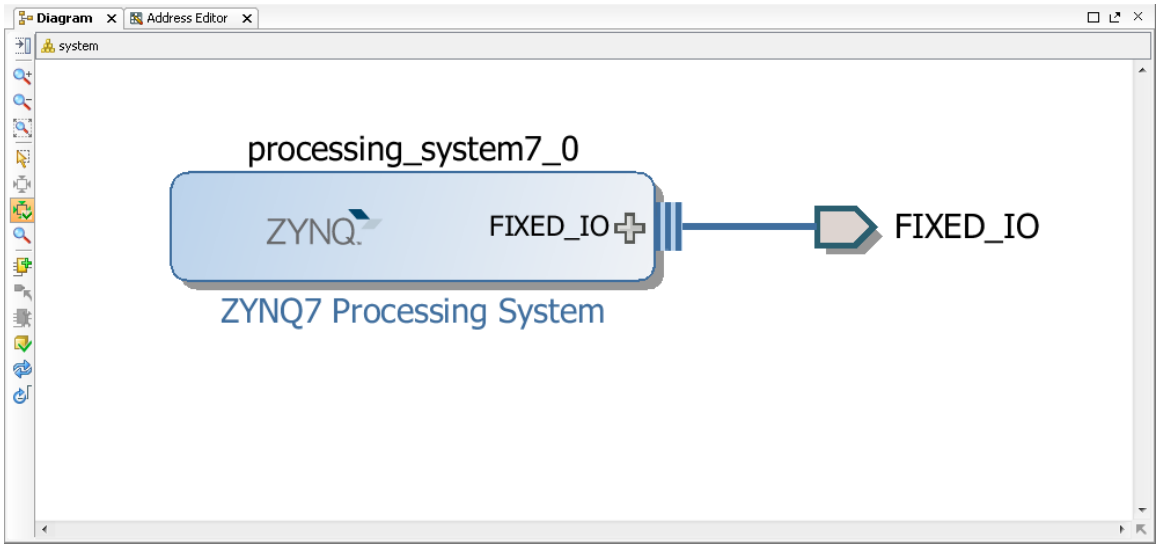


Figure 2-8: Processing System 7

9. Select **Tools > Validate Design,** there should be no warnings or errors.
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. Make sure you select the **Let Vivado manage wrapper and auto-update** option to generate the Verilog file in the sources pane while letting Vivado update the HDL when you change the block diagram design. Click **OK**.

    Notice that *system_wrapper.v* (or .vhd, depending on the HDL type selected during project definition) got created and placed at the top of the design sources hierarchy.
11. Click on **Generate Block Design** in the Flow Navigator to generate the block design. Click **Generate** in the Generate Output Products window.

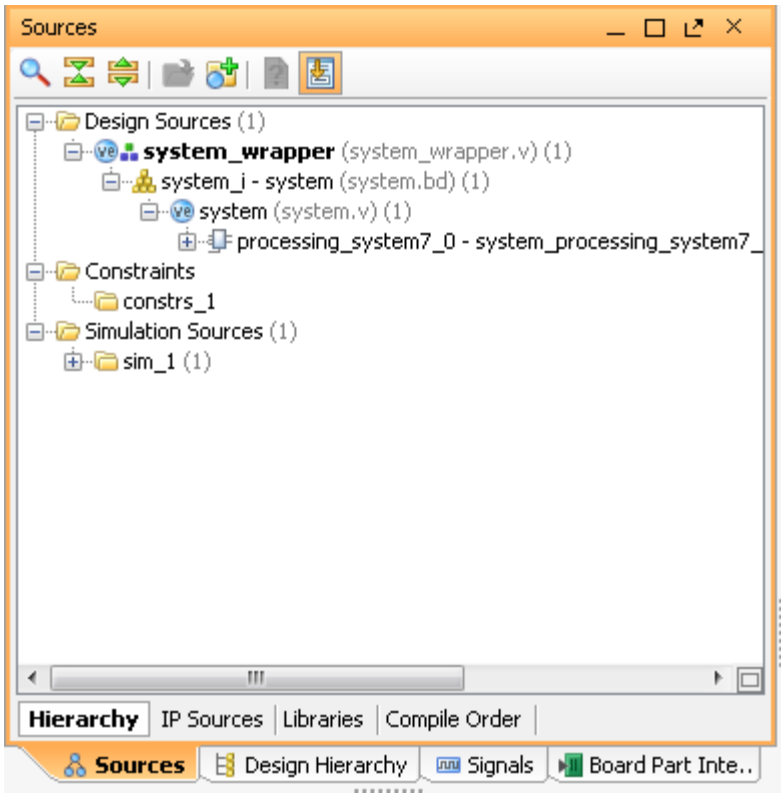


Figure 2-9: Sources pane showing the system.bd file

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.

12. Click **File > Export > Export Hardware**. Notice the Include Bitstream box is not available as synthesis/implementation/bitstream generation has not been run.
13. The Export Hardware dialog box opens. Click **OK** to continue.
14. Click **File > Launch SDK**.
15. The Launch SDK dialog box opens. Click **OK** to continue.

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

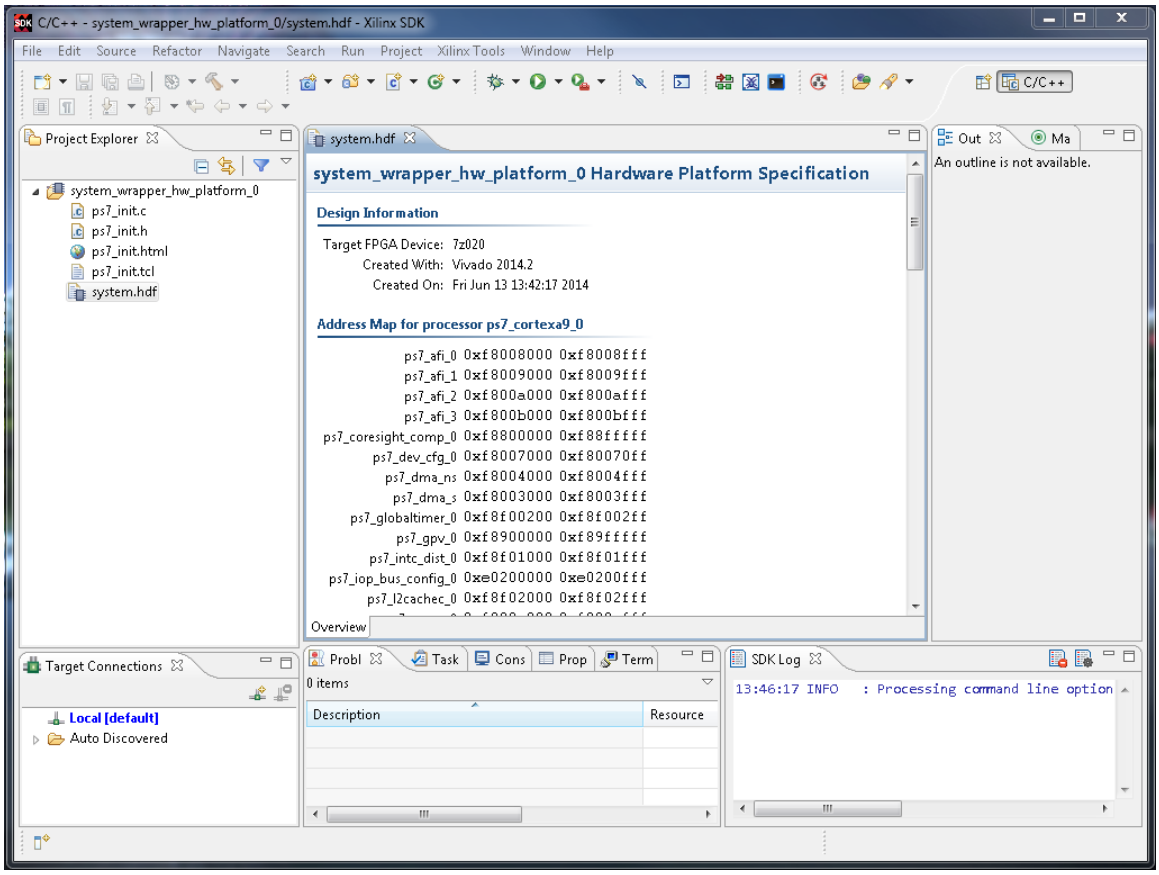


Figure 2-10: The SDK GUI

The hardware description file is automatically read in. The system.hdf tab shows the address map for the entire system.

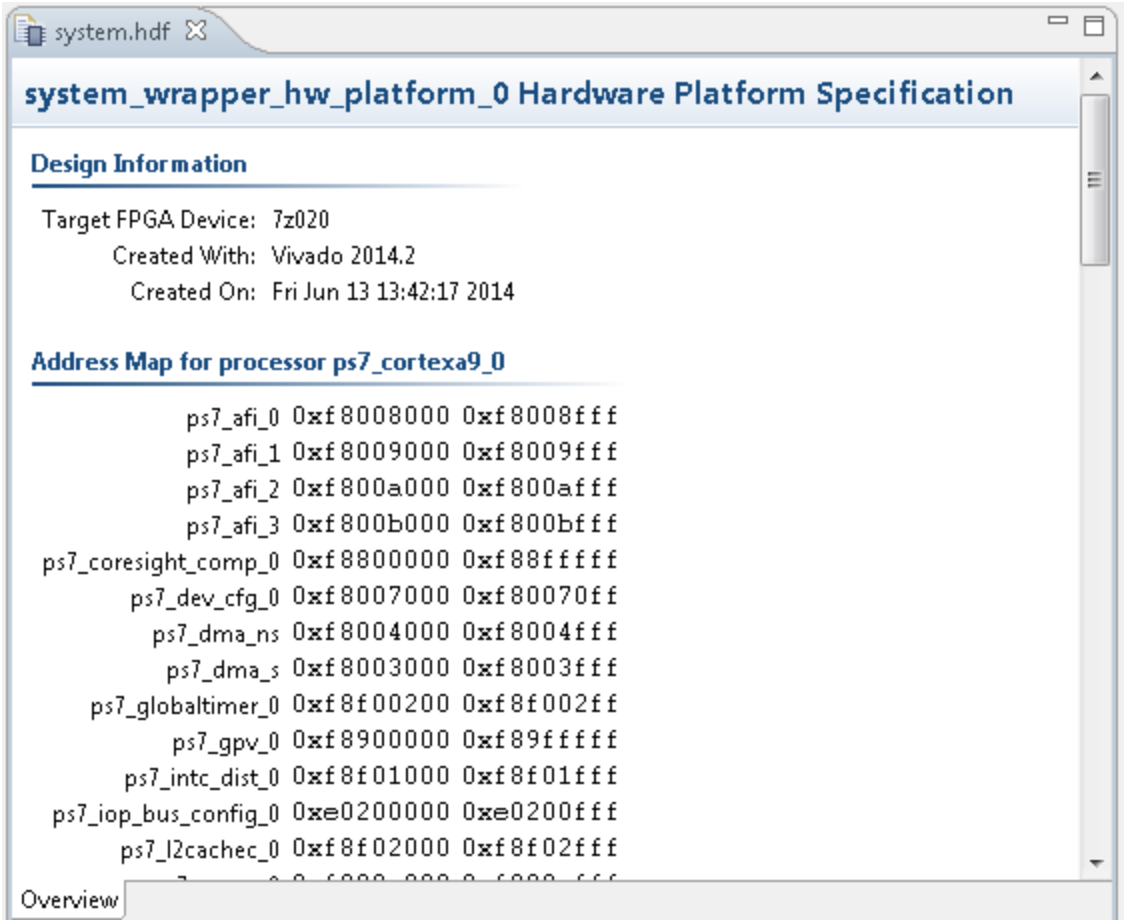


Figure 2-11: Address Map in SDK System.hdf Tab

What Just Happened?

The Vivado design tool exported the Hardware Platform Specification of your design (system.hdf in this example) to the SDK. In addition to system.hdf, 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.hdf 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 files by using them to initialize the processing system so that applications can be run using the configured processing system peripherals.

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.

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

They should be set to 0 0 0 0 0.

---

4. Power-on the board using the switch indicated in Figure 2-12: ZedBoard Power switch and Jumper settings. Note, in some instances the board needs to be ON before the SDK launches in order for the SDK to see which COM port is being used by the OS.

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/1521

To edit JTAG detection settings, use the **Target Connections** view 

. The view appears in the lower left hand corner of the SDK GUI. If it is not there, select **Window > Show View > Target Connections**.

If it is necessary to edit the JTAG cable frequency, simply right click on the **Local [Default]** icon  in the Target Connections window and select **View**. Once the Target Connection Details window is open, click **Advanced >>** to view the JTAG options.

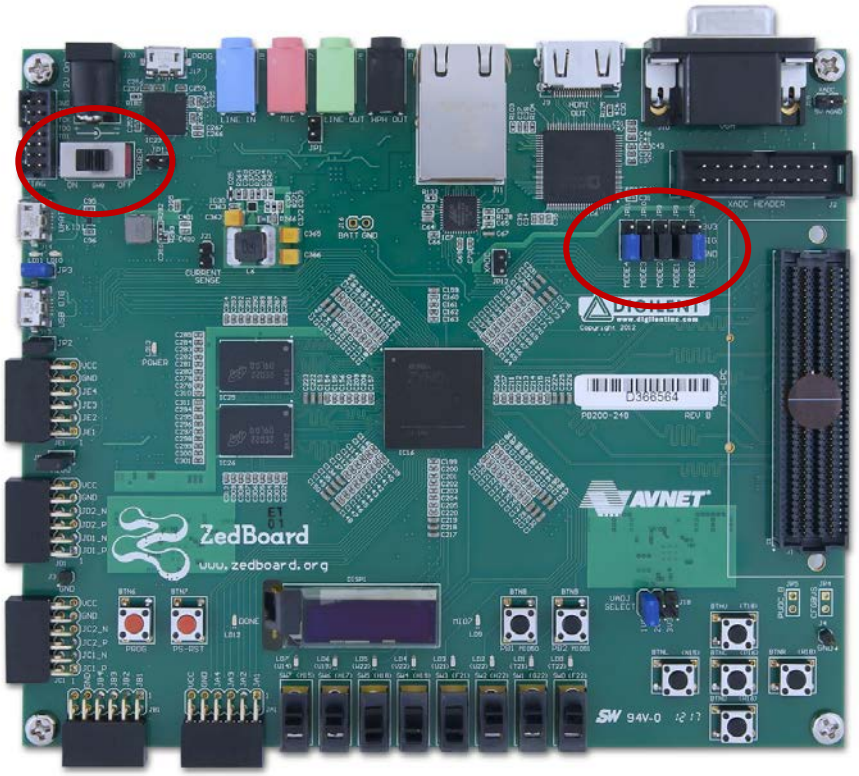


Figure 2-12: ZedBoard Power switch and Jumper settings

Click on the Terminal tab 

by the Console to open a serial communication utility for the COM port assigned on your system. To configure the terminal settings, click on the little notepad icon  . Select the connection type as **Serial**.

Configure it with the parameters as shown below (replacing COM3 with the appropriate COM port number [the COM port will always be 3 or higher] from your computer, verify using **Control Panel > Device Manager**).

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.

