EDK Concepts, Tools, and Techniques

A Hands-On Guide to Effective Embedded System Design

EDK 12.3

XILINX®

Chapter 1: Introduction

Chapter 2: Creating a New Project

Chapter 3: Using Xilinx Platform Studio

Chapter 4: Working with Your Embedded Platform

Chapter 5: Introducing the Software Development Kit

Chapter 6: More on the Software Development Kit: Edit, Debug, and Release

Chapter 7: Creating Your Own Intellectual Property

Chapter 8: Dual Processor Design and Debug

Appendix A: Intellectual Property Bus Functional Model Simulation

Appendix B: Creating an AXI-Based Design in EDK

Appendix C: Glossary

Introduction

About This Guide

The Xilinx® Embedded Development Kit (EDK) is a suite of tools and Intellectual Property (IP) that enables you to design a complete embedded processor system for implementation in a Xilinx Field Programmable Gate Array (FPGA) device.

This guide describes the design flow for developing a custom embedded processing system using EDK. Some background information is provided, but the main focus is on the features of and uses for EDK.

Read this guide if you:

- Need an introduction to EDK and its utilities
- Have not recently designed an embedded processor system
- Are in the process of installing the Xilinx EDK tools
- Would like a quick reference while designing a processor system

Note: This guide is written for the Windows operating system. Linux behavior or the graphical user interface (GUI) display might vary slightly.

Take a Test Drive!

Because the best way to learn a software tool is to use it, this guide provides opportunities for you to work with the tools under discussion. Specifications for a sample project are given in the Test Drive sections, along with an explanation of what is happening behind the scene and why you need to do it. This guide also covers what happens when you run automated functions.

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

Additional Documentation

More detailed documentation on EDK is available at:
http://www.xilinx.com/ise/embedded/edk_docs.html

Documentation on the Xilinx® Integrated Software Environment (ISE®) is available at:
http://www.xilinx.com/support/software_manuals.htm.

Attachments to this Guide

Some examples in this guide require you to access example project files. These files are included in the .zip file for this guide. You can download the .zip file from the Xilinx EDK Documentation website:
http://www.xilinx.com/support/documentation/dt_edk_edk12-1.htm.

The following files are included:

- bus_transaction_bfl_code.txt
- leds.c
- pn.do
- pwm_lights.vhd
- readme_EDK_ctt.txt
- sample.bfl.txt
- user_logic.vhd

How EDK Simplifies Embedded Processor Design

Embedded systems are complex. Getting the hardware and software portions of an embedded design to work are projects in themselves. Merging the two design components so they function as one system creates additional challenges. Add an FPGA design project to the mix, and the situation has the potential to become very confusing indeed.

To simplify the design process, Xilinx offers several sets of tools. It is a good idea to get to know the basic tool names, project file names, and acronyms for these tools. To make this easier for you, see Appendix C, "Glossary," which lists the EDK-specific terms and is provided at the end of this guide.

The Integrated Design Suite, Embedded Edition

Xilinx offers a broad range of development system tools, collectively called the ISE Design Suite. For embedded system development, Xilinx offers the Embedded Edition of the ISE Design Suite. The Embedded Edition comprises:

- Integrated Software Environment (ISE)
- PlanAhead design analysis tool
- ChipScope™ Pro (which is useful for on-chip debugging of FPGA designs)
- Embedded Development Kit (EDK). EDK is also available with the ISE Design Suite: System Edition, which includes tools for DSP design.

For information on how to use the ISE tools for FPGA design, refer to the Xilinx documentation web page: http://www.xilinx.com/support/software_manuals.htm.

The Embedded Development Kit (EDK)

The Embedded Development Kit (EDK) is a suite of tools *and* IP that you can use to design a complete embedded processor system for implementation in a Xilinx FPGA device.

Xilinx Platform Studio (XPS)

Xilinx Platform Studio (XPS) is the development environment used for designing the *hardware* portion of your embedded processor system. You can run XPS using a bash shell

command line, in batch mode, or using the GUI, which is what we will be demonstrating in this guide.

Software Development Kit (SDK)

The Software Development Kit (SDK) is an integrated development environment, complementary to XPS, that is used for C/C++ embedded software application creation and verification. SDK is built on the Eclipse open-source framework and might appear familiar to you or members of your design team. For more information about the Eclipse development environment, refer to http://www.eclipse.org.

Other EDK Components

Other EDK components include:

- Hardware IP for the Xilinx embedded processors
- Drivers and libraries for the embedded software development
- GNU compiler and debugger for C/C++ software development targeting the MicroBlaze™ and PowerPC® processors
- Documentation
- Sample projects

EDK is designed to assist in all phases of the embedded design process.

How the EDK Tools Expedite the Design Process

The diagram below shows the simplified flow for an embedded design.
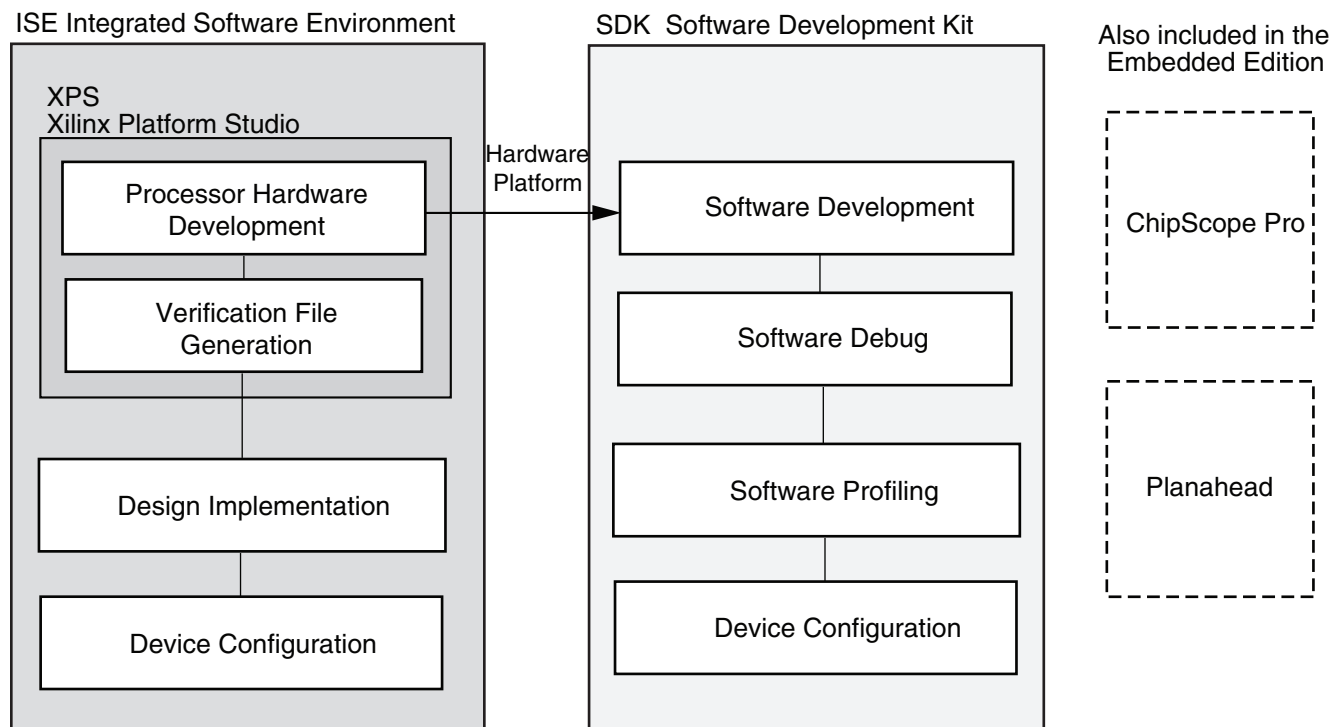
Figure 1-1: **Basic Embedded Design Process Flow**

Typically, the ISE development software is used first to create an Embedded Processor source, which is then added to the ISE project.

- You use XPS primarily for embedded processor hardware system development. Specification of the microprocessor, peripherals, and the interconnection of these components, along with their respective property assignments, takes place in XPS.

- You use SDK for software development. SDK is also available as a *standalone* application. It can be purchased and used without any other Xilinx tools installed on the machine on which it is loaded.

- You can verify the correct functionality of your hardware platform by running the design through a Hardware Description Language (HDL) simulator. You can use the Xilinx simulator ISim to simulate embedded designs.

XPS facilitates three types of simulation:

– Behavioral

– Structural

– Timing-accurate

XPS sets up the verification process structure automatically, including HDL files for simulation. You only have to enter clock timing and reset stimulus information, along with any application code.

- After completing your design in XPS, you return to ISE to generate the FPGA configuration file used to program the target device.

- Once your FPGA is configured with the bitstream containing the embedded design, you can download and debug the Executable and Linkable Format (ELF) file from your software project from within SDK.

For more information on the embedded design process as it relates to XPS, see the "Design Process Overview" in the *Embedded System Tools Reference Manual*, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

What You Need to Set Up Before Starting

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

Installation Requirements: What You Need to Run EDK Tools

ISE and EDK

EDK Installation Requirements

ISE and EDK are both included in the ISE Design Suite, Embedded Edition software. Be sure the software, along with the latest update, is installed. Visit http://support.xilinx.com to confirm that you have the latest software versions.

Bash Shell for Linux

When you use EDK on a Linux platform, you need a bash shell. In addition, be sure to review the supported platforms covered in the Xilinx ISE Design Suite 12: Installation, Licensing, and Release Notes.

Software Licensing

Xilinx software uses FLEXnet licensing. When the software is first run, it performs a license verification process. If it does not find a valid license, the license wizard guides you through the process of obtaining a license and ensuring that the Xilinx tools can use the license. If you are only evaluating the software, you can obtain an evaluation license.

For more information about licensing Xilinx software, refer to the ISE Design Suite 12: Installation, Licensing, and Release Notes.

Simulation Installation Requirements

To perform simulation using the EDK tools, you must have an appropriate simulator installed and simulation libraries compiled:

1. A Secure-IP capable mixed language simulator (ModelSim PE/SE v6.5c or Incisive Enterprise Simulator (IES) IES9.2 or later, or VCS/VCS-MX 2009.12) must be installed for the simulation steps. MXE is not supported for embedded designs. It doesn't have mixed language or SecureIP support.

    You can optionally install the CoreConnect™ toolkit. The CoreConnect toolkit is required only if you intend to run Bus Functional Model (BFM) Simulation. If you do not intend to run BFM simulations, you may skip installation of the CoreConnect Toolkit.
    CoreConnect is a free utility provided by IBM. You can download CoreConnect from the Xilinx website at:
    http://www.xilinx.com/products/ipcenter/dr_pcentral_coreconnect.htm.

    Note: BFM simulation is currently supported only with ModelSim.

Simulation Installation Requirements

2. If you haven't already done so, compile the simulation libraries following the procedure outlined in the EDK help system.

    a. To open the help from XPS, select **Help > Help Topics**. Alternately, the XPS Help is available on the web at
        http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/platform_studio/platform_studio_start.htm.

    b. Navigate to **Procedures for Embedded Processor Design > Simulation > Compiling Simulation Libraries in XPS > Compiling Simulation Libraries in XPS**.

For additional details on the installation process, refer to the ISE Design Suite 12: Installation, Licensing, and Release Notes.

Hardware Requirements for this Guide

In order to complete the design in this guide, you'll need a Spartan®-6 SP605 Evaluation Board and cables.

Creating a New Project

Now that you've been introduced to the Xilinx® Embedded Development Kit (EDK), you'll begin looking at how to use it to develop an embedded system.

The Base System Builder

About the BSB

The Base System Builder (BSB) is a wizard in the Xilinx Platform Studio (XPS) software that quickly and efficiently establishes a working design. You can then customize your design.

At the end of this section, you will have the opportunity to begin your first Test Drive, using the BSB to create a project.

Why Use the BSB?

Xilinx recommends using the BSB wizard to create the foundation for any new embedded design project. While the wizard might be all you need to create your design, if you require more customization, the BSB saves you time by automating common hardware and software platform configuration tasks. After running the wizard, you have a working project that contains all the basic elements needed to build more customized or complex systems.

What You Can Do in the BSB Wizard

Use the BSB wizard to create your project file; choose a board; select and configure a processor and I/O interfaces; add internal peripherals; set up software; and generate a system summary report.

The BSB recognizes the system components and configurations on the selected board, and provides the options appropriate to your selections.

When you create the files, you have the option of applying settings from another project you have created with the BSB.

Selecting a Board Type

The BSB allows you to select a board type from a list or to create a custom board.

Supported Boards

Selecting a Board Type

You can target one of the supported embedded processor development boards available from Xilinx or one of its partners. When you have chosen among the peripherals available on your selected board, the BSB creates a user constraints (UCF) file that includes pinouts for the peripherals you selected. The BSB also creates a completed platform and test application that is ready to be downloaded and run on the board. Each option has functional default values that are pre-selected in Xilinx Platform Studio (XPS). You can further enhance this base-level project in XPS or implement it with utilities provided by ISE®.

When you first install EDK, only Xilinx board files are installed. To target a third party board, you must add the necessary board support files. The BSB Board Selection screen contains a link that helps you find third party board support files. After the files are installed, the BSB drop-down menus display those boards as well.

Custom Boards

If you are developing a design for a custom board, the BSB lets you select and interconnect one of the available processor cores (MicroBlaze™ or PowerPC® processors, depending on your selected target FPGA device) with a variety of compatible and commonly used peripheral cores from the IP library. This gives you a hardware system to use as a starting point. You can add more processors and peripherals, if needed. The utilities provided in XPS assist with this, including the creation of custom peripherals.

Selecting and Configuring a Processor

You can choose a MicroBlaze or PowerPC processor and select:

- Reference clock frequency
- Processor-bus clock frequency
- Reset polarity
- Processor configuration for debug
- Cache setup
- Floating Point Unit (FPU) setting

Selecting and Configuring Multiple I/O Interfaces

The BSB wizard understands the external memory and I/O devices available on your predefined board and allows you to select the following:

- Which devices to use
- Baud rate
- Peripheral type
- Number of data bits
- Parity
- Whether or not to use interrupts

You can open data sheets for external memory and I/O devices from within the BSB wizard.

Adding Internal Peripherals

The BSB wizard allows you to add additional peripherals. The peripherals must be supported by the selected board and FPGA device architecture. For a custom board, only certain peripherals are available for general selection and automatic system connection.

Setting Up Software

Standard input and output devices can be specified in the BSB, and sample C applications are generated. The Software Development Kit (SDK) is recommended for software development, and you'll have the chance to try it as you work through this guide. Sample C applications used in Software Debug Test Drives are generated in SDK.

Viewing a System Summary Page

After you make your selections in the wizard, the BSB displays a system summary page. At this point, you can choose to generate the project, or you can go back to any previous wizard screen and revise the settings.

Device and Board Selections used in Test Drives

This guide uses the Spartan®-6 LX45T Starter Board and targets a MicroBlaze processor. The options you select are listed in "Take a Test Drive! Creating a New Embedded Project," page 14.

If you use a board with an FPGA with a PowerPC 405 (Virtex®-4 FX) or PowerPC 440 (Virtex-5 FXT) processor, either a MicroBlaze or the appropriate PowerPC processor can be used. In almost all cases the behavior of the tools is identical.

The BSB Wizard and the ISE Design Suite

The following test drive walks you through starting your new project in the ISE software and using the New Project wizard to create your project. When your project is created, ISE recognizes that your design includes an embedded processor. ISE automatically starts Xilinx Platform Studio (XPS) and opens the BSB to complete your design.

The Xilinx Microprocessor Project (.xmp) File*

A Xilinx Microprocessor Project (XMP) file is the top-level file description of the embedded system. All project information is saved in the XMP file.

The XMP file is created and handled in ISE like any other source, such as HDL code and constraints files. You'll learn all about that process in the next test drive.

Take a Test Drive! Creating a New Embedded Project

For this test drive, you will start the ISE Project Navigator software and create a project with an embedded processor system as the top level.

1. Start ISE Project Navigator.
2. Select **File > New Project** to open the New Project wizard.
3. Use the information in the table below to make your selections in the wizard screens.

Wizard Screen	System Property	Setting or Command to Use
Create New Project	Name	Choose a name for your project (do not use spaces).
	Location and Working Directory	Choose a location and working directory for your project (again, no spaces).
	Description	You can also add a description for your project (optional).
	Top-level source type	Select **HDL** (default).
Project Settings	Product Category	**All**
	Family	**Spartan6**
	Device	**XC6SLX45T**
	Package	**FGG484**
	Speed	**-3**
	Synthesis Tool	**XST (VHDL/Verilog)**
	Simulator	User-specific[a]
	Preferred Language	**VHDL**
	Accept all other defaults.	
Project Summary	Shows a summary of entries made in the New Project Wizard.	No changes.

a.*Supported simulators are listed in "Installation Requirements: What You Need to Run EDK Tools," page 8.

When you click **Finish**, the New Project Wizard closes and the project you just created opens in ISE Project Navigator.

You'll now use the New Source Wizard to create an embedded processor project.

1. Click the **New Source** button ![icon] on the left-hand side of the Design Hierarchy window. The New Source Wizard opens.

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
Select Source Type	Source Type	Embedded Processor
	File name	system
	Location	Accept the default location.
	Add to project	Leave this checked.
Project Summary	Shows a summary of entries made in the New Source Wizard.	No changes.

After you complete the New Project wizard, ISE recognizes that you have an embedded processor system and starts XPS.

A dialog box appears, asking if you want to create a Base System using the BSB wizard.

3. Click **Yes**.

4. In the Base System Builder Interconnect Type dialog box, select **PLB system** to create a PLB system.

   Note: For information about creating an AXI system, refer to Appendix B, "Creating an AXI-Based Design in EDK."

5. In the Base System Builder wizard, create a project using the settings described in the following table.

   Note: If no setting or command is indicated in the table, accept the default values.

Wizard Screens	System Property	Setting or Command to Use
Welcome to the Base System Builder	Project type options	**I would like to create a new design**.
Board Selection	Board Vendor	**Xilinx**
	Board Name	**Spartan-6 SP605 Evaluation Board**[a]
	Board Revision	**C**
System Configuration	Type of system	**Single-Processor System**.
Processor Configuration	Processor Type	**MicroBlaze**
	System Clock Frequency	**66.67 MHz**
	Local Memory	**16 KB**
	Enable Floating Point Unit	Do not enable the floating point unit.

Wizard Screens	System Property	Setting or Command to Use
Peripheral Configuration	Processor 1 (MicroBlaze) Peripherals list	**Remove** the following peripherals from the "Processor 1 (MicroBlaze) Peripherals" list of default values: • Ethernet_MAC • IIC_DVI • IIC_SFP • PCIe Bridge **Add** the following peripherals: • xps_bram_if_cntlr • Soft_TEMAC • xps_timer (after adding this peripheral, select the **Use Interrupt** check box) • Select the entry for RS232 Uart 1 and select select `xps_uart16550` from the RS232_Uart_1 drop-down list.
Cache Configuration	Instruction/Data caches	Do not enable caches.
Application Configuration	Example Application Options	Do not change the default values for the example applications.