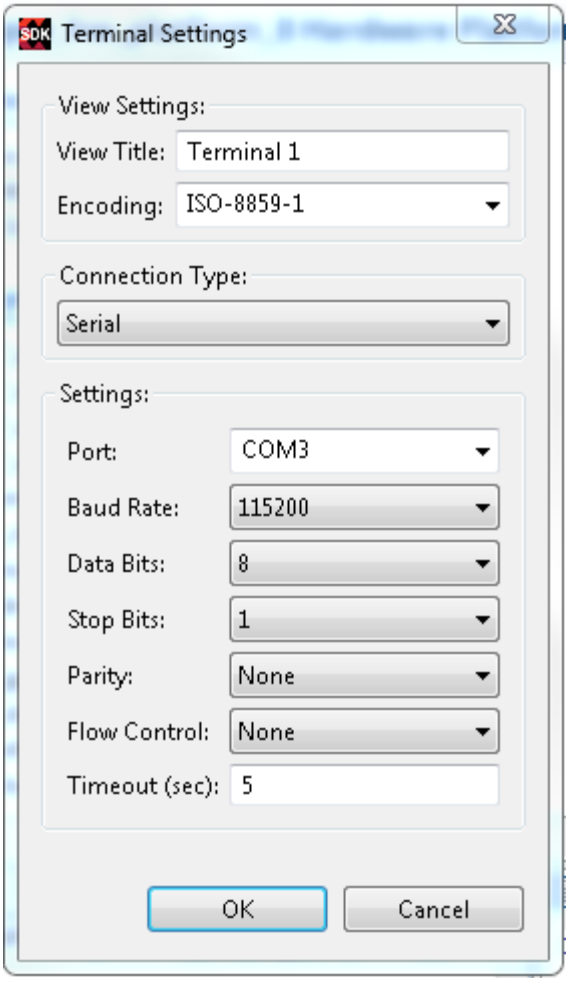


Figure 2-13: Serial Terminal Settings

5. In SDK, select **File > New > Application Project**.
6. 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_wrapper_hw_platform_0
	Processor	ps7_cortexa9_0
	OS platform	standalone
	Language	C
	Board Support Package	**Create New** : Hello_World_bsp
Click **Next**		
Templates	Available Templates	Hello World

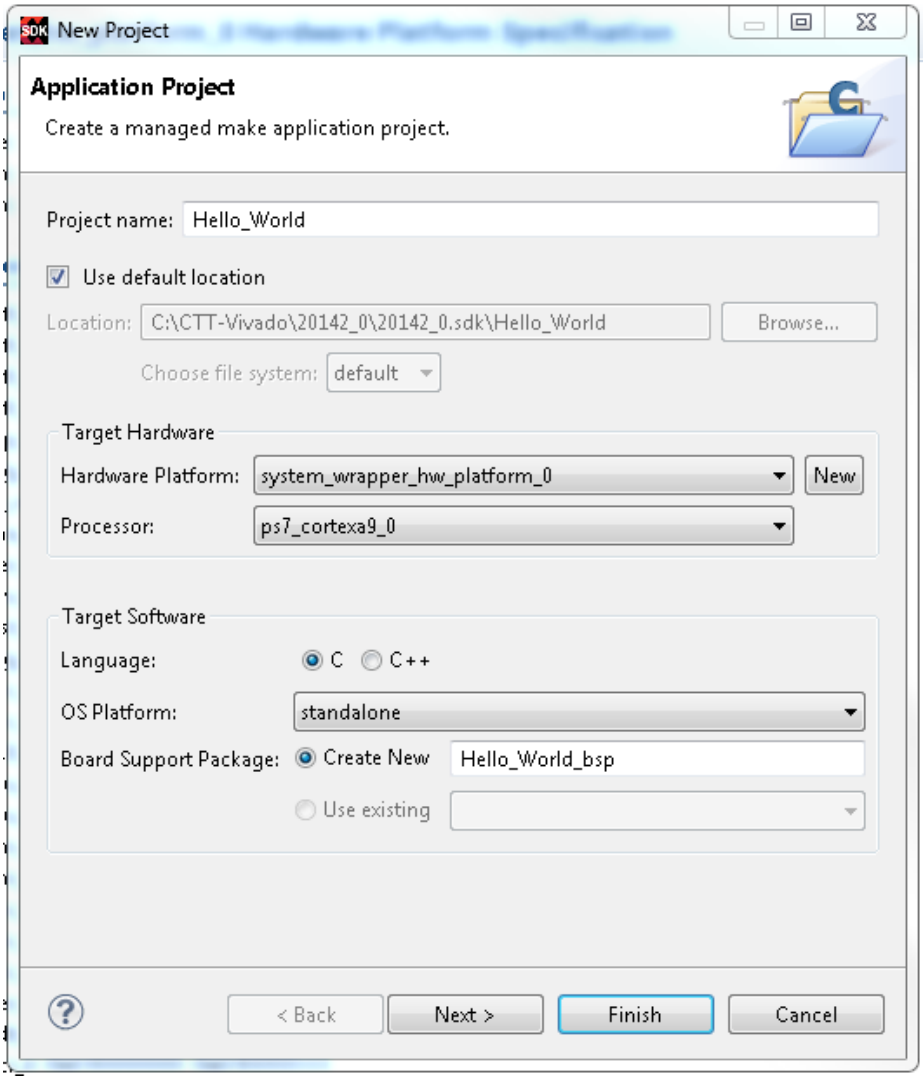


Figure 2-14: New Application Project Wizard

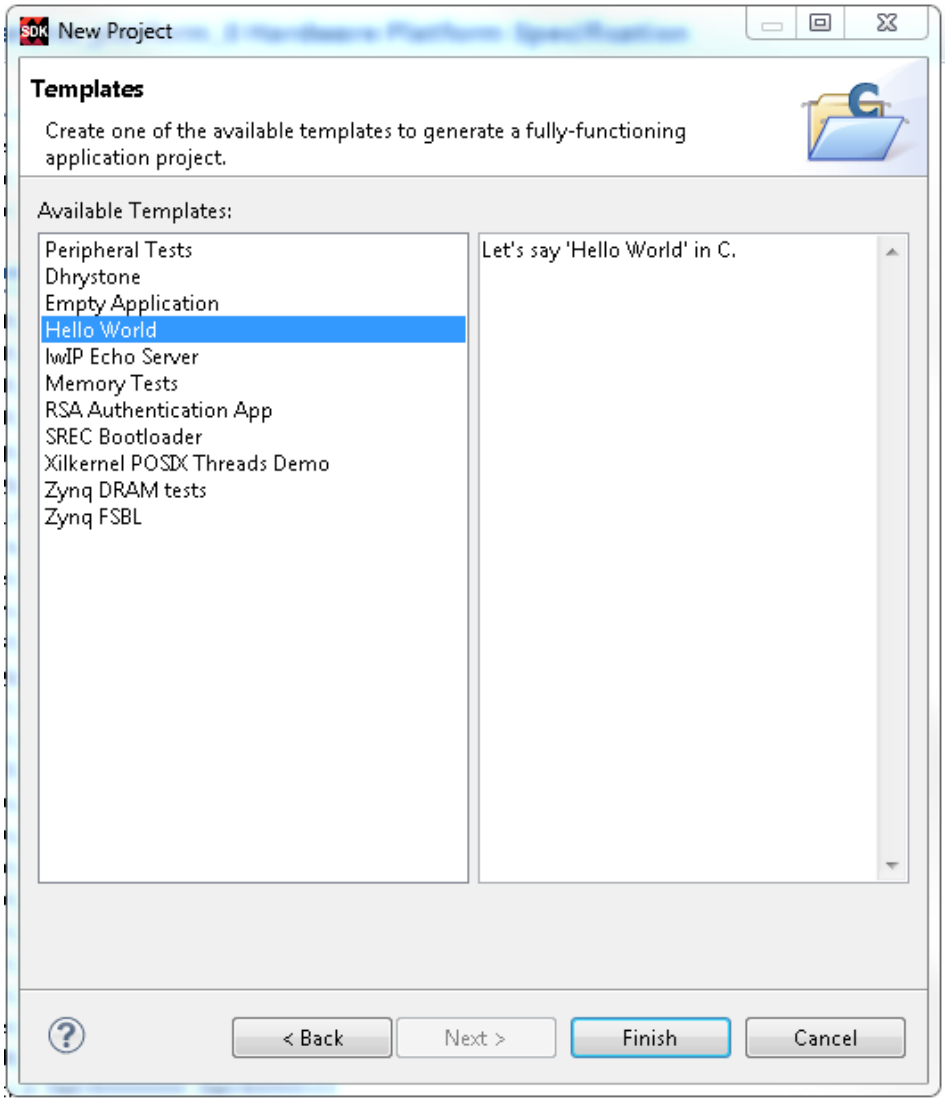


Figure 2-15: Hello World from Available Templates

7. Complete running the new New Application Project Wizard and create the new Application Project by clicking **Finish**.

By doing so, the Hello_World application project and Hello_World_bsp BSP project gets 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.

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

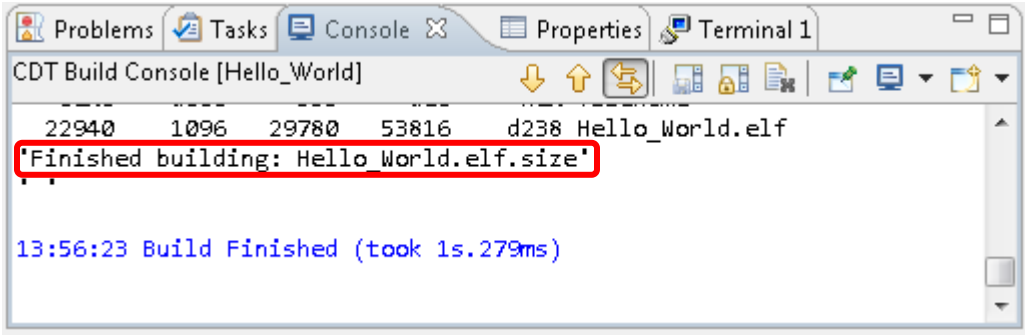


Figure 2-16: Successful Build

9. The application and its BSP are both compiled and the *.ELF file is generated.
10. Right-click **Hello_World** and select **Run As > Run Configurations**.
11. Right-click **Xilinx C/C++ Application (GDB)** and click **New**.
12. The new run configuration is created and named **Hello_World Debug**.

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

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.

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.

13. Leaving all the settings as is, click **Run**.
14. **"Hello World"** appears on the serial communication terminal.
15. Select **File > Exit** to close the SDK.

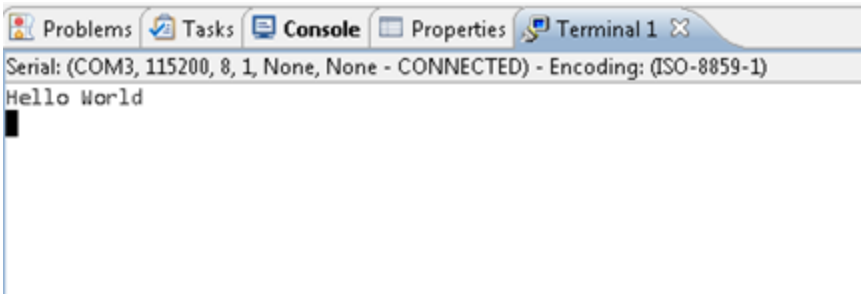


Figure 2-17: "Hello World" on the Serial Terminal

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 automatically for the ZedBoard.

Chapter 3
Embedded system design using the Zynq Processing System and Programmable Logic

One of the unique benefits 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.

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.

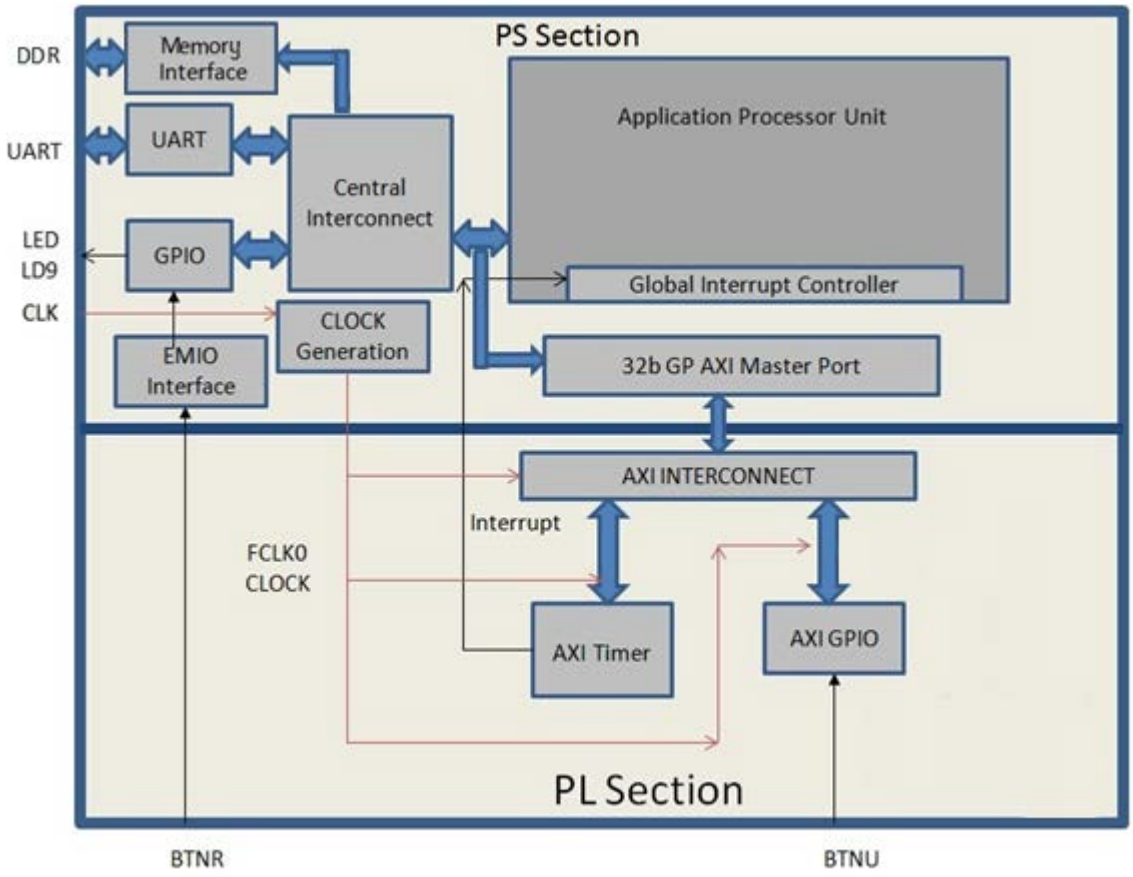


Figure 3-1: System Design Overview

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 turns 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 will 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)**. Since you have previously generated a wrapper HDL file for the block diagram, the block diagram now resides under the system_wrapper file.

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). Double click on the **ZYNQ7_Processing_System** block diagram to edit the block parameters.

2. Click on the **Presets** button near the top of the window, select the **ZedBoard** template and click **OK** to exit the window. This will reset the Zynq PS to default ZedBoard settings. Notice the block inputs and outputs have changed dramatically.

3. Click on the **Add IP** button  . From the IP catalog, double-click **AXI GPIO** to add it. Notice the block has been added to the Block Diagram.

4. Again, click the **Add IP** button and double click on the **AXI Timer** IP to add it to the block diagram.

5. Hit refresh  to allow the software to rearrange the blocks for you.

Now that some of the necessary 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 branch under Fabric Interrupts and expand **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 **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 **OK**.

10. 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.

Connect blocks in the design using the Connection Automation feature in Vivado Run Connection Automation.

11. Click on **Run Connection Automation** in the green bar at the top of the block diagram screen. Select /**axi_gpio_0/S_AXI**. Leave everything as default and click **OK**.

    Notice the connection was made between the processing_system7_0 block and the axi_gpio_0 block. Also notice the AXI Interconnect and the Processing System Reset blocks have been inserted.

12. Repeat the step above for the /**axi_timer_0/S_AXI**.

When not using the Run Connection Automation feature in Vivado, 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.

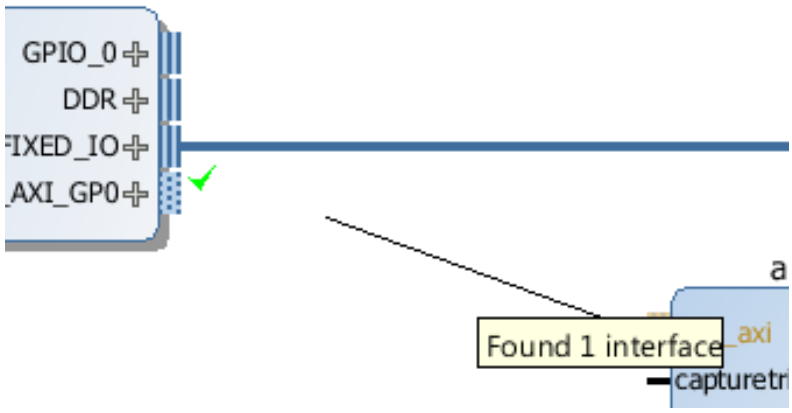


Figure 3-2: Connecting ports on two IP blocks

Note: The figure above does not represent the design being implemented.

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

13. Connect the **IRQ_F2P[0:0]** port on the processing system block to the **interrupt** port on the AXI Timer block.
14. 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 and that the width is **1-bit** (do not check Create Vector). Click **OK** to close the window.
15. 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 and that the width is **1-bit** (do not check Create Vector). Click **OK** to close the window.
16. Connect **BTNR** to the Processing System block's **GPIO_I[0:0]** port (expand GPIO_0 on the processing_system7_0 block.)
17. Connect **BTNU** to the AXI GPIO block's **gpio_io_i[0:0] port** (expand GPIO on the axi_gpio_0 block.)
18. Click on **Run Block Automation**, select **processing_system_7_0**. **Uncheck** Apply Board Preset. Click **OK**.

    Notice that the DDR and FIXED_IO have been pinned out and made external.

Refresh the block diagram to rearrange the blocks. Your connections should be similar to Figure 3-3:

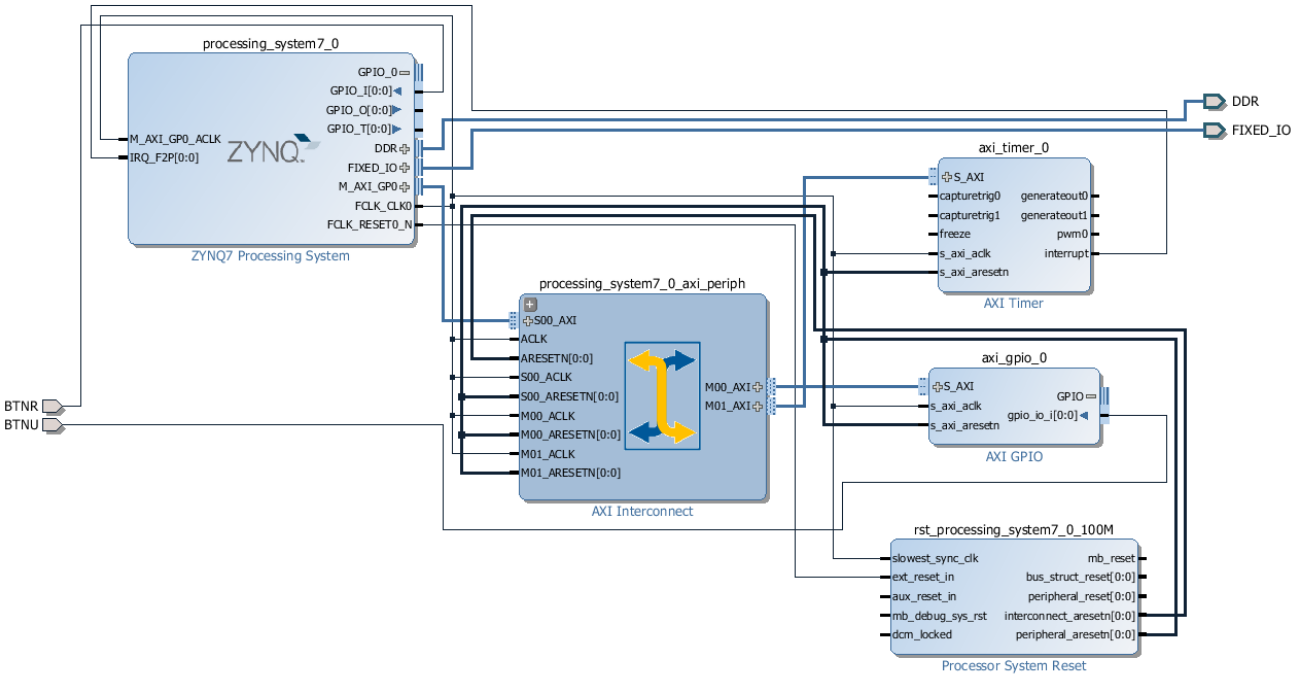


Figure 3-3: Completed Port Connections

19. Click on the Address Editor tab 

. Make sure the **axi_timer_0** and **axi_gpio_0** have Offset Addresses assigned to them (they should not be under Unmapped Slaves). If not, click on the **Automatically Assign Addresses** button  to populate the empty fields. You should have axi_timer_0 assigned **0x42800000** and axi_gpio_0 assigned **0x41200000**.

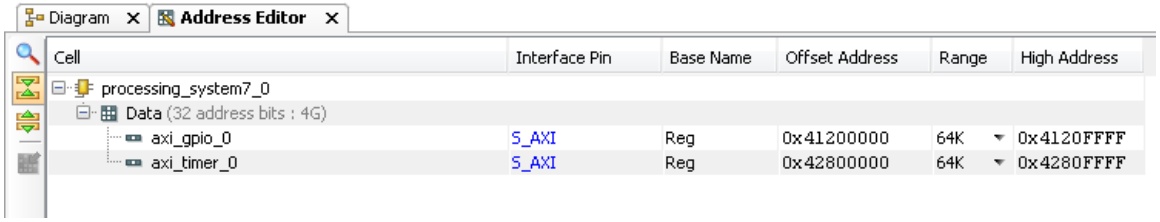


Figure 3-4: Assigned peripheral memory addresses

20. Run **Tools > Validate Design**. Validation will be successful with no warning or error messages. Click **OK** to close the pop-up box.
21. Right click on the **system_i** block diagram (system.bd) in the Sources pane and select **Create HDL Wrapper**. Make sure you let Vivado manage the HDL output when the dialog box appears and click **OK**.
22. Click **Generate Block Design** and then click **Generate**.
23. Click the **Add Sources** button in the Flow Navigator. Select **Add or Create Constraints**. Click **Next**.
24. 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.xdc**. Click **OK** to exit the sub-window and make sure the constraints are copied into the project. Click **Finish** to complete the process.

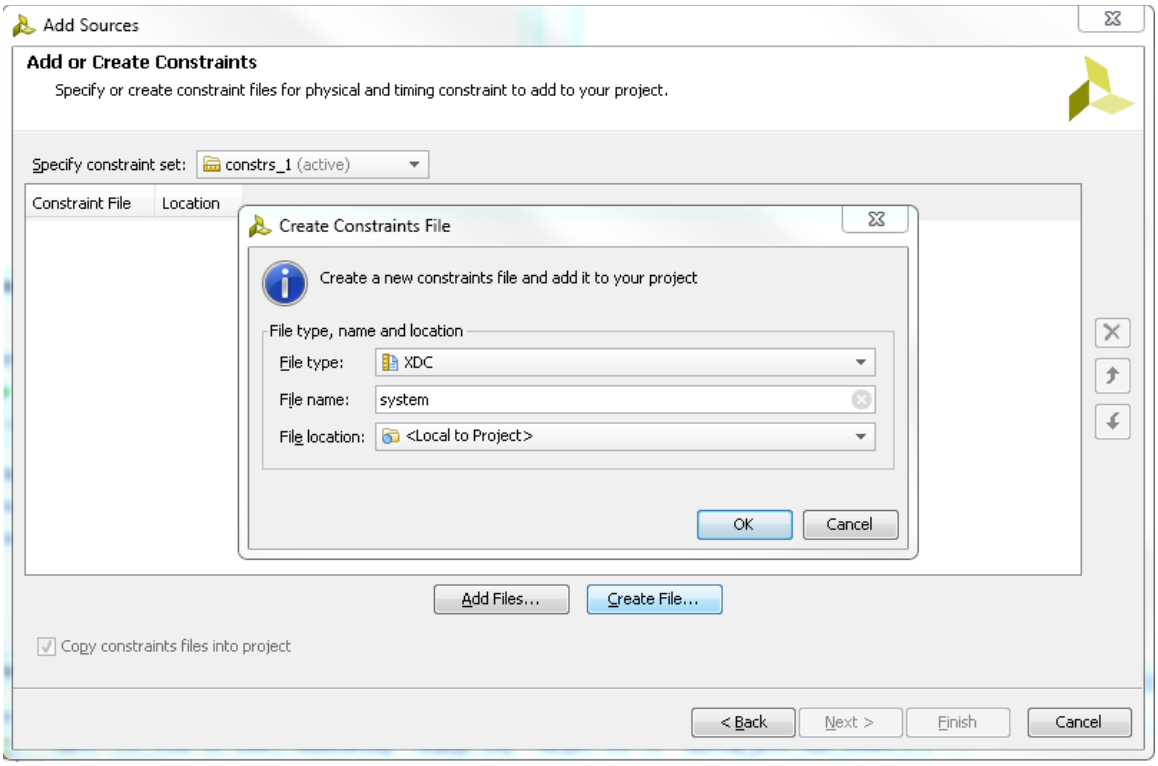


Figure 3-5: Adding a constraints file

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

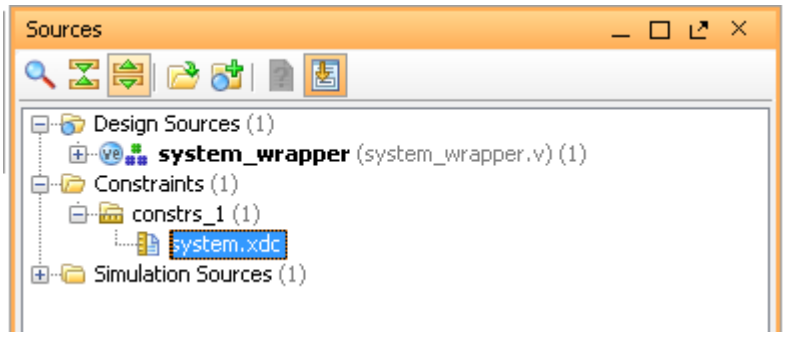


Figure 3-6: Constraints file added into project

25. Double click on **system.xdc** and add the following location constraints (copy and paste) to the empty file:

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

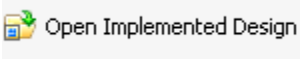
set_property IOSTANDARD LVCMOS25 [get_ports BTNR]
set_property IOSTANDARD LVCMOS25 [get_ports BTNU]

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.

26. **Save** the edited constraints file.
27. In the Program and Debug list in the Flow Navigator pane, click **Generate Bitstream**. This step is necessary since PL resources will be used in the project and will need to be implemented in hardware before launching the SDK. Save the block diagram when prompted.

    A dialog box will appear to ask you to save the project, make sure the **Block Design – system** is checked and hit **Save**. Another box will appear to warn you that synthesis is not run on the updated files, click **Yes** to run synthesis. Generating the bitstream will invoke the entire implementation process after synthesis, click **Yes** to run implementation as well if prompted. Synthesis may take a while to complete.
28. After the Bitstream generation completes select **Open Implemented Design** in the dialog box and click **OK.** If you do not get this dialog box, click on the Open Implemented Design button in the Flow Navigator bar

    .

    This allows you to get a graphical overview of the PL resource usage and routing.
29. Make sure the block diagram view has been opened before launching SDK. If the block diagram is not open, double click on system_i in the Design Sources hierarchy.
30. Connect and power-on the ZedBoard (you should at least power cycle it if the board has been on from the previous exercise).

    Export Hardware the making sure that the “Include Bitstream” option is selected. For this design, since there is a bitstream generated for the PL, the bitstream will also be exported to SDK.
31. **Launch SDK**.

3.1.2 Take a Test Drive! Working with SDK

SDK launches with the "Hello World" project you created with the Standalone OS 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. Create a new application project called **Hello_World_Prompt**. Leave everything else as default, click **Next**.
2. Select **Hello World** in the next screen and click **Finish**.
3. Open the **helloworld.c** file (under **Hello_World_Prompt > Src** in the Project Explorer pane) and modify the application software code. Refer to **Appendix A, Application Software** for the application software details. You can copy and paste the code from Appendix A into helloworld.c.
4. Make sure you **save** the modified helloworld.c file.
5. Open the serial communication utility with baud rate set to **115200**.
6. Right click on the Hello_World_Prompt application and select **Run As > Run Configurations….**
7. Right click on **Xilinx C/C++ application (GDB)** and select **New**. A new configuration named Hello_World_Prompt_Debug is created.
8. Because you have a bitstream for the PL, you must download the bitstream. To do this, change **Reset Processor** to **Reset Entire System**. Notice the **Program FPGA** option becomes available, check the box next to it. Pay attention to the summary of applications next to it where the PL will be programmed.
9. Leave everything as default, hit **Apply**. After that click **Run** to run the application.

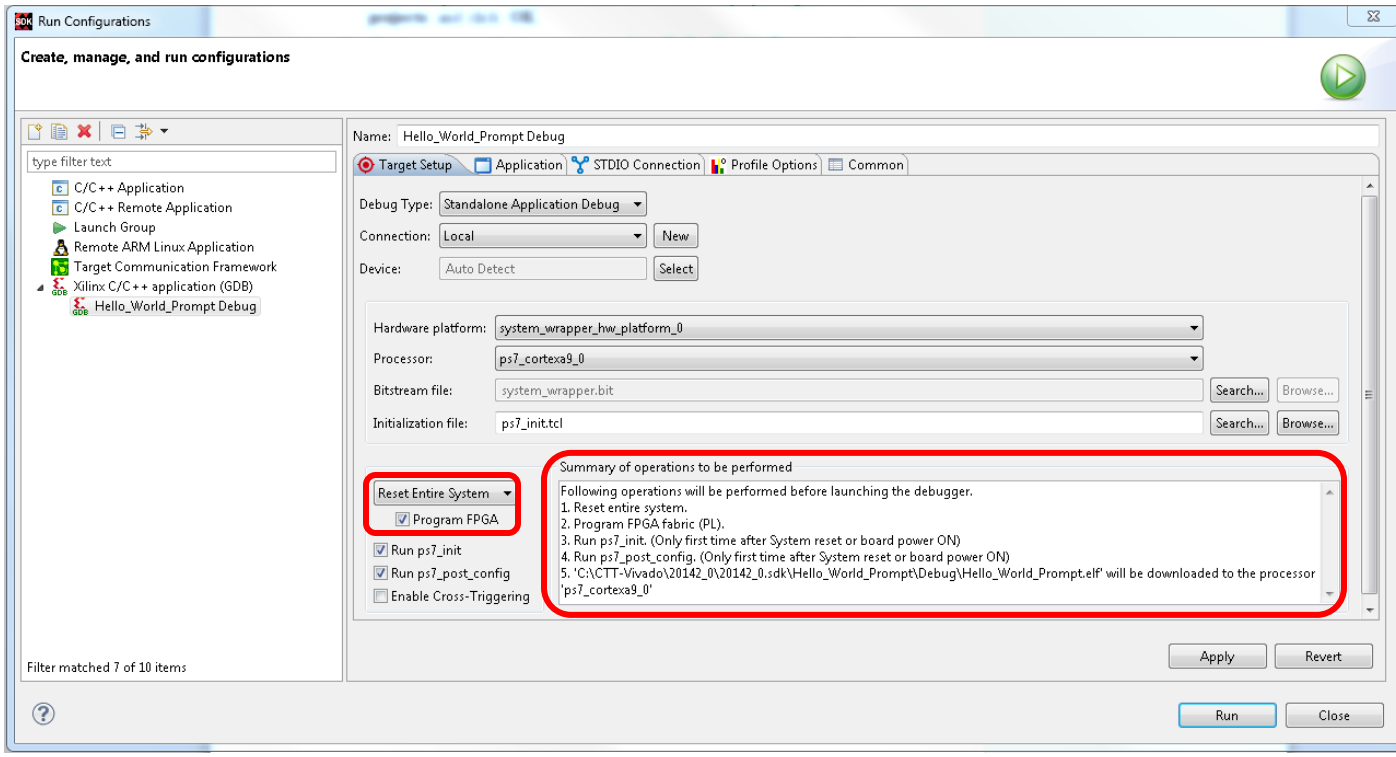


Figure 3-7: Run Configurations menu with the Program FPGA operation checked

10. The blue DONE LED will light up. Pay attention to the serial terminal window.

    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.

11. Follow the instructions shown on the serial terminal to run the application by pressing the keys mentioned in the terminal window. After you are satisfied with the application, **do not** exit the application just yet (for the exercises in the next chapter).