Wizard Screens	System Property	Setting or Command to Use
Summary	System Summary page	After you've selected and configured all of your system components, the BSB displays an overview of the system for you to verify your selections.
		You should have a processor system with the following components:
		• MicroBlaze processor
		• DIP Switch interface
		• Flash interface
		• IIC EEPROM interface
		• 4-bit LED interface
		• DDR3 interface
		• 4-bit pushbutton interface
		• UART
		• Soft TEMAC
		• Compact Flash Interface
		• Two LMB Block RAM interfaces
		• One PLB Block RAM interface
		• Timer
		You can go back to any previous wizard page and make revisions.
		The BSB creates a default memory map. The memory map cannot be modified inside the BSB, but it can be changed after the BSB is finished.

a. The SP605 board contains a Spartan-6 device, which means that the BSB allows you to configure one or more MicroBlaze processors.

6. After reviewing the system summary, click **Finish**.

Read and then dismiss the dialog boxes that appear after you exit the BSB Wizard.

If you've used earlier revisions of this guide, you might notice that the sample design you create here is more complex than previous designs that we've done. There are two reasons for this. First, with the BSB, it's as easy to create a complex design as it is to create a simple one. When a design is created using the BSB, it is guaranteed to close timing and work in hardware. The MicroBlaze design you just created is effectively the same as that used in the targeted design platforms that Xilinx offers.

A Note on the BSB and Custom Boards

If you plan to create a project that includes a customer board, you can create a Xilinx Board Description file (*.xbd) for your custom board library and place it in the Global Repository. For more information about Global Repositories, refer to "Directory Structures for EDK Libraries" in the Platform Studio Online Help, available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/platform_studio/platform_studio_start.htm.

What's Next?

The upcoming sections address Hardware Fundamentals.

- In Chapter 3, "Using Xilinx Platform Studio," you will use the XPS software.
- In Chapter 4, "Working with Your Embedded Platform," you will continue with the hardware design and learn how you can view and modify your new project in XPS.

Chapter 3

Using Xilinx Platform Studio

Now that you have created a baseline project with the Base System Builder (BSB) wizard, it's time to take a look at the options available in Xilinx® Platform Studio (XPS). Using XPS, you can build on the project you created using the BSB. This chapter takes you on a tour of XPS, and subsequent chapters describe how to use XPS to modify your design.

Note: Taking the tour of XPS provided in this chapter is recommended. It enables you to follow the rest of this guide and other documentation on XPS more easily.

What is XPS?

XPS includes a graphical user interface that provides a set of tools to aid in project design. This chapter describes the XPS software and some of the most commonly used tools.

The XPS Software

From the XPS software, you can design a complete embedded processor system for implementation within a Xilinx FPGA device. The XPS main window is shown in the following figure.

Optional Test Drives are provided in this chapter so you can explore the information and tools available in each of the XPS main window areas.
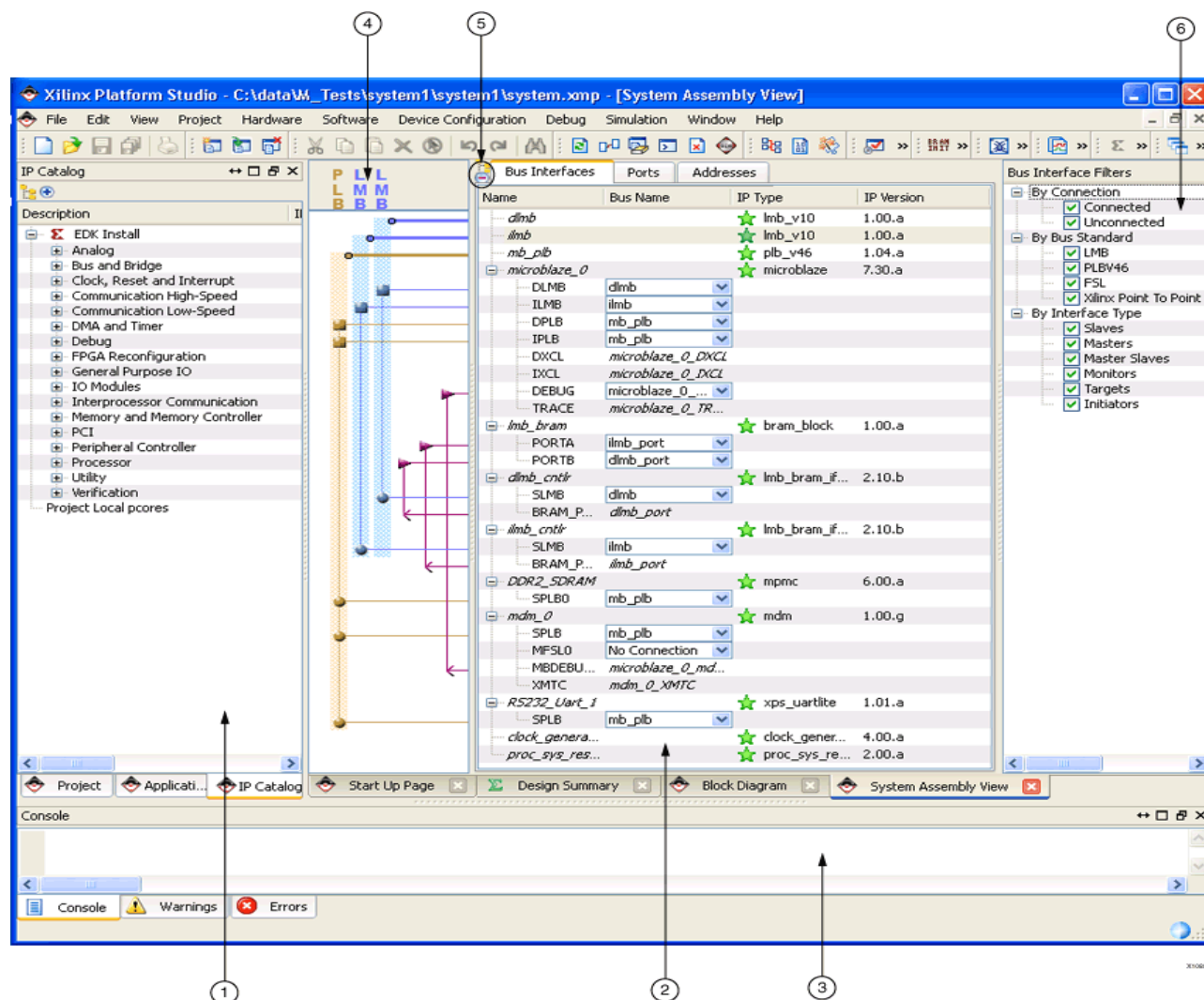
Figure 3-1: **XPS Project Window**

Using the XPS User Interface

The XPS main window is divided into these three areas:

- Project Information Area (1)
- System Assembly View (2)
- Console Window (3)

The XPS main window also has labels to identify the following areas:

- Connectivity Panel (4)
- View Buttons (5)
- Filters Pane (6)

Project Information Area

The Project Information Area offers control of and information about your project. The Project Information Area includes the Project, Applications, and IP Catalog tabs.

Project Tab

The Project Tab, shown in the following figure, lists references to project-related files. Information is grouped in the following general categories:

- Project Files

  Project-specific files such as the Microprocessor Hardware Specification (MHS) files, Microprocessor Software Specification (MSS) files, User Constraints File (UCF) files, iMPACT Command files, Implementation Option files, and Bitgen Option files.

- Project Options

  Project-specific options, such as Device, Netlist, Implementation, Hardware Description Language (HDL), and Sim Model options.

- Design Summary

  A graphical display of the state of your embedded design and gives you easy access to system files.
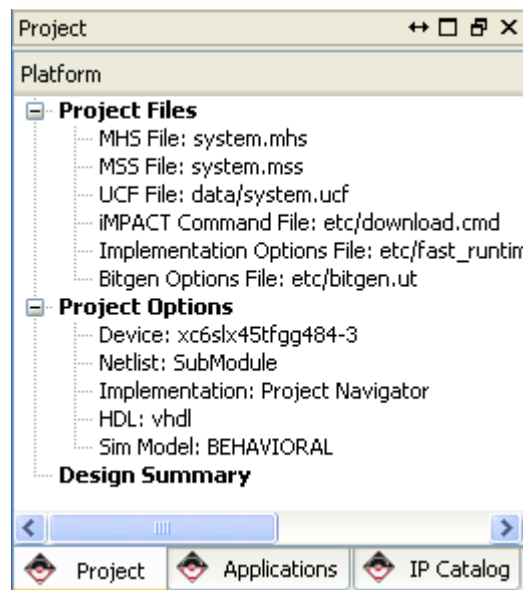
Figure 3-2: **Project Information Area, Project Tab**

Applications Tab

The Applications tab, shown in Figure 3-3, lists the software application option settings, header files, and source files that are associated with each application project. With this tab selected, you can:

- Create and add a software application project, build the project, and load it to the block RAM.

- Set compiler options.

- Add source and header files to the project.

Note: You can create and manage software projects in XPS; however, SDK is the recommended tool for software development.
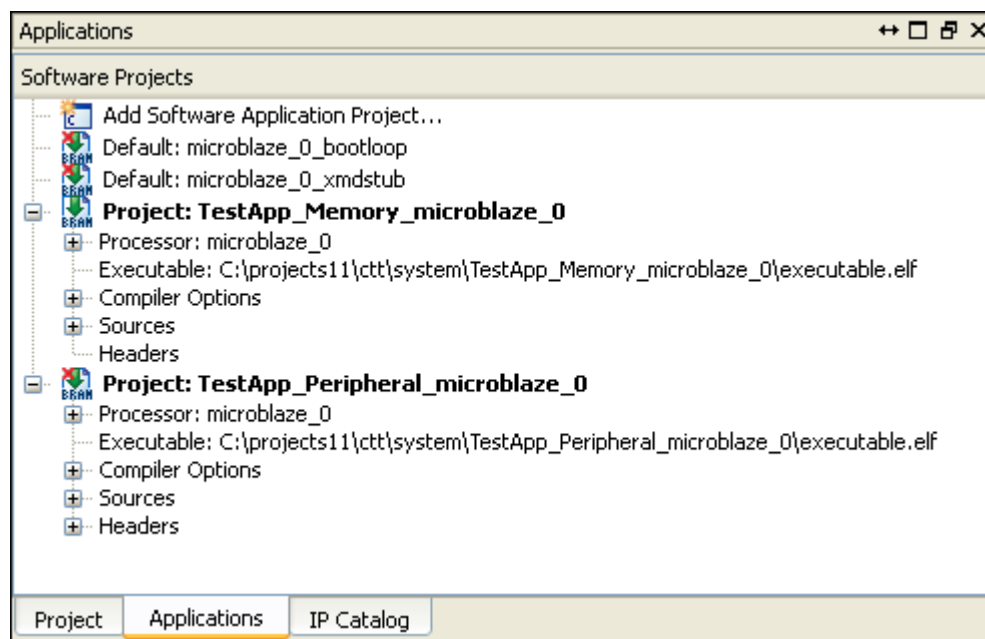
Figure 3-3: **Project Information Area, Applications Tab**

IP Catalog Tab

The IP catalog tab (shown in Figure 3-1), lists information about the IP cores, including:

- Core name and licensing status (not licensed, locked, or unlocked)

- Release version and status (active, early access, or deprecated)

- Supported processors

- Classification

Additional details about the IP core, including the version change history, data sheet, and the Microprocessor Peripheral Description (MPD) file, are available when you right-click the IP core in the IP Catalog tab. By default, the IP cores are grouped hierarchically by function.

Note: You might have to click and drag to expand the pane to view all details of the IP.

Take a Test Drive! Reviewing the Project Information Area

1. With your project open in XPS, click the **Project** tab.

2. Right-click any item under Project Files and select **Open**. In future Test Drives, you will edit some of these files. In particular, the system.mhs file contains a text representation of your entire embedded system.

3. Close the file by selecting **File > Close**.

4. Right-click any item in the Project Options category to open the Project Options dialog box. Alternatively, you can select **Project > Project Options**.

5. Close the Project Options dialog box.

6. Click the **IP Catalog** tab.

7. At the top left of the IP Catalog window, note the two buttons. Click them and observe how they change how the IP catalog displays.

8. Right-click any item in the IP Catalog to see what options are available.

   Note: You might need to expand the selection by clicking the plus sign next to the IP description.

   Notice a few parts of the IP Catalog in particular:

   – **Add IP**, which adds the selected IP to your design

   – **View PDF Datasheet**, which brings up the datasheet for the IP

   – **View IP Modifications (Change Log)**, which lists the revision history for the selected IP.

9. Find and expand the **Communication Low-Speed** IP category.

10. Right-click the xps_uart16550 IP Type peripheral and select **View PDF Datasheet** to view the related PDF datasheet in your PDF viewer. Similar data sheets are available for all embedded IP.

System Assembly View

The System Assembly View allows you to view and configure system block elements. If the System Assembly View is not already maximized in the main window, click and open the **System Assembly View** tab at the bottom of the pane.

Bus Interface, Ports, and Addresses Tabs

The System Assembly View comprises three panes, which you can access by clicking the tabs at the top of the view.

- The **Bus Interface** tab displays the buses in your design. Use this view to modify the information and connections for each bus.

- The **Ports** tab displays ports in your design. Use this view to modify the details for each port.

- The **Addresses** tab displays the address range for each IP instance in your design. Click **Generate Addresses** to automatically generate the system address map.

Connectivity Panel

With the Bus Interfaces tab selected, you'll see the Connectivity Panel (label 4 in Figure 3-1, page 20), which is a graphical representation of the hardware platform connections. You can hover your mouse over the Connectivity Panel to view available bus connections.

A vertical line represents a bus, and a horizontal line represents a bus interface to an IP core. If a compatible connection can be made, a connector is displayed at the intersection between the bus and IP core bus interface.

The lines and connectors are color-coded to show bus compatibility. Differently shaped connection symbols indicate whether IP blocks are bus masters or bus slaves. A hollow connector represents a connection that you can make. A filled connector represents an existing connection. To create or disable a connection, click the connector symbol.

Filters Pane

XPS provides filters that you can use to change how you view the Bus Interfaces and Ports in the System Assembly View. The filters are listed in the Filters pane (label 6 in Figure 3-1, page 20) when the Bus Interfaces or Ports tabs are selected. Using these filters can unclutter your connectivity panel when creating a design with a large number different buses.

View Buttons

So you can sort information and revise your design more easily, the System Assembly View provides two buttons that change how the data is arranged (label 5 in Figure 3-1, page 20):

- **Change to Hierarchical/Flat View** button
  - The default display is called *hierarchical view.* The information that is displayed for your design is based on the IP core instances in your hardware platform and organized in an expandable tree structure.
  - In *flat view,* you can sort the information alphanumerically by any column.

- **Expand/Collapse All Tree Nodes** button

  The +/- icon expands or collapses all nets or buses associated with an IP to allow quick association of a net with the IP elements.

Take a Test Drive! Exploring the System Assembly View

1. Click the **Ports** tab located at the top of the screen.

2. Expand the **External Ports** category to view the signals that leave the embedded system (and ultimately the FPGA device).

3. Note the signal names in the **Net** column and find the signals related to the `fpga_0_RS232_Uart_1` ports. (You might need to drag the right side of the **Net** column header to see its entire contents.) These signals are referenced in the next step.

4. Scroll down, locate, and expand the `RS232_Uart_1` peripheral.

   Note the net names and how they correspond to the names of external signals. The **sin (serial in)** and **sout (serial out)** net from the UART are name-associated with the external ports.

5. Right-click the `RS232_Uart_1` peripheral and select **Configure IP** to launch the associated IP Configuration dialog box. You can open a similar configuration dialog box for any peripheral in your system.

    a. Observe what happens when you hold the mouse cursor over a parameter name.

    b. Note the three top tabs and buttons available for this core.

    c. Close this dialog when finished.

6. Click the **Change to Hierarchical/Flat View** button and see how the display changes.

Any changes you make in the System Assembly View immediately cause XPS to update the `system.mhs` file. You can open this file from the Project Files area, as shown in Figure 3-2.

Console Window

The Console window (label 3 in Figure 3-1, page 20) provides feedback from the tools invoked during runtime. Notice the three tabs: Console, Warnings, and Errors.

Start Up Page

The Start Up page has information relevant to your version of XPS, including sets of links for release information and design flows. There is also a tab to help you locate EDK documentation.

If the Start Up page isn't already open, select **Help > View Start Up Page** to open it.

XPS Tools

In addition to the software interface, XPS includes the underlying tools you need to develop the hardware and software components of an embedded processor system:

- The Base System Builder (BSB) wizard, for creating new projects. You can start the BSB from the New Project dialog box that appears when you start XPS, or you can select **File > New Project** from the XPS main window.

- The Hardware Platform Generation tool, Platgen, for generating the embedded processor system. To start Platgen, click **Hardware > Generate Netlist**.

- The Simulation Model Generation tool, Simgen, generates simulation models of your embedded hardware system based on your original embedded hardware design (behavioral) or finished FPGA implementation (timing-accurate).
Click **Simulation > Generate Simulation HDL Files** to start Simgen.

- The Create and Import Peripheral wizard helps you create your own peripherals and import them into EDK-compliant repositories or XPS projects. To start the wizard, click **Hardware > Create or Import Peripheral**.

- The Library Generation tool, Libgen, configures libraries, device drivers, file systems, and interrupt handlers for the embedded processor system. Click **Software > Generate Libraries and BSPs** to start Libgen.

Take a Test Drive! Reviewing the XPS Structure

You can access the XPS tools described above in the **Hardware** and **Simulation** toolbar menus. Take a look at them and familiarize yourself with the options that are available.

XPS Directory Structure

For the Test Drive design you started, the BSB has automated the set up of the project directory structure and started a simple but complete project. The time savings that the BSB provides during platform configuration can be negated if you don't understand what the tools are doing behind the scenes. Take a look at the directory structure the BSB created and see how it could be useful as the project development progresses.