12. When you are satisfied, **do not** close the SDK window.

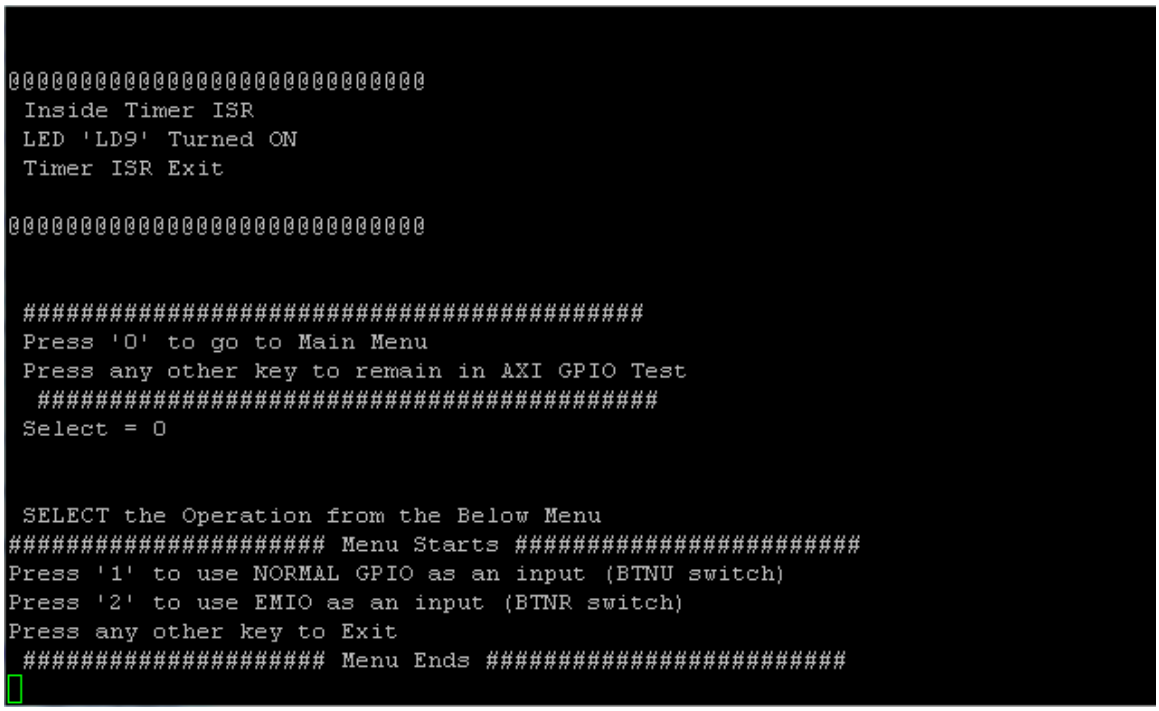


Figure 3-8: Terminal showing application output

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 the SDK

First you will try debugging with software using SDK. This step assumes the PL is still configured from the previous chapter. The serial terminal window should be open from the previous session/exercise. **Do not** power-cycle the ZedBoard!

1. In the C/C++ Perspective, right-click on the **Hello_World_Prompt** Project and select **Debug As > Debug Configurations**. The window automatically loads the Hello_World_Prompt_Debug screen. Check that settings are correct for your debug operation (it should be the same as your run configuration).
2. Click **Debug**.
3. A warning will appear asking to terminate the previous launch, click **Yes**.

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

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

Figure 4-1: Debug Perspective Suspended

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

The program counter is currently sitting at the beginning of main() with program execution suspended at address 0x0010078c. You can confirm this information with the Disassembly view, which shows the assembly-level program execution also suspended at 0x0010078c.

Note: If the 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.

5. Select the Registers view to confirm that the program counter, pc register, contains **0x0010078c**. You may need to expand Main to see the registers.

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

6. Double-click in the blue 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**.

7. 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 **0x001007b4**.

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

Program execution suspends at location **0x00100dc8**. The call stack is now two levels deep.

9. Select **Run** > **Resume** again to run the program to conclusion by hitting any other key (*Enter* for example) 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.

10. Re-run your code several times by again following the steps from the beginning of this section (you may need to return to the C/C++ perspective). Experiment with single-stepping, examining memory, changing breakpoints, modifying code, and adding print statements. Try adding and moving views.
11. After you are satisfied with the design behavior, you may power-cycle the ZedBoard.
12. **Do not** close the SDK window!

4.2  Take a Test Drive! Debugging hardware using the Vivado Logic Analyzer tool

This section is only applicable if you have the Logic Analyzer license 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 Manager 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 Manager is open.

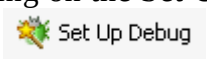
Go back to the Vivado window (it should still be running; do not close the SDK window). Recall that Synthesis and Implementation have ran successfully to generate the bitstream.

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. You may ignore the warning messages when the synthesized design is loading.

2. Expand the **system_i > Nets** branch. This shows you a list of nets in the top most level of the design.
3. Find **processing_system7_0_axi_periph_M01_AXI_ARVALID**, right click and select **Mark Debug**. 

When prompted, confirm by clicking **OK** to debug the net.
4. Repeat the above steps for the following signals:
    - **processing_system7_0_axi_periph_M01_AXI_ARREADY**
    - **processing_system7_0_axi_periph_M01_AXI_AWADDR (5-bits wide bus)**
    - **processing_system7_0_axi_periph_M01_AXI_WDATA (32-bits wide bus)**
5. Launch the **Set Up Debug** wizard by clicking on the Set Up Debug button in the Flow Navigator under Synthesized Design 

.
6. Click **Next** in the first page.
7. Notice the nets selected for Debug are shown. Click **Next**.
8. Leave the **ILA (Integrated Logic Analyzer) General Options** window options as defaults and click **Next**.
9. Click **Finish** to exit the wizard. This creates the necessary debug cores for the design.
10. Select **File > Save Constraints**. Click **OK** to acknowledge the warning about synthesis going out of date. Save to **system.xdc** and click **OK.**

    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.
11. Click on the **Design Runs** tab. Notice despite Vivado's warning about synthesis going out of date, there still is a green check mark next to the operation (indicating it is up-to-date).

    If synthesis is out of date, right click on the **synth_1** row and select **Force-Up-to-Date** since the source design was not modified. This forces Vivado to keep the synthesized results up to date.

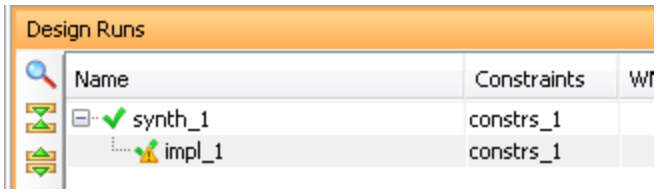
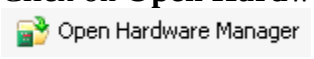


Figure 4-2: Synthesis is up-to-date

12. Run implementation and re-generate the bitstream by clicking on **Generate Bitstream** under Program and Debug in the Flow Navigator. Click **Yes** to re-run implementation.
13. Once the bitstream has been generated, click on **Open Implemented Design** in order to port the bitstream to SDK later on. You should click **Yes** to close the synthesized design to view the implemented design.

    Make sure the ZedBoard is powered-on (if you haven't already done so, please power-cycle the board).
14. Click on **Open Hardware Manager** under Program and Debug.

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

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

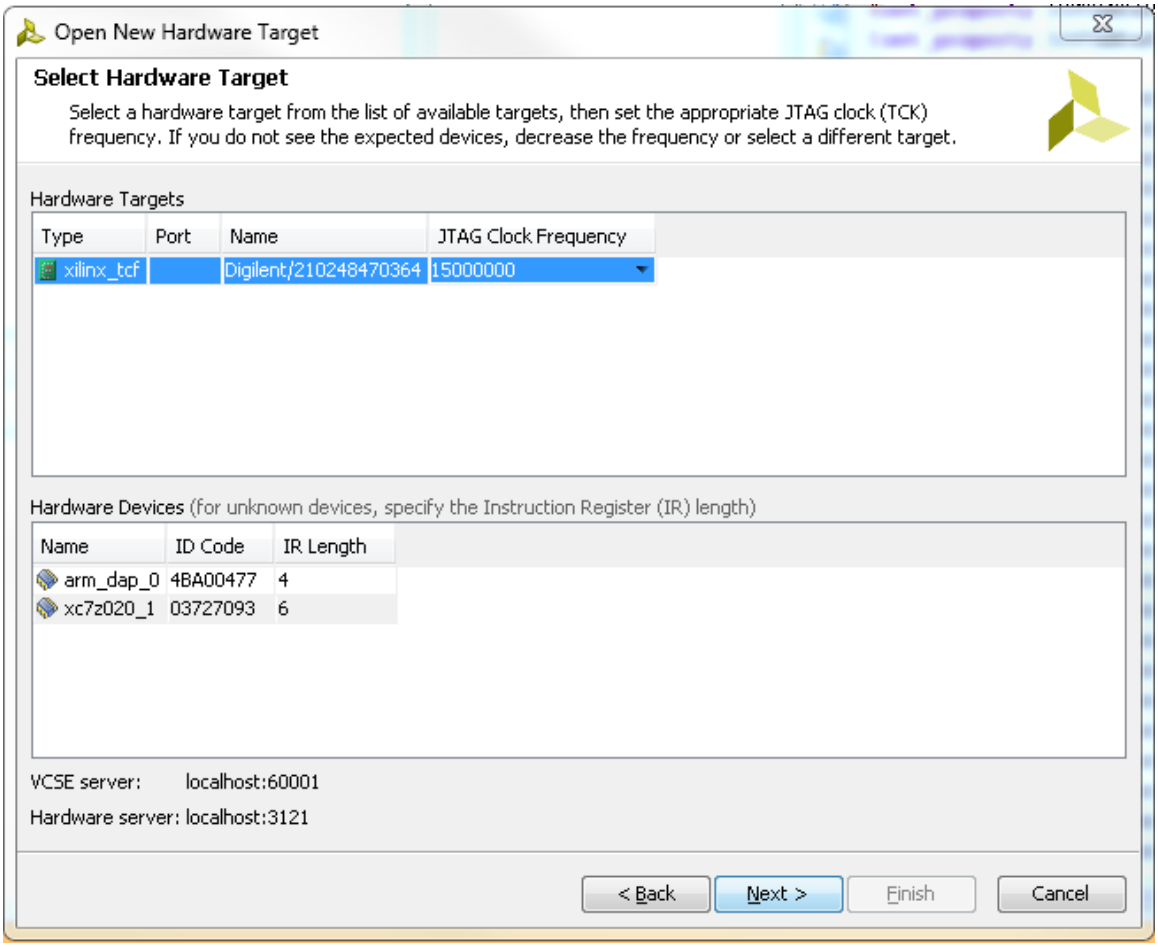


Figure 4-3: Open New Hardware Target screen

15. Click **Next** to exit the welcome screen. Go with the default Vivado Local server option, click **Next** again.
16. Click **Next** again at the Select Hardware Target screen. Keeping the JTAG clock frequency at 15000000 Hz.
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.

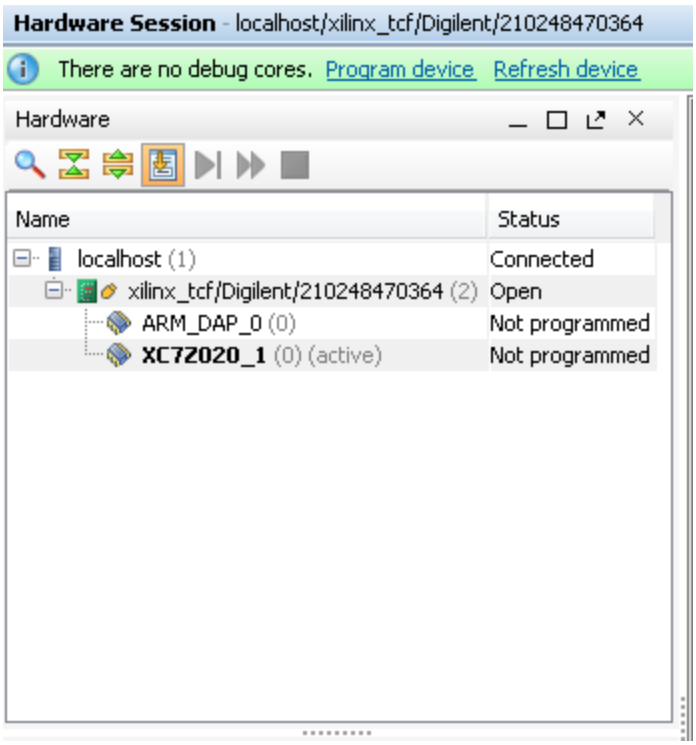
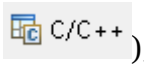


Figure 4-4: Identified devices in Hardware Manager

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

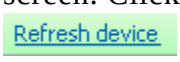
    After the device is programmed, notice the blue DONE LED lighting up on the ZedBoard. Vivado may complain that the clock cannot be found in the debug core. This is not a problem since the software design is not yet running. The next step is to run the design in SDK (which should not have been closed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

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

20. Go to the SDK window that is still running (you will need to be in the *C/C++* perspective 

), run the Hello_World_Prompt design again. Right click on the Hello_World_Prompt application project, click **Run As > Run Configurations**. Since the target device has been programmed already, **uncheck** the Program FPGA box. and select **Reset Processor** from the drop down menu. Click **Apply** and then 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 the Vivado window which should be showing the Hardware Manager screen. Click **Refresh Device** in the green information bar at the top.
    

    Select the **XC7Z020_1** target device.

Notice in the Debug Probes tab next to the debug window signals that were previously marked for debug.

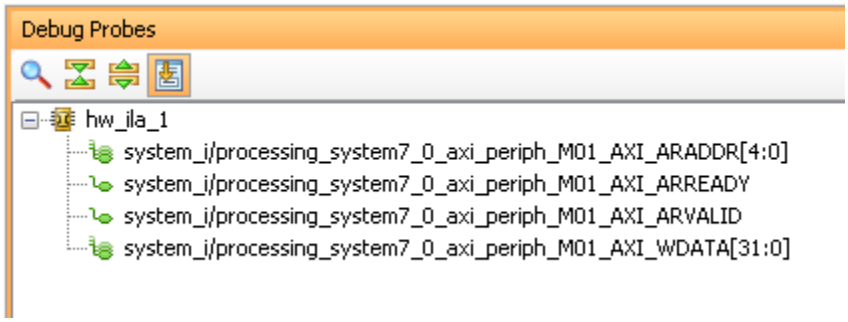


Figure 4-5: Debug Probes available for the waveform view

23. Drag **ARVALID** into the empty **Basic Trigger Setup** window to the right. This specifies ARVALID as a trigger signal.
24. In the **Basic Trigger Setup** window, set the **Compare Value** (trigger value) to **1** (1-bit logic '1') by selecting the drop down menu == [B] X. Click **OK.**

    This means the waveform will only be captured when ARVALID is triggered at logic '1'.
25. Click the **Run Trigger** button. This causes the ILA to look for a logic '1' on **ARVALID**. Notice the hw_ila_1(ILA) being shown as Waiting For Trigger Waiting Fo... .
26. Now press BTNU on the ZedBoard. Notice hw_ila_1 is first Full and now Idle Idle .
27. The waveforms will be captured in the hw_ila_data_1.wcfg window.

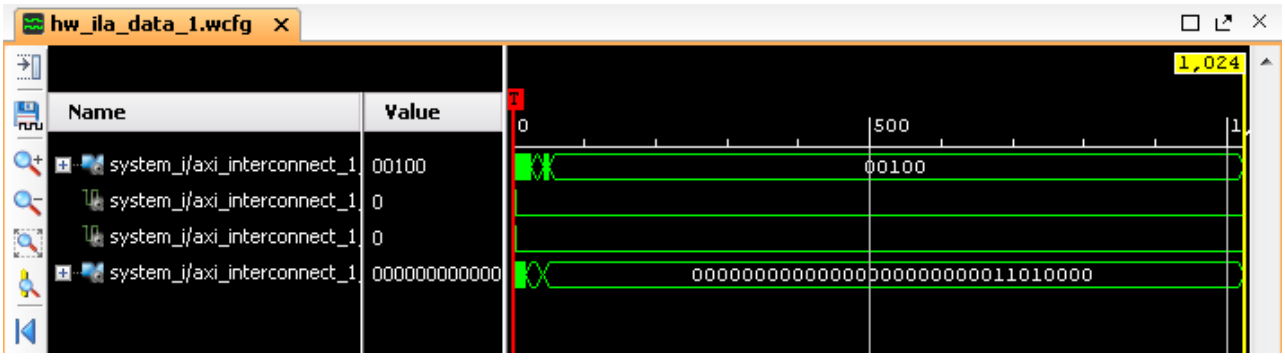


Figure 4-6: Captured waveforms from the triggered run

28. 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, **Terminate**  the application program in the SDK, **exit** the serial terminal, **exit** the Hardware Manager (click **OK**, no need to save changes to the *.WCFG file). Power **OFF** the board.