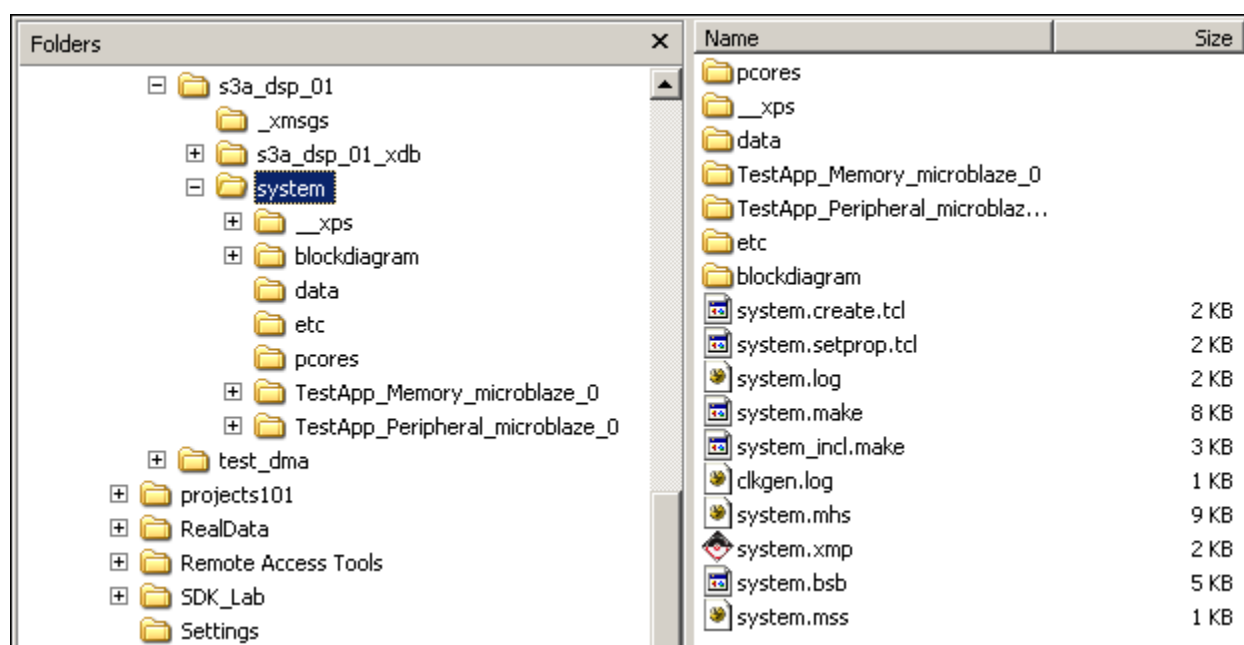**Note:** The files are stored in the location where you created your project file.

Figure 3-4: **File Directory Structure**

Directory View

The BSB automatically creates a project directory with the name of your embedded system source. This directory contains the subdirectories for your project in the repository search path, shown in Figure 3-4:

`__xps`	Contains intermediate files generated by XPS and other tools for internal project management. You will not use this directory.
`blockdiagram`	Contains files related to the block diagram.
`data`	Contains the user constraints file (UCF). For more information on this file and how to use it, see the ISE® UCF help topics at: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/manuals.pdf.
`etc`	Contains files that capture the options used to run various tools. This directory is empty because no actions outside of the BSB have been performed.
`pcores`	Used for including custom hardware peripherals. The `pcores` directory is described in more detail in Chapter 5, "Introducing the Software Development Kit."

Two other directories contain the files generated by the BSB:

• `TestApp_Memory_microblaze_0`

• `TestApp_Peripheral_microblaze_1`

These directories contain test application C-source code, header files, and linker scripts. Although they are there and available for your use, you are going to use sample applications created in SDK in this guide. These topics are covered in upcoming chapters.

In the main project directory, you will also find a few individual files. Those of interest are:

`system.xmp`	This is the top-level project design file. XPS reads this file and graphically displays its contents in the XPS user interface.
`system.mhs`	The system Microprocessor Hardware Specification, or MHS file, captures the system elements, their parameters, and connectivity in a text format. The MHS file is the hardware foundation for your project.
`system.mss`	The system Microprocessor Software Specification, or MSS file, captures the software portion of the design, describing the system elements and various software parameters associated with the peripheral in a text format. The MSS file is the software foundation for your project.

The MHS and MSS files are the main products of your XPS design. Your entire hardware and software system is represented by these two files.

Take a Test Drive! Exploring the Directory Structure

In this Test Drive, you'll take a first-hand look at the XPS directory structure.

1. Using a file explorer utility, such as Window Explorer, navigate to the top-level directory of your project.

2. Open the various subdirectories and become familiar with the basic file set.

What's Next?

Now that you know your way around XPS, you are ready to begin working with the project you started. You'll continue with Chapter 4, "Working with Your Embedded Platform."

Working with Your Embedded Platform

What's in a Hardware Platform?

The embedded hardware platform includes one or more processors, along with a variety of peripherals and memory blocks. These blocks of IP use an interconnect network to communicate. Additional ports connect to the "outside world," which could be the rest of the FPGA or outside of the FPGA entirely. The behavior of each processor or peripheral core can be customized. Implementation parameters control optional features and specify how the hardware platform is ultimately implemented in the FPGA.

Hardware Platform Development in Xilinx Platform Studio

About the Microprocessor Hardware Specification (MHS) File

Xilinx® Platform Studio (XPS) provides an interactive development environment that allows you to specify all aspects of your hardware platform. XPS maintains your hardware platform description in a high-level form, known as the Microprocessor Hardware Specification (MHS) file. The MHS, which is an editable text file, is the principal source file representing the hardware component of your embedded system. XPS synthesizes the MHS source file into netlists ready for the FPGA place and route process using an executable called Platgen.

The MHS file is integral to your design process. It contains all peripheral instantiations along with their parameters. The MHS file defines the configuration of the embedded processor system. It includes information on the bus architecture, peripherals, processor, connectivity, and address space. For more information about the MHS file, refer to the "Microprocessor Hardware Specification (MHS)" chapter of the *Platform Specification Format Reference Manual*, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

Take a Test Drive! Examining the MHS File

In this Test Drive, you'll take a quick tour of the MHS file that was created when you ran the BSB wizard.

1. Select the **Project** tab in the Project Information Area of the XPS software.

2. Look under the Project Files heading to find `MHS File:system.mhs`. Double-click the file to open it.

3. Search for `xps_uart16550` in the `system.mhs` file by selecting **Edit > Find** and using the Find tool that appears below the main window area.

   Note the line in the MHS file that states:

   `PORT sin = fpga_0_RS232_Uart_1_sin_pin`

4. Search the file for another instance of the port name `fpga_0_RS232_Uart_1_sin_pin`. You'll find it at the top of the file as a PORT.

   When a PORT is shown inside of a BEGIN/END pair, as it is here, it's a port on a piece of IP. When you see a PORT at the top of the MHS, it connects the embedded platform to the outside world.

5. Take some time to review other IP cores in your design. When you are finished, close the `system.mhs` file.

The Hardware Platform in System Assembly View

The System Assembly View in XPS displays the hardware platform IP instances in an expandable tree and table format.

XPS provides extensive display customization, sorting, and data filtering capability so you can easily review your embedded design. The IP elements, their ports, properties, and parameters are configurable in the System Assembly View and are written directly to the MHS file.

Editing a port name or setting a parameter takes effect when you press **Enter** or click **OK**. XPS automatically writes the system modification to the hardware database, which is contained in the MHS file.

Hand-editing the MHS file is not recommended, especially when you're just starting out with XPS. The recommended method of forcing changes in the MHS file is to use the features of the System Assembly View. As you gain experience with XPS and the MHS file, you can also use the built-in text editor to make changes.

Note: Additional information about adding, deleting, and customizing IP are described in Chapter 7: "Creating Your Own Intellectual Property."

Converting the Hardware Platform to a Bitstream

For a design to work in an FPGA, it needs to be converted to a bitstream. This conversion is a three-step process. First, XPS generates a netlist that is representative of your embedded hardware platform. Next, the design is implemented (mapped into FPGA logic) in the ISE® Design Suite tools. In the final step, the implemented design is converted to the bitstream that can be then downloaded to the FPGA.

Note: In the examples used in this guide, the design implemented in the FPGA consists only of the embedded hardware platform. Typical FPGA designs also include logic developed outside of XPS.

Generating the Netlist

When you instruct XPS to generate the netlist, it invokes the platform building tool, Platgen, which does the following:

- Reads the design platform configuration MHS file.

- Generates a Hardware Description Language (HDL) representation of the MHS file written to `system.[vhd|v]` along with a `system_stub.[vhd|v]` file. The system file is your MHS description written in HDL format.

  Note: The `system_stub` file is a top-level HDL template file that could be used to instantiate your processor system as a component in a larger, HDL-based design. The easier way, however, is to just use the `system.xmp` file, as you'll use in the examples in this guide. To see the created HDL files, look in the *<project_name>*\system\hdl directory.

- Synthesizes the design using Xilinx Synthesis Technology (XST).

- Produces a netlist file (with an `.ngc` extension)

More information about Platgen is provided in the "Platform Generator (Platgen)" chapter of the *Embedded System Tools Reference Manual*, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

You can control netlist generation from within XPS or from the ISE Project Navigator interface. In the sections ahead, we will be doing the actual netlist generation from within the ISE interface.

Exporting Your Hardware Platform

In the Embedded Development Kit (EDK), your hardware platform is designed in XPS and your software is developed and debugged in the Software Development Kit (SDK). The information about your hardware platform is required for SDK, so you will export a file called `system.xml`.

The system.xml File   The `system.xml` file has the information SDK requires for you to do software development and debug work on the hardware platform that you designed.

Take a Test Drive! Exporting Your Hardware Platform to SDK

1. In the XPS software, select **Project > Export Hardware Design to SDK**.

2. A directory location is selected for you and can't be changed when the design is exported from XPS. The ultimate repository search path to the `system.xml` file is then `\system\SDK\SDK_Export\hw\..` in your project directory.

   Note: There are other XML files used in XPS, so it is important to know the location of the XML file that you will be using.

3. Select **Export Only**. You'll run several Test Drives of SDK in upcoming chapters.

What Just Happened?

It is important to understand the details of the export operation, especially if you are managing multiple hardware versions.

When you select **Export Only**, a utility creates a number of files used by SDK. In addition to the XML file, documentation on the software drivers and hardware IP is included so you can access necessary information from within SDK.

The other option, **Export & Launch SDK**, automatically overwrites any existing XML files that already exist in the export directory. Any existing bitstream (BIT) and Block Memory Map (BMM) files in the export directory are erased. Because your project was instantiated in Project Navigator, the option to automatically include the bitstream and BMM files is unavailable.

Note: When, in future projects, you export a project created in BSB and select the **Include bitstream and BMM files** option, XPS saves these files in the export directory. This process prevents the export directory from containing hardware files that are out of synchronization.

In the `\system\SDK\SDK_Export\hw` directory, a number of HTML files are created in addition to the `system.xml` file. Opening the `system.html` file shows a hyperlink-enabled block diagram with all of the details of your embedded hardware platform.

Take a Test Drive! Generating the Bitstream

Now that you've described your Hardware Platform in XPS, you'll use the ISE Project Navigator software to implement the design and generate the bitstream.

Implementing the Design in ISE using Project Navigator

Compiled C code is not part of that bitstream. It is added later in SDK.

1. Review the Project Navigator main window. The Design panel on the left side should look like this:
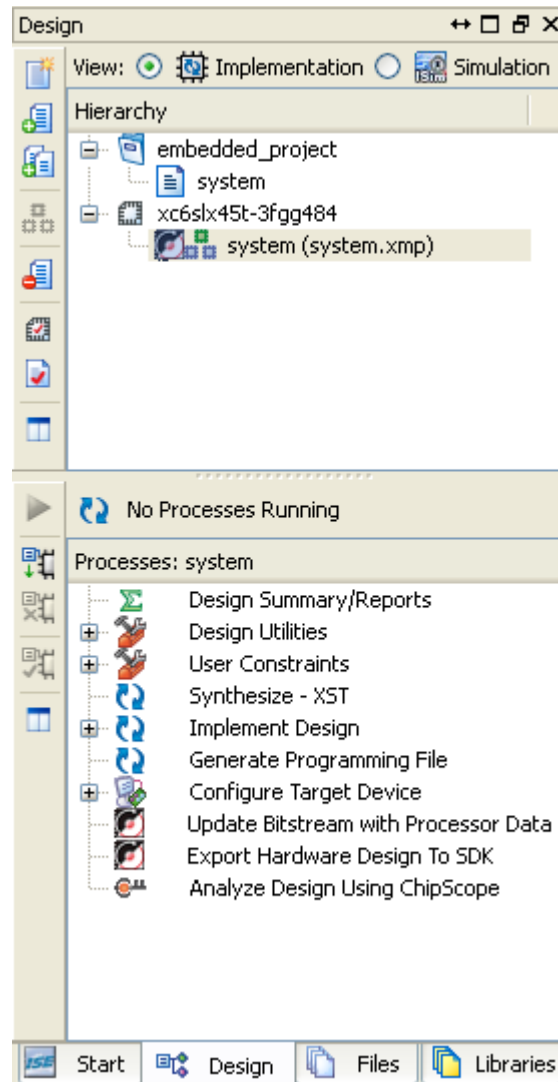
Figure 4-1: **ISE Project Navigator Design Area**

Generating a Bitstream and Creating a UCF file

You're about to run the design through to the point at which a bitstream is generated. But before you can do that, you need to add some information so that the ISE Place and Route (PAR) tool has information about your design, such as the pinout and the speed at which it needs to run.

As you learned earlier, that information is included in the UCF file. When you run the BSB, a UCF is created for you.

2. In the Project Navigator software, select **Project > Add Source**, navigate from your project search repository to `system/data`, and add the `system.ucf` file.

    This step is only necessary when the XMP source is the top-level design in an ISE project. If the XMP is instantiated in a VHDL or Verilog file, ISE manages the EDK UCF file.

3. The Adding Source Files dialog box appears to show the progress of processing the UCF file. When the file processing completes, click **OK**.

    Your Sources window should now look like this:
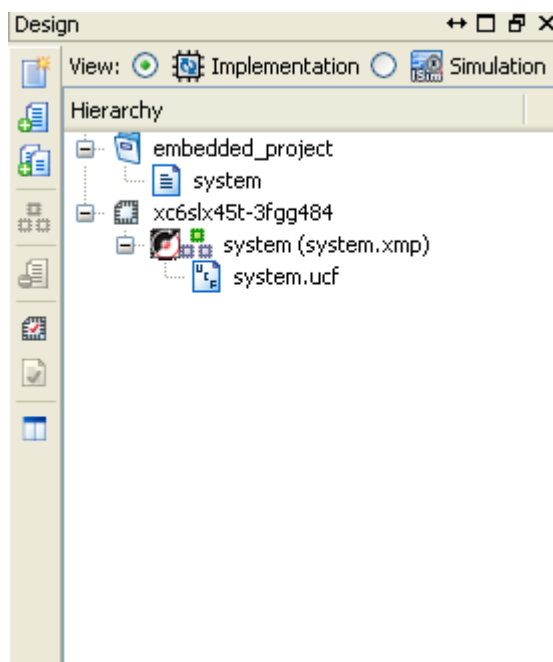
Figure 4-2: **The system.xmp and the system.ucf Files**

Notice that the file `system.ucf` is added and is associated with the embedded system design (`system.xmp`).

4. Click to select the system.xmp item in the Design window.

EDK system.bit and BmmFile_bd.bmm Files

5. In the Processes pane, double-click **Generate Programming File** to create your bitstream. It takes a few minutes and should conclude with the message "Process 'Generate Programming File' completed successfully."

    The generated bitstream is called `system.bit`. There is another file generated called `edkBmmFile_bd.bmm`, which is used by the SDK for loading memory onto your target board.

Make a mental note of both of these files and their locations in the root of your hardware project. These files are used in subsequent chapters.

What's Next?

Now you can start developing the software for your project using SDK. The next two chapters explain embedded software design fundamentals.

Introducing the Software Development Kit

The Xilinx® Software Development Kit (SDK) facilitates the development of embedded software application projects. SDK is a complementary program to XPS. Using SDK, you develop the software that is used on the embedded platform built in XPS. SDK is based on the Eclipse open source tool suite. For more information about Eclipse, see http://www.eclipse.org.

About SDK

In the ISE 12 release, some of the terminology is different than in previous versions of ISE.

The terms used in SDK are:

- Software project
- Hardware platform
- Board support package
- Perspectives
- Views

SDK Terminology

You start your design by creating a *software project*. The SDK environment can manage multiple software projects. When you create your software project, SDK prompts you to create a *hardware platform* and a *board support package* (BSP).

The hardware platform is the embedded hardware design that is created in XPS. The hardware platform includes the XML-based hardware description file, the bitstream file, and the BMM file. When you import the XML file into SDK, you import the hardware platform. In the ISE Design Suite 12 release, multiple hardware platforms can exist in a single workspace.

The BSP is a collection of libraries and drivers that form the lowest layer of your application software stack. Your software applications must link against or run on top of a given software platform using the provided Application Program Interfaces (APIs).

You can have SDK create board support packages for two different run-time environments:

Board Support Package Types in SDK

- **Standalone** - A simple, semi-hosted and single-threaded environment that provides basic features such as standard input/output and access to processor hardware features.

- **Xilkernel** - A simple and lightweight kernel that provides POSIX-style services such as scheduling, threads, synchronization, message passing, and timers.

In SDK, multiple board support packages can exist simultaneously. For example, you might have a BSP for a design that runs on the standalone environment, and one that uses Xilkernel.

After you set up your board support package (or packages), SDK creates *software projects*.

Perspectives and Views

SDK looks different depending on what activity you are performing. When you are developing your C or C++ code, SDK displays one set of windows. When you are debugging your code on hardware, SDK appears differently and displays windows specific to debugging. When you are profiling code, you use the gprof view. These different displays of windows are called *perspectives*, and each window in the perspective is called a *view*.

Note: Profiling is not covered in this guide. For more information about Profiling in EDK, refer to the EDK Profiling User Guide.

Take a Test Drive! Creating a Hardware Platform

1. Double-click the Xilinx Software Development Kit desktop icon, or select **Programs > ISE Design Suite 12 > EDK > Xilinx Software Development Kit** from the Windows Start menu.

2. Select a workspace. This is the folder in which your software projects are stored. To create a new workspace, make a new folder in your project directory. For this example, create a new workspace called **SDK_Workspace**.

   Caution! Make sure the path name does not have spaces.

3. SDK opens to the Welcome screen. We won't spend a lot of time looking at this right now. You can re-open it at any time by selecting **Help > Welcome**.

4. Create a new Hardware Platform by selecting **File > New > Xilinx Hardware Platform Specification**.

   The New Hardware Project dialog box opens.

5. For this example, name the project **CTT_Hardware_Platform** and identify the system.xml file that you exported in Chapter 4. The default location for this file is in the SDK/SDK_Export/hw subdirectory of the ISE Project directory. Check the timestamp on the file to make sure that you are pointing to the correct file.

6. Click the arrow next to **Bitstream and BMM Files** to expand that section of the dialog box.

7. In the **Bitstream** and **BMM File** fields, select the `system.bit` and `edkBmmFile_bd.bmm` files, respectively, for your project. These files are located in the top-level folder of the ISE project you created.

8. Click **Finish**.

   SDK opens with a number of views. The most notable of these views are the Project Explorer, which at this time only displays your hardware platform, and the `system.xml` file, which opens in its own view. Take a moment to review the contents of the `system.xml` file.

Take a Test Drive! Creating a Board Support Package

Now that you have imported your hardware platform, you need to create a corresponding software platform.

1. Still working in SDK, select **File > New > Xilinx Board Support Package**.

   Recall that there can be multiple BSPs for a given embedded design. The first one you create in this Test Drive is a standalone (as opposed to a Xilkernel) project.

2. Populate the new BSP Project with the following selections:

   – Project name: **standalone_bsp_0**

   – Project Location: **Use default location**

   – Hardware Platform: **CTT_Hardware_Platform**

   Note: If your embedded system had more than one processor, indicate in the CPU field which processor the board support package is targeting.