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 7 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 zip file 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
- **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
- **u-boot.elf**: U-Boot file used to create the BOOT.BIN image
- **zimage**: Linux kernel image, loaded into memory by U-Boot
- **zimage.bin**: Linux kernel image, loaded into memory by U-Boot. A copy of the previous file with file extension (for specific exercises).
- **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

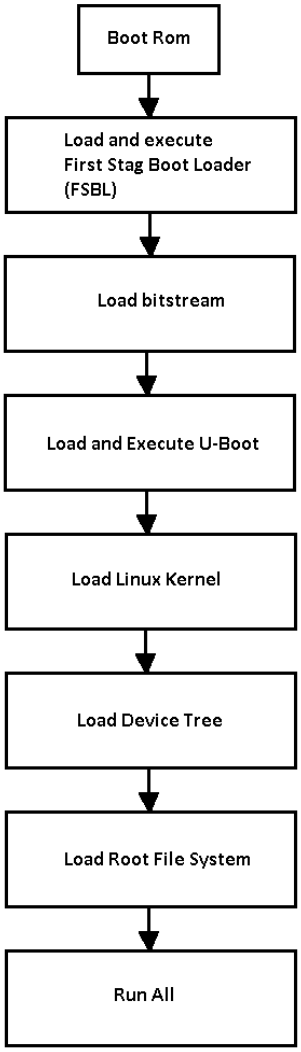


Figure 5-1: Linux Boot Process on the ZedBoard

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, various 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 DDR memory. 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. In this tutorial we use the Non Secure mode.

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

5.2.2 Booting Linux from JTAG

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.

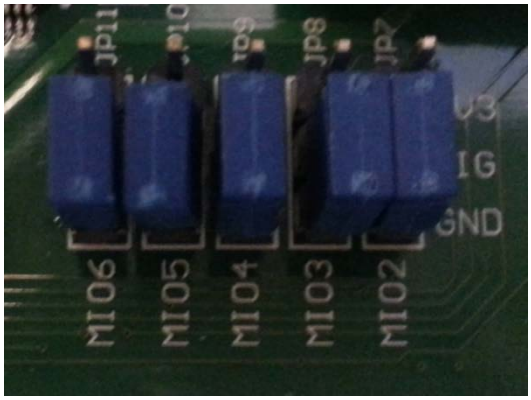


Figure 5-2: Jumper Settings to boot in JTAG mode

b. Connect the power cable to the board.

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

d. Connect a micro USB 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, if it was closed, 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**

4. If the external 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/system_wrapper_hw_platform_0/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).

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

   d. 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

e. Press any key.

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

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

The U-Boot execution is stopped.

g. Type **dow -data <directory>/zimage 0x10000000** to download the Linux Kernel image (zImage) at location 0x10000000 (seven zeroes).

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

i. Type **dow -data <directory>/devicetree.dtb 0x1000000** to download the Linux device tree at location 0x1000000 (six zeroes).

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

7. At the command prompt of the serial terminal, type **go 0x10000000** (seven zeroes).

   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. Type **ping 192.168.1.10**. This is to have the device ping itself and not with an external host. The following ping response displays in a continuous loop:

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

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

   Linux booting completes on the target board.

   Power-off the ZedBoard but keep the SDK open.

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 using SDK.
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.** Keep everything else as defaults and click **Next**.

2. Select **Zynq FSBL** in the Template list.
3. Click **Finish** to generate the FSBL.

   The Zynq FSBL compiles and an .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. Make sure to select **Create new BIF file** and specify a path where the *.BIF file (and the resultant *.MCS file) will reside under **BIF file path**. The default file name is **output.bif**.

3. Click **Add** and add the **zynq_fsbl_0.elf** file by specifying the correct path to it. The file is located in **<your project path>/<project name>.sdk/zynq_fsbl_0/Debug/** Make sure the partition type is **bootloader**. Do not specify any addresses in the **Other** section of the window. Click **OK** when done.

4. Add the provided **u-boot.elf** image as a **datafile**. Leave everything else blank or as default.

5. Add the Linux Kernel image (zImage.bin) as a datafile, while keeping everything as default provide the offset **0x100000** (five zeroes).

6. Add the device tree image (devicetree.dtb) as a datafile and provide offset - **0x3c0000** (four zeroes) with everything else set as default.

7. Add the root file system image (ramdisk8M.image.gz) as a datafile and provide offset **0x400000** (five zeroes) with everything else set as default.

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.

8. You can provide the absolute path to the output folder and file name in the **Output path** text box. This is not always necessary as the file will by default reside where the *.BIF file is saved by default as BOOT.bin.

   However, we are creating a *.MCS file to program the QSPI flash. In the Output path text box rename output.bin to **BOOT.mcs**.

   The tool will automatically create a MCS formatted output file just by you changing the extension.You may have to make the window bigger to see the output file path field.

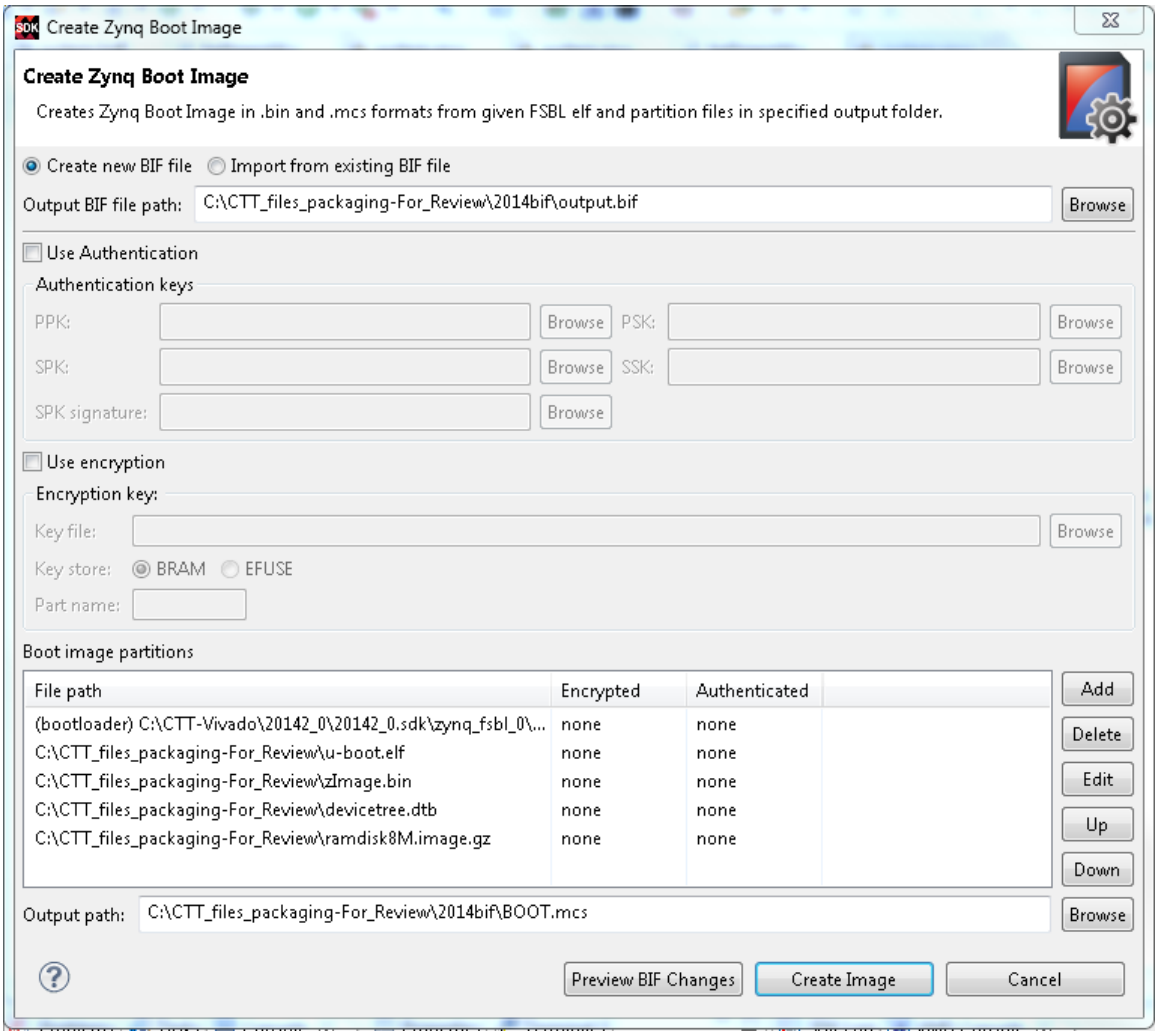


Figure 5-3: Creating a Zynq QSPI Boot Image

9. Click **Create Image.**

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

BOOT.mcs

output.bif

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

1. Set the Jumpers JP7-11 to the JTAG boot mode:

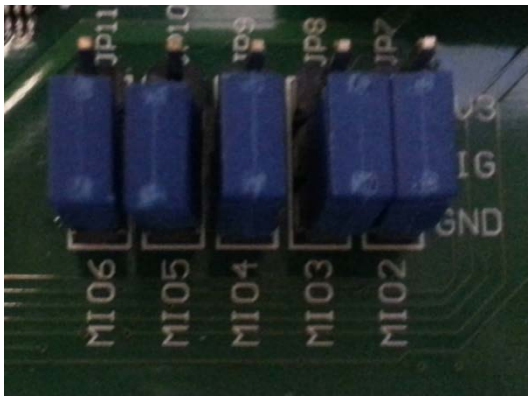


Figure 5-4: Jumper Settings to boot in JTAG mode

2. Power on the ZedBoard.
3. In the SDK GUI, select the **Xilinx Tools → Program Flash** option. This will launch the Program Flash Memory window.
4. For the Image File, browse to where you created the **BOOT.mcs** file. Select the Device as **2-xc7z020** and leave the other options as default. The Blank check after erase option checks that the PROM is blank. Also, the Verify after flash option to verify the QSPI flash. The selecting both the blank check and verify operation will almost double the programming time.
5. Click **Program** to continue.

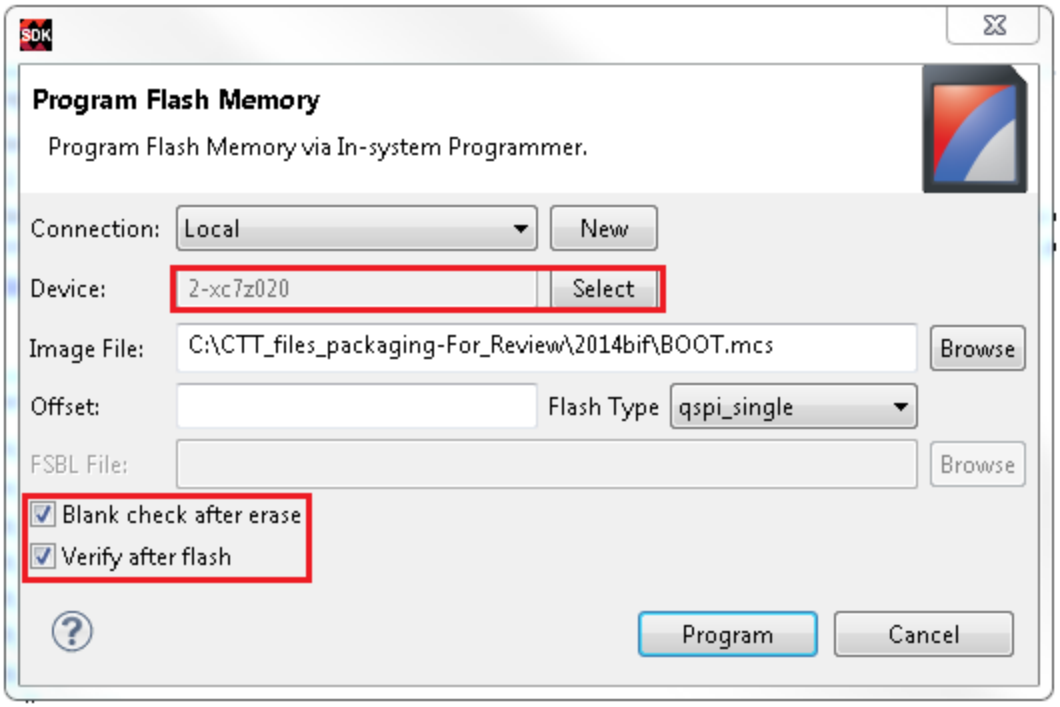


Figure 5-5: QSPI programming interface

6. Once flash programming is complete, power-off the ZedBoard.

4. Booting Linux from the QSPI Flash

1. Set the jumper settings (JP7-11) on the ZedBoard. Jumper settings for QSPI:

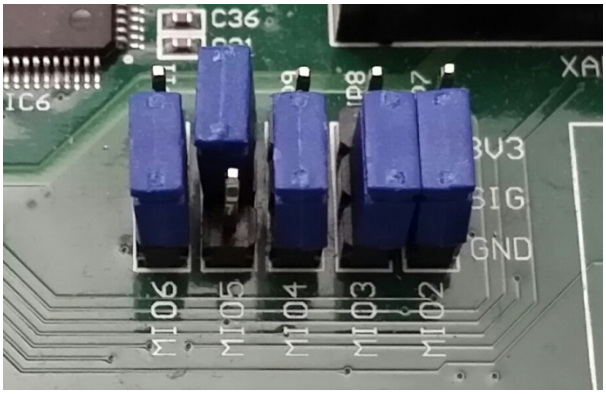


Figure 5-6: Jumper Settings to boot in QSPI mode

1. Connect the Serial terminal with a 115200 baud rate setting.
2. Power-on the ZedBoard. You may need to wait for around five seconds before the first text appears on the console.

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

3. Power-off the ZedBoard after you are satisfied with the Linux boot.

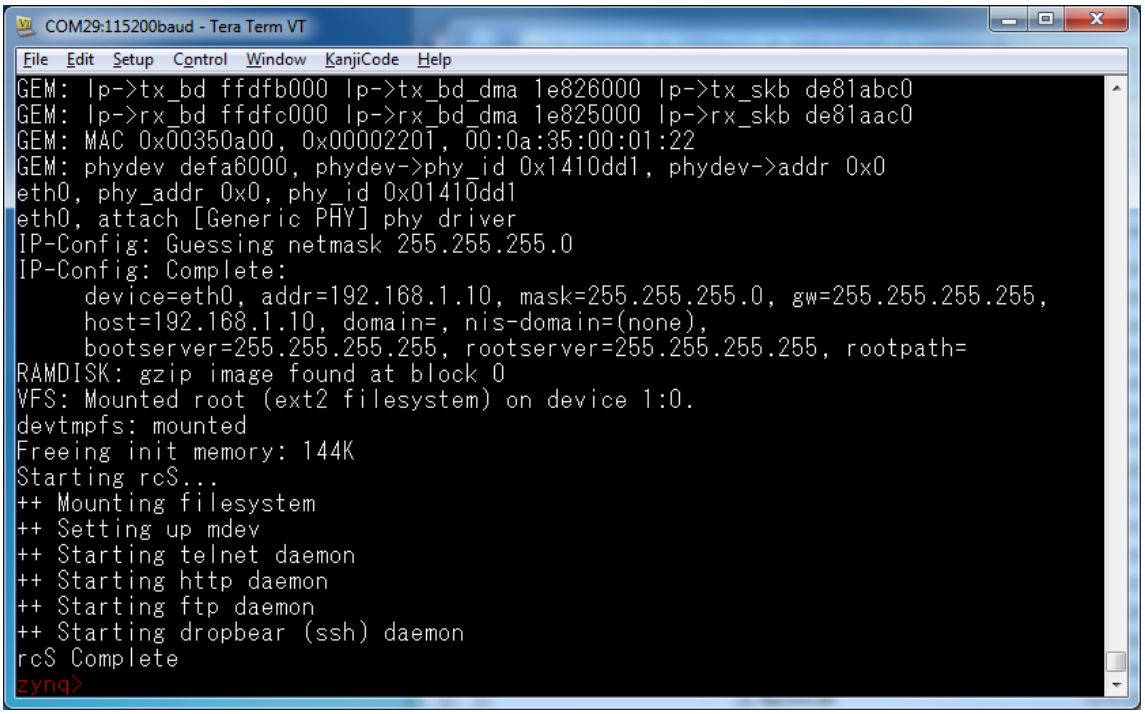


Figure 5-7: Serial Terminal Window showing Linux Booting

5.2.6 Booting Linux from the SD card

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

With the board powered-off, ensure that the jumper settings (JP7-11) are set to boot from SD card as shown in the figure.

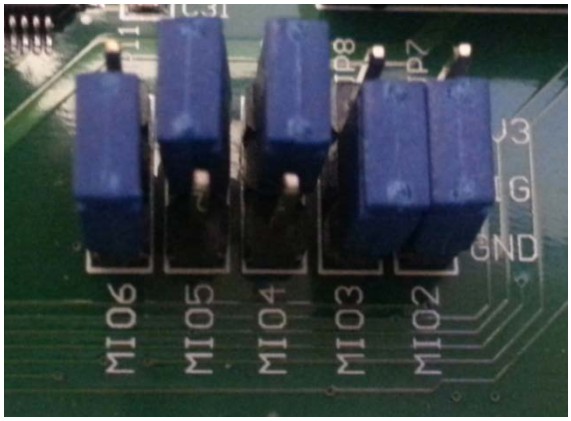


Figure 5-8: Jumper Settings to boot from SD Card

For the steps below, you can use the FSBL that was created for the previous exercise. Alternatively, you can use the zync_fsbl_0.elf file from the CTT zip file.

1. In SDK select **Xilinx Tools > Create Zynq Boot Image** to open the "Create Zynq Boot Image" wizard. As in the exercise before, you will need to create a new *.BIF file and specify a path to it. The default name is **output.bif**.

   You can skip this process by using the u-boot.bin from the CTT zip file, rename the file as BOOT.bin and skip to step 6.

2. Add the **zynq_fsbl_0.elf** (the file is located in **<your project path>/<project_name>.sdk/zynq_fsbl_0/Debug/**) as the boot loader.

3. Add the provided **u-boot.elf**. Keep everything as default in the Add Parition windows (no offsets needed).

4. Provide the location to store the generated files in the **Output path** field and the default name of the output file is BOOT.bin.

5. Click **Create Image**. SDK generates the BOOT.bin file in the specified output folder.

6. Copy **BOOT.bin** (generated), **zimage** (provided), **devicetree.dtb** (provided) and **ramdisk8M.image.gz** (provided) to the SD card. Notice that zimage doesn't have any extensions. Alternatively, you can rename zImage.bin to just zimage (all lower case and no file extension) in the SD card. Make sure the SD card is FAT32 formatted before copying the files into it.

7. Insert the SD card into the SD card slot on the ZedBoard.

8. Power-on the ZedBoard.

9. Immediately connect the serial terminal. Again, you may need to wait 5 to 10 seconds for the bootup text to appear on the terminal. The **zynq>** prompt appears after Linux booting is complete on the target board.

10. Power-off the ZedBoard once you are done with Linux. You **do not** need to exit from SDK.

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.

2. Assuming that SDK was not closed from the previous exercise, select **File > New > Application Project.**

3. Enter **Hello_World_ap** in the **Project name** field

4. Select **Linux** as the OS Platform in the Target Software and select **C** as the language. Click **Next.**

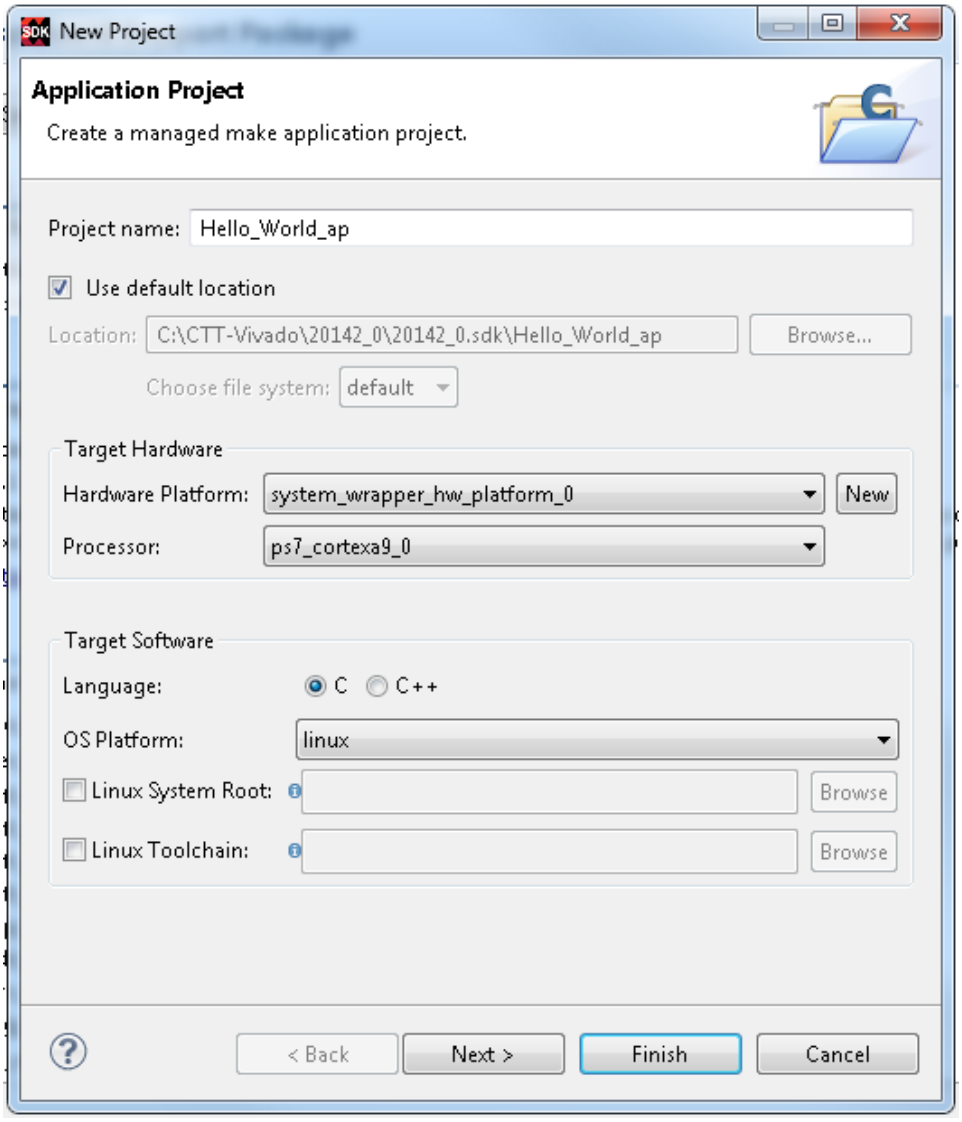


Figure 5-9: Application Project

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

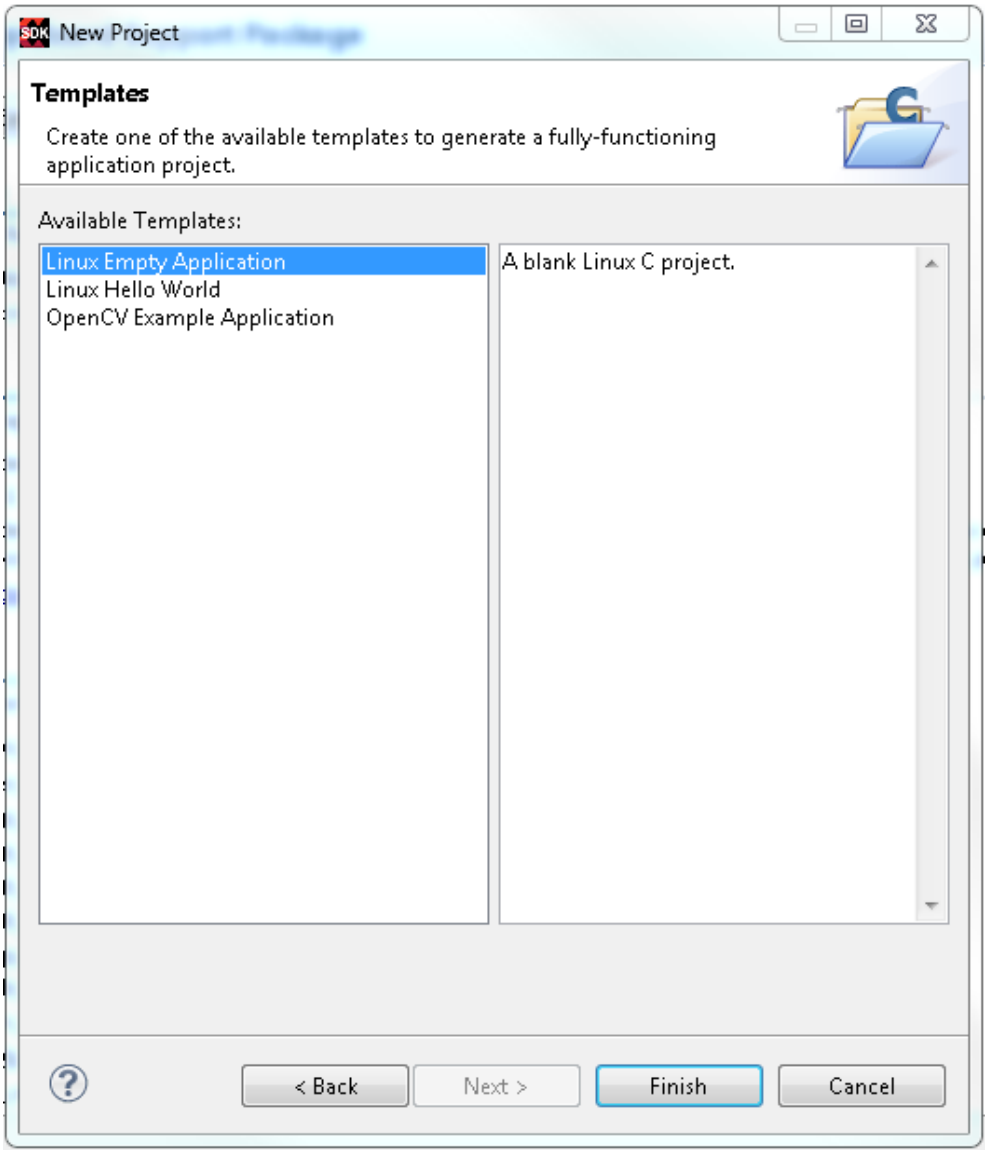


Figure 5-10: Add An Empty Application

At this point, you would have created a software platform and an empty software project for the hardware. You will next import the hello_world_linux.c into the project, and SDK will automatically build and produce an *.ELF (Executable and Load Format) file.

6. Right Click **Hello_World_ap** and select **Import.**

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

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

Check that the application is built without errors. The message you should get is 'Finished building: Hello_World_ap.elf.size' in the Console window:

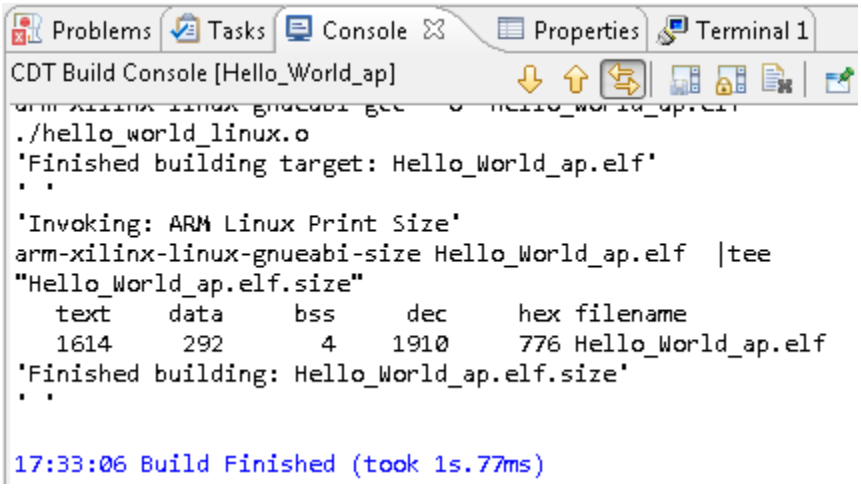


Figure 5-11: Serial Terminal Window showing Linux Booting

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

   C:\<project_path>\<project_name>\<project_name>.sdk\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 immediately open a serial terminal window.

6. You should see Linux booting on the ZedBoard from the SD card with the pre-built image. Again, it may take five to ten seconds for the boot text to appear.

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

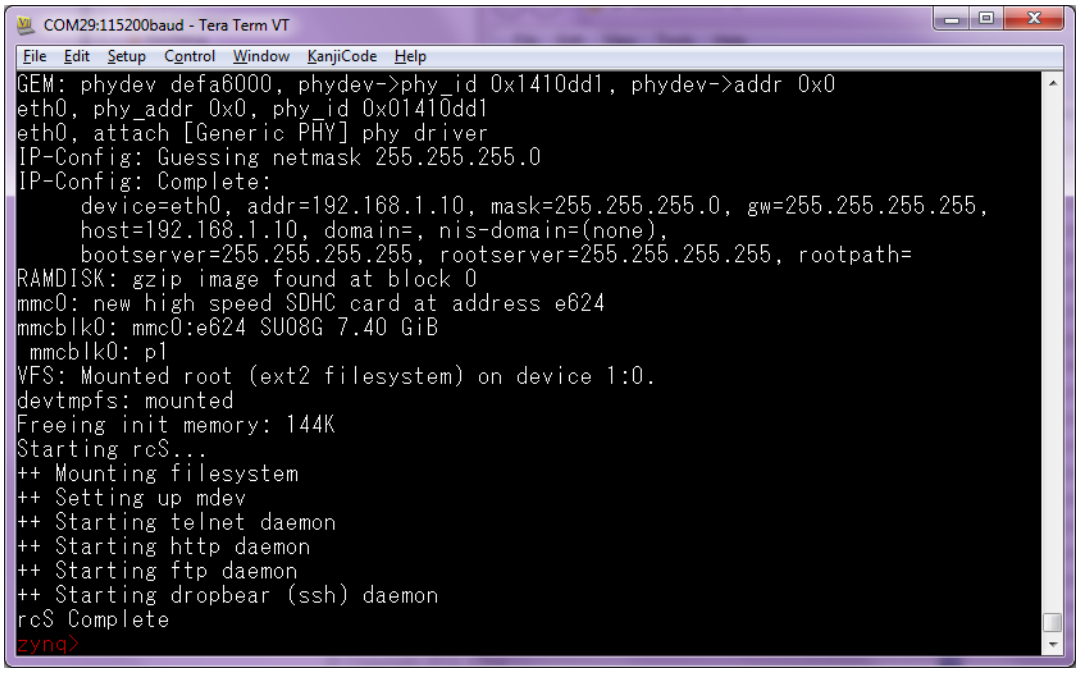


Figure 5-12: Serial Terminal Window showing Linux Booting

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

   zynq> cd dev

   zynq> mount /dev/mmcblk0 /mnt

   Or if the SD card was partitioned: **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. You may power-off the ZedBoard, close the terminal window and exit the SDK at this point.