   – Board Support Package OS: **standalone**

3. Click **Finish**.

4. Click **OK** in the Board Support Package Settings dialog box.

   To re-open this dialog box later, right-click the project name and select **Board Support Package Settings**.

What Just Happened?

SDK examined your hardware specification file (`system.xml`), along with the type of board support package that you selected, and compiled the appropriate libraries corresponding to the components of your hardware platform. You can view the log for this process in the Console view.

Expand the **microblaze_0** section under `standalone_bsp_1` in the Project Explorer tab. The `code`, `include`, `lib`, and `libsrc` folders contain the libraries for all of the hardware in your embedded design.

Double-click any of the files in this view to view them in the SDK Editor area.

Take a Test Drive! Setting Up the Software Environment

Now you can begin writing code that targets the embedded processor design which you built in the previous chapters. To do this, create a Xilinx C Project for `standalone_bsp_0`.

1. Select **File > New > Xilinx C Project**.

    There are a number of sample applications available. You are going to start with a simple "Hello World" application.

2. Select the "Hello World" Sample Project Template. The Project name fills in automatically with **hello_world_0**.

3. For the project location, make sure that the **Use default location** check box is selected

4. Click **Next**.

5. Select the **Target an existing Board Support Package** check box.

6. Click **Finish**.

    The `hello_world_0` sample application builds automatically, producing an ELF file suitable for downloading onto the target hardware.

    The C/C++ Projects tab now contains information related to the software platform and the software project. The relevant project management information is displayed in this window.
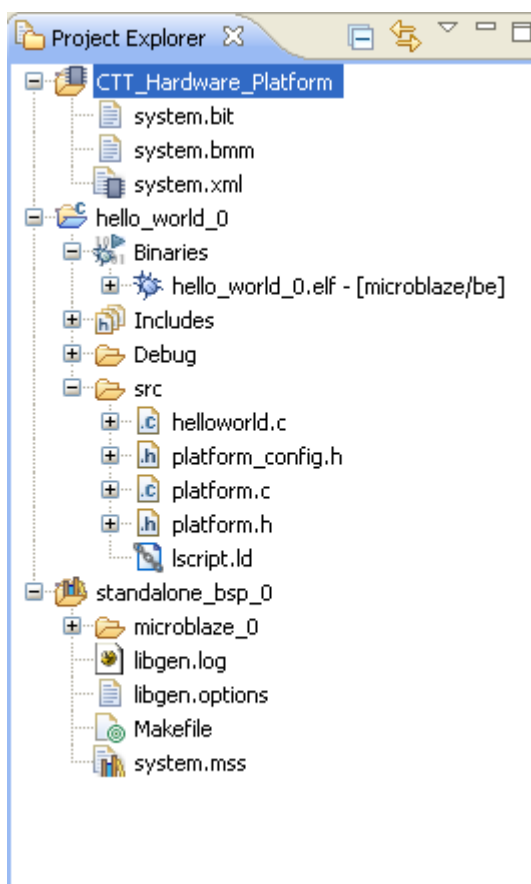

Figure 5-1:   **Project Files Displayed in the Project Explorer Tab**

7. In the Project Explorer window, expand the `src` folder in the `hello_world_0` software project. Notice the `Binaries` folder, under which is the ELF file that will be downloaded to the target board.

8. Double-click the `helloworld.c` file to open it in the SDK Editor window. You can modify the sample code or create your own.

You can also see lscript.`ld`, the default linker script that was generated for this project. A linker script is required to specify where your software code is loaded in the hardware system memory. You can customize this linker script for your design by right-clicking the `lscript.ld` file and selecting **Properties**.

You now have a complete framework for editing, compiling, and building a software project. The next step is debugging, which you will do in the next Test Drive.

Take a Test Drive! Debugging in SDK

Debugging is the process of downloading and running C code on the target hardware to evaluate whether the code is performing correctly. Before you can begin debugging, you must set up your SP605 board as follows:

1. Connect two mini-USB cables between your computer and the two mini-USB jacks on the SP605 board.

    One of the USB connections connects to a JTAG download and debug interface built into the SP605 board.

    The other USB connection is a USB-to-RS232 Bridge. In order for your PC to map the USB port to a COM port, you must download the appropriate driver from Silicon Labs.

2. When the Windows Found New Hardware Wizard opens, select the option to have the wizard find the driver for the hardware. You will have to do this multiple times.

3. Install the CP210x VCP drivers that came with your SP605 board. You can also find these drivers on the Silicon Labs website:
    http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx

4. Turn on the power to your SP605 board.

Download Bitstream with Bootloop

Because this is an FPGA, you must configure it with a bitstream that loads a design into the FPGA. In this case, the design is an embedded processor system.

1. In SDK, select **Xilinx Tools > Program FPGA**.

    The bitstream (BIT) and block memory map (BMM) files are automatically populated for you.

2. Select **Program**. When the Programming completes, your FPGA is configured with your design.

    At this point, you have downloaded the bitstream to the FPGA and initialized the microprocessor with a single-instruction "branch-to-itself" program called "bootloop."Bootloop keeps the processor in a known state while it waits for another program to be downloaded to run or be debugged.

3. In the Project Explorer, under Binaries, right-click `hello_world_0.elf` and select **Debug As > Launch on Hardware**.

The executable is downloaded to the hardware where specified in the linker script. The "STDIO connections" are disabled by default.

A dialog box appears, informing you that the perspective is about to change from C/C++ to Debug.

4. Open a Hyperterminal (or another terminal emulation program) and set the display to 9600 baud, 8 bit data, 1 stop bit. Be sure to set the COM port to correspond to the COM port that the Silicon Labs driver is using.

The Debug Perspective

5. In the Debug Perspective, the C code is now highlighted at the first executable line of code (you might need to scroll to view helloworld.c). The debug window shows that for `Thread[0]` the `main()` function is currently sitting at line28 because there is an automatically-inserted breakpoint.

Note: If your display does not show line numbers, you can turn them on by right-clicking in the window, selecting **Preferences**, then navigating to **General > Editors > Text Editors** and selecting the **Show line numbers** check box.

6. Execute the code by clicking the **Resume** button or pressing **F8** on your keyboard.

7. Terminate the debug session by clicking the **Terminate** button or pressing **Ctrl + F2** on your keyboard.

8. The output in the terminal window displays "Hello World." When you are finished, close SDK.

What Just Happened?

The code you executed in SDK displays a classic "Hello World" message in the terminal window to demonstrate how simply software can be executed using SDK.

What's Next?

This chapter showed you how to set up an SDK project, download a bitstream to a target board, and execute a simple program.

In the next chapter, you'll be digging deeper into SDK as you create a new software project, use the source code management, and explore debugging in greater depth.

More on the Software Development Kit: Edit, Debug, and Release

The Xilinx® Software Development Kit (SDK) can be used for the entire lifecycle of the software development process. This lifecycle consists of creating, editing, and building your software projects, debugging your software on target hardware, perhaps profiling it on your target hardware, and then releasing your software and optionally programming it into Flash memory. All of these activities can be done in SDK. In this chapter, we'll look more at the first two items on this list: software development and debug.

SDK Drivers and Windows

Before getting started with SDK, you need to know about the "low-level" drivers that Xilinx provides. You also need to understand the layout of windows in the SDK software.

More on Drivers

The "low-level" drivers that Xilinx provides are located in the `\EDK\sw\XilinxProcessorIPLib\drivers` directory of your EDK installation area. Here, you will see a directory for each peripheral's driver. There are drivers corresponding to each piece of hardware available to you in Platform Studio. For each driver, the directory contains source code, HTML documentation on the driver, and examples of how the drivers can be used.

SDK Windows

As demonstrated in the previous chapter, SDK has different workspaces, called *perspectives*.

So far we've worked in the C/C++ perspective and the Debug perspective. The other perspective built into SDK is the Profiling perspective.

Note: Profiling in SDK is not covered in this guide. For information about profiling, refer to the *EDK Profiling Guide*.

Whether you are working in the C/C++ Perspective or the Debug perspective, you'll find the SDK windowing system very powerful. There are two kinds of windows within perspectives: *editing* windows and *informational* windows. The editing windows, which contain C or C++ source code, are language-specific and syntax aware. Right-click an item in an editing window to open a comprehensive list of actions that can be done on that item.

Informational windows are particularly flexible. You can have as many informational windows as you like. An informational window can have any number of views, each of which is indicated by a tab at the top of the window. Views in the Debug perspective include Disassembly, Register, Memory, and Breakpoints.

Views can be moved, dragged, and combined in any number of ways. Click any tab on any window in either the C/C++ or Debug Perspective or drag it to another window. Its contents are displayed in the new window. To see the views available for a given perspective, select **Window > Show View**.

Experiment with moving windows around. The ability to easily customize your workspace is one of the more powerful features of SDK. SDK remembers the position of the windows and views for your perspective within a project.

Take a Test Drive! Editing Software

In the previous chapter, you compiled and debugged a sample software module. In this next test drive, you'll run two more sample modules and create a third software module from scratch to call the first two routines. This will give you a bit more experience managing source files for multiple projects.

Setting Up Your Workspace

1. Start a new SDK Session by double-clicking the SDK icon on your desktop.

2. Create a new workspace and save it anywhere on your system. Do not use spaces in the filename or path.

3. Import the same hardware platform used in "Take a Test Drive! Creating a Hardware Platform" in Chapter 5. Select **File > New > Xilinx Hardware Platform Specification**. Name the project **Advanced_CTT_Project**. The system.xml file is located in the <project home>\system\SDK\SDK_Export\hw subdirectory.

Creating New Xilinx C Projects

Now that the SDK project space is set up correctly, you can create a new Xilinx C project. SDK automatically creates the Board Support Package if one hasn't been created yet.

1. Create a new Xilinx C Project. Call it **`Editor_C_Project`** and make it a **Hello World** sample application.

2. Click **Next**.

   A dialog box opens, asking if you want a new Board Support Package. For this test drive, click **Finish** to have SDK create the new BSP.

   In the next few steps, you will create two more Xilinx C Projects, each with a different Sample Application. We will then show how to call them from the hello_world applications. While this isn't a complex process, you must be familiar with this fundamental type of file management in order to create larger, real-life projects. If you need a refresher at any time, review the project management steps done in the Test Drives in Chapter 5.

3. Create two more Xilinx C Projects using the same technique that was used in step 1. Use the Memory Test and Peripheral Test sample applications. For both projects, select the **Target an Existing Board Support Package** check box and identify the "hello_world_bsp_0{OS:standalone}" BSP.

Running Your Applications

Before you can run these two applications, download the FPGA's bitstream to the board, as you did in Chapter 5.

1. Select **Xilinx Tools > Program FPGA**.

2. Populate the Program FPGA dialog box by selecting the locations of the Bit and Bmm files for your project, and then click **Save and Program**.

   We will now observe what the two sample programs do. You'll run the memory_test application and then the peripheral_tests application.

3. Open a hyperterminal session and be sure it's set to 9600-8-N-1.

4. In the project management area, right-click `memory_tests_0.elf` under the hierarchy of `memory_tests_0/Binaries/`.

5. Select **Debug As > Launch on Hardware**. If a confirmation dialog box appears, click **Yes** to confirm the Perspective Switch. The Debug perspective opens.

6. Select **Run > Resume** to run the program. The program output displays on your terminal window.

   If the flash memory test runs, it will fail, because writing to flash requires different calls to be used.

   Note: This functionality will be supported in a future version of the sample applications.

7. Select **Run > Terminate** to end your debug session.

8. Open the C/C++ perspective and run the `peripheral_tests_0` C project in the same way that you just ran the `memory_tests_0` project. The program displays on your terminal window. Note that `peripheral_tests_0` will take longer to run.

9. Select **Run > Terminate** to end your debug session.

Now that the two applications have run successfully, we will modify `hello_world` to individually call each application.

Take a Test Drive! Working with Multiple Source Files and Projects

You'll now modify your existing two software applications so that they can be called by `helloworld.c`. We'll change the name of `main()` in each application to something that a new `main()` function can call.

1. In the C/C++ perspective, double-click the `memorytest.c` and `testperiph.c` applications to open them. When opened, they will appear in an edit window.

    Note: These applications are located in the src folder for the respective projects.

2. In `memorytest.c`, change the name of `main()` to `memorytest_main()`. This should be around line number 53.

    Note: If line numbers aren't showing, right-click in the left margin of the edit window and select **Show Line Numbers**.

    As you change the name of `main()`, notice that this new name shows up in the Outline view. If an Outline isn't visible, select **Window > Show View > Outline**.

3. In `testperiph.c`, change the name of `main()` to `peripheraltest_main()`. This should be around line 58.

4. Save both files.

    The files build automatically. They will fail since there is no longer a `main` function, which the build is looking for. If you were to change either function's name back to `main`, the build would proceed error-free.

    We will now modify `helloworld.c` to have it call the `memorytest_main()` and `peripheraltest_main()` functions.

5. Open `helloworld.c` and modify it as shown in Figure 6-1. The `helloworld.c` file is in the `src` folder in the C Project called Editor_C_Project.

```
19/*
20 * helloworld.c: simple test application
21 */
22
23#include <stdio.h>
24#include "platform.h"
25
26int main()
27{
28     init_platform();
29
30     print("Hello World\n\r");
31
32     memorytest_main();
33     peripheraltest_main();
34
35     cleanup_platform();
36
```

Figure 6-1: **Modified Version of helloworld.c File**

6. Save the file, and observe that it, too, builds automatically.

   Note: You can turn automatic building on and off by selecting **Project > Build Automatically**.

   SDK will error out, since it has no knowledge of where the peripheral test or memory test functions are. (They're in their own C Projects). We will now drag and drop the relevant source files to `Editor_C_Project` so that `helloworld.c` can access them.

7. Drag and drop source files from `memory_tests_0` and `peripheral_tests_0` into the `src` subfolder of the Editor _C_Project folder. Figure 6-2 shows the source files that the directory should contain.

   Note: Do not move over the `platform_config.h`, `platform.c`, or `platform.h` files. These files are already part of Editor_C_Project.
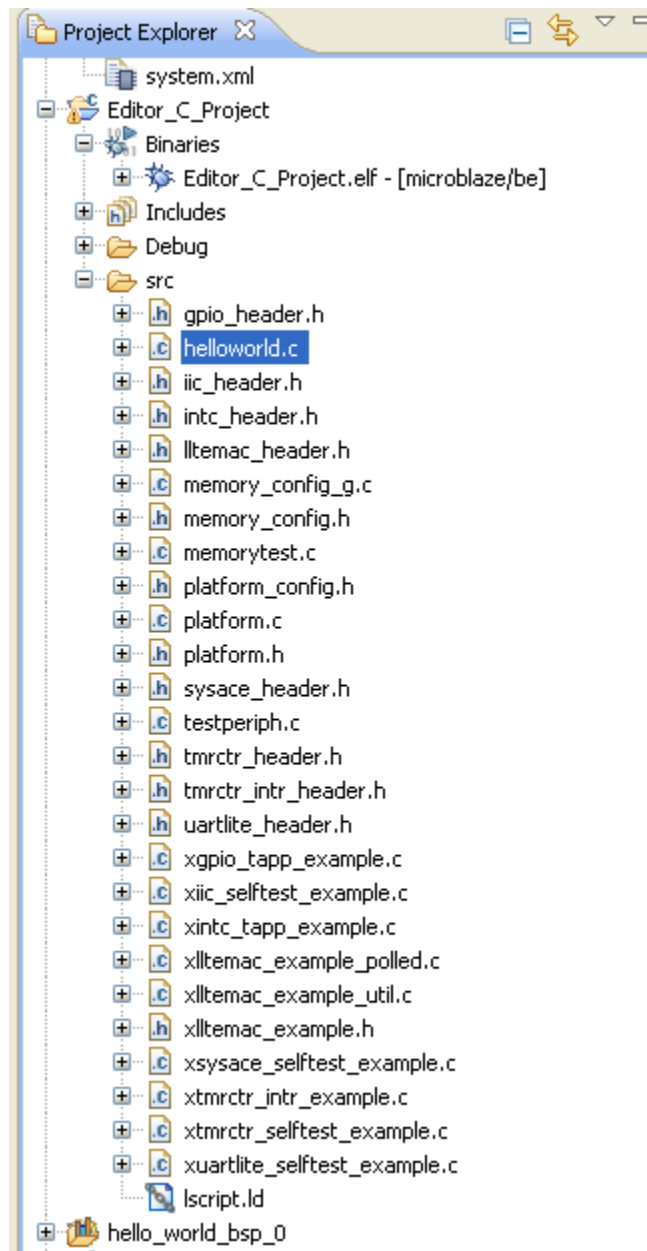

Figure 6-2: **Source Files in Editor_C_Project**

As you drag and drop the files, the "Editor_C_Project" will build after each file. After you've dropped the last file and the ELF file successfully builds, the following message displays in the Console View:

```
Invoking: MicroBlaze Print Size

mb-size Editor_C_Project.elf   |tee
"Editor_C_Project.elf.size"

   text    data     bss    dec    hexfilename

  45674     528    5766   51968    cb00Editor_C_Project.elf

Finished building: Editor_C_Project.elf.size
```

Note the size: 51968 (decimal). Up until now, our applications have all run from block RAM, which is memory on the FPGA. Recall from Chapter 3 and Chapter 4 that we have 16K of memory local to the MicroBlaze processor and another 8K of memory on the PLB for a total of 24K. Our application has grown to 51K, meaning some of it will have to reside in external RAM. The problem is that the RAM test is destructive: if part of the application is in external RAM, it could crash. So next you'll fix that problem before it occurs.

8. Open `memorytest.c` and scroll down to `memorytest_main( )`.

9. Position the cursor over `&memory_ranges[i]`. An informational window opens to display information about `memory_ranges`. You can click in the window and scroll up and down. Note that memory_ranges contains the starting address and size (in decimal) of each memory to be tested.

10. Right click `memory_ranges` and select **Open Declaration** to open the `memory_config.c` file, in which `memory_ranges` is defined. Note that whenever a C file opens for editing, the Outline window, if visible, shows the variables and header files used with the C file. You can right-click any variable in the outline view to view a call hierarchy that shows the calling path back to `main( )`.

11. To change where the external memory test starts, modify the data structure in `memory_config.c` as follows:

```
{
"MCB_DDR3",
"mpmc",
0x89000000, /*Change from 0x88000000 to 0x89000000*/
134217728,
},
```

12. Save the file. It should recompile without errors.

13. Download and run the `Editor_C_Project.elf` application. Confirm that it runs correctly. The terminal window displays a message to indicate that both the memory test and the peripheral test are working.

You can also control the running of your software in the Debug window. In the Debug perspective, hover your cursor of the buttons on the top of the Debug window to see what each button controls.

Working with the Debugger

Now that you have done some file manipulation in the C/C++ Perspective, let's look at some of the features of the Debugger.

The purpose of a debugger is to allow you to see what is happening to a program while it is running. You can set breakpoints and watchpoints, step through program execution in a variety of ways, view program variables, see the call stack, and view or edit the contents of the memory in the system.