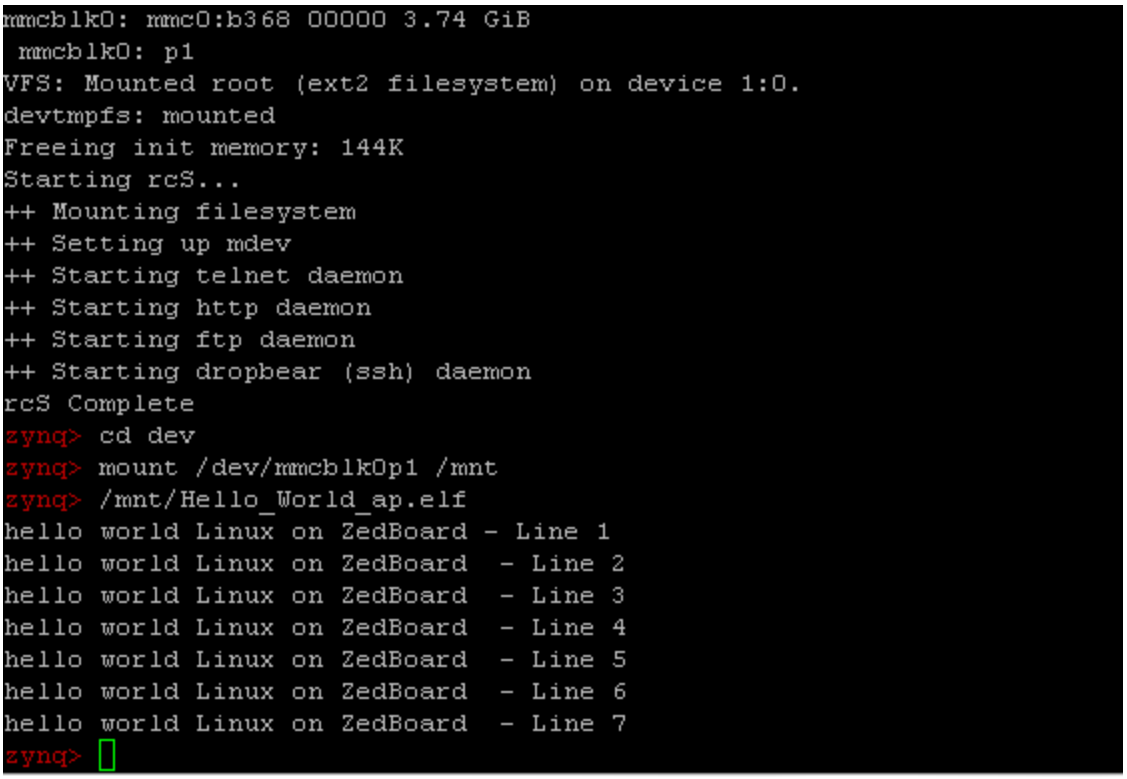


Figure 5-13: Serial Terminal Window with hello_world_linux running

5.4 Controlling LEDs and switches in Linux example

This example shows you how to create a simple Linux application that drives the LEDs and prints the value of the switch settings. 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.

This example assumes you have completed the previous exercises and will not provide explicit instructions on doing certain tasks.

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 Vivado 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. Click **Next.**
	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**. Click **Next.**
Add Sources	Do not make any changes on this screen. Click **Next.**	
Add Existing IP	Do not make any changes on this screen. Click **Next.**	
Add Constraints	Do not make any changes on this screen. Click **Next.**	
Default Part	Specify	Select **Boards**.
	Board	Select **ZedBoard Zynq Evaluation and Development Kit.** Select either Rev.C or D. Click **Next.**
New Project Summary	Project summary	Review the project summary before clicking **Finish** to create the project.

When you click **Finish**, the New Project wizard closes and the project you just created opens in the Vivado GUI. You can close the Vivado project from the previous exercises.

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. Click on the **Add IP** button. From the IP catalog, drag the **AXI GPIO** item to add it. Notice the block has been added to the Block Diagram.

6. Drag two more AXI GPIO blocks to the block diagram with the IP Catalog open.
7. Click the **Add IP** button again, double click on the **AXI Interconnect** IP.
8. Click the **Add IP** button again, double click on the **Processing System Reset** IP.
9. Double click on the **AXI Interconnect** block to edit the IP properties. Set the number of Master Interfaces to **3**. Click **OK** to exit the screen.
10. Connect **FCLK_RESET0_N** on the **ZYNQ7 Processing System** block to **ext_reset_in** on the **proc_sys_reset_0** block.
11. Connect **FCLK_CLK0** to the **slowest_sync_clk** of the **proc_sys_reset_0 block**.
12. Also connect **FCLK_CLK0** to all the clock ports in the block diagram.
13. Connect the **peripheral_aresetn[0:0]** on the **proc_sys_reset_0** block to all the GPIO reset ports.
14. Connect the **interconnect_aresetn[0:0]** on the **proc_sys_reset_0** block to all the **axi_interconnect_0** reset ports.
15. Connect the **M_AXI_GP0** on the **ZYNQ7 Processing System** block to **S00_AXI** of the **AXI Interconnect** block.
16. Click on **Run Block Automation**, select **processing_system7_0**. In the Run Block Automation dialogue box, make sure **Apply Board Preset** is checked. Click **OK**.

    This will apply the ZedBoard presets to the block.
17. Click on **Run Connection Automation**, select **/axi_gpio_0/S_AXI**. Click **OK**. Notice the slave side ports of the **axi_gpio_0** block have been connected to the interconnect, the processing system and the just inserted **Processor System Reset** block. The Run Connection Automation functionality saves time by automatically connecting the necessary signals/ports in the block design for you.
18. Repeat the step above for **/axi_gpio_1/S_AXI** and **/axi_gpio_2/S_AXI**.
19. 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.
20. 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.
21. 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.

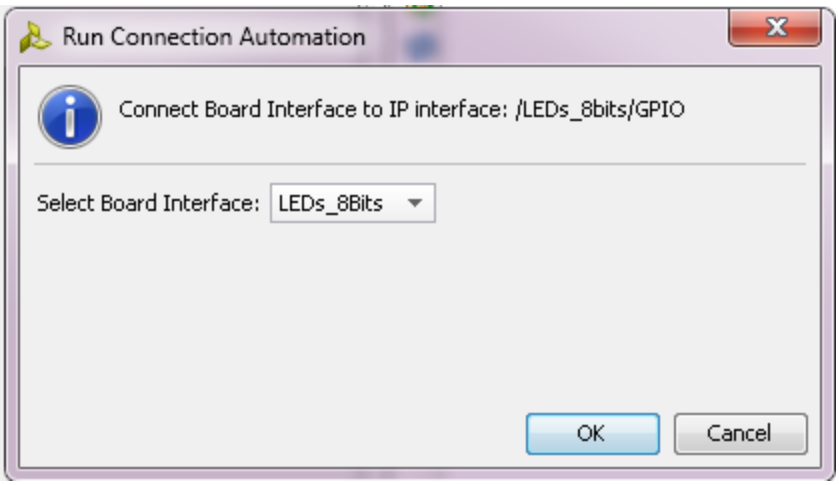


Figure 5-14: Automatically connecting the LEDs_8bits GPIO to external pins

22. Click on **Run Connection Automation** and select **/leds_8bits/GPIO**. Select the board interface for LEDs_8bits. Click **OK**. Notice that LEDs_8bits now has an external port associated with it.
23. Repeat the same procedure as above but with the **sws_8bits** block.
24. Repeat the same procedure as above but with the **btns_5bits** block.

Hit refresh and the block diagram connections should now look like this:

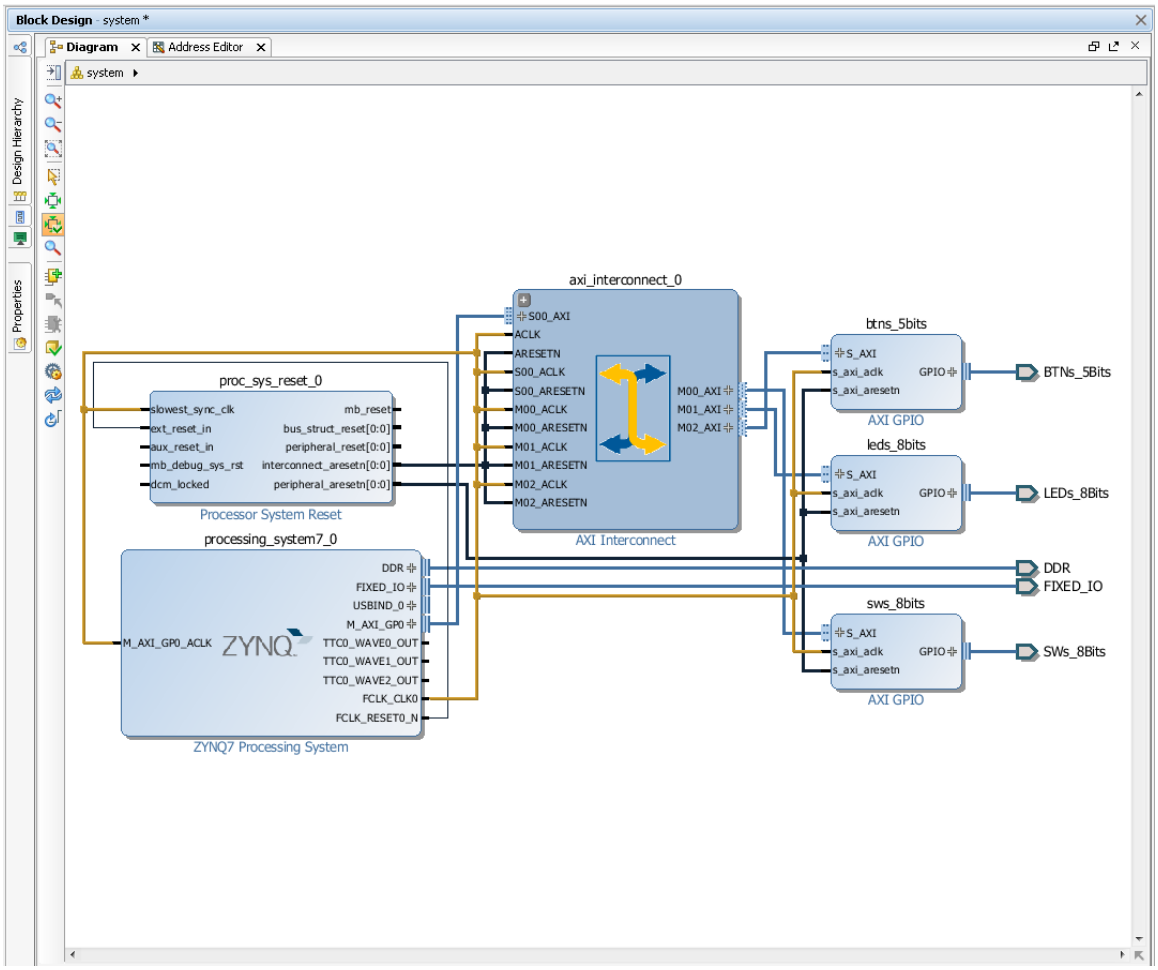


Figure 5-15: System block diagram

1. The offset addresses have been specified for the design in the Address Editor. There is no need to auto-generate the addresses.

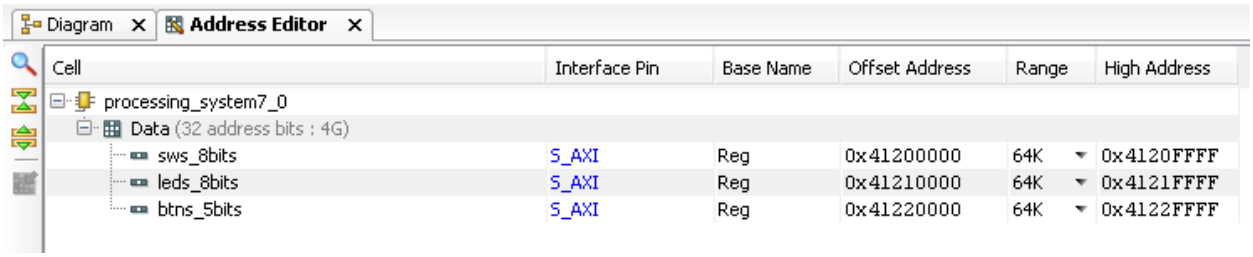
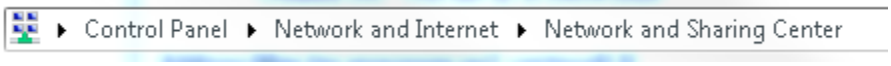


Figure 5-16: Address Editor Offset Addresses for GPIO peripherals

2. **Save** the block design and run **Tools > Validate Design**. Click **OK** in the Validation successful window.
3. **Generate a new HDL wrapper** for the block diagram, letting Vivado manage the wrapper file.

4. Click on **Generate Block Design.** Hit **Generate** when prompted.
5. Click on the **Generate Bitstream** under the **Program and Debug** group to synthesize, implement and generate the bitstream for the project.
6. When bitstream generation is complete, make sure to open the implemented design before exporting the design to SDK.
7. Select **File > Export > Export Hardware.** Make sure that the **Include Bitstream** check-box is selected. Click **OK**.
8. Select **File > Launch SDK** and click **OK**. SDK opens.

For the subsequent steps in the exercise below, the ZedBoard is going to be connected directly to the host PC via an Ethernet connection. This connection will be made from the ZedBoard's Ethernet port to the host PC's Ethernet port via a cable.

> To successfully run the exercise, please manually set your host computer's static IP address to **192.168.1.1** with the subnet mask set to **255.255.255.0**. You will need to disable any wireless Ethernet on the host PC. To do this in Windows 7, you need to access the Network and Sharing Center…
>
> 
>
>
> and click **Change adapter settings** on the side bar. Right click on **Local Area Connection** and select **Properties**. You will need administrator access to change the properties.
>
> Select **Internet Protocol Version 4 (TCP/IPv4)** and select **Properties**. Make a note of the original settings shown in the window. Change the IP address and subnet mask to the values above. Click **OK** and **Close** to get out of the properties menues.

Remember to change the IP address back to the original setting once the exercise is done.

Continuing Your Design in SDK

1. Connect the 12V AC/DC converter power cable to the ZedBoard power 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's Ethernet ports.
5. **Do not** power-up the board yet.

6. A FSBL *.ELF file will need to be created using the **system_wrapper_hw_platform_0** as the base platform. Please name the file as **zynq_fsbl_0**.

7. Create a new BOOT.bin under the **Xilinx Tools > Create Zynq Boot Image** utility. Include the system_wrapper.bit that was exported to SDK under the **…\<project_name>.sdk\system_wrapper_hw_platform_0** directory.

   The exact order of the *.ELF file and the rest of the files is important, make sure they appear in the following order:

   zynq_fsbl_0.elf (generated)

   system_wrapper.bit (generated in Vivado before SDK export)

   u-boot.elf (provided with the CTT)

   No offset is needed for the system_wrapper.bit file. Name the output file as BOOT.bin and copy it to the SD card (assuming the files from Section 5.2.6 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.