SDK provides full source-level debugging capabilities. If you've used other debuggers, you will see that the SDK debugger has most, if not all, of the features that you are used to.

The Debug window, a key part of the Debug Perspective, contains information about the state of your debug session. In this particular example, you'll easily see that our call stack is three deep. Specifically, main(), at address 0x880001dc, called memorytest_main(), at address 0x880004d8, which then went on to call test_memory_range() at address 0x880003fc. In addition, the program state is currently suspended, which indicates that a breakpoint was encountered. Each item on the call stack also shows which line of code contained the calling routine.

Software execution can also be controlled from the Debug window. In the Debug Perspective, hover your cursor of the buttons on the top of the Debug window to view what each button controls.

Take a Test Drive! Working with the Debugger

To begin this test drive, make sure that you've completed "Take a Test Drive! Working with Multiple Source Files and Projects," page 44 and have a binary file called Editor_C_Project.elf.

1. From the C/C++ Perspective, right-click on the executable file and select **Debug As > Launch on Hardware** to download Editor_C_Project.elf to your target board. The Debug Perspective automatically opens.

    When the Debug Perspective opens, it should look similar to Figure 6-3. If some of the views such as Disassembly and Memory are not visible, select **Window > Show View** and select the view that you want to see. If the view doesn't show up in the window that you intended, click and drag it into place.
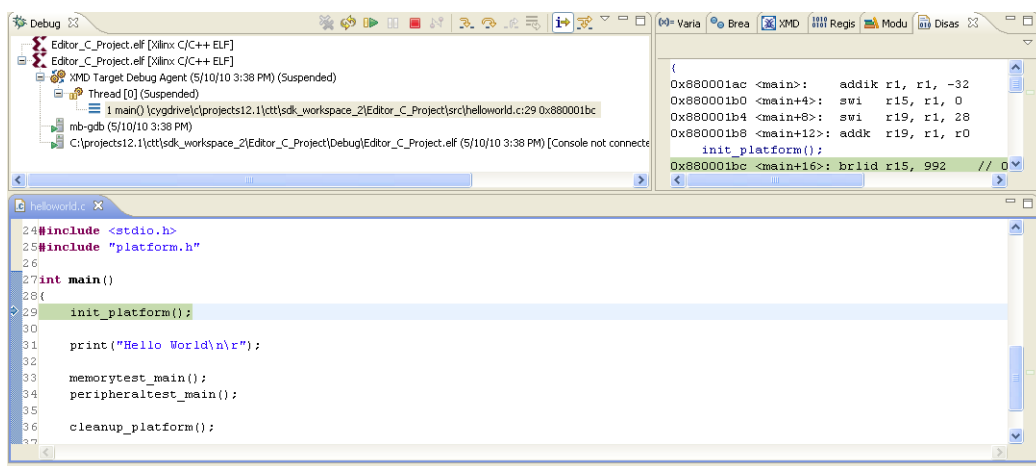

Figure 6-3: **Debug Perspective**

As you can see, MicroBlaze is currently sitting at the beginning of `main()` with program execution suspended at line 0x880001bc. You can correlate that with the Disassembly view, which shows the assembly-level program execution also suspended at 0x880001bc. Finally, the `helloworld.c` window also shows execution suspended at the first executable line of C code. Select the Registers view to confirm that the program counter, RPC register, contains 0x880001bc.

Note: If the Registers window isn't showing, select **Window > Show View > Registers**.

2. Double-click in the margin of the `helloworld.c` window next to the line of code that reads `peripheraltest_main();`. This sets a breakpoint at `peripheraltest_main()`.

   To confirm the breakpoint, review the Breakpoints window.

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

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

   Program execution stops at the line of code that includes `peripheraltest_main();`. Disassembly and the debug window both show program execution stopped at 0x880001dc.

4. Select **Run > Step Into** to step into the `peripheraltest_main()` routine. Program execution is suspended at location 0x88000628. The call stack is now 2 deep.

5. Select **Run > Resume** again to run the program to conclusion. 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. Review your terminal output, which indicates that both `peripheraltest_main()` and `memorytest_main()` have run.

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

What's Next?

The goal of this chapter was to provide you a C project with multiple files to work with, and enough exposure to the debugger to experiment and customize SDK to work the way you do.

In the next chapter, you will create your own IP.

Creating Your Own Intellectual Property

Creating an embedded processor system using Xilinx® Platform Studio (XPS) is straightforward because XPS automates most of the design creation. The Base System Builder (BSB) wizard reduces the design effort to a series of selections.

Benefits of XPS and BSB

You can use the BSB to create most of the embedded processor design. You can then further customize your design in XPS. Design customization can be as simple as tweaking a few parameters on existing intellectual property (IP) cores (for example, changing the baud rate for the UARTLite), or as complex as designing custom IP and integrating it into the existing design.

Benefits of CIP Wizard

While you are the expert regarding the functionality of the required custom IP, you might need additional information about CoreConnect™ bus protocols, the `/pcores` directory structure required by XPS, or the creation of Bus Function Model simulation frameworks. This chapter clarifies these important system distinctions and guides you through the process of creating custom IP using the Create and Import Peripheral (CIP) wizard.

Using the CIP Wizard

The CIP wizard is designed to provide the same benefits as the BSB wizard. It creates the framework of the design, including bus interface logic, and provides an HDL template so that you can integrate your custom logic in an understandable manner. All files necessary to include your custom peripheral core (pcore) into the embedded design are supplied by the CIP wizard.

Creation of custom IP is one of the least understood aspects of XPS. Though the CIP wizard steps you through the creation of your pcore framework, it is important to understand what is happening and why. This chapter provides a basic explanation and guides you through the initial process. It also includes completed pcore design for study and analysis.

Overview of IP Creation

The Bus Interface tab in the XPS System Assembly View (shown in Figure 3-1, page 20) shows connections among buses, processors, and IP. Any piece of IP you create must be compliant with the system you design.

To ensure compliance, you must follow these steps:

1. Determine the interface required by your IP. The bus to which you attach your custom peripheral must be identified. For example, you could select one of the following interfaces:

   – Processor Local Bus (PLB) version 4.6. The PLBv.46 provides a high-speed interface between the processor and high-performance peripherals. PLBv4.6 is used in both the PowerPC® and the MicroBlaze™ processor systems.

   – Fast Simplex Link (FSL). The FSL is a point-to-point FIFO-like interface. It can be used in designs using MicroBlaze processors, but generally is not used with PowerPC processor-based systems.

2. Implement and verify your functionality. Remember that you can reuse common functionality available in the EDK peripherals library.

3. Verify your standalone core. Isolating the core ensures easier debugging in the future.

4. Import the IP to EDK. Your peripheral must be copied to an EDK-appropriate repository search path. The Microprocessor Peripheral Definition (MPD) and Peripheral Analyze Order (PAO) files for the Platform Specification Format (PSF) interface must be created, so that the other EDK tools can recognize your peripheral.

5. Add your peripheral to the processor system created in XPS.

Using the CIP Wizard for Creating Custom IP

The CIP wizard assists you with the steps required in creating, verifying, and implementing your Custom IP. It supports the same buses that are supported by XPS.

The most common design case is the need to connect your custom logic directly to a PLBv46 bus. With the CIP wizard, you can make that bus connection even without understanding bus protocol details. Both slave and master connections are available.

The CIP Wizard Creates HDL Templates and BFM Simulation for your IP

The CIP wizard helps you implement and verify your design by walking you through IP creation. It sets up a number of *templates* that you can populate with proprietary logic.

Besides creating HDL templates, the CIP wizard can create a pcore verification project for Bus Functional Model (BFM) verification. The templates and the BFM project creation are helpful for jump-starting your IP development and ensuring that your IP complies with the system you create. For details of BFM simulation, refer to Appendix A, "Intellectual Property Bus Functional Model Simulation."

What You Need to Know Before Running the CIP Wizard

Before creating a project with the CIP wizard, here are some things you need to know.

Supported Peripherals in the CIP Wizard

In the CIP wizard, you can create four types of PLB v4.6 peripherals using predefined IP interface (IPIF) libraries:

- PLB v4.6 Slave for single data beat transfer
- PLB v4.6 Slave for burst data transfer
- PLB v4.6 Master for single data beat transfer
- PLB v4.6 Master for burst data transfer

The CIP wizard also supports creation of Fast Simplex Link (FSL) peripherals.

Documentation

Before launching the CIP wizard, review the documentation specific to the bus interface you intend to use. Reviewing this information can help eliminate much of the confusion often associated with bus system interfaces. To review the XPS Help topics related to the CIP wizard, select **Help > Help Topics** and navigate to **Procedures for Embedded Processor Design > Creating and Importing Peripherals**.

Accessing IP Datasheets

XPS provides data sheets related to the IP in your system. To access these data sheets, select **Help > View Start Up Page**. In the Start Up page, select the **Documentation** tab, expand **IP Reference and Device Drivers Documentation**, and click the **Processor IP Catalog** link.

If you plan to create a PLBv46 peripheral, examine one of the following data sheets for your custom peripheral:

- plbv46_slave_single
- plbv46_master_single
- plbv46_slave_burst
- plbv46_master_burst

The sections discussing the IP Interconnect (IPIC) signal descriptions are useful in helping identify the IPIF signals that interface to your custom logic.

Note: Normally the CIP wizard is launched from within XPS, as described in the next Test Drive, but the CIP wizard can also run outside of XPS.

Take a Test Drive! Generating and Saving Templates

In this Test Drive, you'll use the CIP wizard to create a template for a custom peripheral. For simplicity, you'll accept the default values for most steps, but you will review all the possible selections you can make.

> *Caution!* Unless you are an advanced user, before starting this Test Drive, make sure that you have read through and completed the Test Drives in Chapter 4, "Working with Your Embedded Platform" and Chapter 5, "Introducing the Software Development Kit."

1. Start the CIP Wizard and determine the location in which to store the custom peripheral files:

   a. Open Xilinx ISE® Project Navigator, load your project, select system.xmp, and double-click the **Manage Processor Design (XPS)** process (located under **Design Utilities**) to launch XPS.

   b. In XPS, select **Hardware > Create or Import Peripheral**.

   After the Welcome page, the Peripheral Flow page opens. On this page, you can either create a new peripheral or import an existing peripheral.
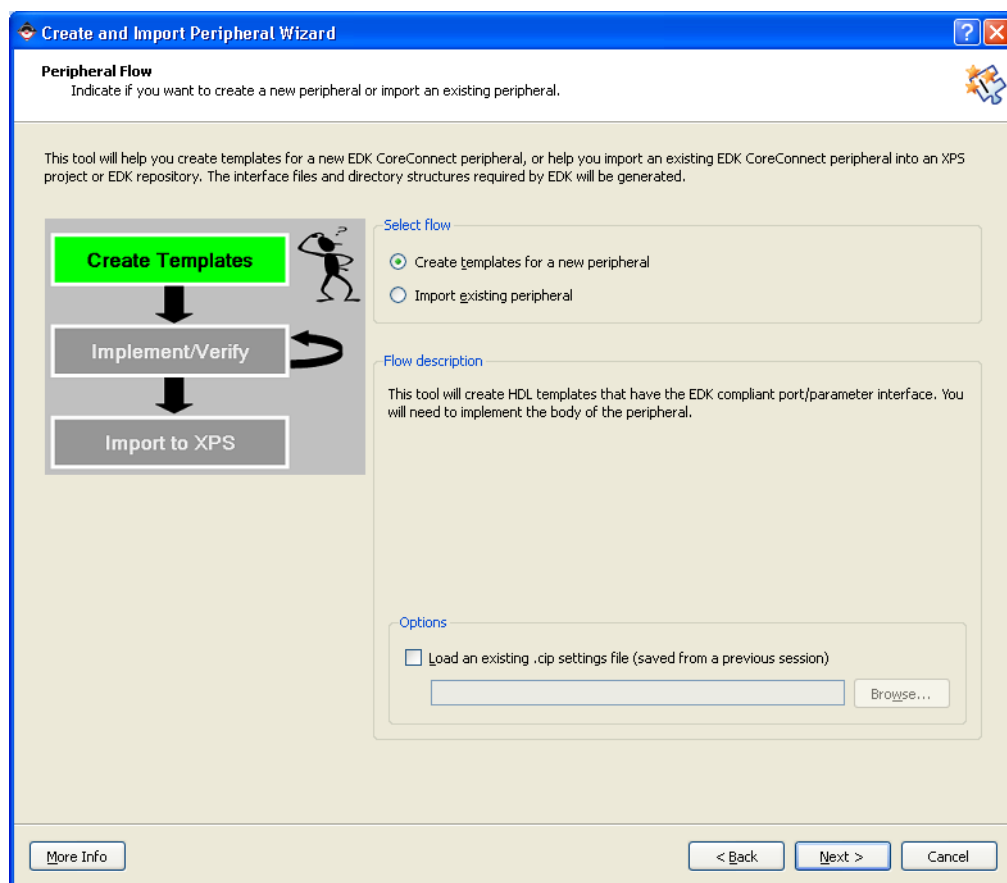

Figure 7-1: **Peripheral Flow Page**

2. Select **Create templates for a new peripheral**. Before continuing through the wizard, read through the text on this page.

   > *Note:* Each CIP wizard screen is full of useful information. You can also click **More Info** to view the related XPS help topic.

3. On the Repository or Project page, specify where to store the custom peripheral files.

   For this example, you will use this peripheral for a single embedded project.

4. Select **To an XPS project**.

   Because you launched the CIP wizard from within XPS, the directory location is filled in automatically.

   Note: If the custom pcore will be used for multiple embedded projects, you can save the file in an EDK repository.

5. Click **Next** to open the Name and Version page.

Figure 7-2: **Name and Version Page**

6. Use the Name and Version page to indicate the name and version of your peripheral. For this example design, use the name `pwm_lights`.

   A version number is supplied automatically. You can also add a description of your project.

7. On the Bus Interface page, select the processor bus that connects your peripheral to your embedded design. For this example, select PLB v46.

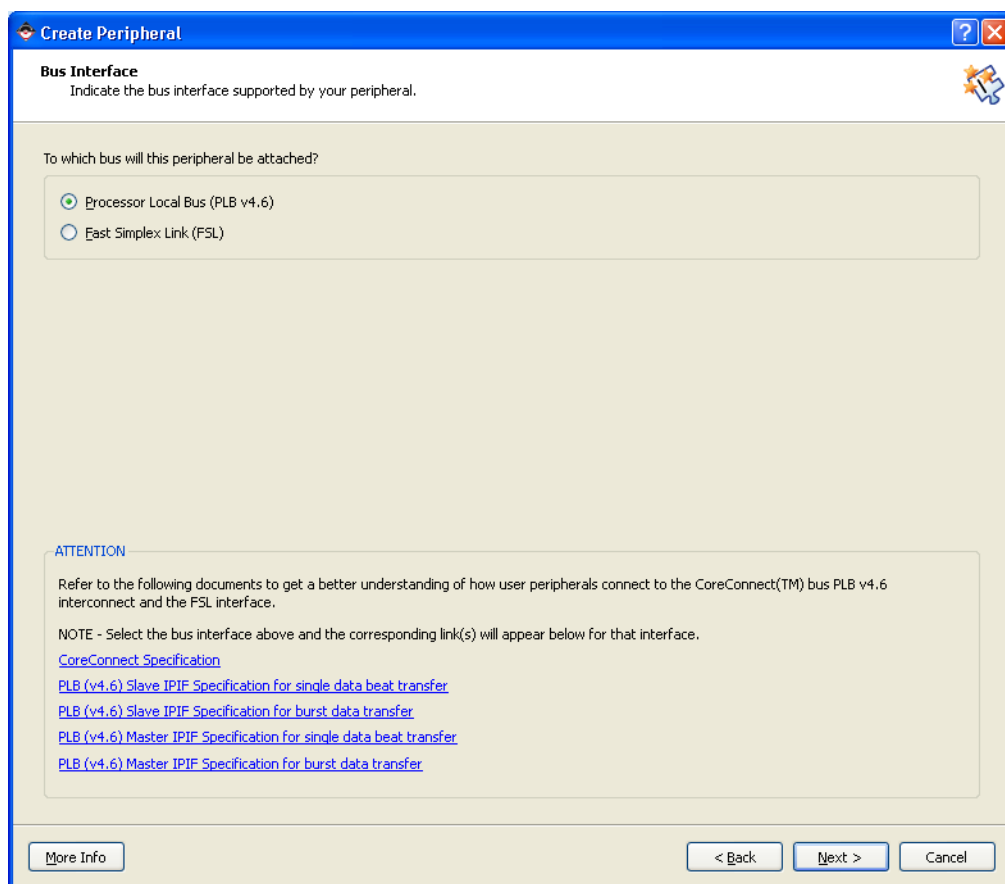   ***Note:*** You can access related data sheets from the Bus Interface page.

Figure 7-3: **Bus Interface Page**

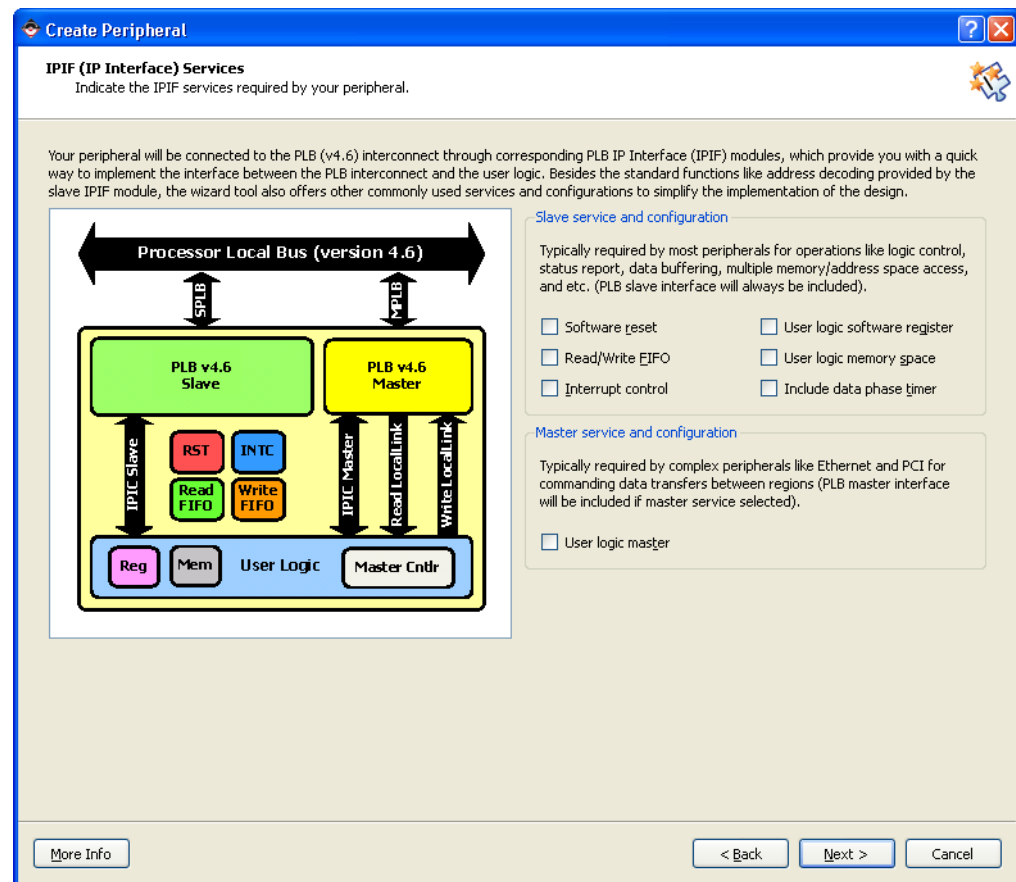8. Click **Next** to open the IPIF (IP Interface) Services page.