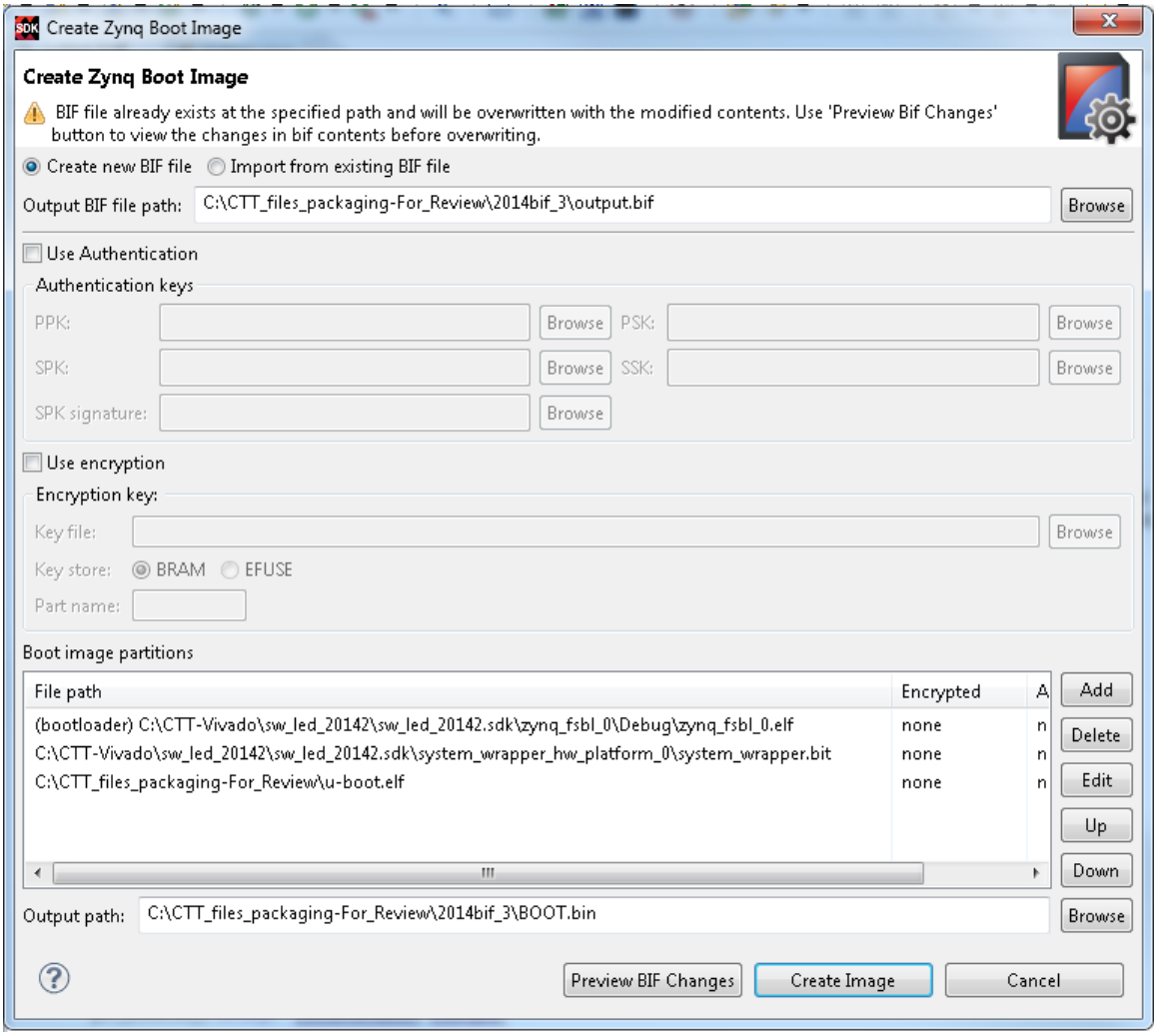


Figure 5-17: Creating a new BOOT.bin for the SD card

8. Use the jumper settings to boot from SD card.

   MIO6: 0

   MI05: 1

   MIO4: 1

   MIO3: 0

   MIO2: 0

9. Power the ZedBoard.

10. Immediately 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**

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

12. 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.

13. Optionally, **ping 192.168.1.1** to make sure the Ethernet cable connection is good.

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 leds_switches.c into the project, and SDK will automatically build and produce an .ELF (Executable and Load Format) file.

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

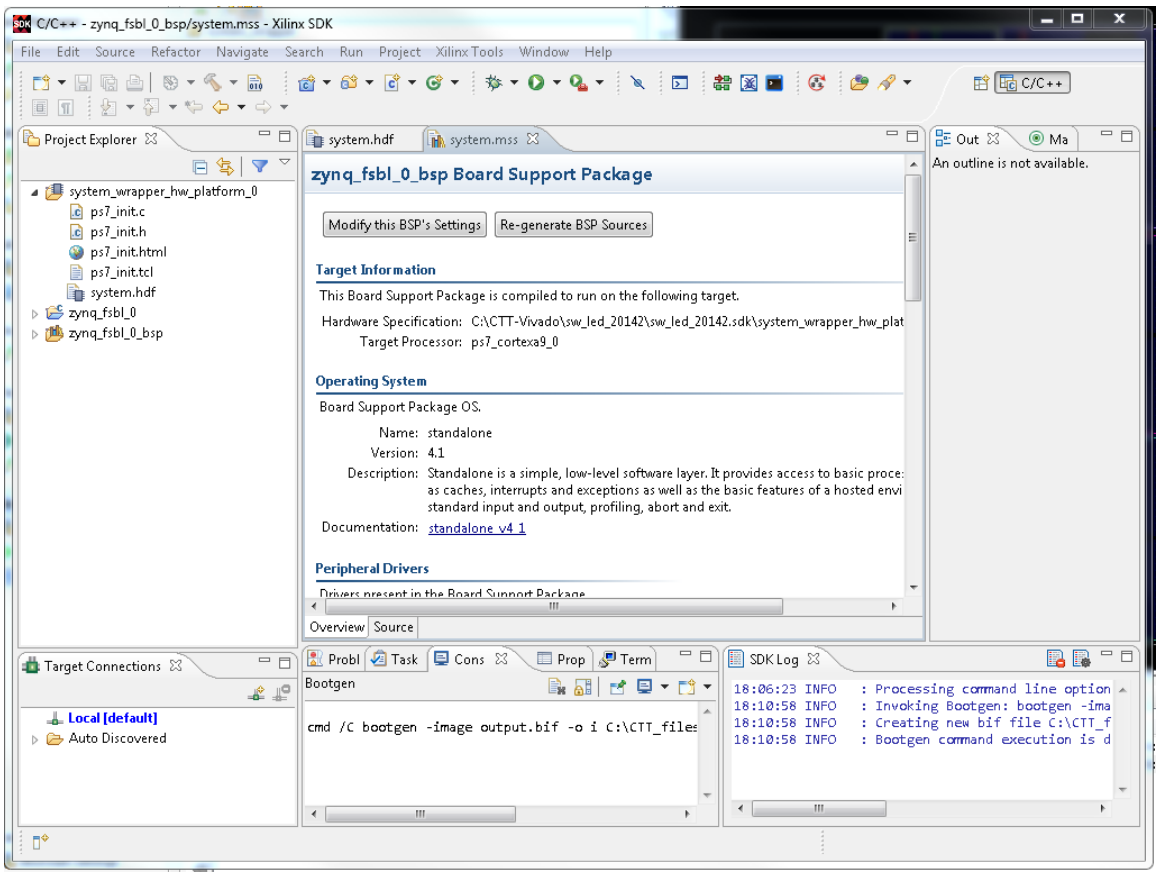


Figure 5-18: New Project Selection

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

16. Select the Hardware Platform as **system_wrapper_hw_platform_0.**

17. Select **Linux** as the OS Platform in the Target Software.

18. Select **C** as the Language.

19. Click **Next**.

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

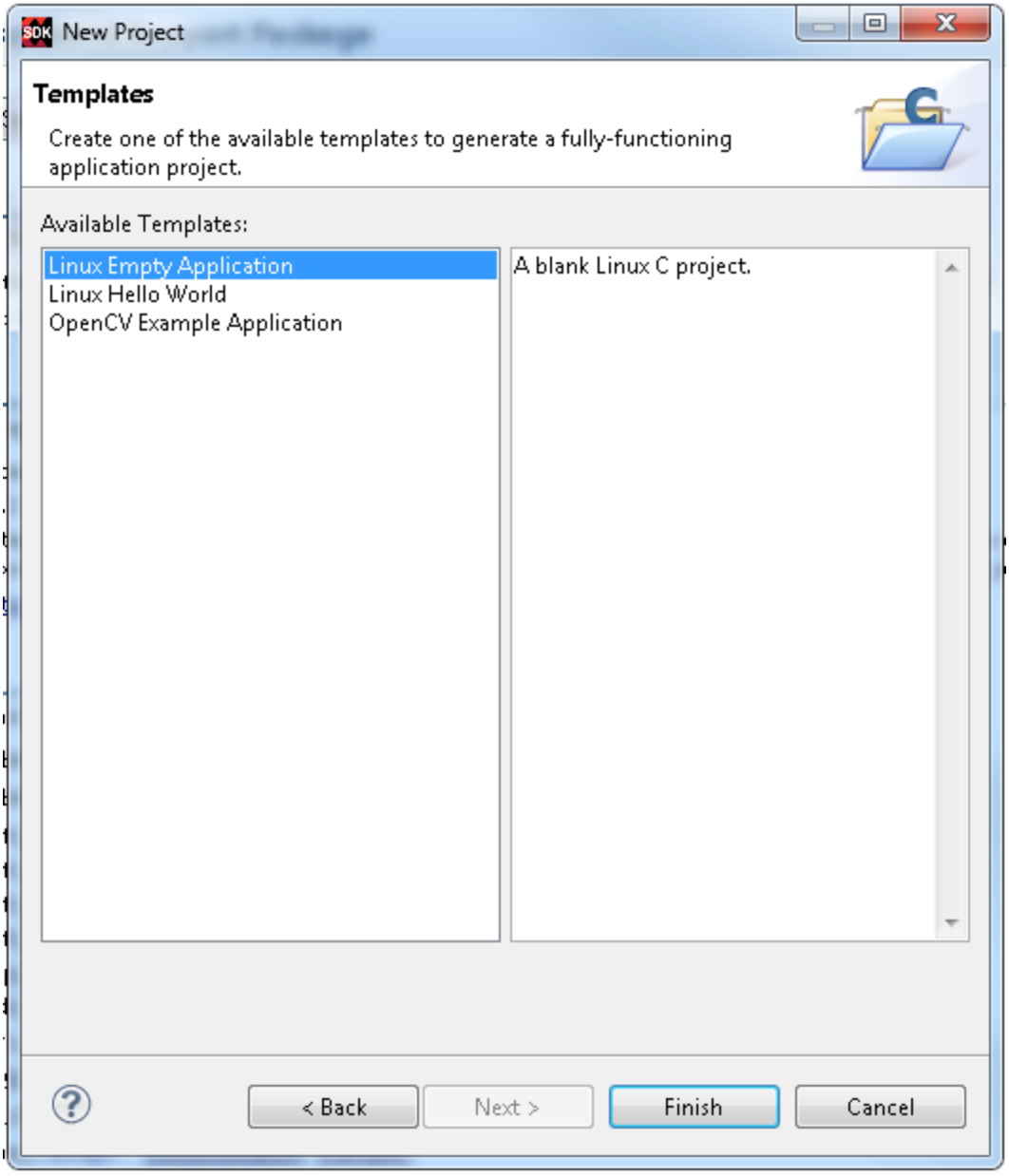


Figure 5-19: Add An Empty Application

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

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

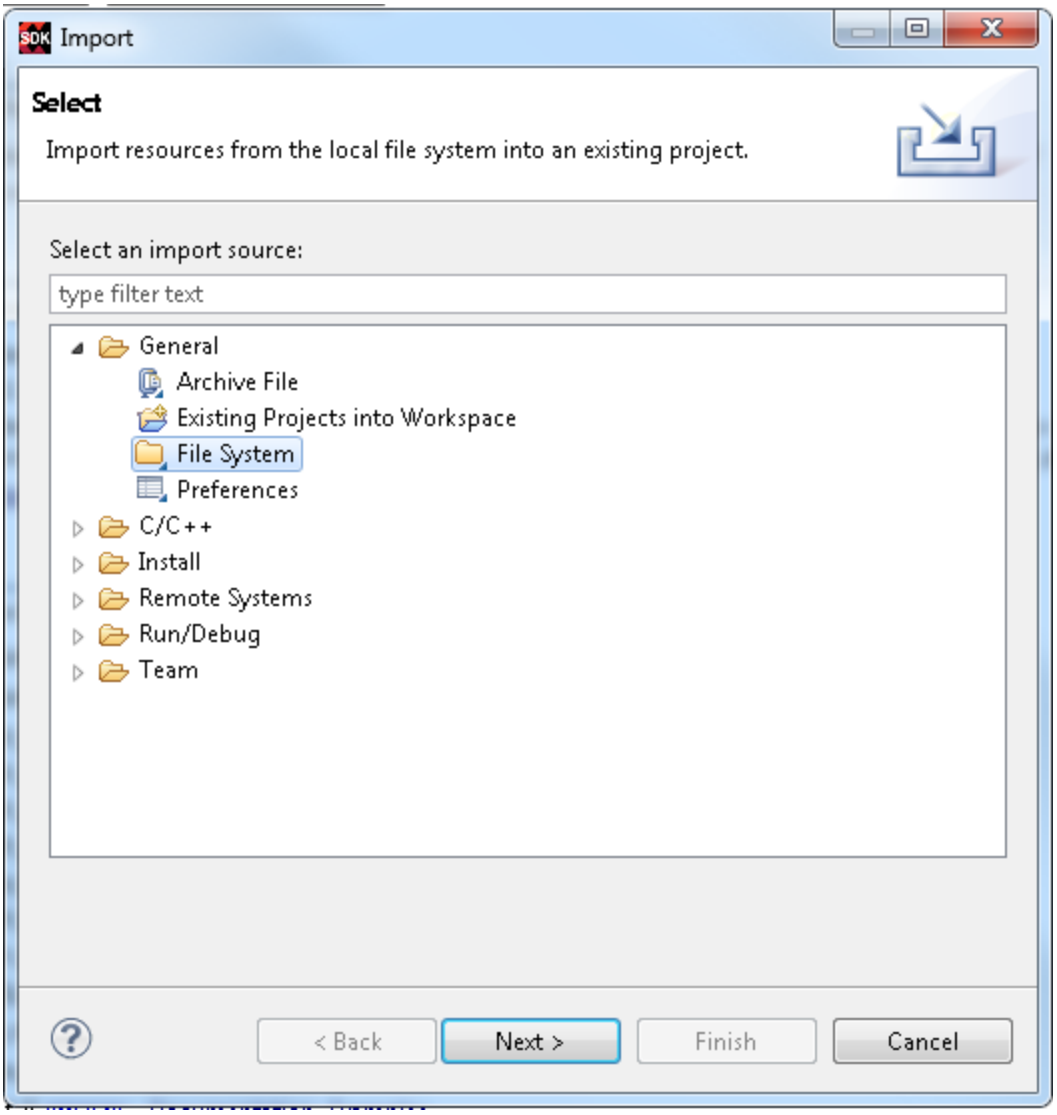


Figure 5-20: Import .C file

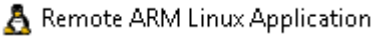
23. Browse to the directory in which you saved the CTT files that you unzipped. Select **leds_switches.c** and select **Finish**. Do not modify any other settings.

Check that the application is built without errors via 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. Next to the Connection drop-down list, click **New**.
4. The New Connection wizard opens.
5. Click the **SSH Only** icon and click **Next**.
6. In the **Host Name** field, type the target board IP (it should be 192.168.1.10).
7. Set the connection name and description in the respective fields. Verify host name should be selected.
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. Perform the following steps:
    a. Expand the root directory. This action 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**.
    d. Right-click on the "/" in the path name and create a new folder; name it **Apps**.
    e. Drill down to the Apps directory, right click and create a new file named **leds_switches_0.elf**. Click **Finish** and click **OK**.
    f. Double check that the application absolute path is **/Apps/leds_switches_0.elf**.
11. Click **Apply**.
12. Click **Debug**. SDK may notify you that it will open the Debug Perspective, click **Yes**. The Debug Perspective opens. Turn off the Verbose console mode  in the console window.
13. Click on resume to run  the code, and watch the messages in the console window display the values of the LEDs and Switches.
14. 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.
15. 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 and from other resources on the internet. The Zynqgeek Blog (http://zedboard.org/zynqgeek) contains helpful step-by-step instructions on several topics (please note the blog has not been updated for a while).

Another good resource is ZedBoard.org’s own blog (http://www.zedboard.org/blog) that extends its focus to beyond just the ZedBoard. Also useful is the New Horizons blog Zynq Design From Scratch (http://svenand.blogdrive.com/archive/160.html) that features the ZedBoard in a series of blog posts.

In addition to blogs, there are other useful links for the registered ZedBoard user on ZedBoard.org.

1. 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.

2. 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.

3. 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_0_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_0_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_0_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_0_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_0_DEVICE_ID,
    (XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION ));
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Step-7 :PS GPIO Intialization
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_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_0_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, 0x41210000);
        printf("LEDs mmap Good... \r\n");

        buttons = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd,
0x41220000);
        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;
}
```