Figure 7-4: **IP Interface Services Page**

The CIP wizard automatically creates the following:

– Master or slave connections to the PLB bus

– Necessary bus protocol logic

– Signal sets used to attach your custom HDL code

In addition to this base set of capability, you can add optional services.

Click **More Info**, and on the help page that opens, click the **IPIF Features** link. You can read details on each of these services to help you determine whether the features are necessary for your IP.

9. Unselect all check boxes on this page. For this example, none of the services is required.

The next page is the Slave Interface page, on which you can set up burst and cache-line support. Although you won't use this support for this example, take a moment to review the content about slave peripherals and data width, then move on to the next page of the wizard.

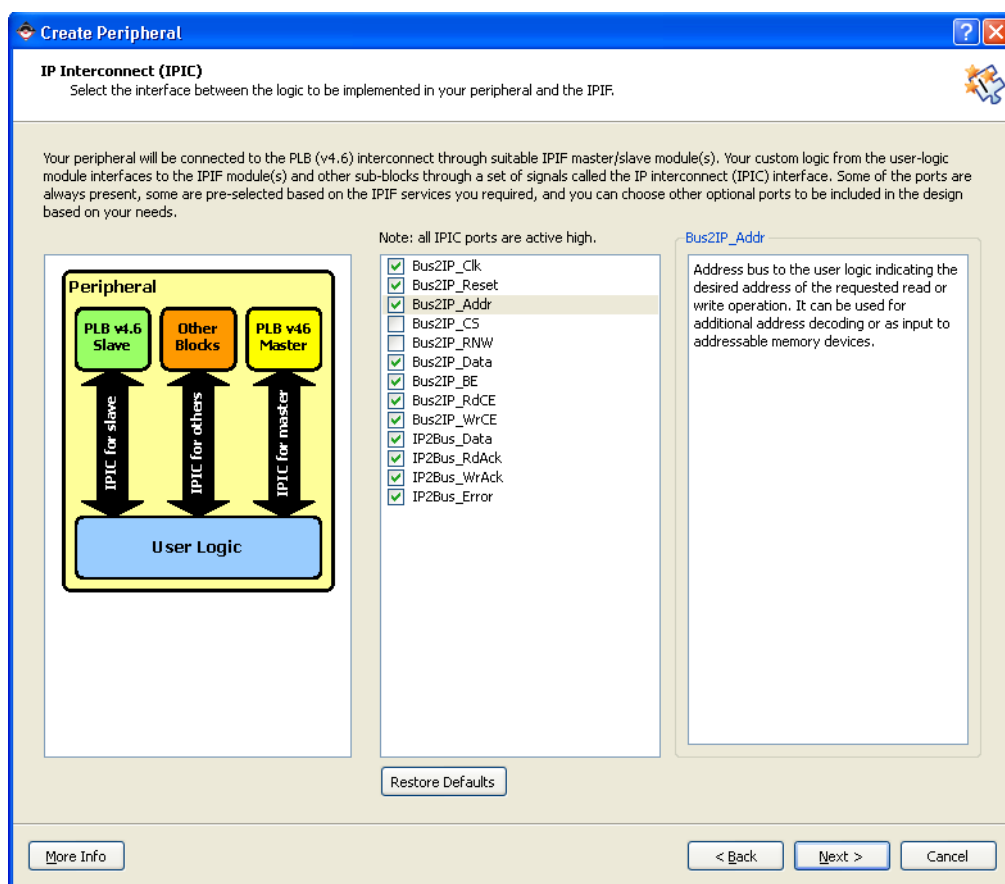10. Click **Next** to open the IP Interconnect (IPIC) page.

Figure 7-5: **IP Interconnect Page**

On the IPIC page, review the set of IPIC signals that the CIP wizard offers for your custom peripheral. If you don't understand what these signals do, review the appropriate specification. The signals selected should be adequate to connect most custom peripherals.

In this design, multiple writes to different addresses must be decoded, so you'll add Bus2IP_Addr signals to create the decode logic inside your HDL.

Alternatively, you can also select **User logic memory space** on the IPIF Services page to open a wizard page specific to managing user memory space.

11. Click **Bus2IP_Addr**. Leave the other boxes unchanged.

12. Click **Next** to open the Peripheral Simulation Support page.
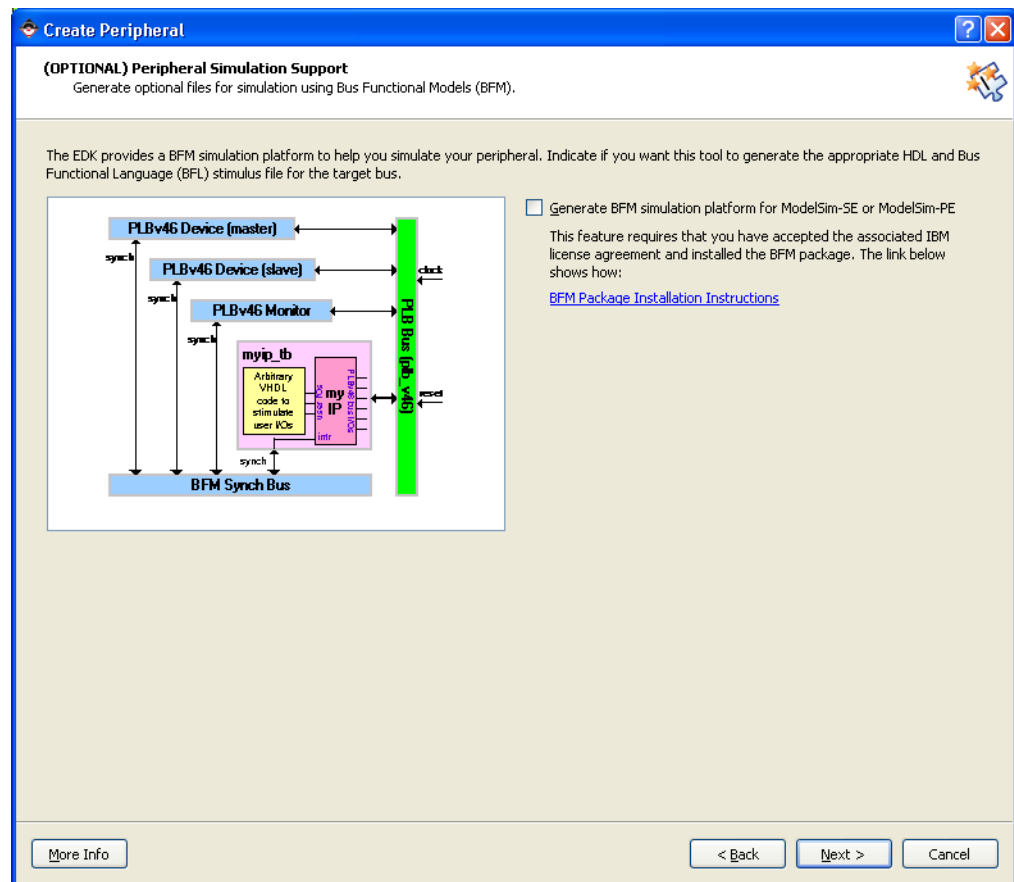

Figure 7-6:   **Peripheral Simulation Support Page**

On the Peripheral Simulation Support page, you can elect to have the CIP generate a BFM simulation platform for your project. Generating a BFM simulation platform requires that you have installed the following:

– The BFM simulation package for EDK

– ModelSim-SE or ModelSim-PE

If you'd like, you can stop here and click the **BFM Package Installation Instructions** to go through the steps necessary to license, download, and install the BFMs. BFM simulation is described in Appendix A, "Intellectual Property Bus Functional Model Simulation." If you think you might want to run a BFM simulation on this IP example, generate the BFM platform now.

The CIP wizard creates two HDL files that implement your pcore framework:

– The `pwm_lights.vhd` file, which contains the PLBv46 bus interface logic. Assuming your peripheral contains ports to the outside world, you must modify this file to add the appropriate port names. This file is well documented and tells you exactly where to add the port information.
If you are a Verilog designer, *don't panic,* but realize that you must write the port names using HDL syntax. For this example, you can find the source code in an upcoming Test Drive and use that source as a template for future pcore creation.

– The `user_logic.vhd` file, which is the template file where you add the custom RTL that defines your peripheral. Although you can always create additional source files, the simple design example you are using requires only the `user_logic.vhd` file.

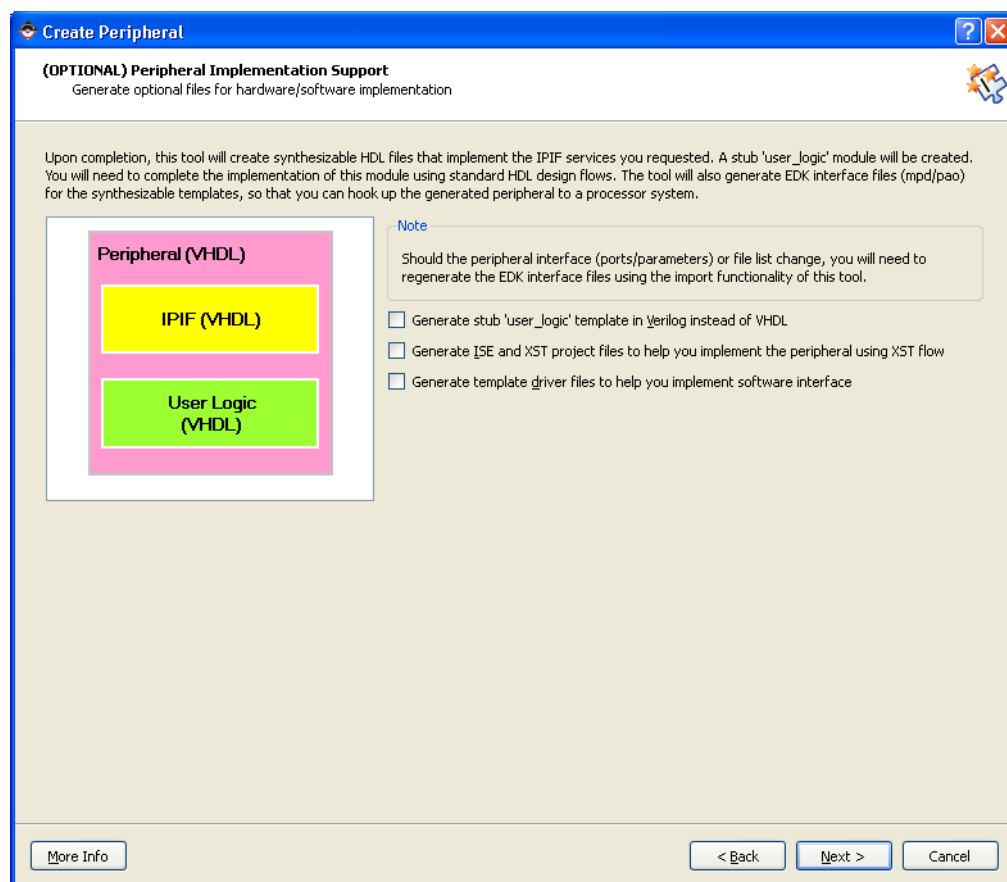13. Click **Next** to open the Peripheral Implementation Support page.

Figure 7-7: **Peripheral Implementation Support Page**

Verilog Support

The Peripheral Implementation Support page lists three options for creating optional files for hardware and software implementation.

– The CIP wizard can create the user_logic template in Verilog instead of VHDL. To create the template in Verilog, select the **Generate stub 'user_logic' template in Verilog instead of VHDL** check box.

– If you intend to implement your pcore design to completion (for timing analysis or timing simulation), click the **Generate ISE and XST project files to help you implement the peripheral using XST flow** check box. The CIP wizard creates the necessary ISE project files. However, if your peripheral is low-speed or very simple, this step is not necessary.

– If your peripheral requires more complex software drivers, click the **Generate template driver files to help you implement software interface** check box. The CIP wizard creates the necessary driver structure and some prototype drivers based on the services selected.

For this example design, leave all three boxes unchecked. The final screen displays a summary of the CIP wizard output – files created and their locations.
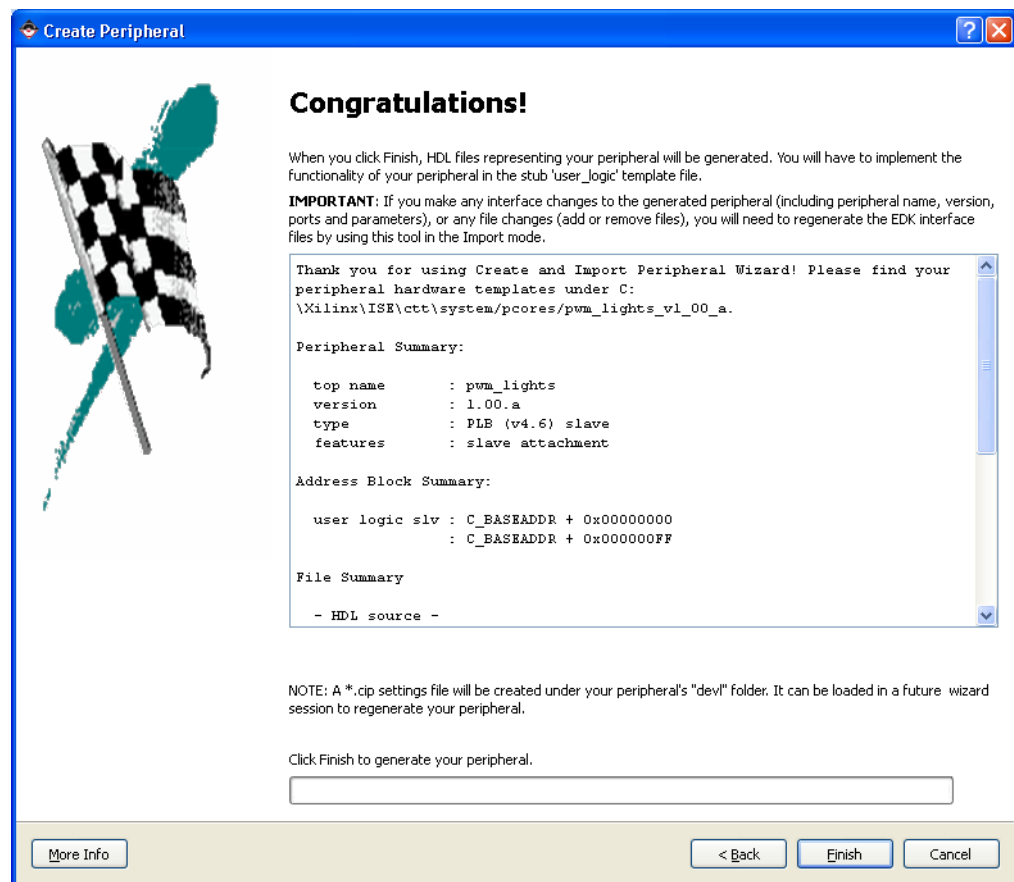

Figure 7-8: **Create and Import Peripheral Wizard Summary Page**

Important Summary Information

14. Review this information and click **Finish**. You can observe the file creation status in the Console window.

What Just Happened?

Precisely what did the CIP wizard do? Let's stop for a moment and examine some concepts and the resulting output.

EDK uses PLB slave and burst peripherals to implement common functionality among various processor peripherals. The PLB slave and burst peripherals can act as bus masters or bus slaves.

In the Bus Interface and IPIF Services Panel, the CIP wizard asked you to define the target bus and what services the IP needs. The purpose was to determine the PLB slave and burst peripheral elements your IP requires.

PLBv46 Slave and Burst Peripherals

The PLB slave and burst peripherals are verified, optimized, and highly parameterizable interfaces. They also give you a set of simplified bus protocols. Your custom RTL interfaces to the IPIC signals, which are much easier to work with when compared to operating on the PLB or FSL bus protocols directly. Using the PLB slave and burst peripherals with parameterization that suits your needs greatly reduces your design and test effort.

Figure 7-5 illustrates the relationship between the bus, a simple PLB slave peripheral, IPIC, and your user logic.

The following figure shows the directory structure and the key files that the CIP wizard created. These file reside in the `/pcores` subdirectory of your project directory..
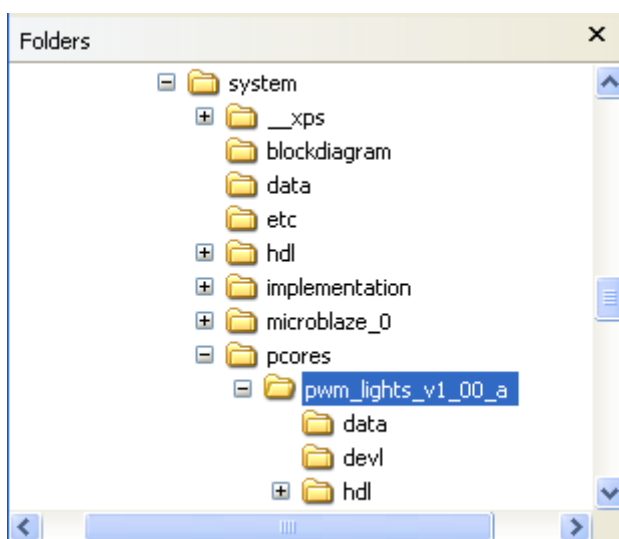

Figure 7-9: **Directory Structure Generated by the CIP Wizard**

Information about the files generated by the CIP wizard:

- The wizard created two HDL template files: `pwm_lights.vhd` and `user_logic.vhd`. These files are located in the `hdl` folder.
- The `user_logic` file connects to the PLB v4.6 bus using the PLB slave/burst cores configured in `pwm_lights.vhd`.
  - The `user_logic.vhd` file is equivalent to the "Custom Functionality" block.
  - The `pwm_lights.vhd` file is equivalent to the "PLBv.46 slave/burst" blocks.
- Your custom logic interfaces using the IPIC signals.

To complete your design, you must add your proprietary logic to the two files.

Example Design Description

You can use the CIP wizard to create a fully functional peripheral, assuming that reading and writing registers provides adequate functionality. You can choose to create a simple peripheral this way. However, having an actual, functioning example that you can modify is much more valuable, so now you'll define a simple PLBv46 peripheral.

You'll open and modify the source code files for this peripheral in the next Test Drive. These files are located in the `/pcores` directory on your system. You'll also use some example files that are included in the .zip file for this guide. Find and review the files listed there. You'll open them in the Test Drive.

Note: You can find information about downloading the .zip file for this manual in "Attachments to this Guide," page 6.

The custom peripheral controls the 4 LEDs on the evaluation board. To make the design interesting, the `pwm_lights` circuit will:

- Turn LEDs off by a write to offset 0.
- Turn LEDs on by a write to offset 4.
- Control the intensity of the lights using a simple PWM circuit. The intensity varies over 16 discreet brightness values:
  - A write to offset 8 uses a log intensity drive scale.
  - A write to offset 12 uses a linear intensity log scale.
- Read back the control circuit status.

In addition to the hardware design, a simple software application gives you control over the various settings and the read back status.

Take a Test Drive! Modifying the CIP Wizard Template Files

Conceptually, what you are going to do in this final Test Drive is simple: you're going to load and run some C code that controls the pcore created by the CIP wizard. Before starting, you should be aware of the following software features:

- SDK tracks the `system.xml` file that is used in its workspace. You introduced this file in Chapter 4, "Working with Your Embedded Platform." If that file changes for any reason, SDK flags the change. You'll see an example of that feature in this Test Drive when you add hardware to the file.

- By default, SDK maps all of your C code to block RAM. In this Test Drive, the piece of C code you are using is larger (in terms of memory use) than that used previously. Consequently, the available block RAM to which SDK mapped is too small. Therefore, you must modify the Linker script. SDK has a built in GUI that simplifies modifying your Linker Script.

Now, you'll open and modify the template files that the CIP wizard created for your project.

1. In XPS, select **File > Open**.

2. Navigate to the `pcores\pwm_lights_v1_00_a\hdl\vhdl` directory and locate the `pwm_lights.vhd` file and the `user_logic.vhd` file.

   Note: You might have to change the **Files of type** drop-down list to view and open these files.

3. Open the `pwm_lights.vhd` file.

   Add the external port names in two places in this file:

   – The top level entity port declaration

   – The port map for the instantiation of the `user_logic`

4. Scroll down to approximately line 165. In the code segment shown in the figure below, the user port LEDs are displayed in the appropriate location. Add the LEDs port declaration for the top-level entity in your file as shown here.

```
);
port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
  LEDs                              : out std_logic_vector(0 to 7);
  -- ADD USER PORTS ABOVE THIS LINE ------------------

  -- DO NOT EDIT BELOW THIS LINE ---------------------
```

Figure 7-10: **Add User Ports**

5. Scroll down to approximately line 390. In the code segment shown in the figure below, the user port LEDs are displayed in the appropriate location. Add the LEDs port declaration into the `user_logic` port mapping in your file as shown here.

```
)
port map
(
  -- MAP USER PORTS BELOW THIS LINE ------------------
  LEDs(0 to 7)                      => LEDs(0 to 7),
  -- MAP USER PORTS ABOVE THIS LINE ------------------
```

Figure 7-11: **Add Port Mapping**

6. Save the file.

   Where user information is required in the two template files (*<ip core name>*.vhd and user_logic.vhd), comments within the file indicate the type and placement of required information.

   In most cases, adding user ports to the top-level entity and then mapping these ports in the user_logic instantiation are the only changes required for *<ip core name>*.vhd.

7. In XPS, select **File > Open** and navigate to the pcores\pwm_lights_v1_00_a\hdl\vhdl directory.

8. Open and examine the user_logic.vhd file.

9. A completed user_logic.vhd file is provided in the .zip file for this guide. Replace the contents of the currently opened user_logic.vhd file with the contents of the provided user_logic.vhd file and save the file.

Reviewing the File Contents

Assuming you are familiar with VHDL, the code that makes up pwm_lights is easy to understand.

The user_logic.vhd file is similar to the top-level pwm_lights.vhd file, in that the template contains many comments and instructs you where to add custom RTL. If you have never used the CIP wizard before, take a few minutes to study the comments, the list of interface signals, and locations where you are instructed to add your RTL.

It is essential that you *do not* modify the auto-generated generics and ports. Add your custom generics and ports only where instructed.

At approximately line 100, notice that the user port LEDs (0 to 3) were added. This output vector drives the four LEDs on the evaluation board. Anytime you add signals specific to your design, you must add these ports in this location. You also need to add these ports in the top-level file and map them through to user_logic.

Most of the code after the architecture declaration is custom code.

After declaring the necessary internal signals and constants, the first block in the design drives a simple counter. Two output signals are tapped off the counter:

- The PWM update clock (selected to be approximately 1 Khz)
- The LED update clock (selected to be approximately 4 Hz)

Modifying Clock Rates

If you want to modify the design later, you can change these clock rates by modifying one or both of the constants: PWM_tap and slow_clock_tap.

The decode process is used to decode the interface signals from the IPIC to select the appropriate function. A write to the custom block occurs when Bus2IP_WrCE(0) is active (high). Adding a few address signals to the decode logic implements the following behavior:

- Write to offset 0x00: All LEDs turned off
- Write to offset 0x04: All LEDs turned on
- Write to offset 0x08: LED brightness varies, using a square function drive signal
- Write to offset 0x0C: LED brightness varies, using a linear function drive signal
- Write to offset 0x1x: LED brightness is set to a constant value (the range is 0 to 0xFF)

The PWM process updates the drive signal based at the update rate of the `slow_clock`.

- The first case statement updates the drive values using a square function
- The second case statement updates the drive values in a linear manner

Feel free to change the drive values as part of your experimentation. As designed, 16 discrete drive values are used.

The LEDs are driven with a PWM-generated drive signal. The duty cycle of the drive signals varies from 0% (no drive) to almost 100% (`0xFF` or `255`).

The assignment to LEDs (0 to 7) near line 247 controls the LEDs based on the last instruction written to the circuit.

All the code described above is simple and can be modified if you want to experiment later. However, the interface signals starting at line 256 are required to have very explicit behavior. Incorrect logic driving these signals will cause the custom pcore to interfere with proper bus operation, and could result in unexpected behavior during debug.

`IP2Bus_Data` is the bus that is read by the processor during a read operation. Correct PLB operation requires that this bus be driven with all logic zeros except when an active read is occurring from the custom pcore. An active read is decoded correctly when reset is inactive and the `Bus2IP_RdCE` is active high. When this condition occurs, the custom circuit drives the specified value onto the bus; otherwise, zeros are driven.

For this example design, doing a read of any address within the peripheral address map returns a 32-bit value which looks like this:

Bits 0 to 15:	0xF0F0
Bits 16 to 23:	One byte value written to LED drive register
Bits 24 to 27:	0x0
Bits 28 to 31:	all_off (1 bit), run (1 bit), linear (2 bits)

The final signals, `IP2Bus_WrAck` and `Bus2IP_WrCE(0)`, are also critical. `IP2Bus_WrAck` is a write acknowledge that must be returned by the custom logic. `IP2Bus_WrAck` must be driven high only for a single cycle, but can be delayed if your custom logic needs to add wait states to the response. For this example, no wait states are necessary. Connecting `IP2Bus_WrAck` directly to `Bus2IP_WrCE(0)` provides a simple, zero wait state response. The logic for the read acknowledge signal is identical. The peripheral can add wait states if necessary.

The `pwm_lights` pcore contains a `C_INCLUDE_DPHASE_TIMER` parameter that can be set to a logic one, which generates an automatic bus timeout signal if the peripheral does not respond to a bus request.

As implemented, the data phase timer is not included. To add this logic, add the `C_INCLUDE_DPHASE_TIMER` parameter to the `pwm_lights` peripheral in the file and set the value to 1. Doing so guarantees that an unacknowledged bus transfer will time-out after 128 PLB clock cycles.

The `IP2Bus_Error` is driven with a constant logic zero, implying that no error condition is returned. If your custom peripheral could potentially time out based on having to wait for other external logic, you can connect logic to drive `IP2Bus_Error` to terminate the bus transfer.

Adding Your Custom IP to Your Processor System

When you modified `pwm_lights.vhd` and `user_logic.vhd`, you added new ports to the template design. Any time you modify the design files in a manner that modifies the ports or parameters in the MPD file, the CIP wizard must be run again in Import mode. This regenerates the correct PSF files, MPD and PAO, which are the interface files to EDK. When the Import flow is completed, the custom pcore can be added to your embedded design.

Before taking this test drive, let's do a quick review of where you are in the IP creation process:

- The first time you ran the CIP wizard, you created the `pwm_lights` peripheral, set up the bus interface, and generated the required template files.

- Next you will add `pwm_lights` to your project, again using the CIP wizard. In the process, `pwm_lights` is imported to an XPS-appropriate directory and the CIP wizard creates the MPD and PAO files. For more information about PSF files, see the *Platform Specification Format Reference Manual*, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

Take a Test Drive! Using the CIP Wizard to Re-Import the Modified File into Your XPS Project

1. Open the CIP wizard and specify that you want to import an existing peripheral to an XPS project.

2. On the Repository or Project page, select the current project.

3. On the Name and Version page, select `pwm_lights` from the **Name** drop-down list. A version is not required, but for this example it is already selected. Use either the default setting or a custom version number.

4. If the CIP wizard asks if you want to overwrite an existing peripheral with this name, select **Yes**.

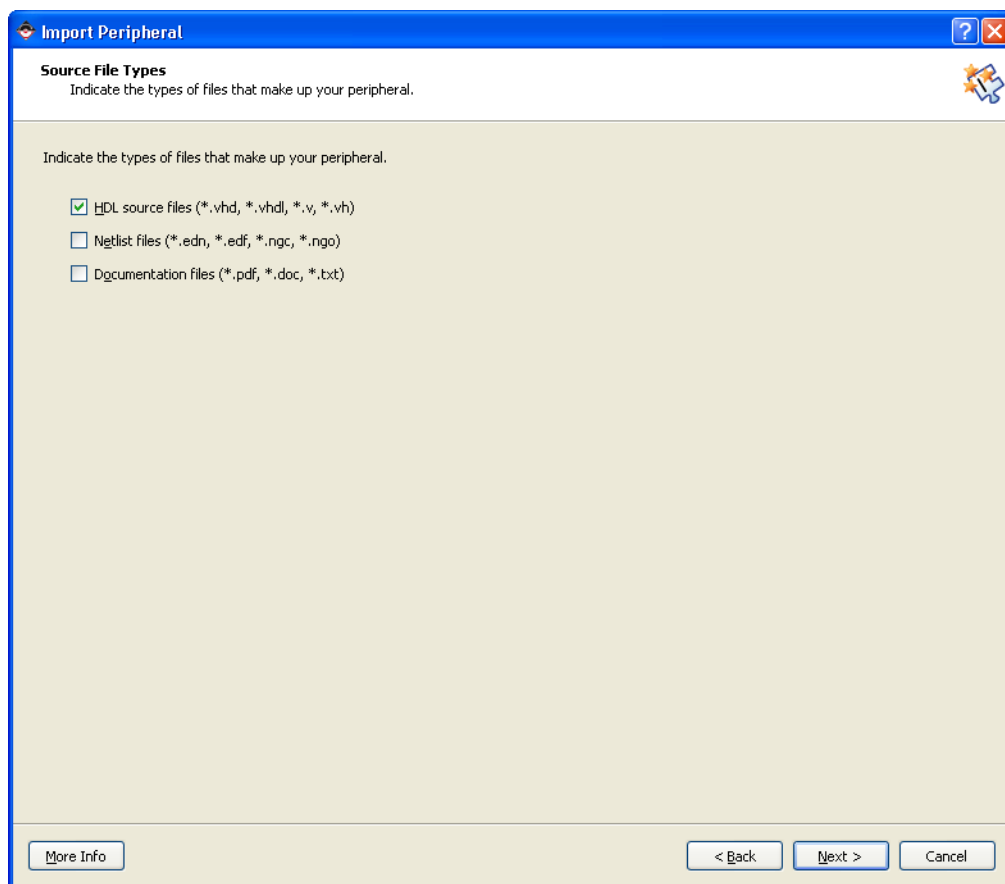5. Select HDL source files. You can create pcores using RTL or existing netlists.

Figure 7-12: **Source File Types Page**

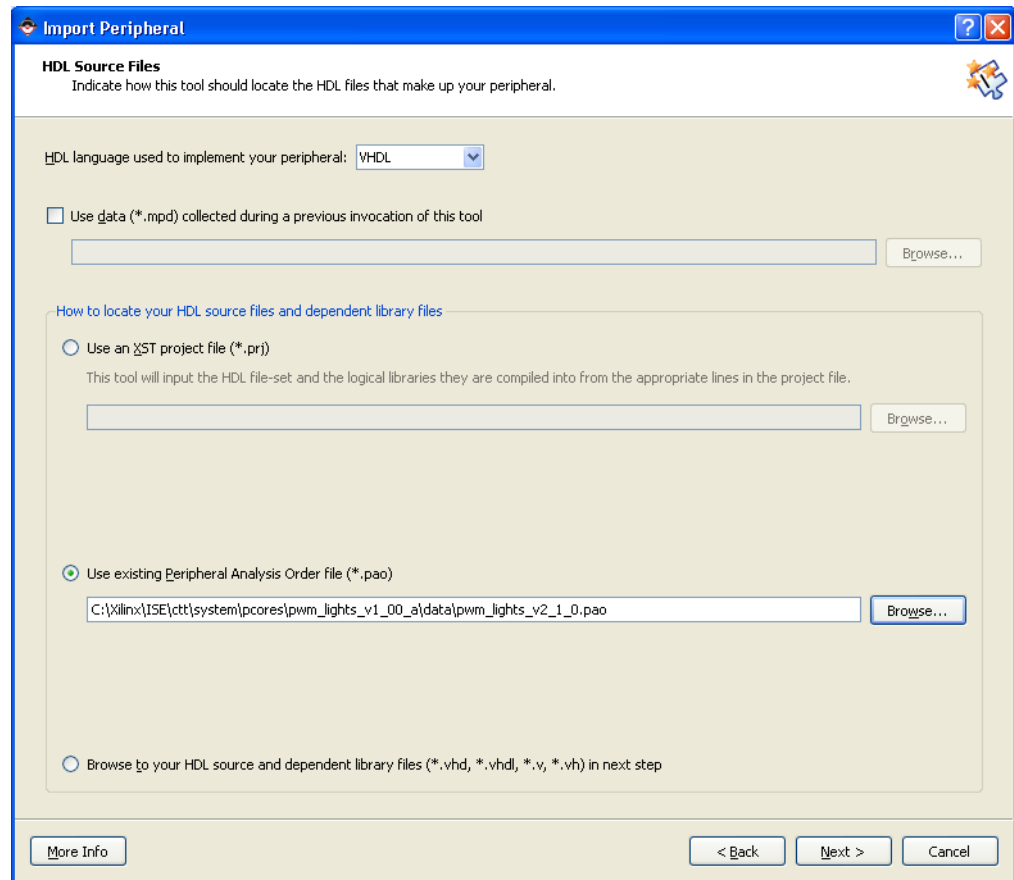6. Click **Next** to open the HDL Source Files page.

Figure 7-13: **HDL Source Files Page**

The CIP wizard imports pcores that were created in a variety of ways. If you used the CIP wizard to initially create the pcore, the most straightforward manner to properly locate and identify source files is to use the Peripheral Analysis Order (PAO) file.

7. Select **Use existing Peripheral Analysis Order file (*.pao)**.

8. Browse to the PAO file. By default, the browse function opens at the top-level `pcores` directory. The PAO file is located in the in the `/pwm_lights/data` subdirectory.

When you use the PAO file to locate the necessary source files (including lower level libraries), it is not necessary to add any additional files or libraries.

If you are importing a complex peripheral made up of many files, look through the listing of libraries and HDL source file paths to verify that any necessary files and libraries are included.

Note: You might have to change the Files of Type drop-down to see and open the files.

9. Click **Next**.

10. In the HDL Analysis Information page, make sure that `user_logic.vhd` and `pwm_lights.vhd` show at the bottom of the list.
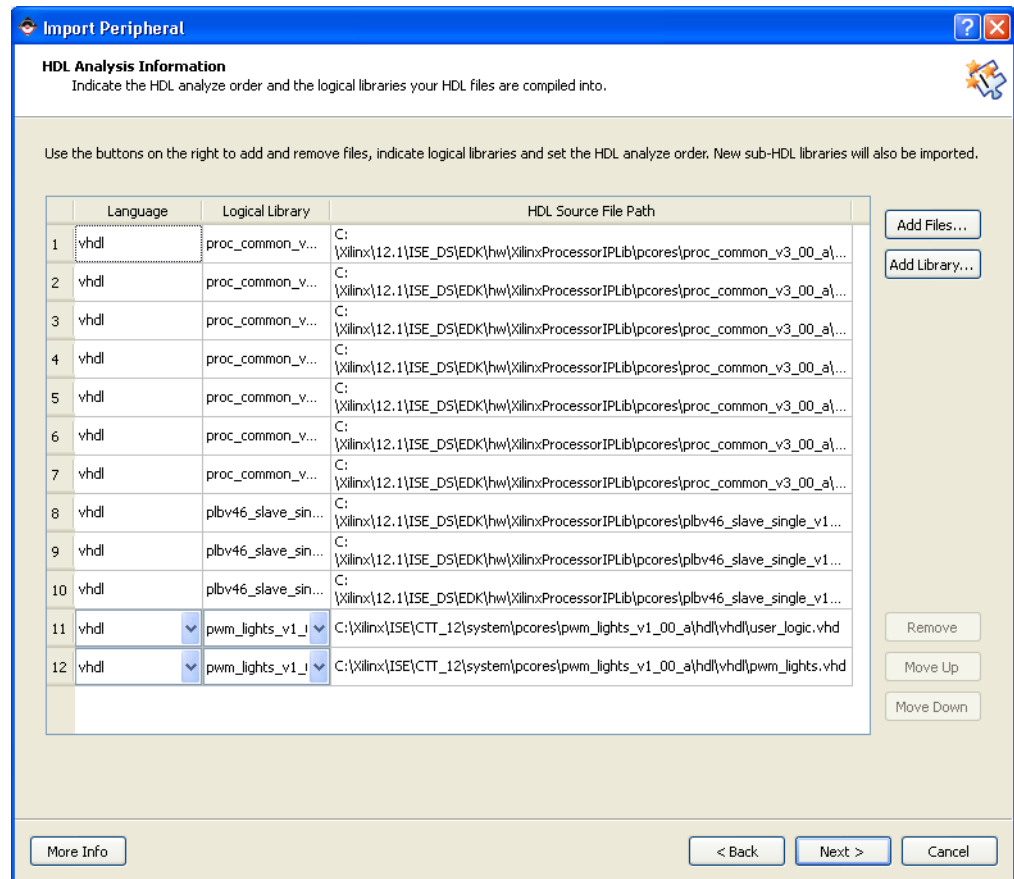

Figure 7-14: **HDL Analysis Info Page**

11. If the Current Logical Library dialog box indicates that the two VHDL files that you added have not yet been compiled, click **Next** to compile them and open the Bus Interfaces page.
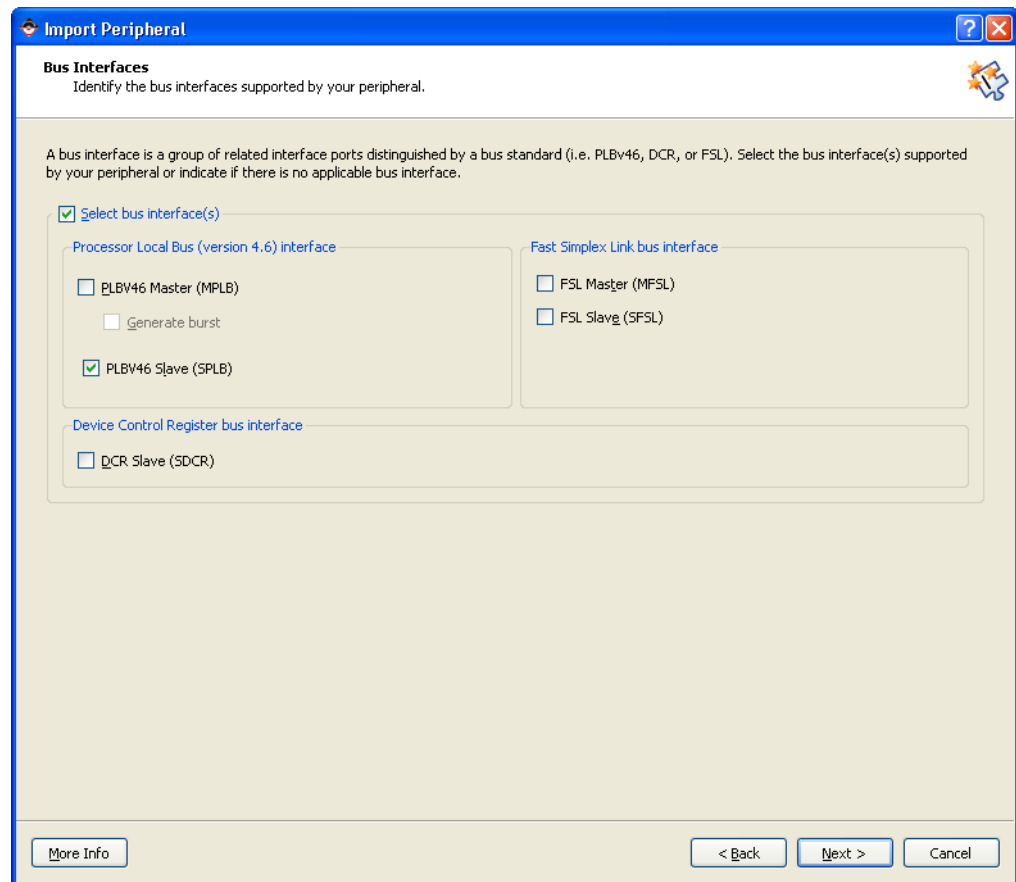

Figure 7-15: **Bus Interfaces Page**

12. Select the appropriate bus interface. The `pwm_lights` peripheral is a PLBv46 Slave (SPLB).

13. Click **Next** to open the Port Details page.
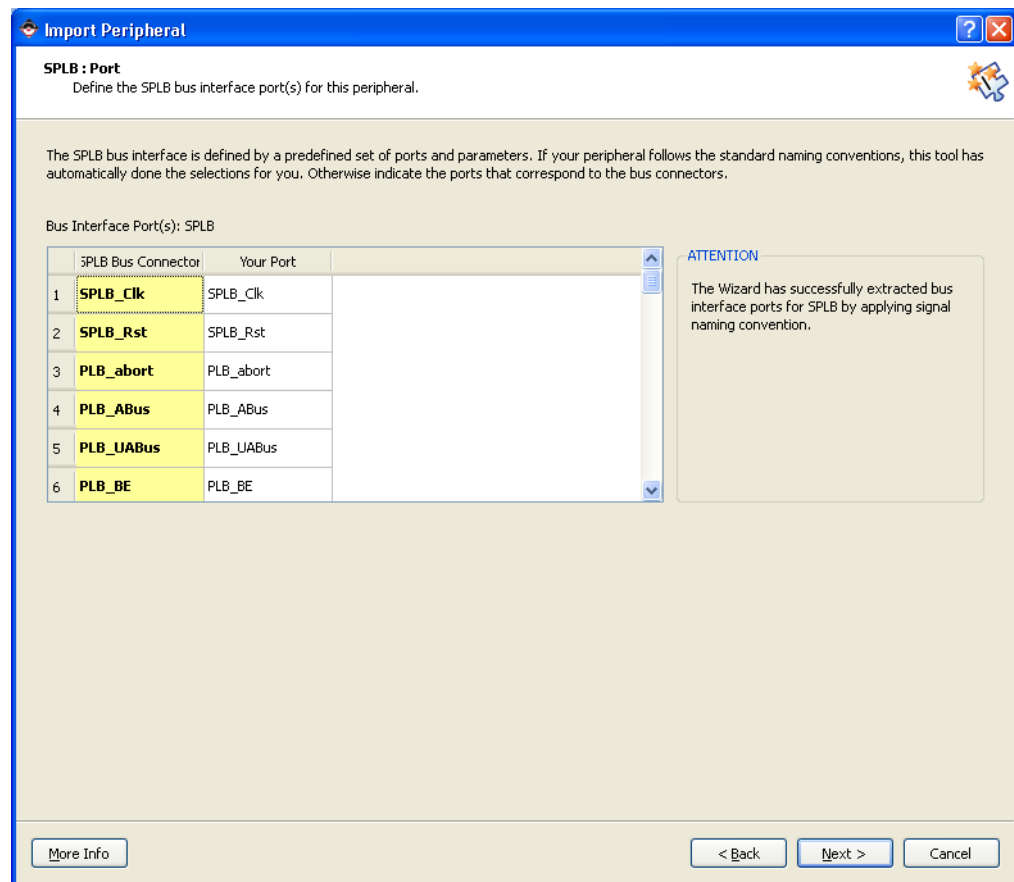
Figure 7-16: **SPLB Port Details**

This page shows a complete listing of all the PLBv46 bus signals that are used in this design. You can scroll through the list of signals and associated bus protocols that the Create and Import IP wizard automatically configured.

If the core was originally created by the Create and Import IP wizard, all of the necessary signals are included. If a core was created with another tool or contains a complex or custom bus interface, this wizard page is useful for analyzing the bus signals.

14. Review the Port details and click **Next**.

Figure 7-17: **SPLB Parameter Details**

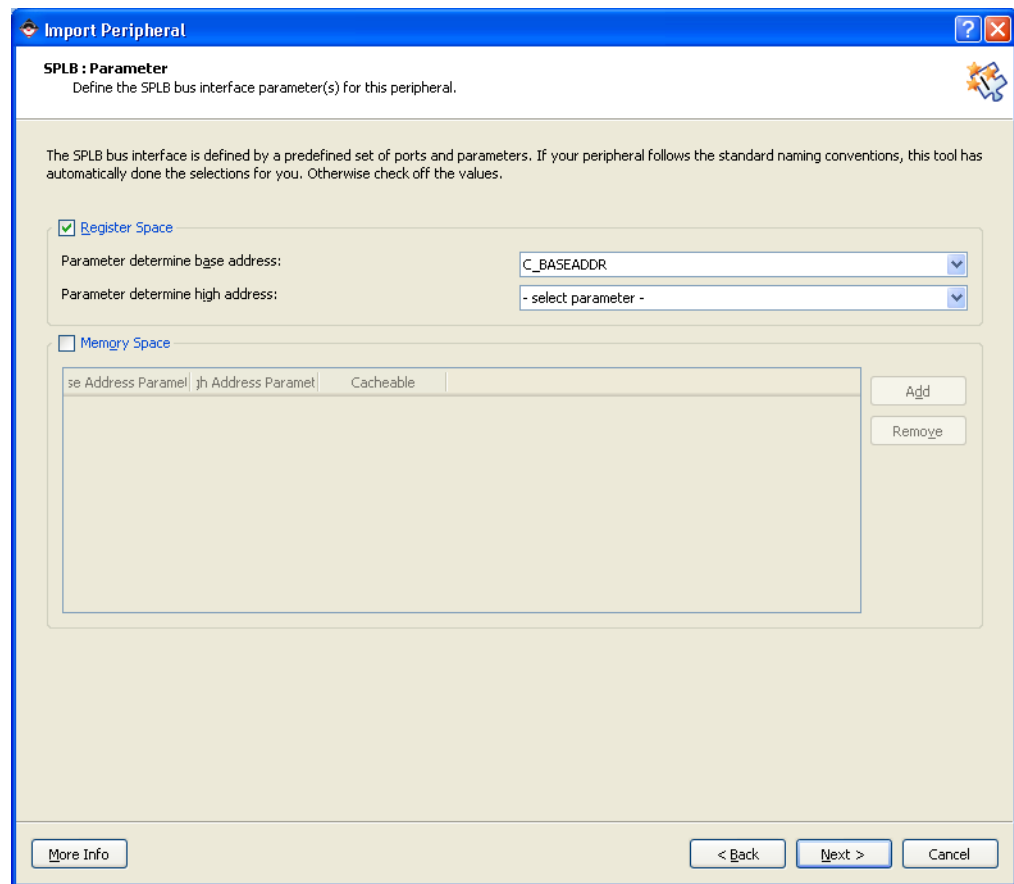The `pwm_lights` peripheral is mapped to a single address range, which XPS selects when the pcore is included in the design. More complex peripherals might also contain memory blocks or decode ranges that must be accessible.

15. Accept the default parameter settings and click **Next**.
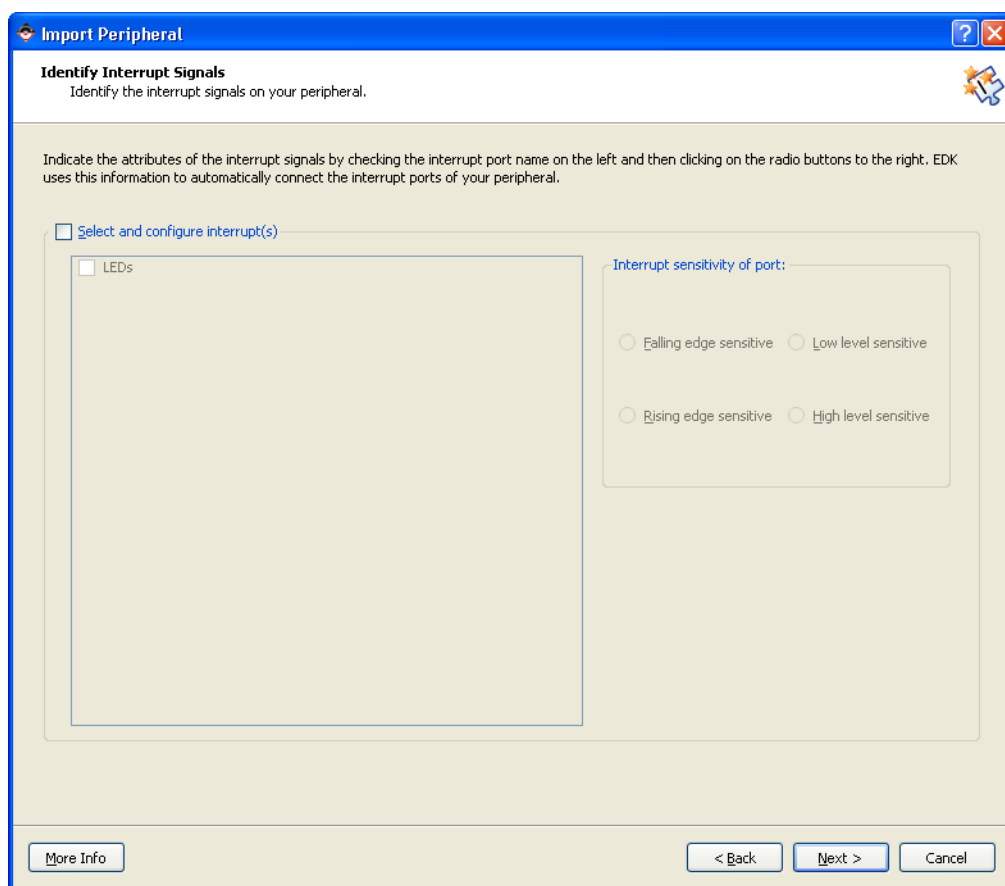
Figure 7-18: **Identify Interrupt Signals Page**

16. The pwm_lights peripheral contains no interrupt sources. Unselect the **Select and configure interrupts** check box, and click **Next** to open the Parameter Attributes page.
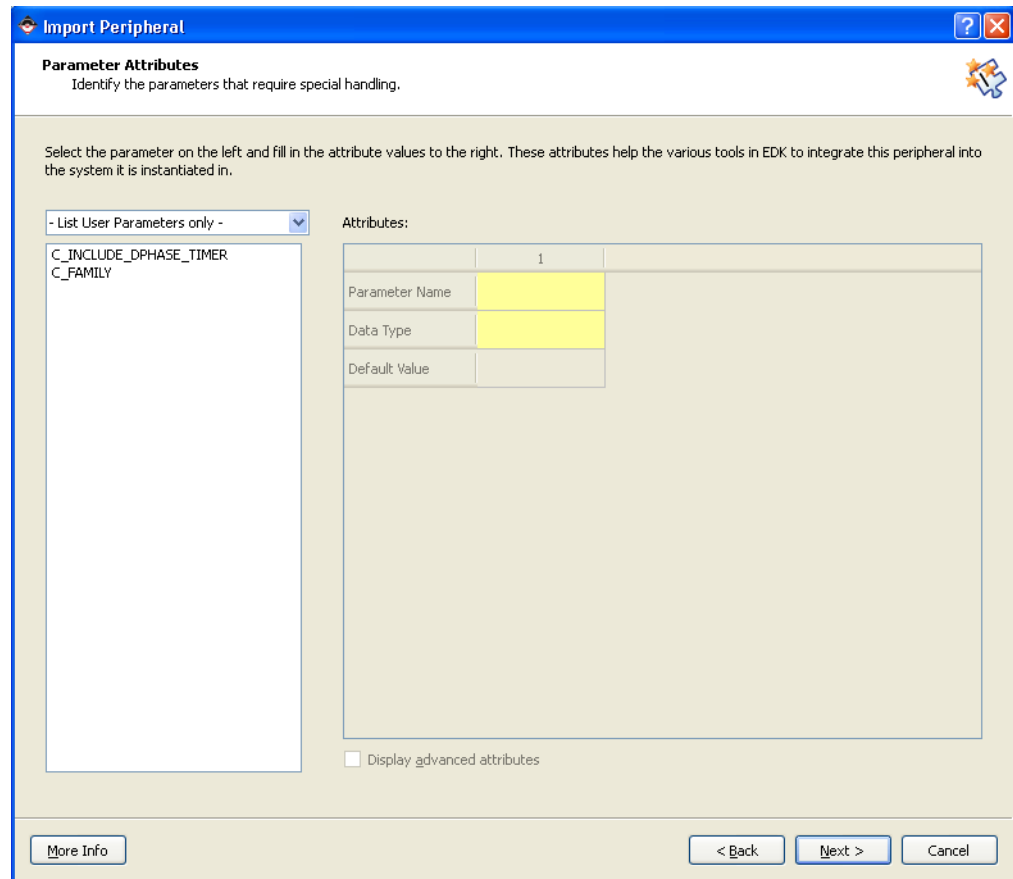
Figure 7-19: **Parameter Attributes Page**

Complex peripherals might include a large number of parameters and require careful control of PLB bus behavior. Use the Parameter Attributes page to view and control available settings for all parameters.

Use the drop-down menu to view parameters for your custom pcore, parameters for the selected bus interface, and a combination of the two. The pcore parameters were generated from an earlier CIP Wizard screen. The bus interface parameters were automatically generated.

For this example, no changes are required.

17. Click **Next** to open the Port Attributes page.

18. On the Port Attributes page, click **LEDs**, and then select the **Display advanced attributes** check box to get the full display. This setting gives you a finer degree of control over the port attributes as they appear in the `.mpd` file.
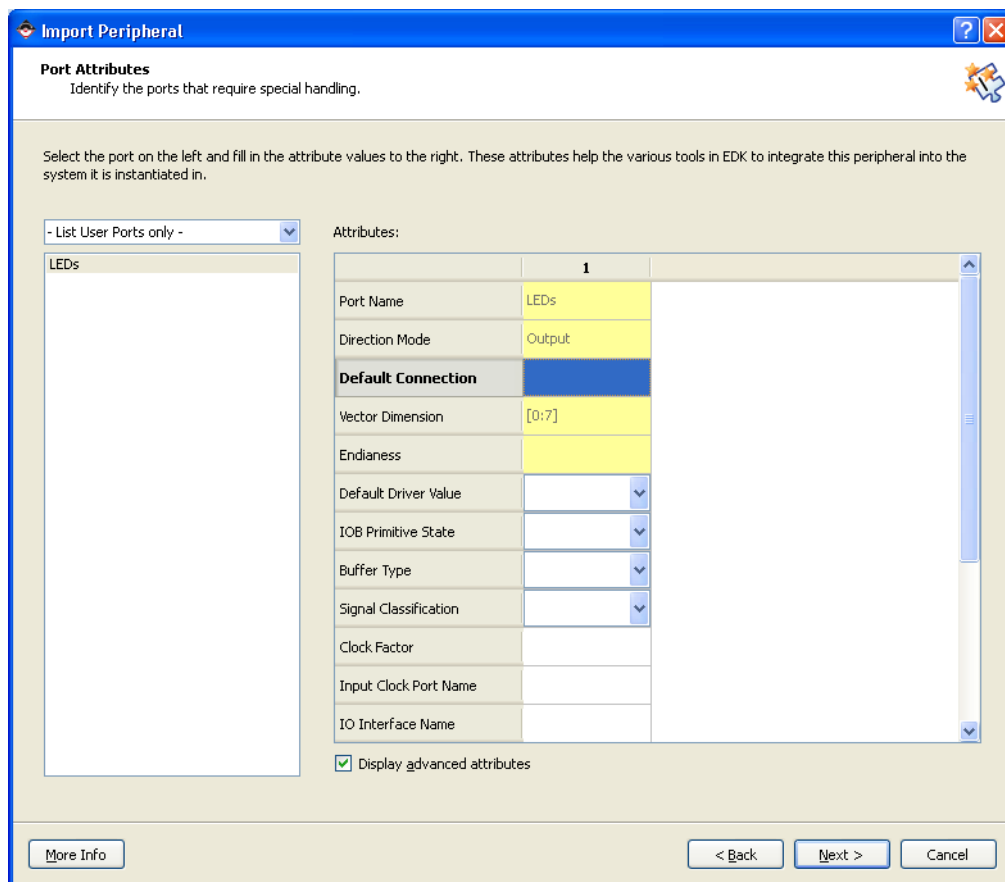
Figure 7-20: **Port Attributes Page**

You can see all ports that will be used by the interface between your custom pcore and your embedded processor subsystem by selecting **List All Ports** from the drop-down list. The Create and Import IP wizard managed this part of your design for you.

19. Click **Next**. The Project Details page appears, displaying the details of your project.
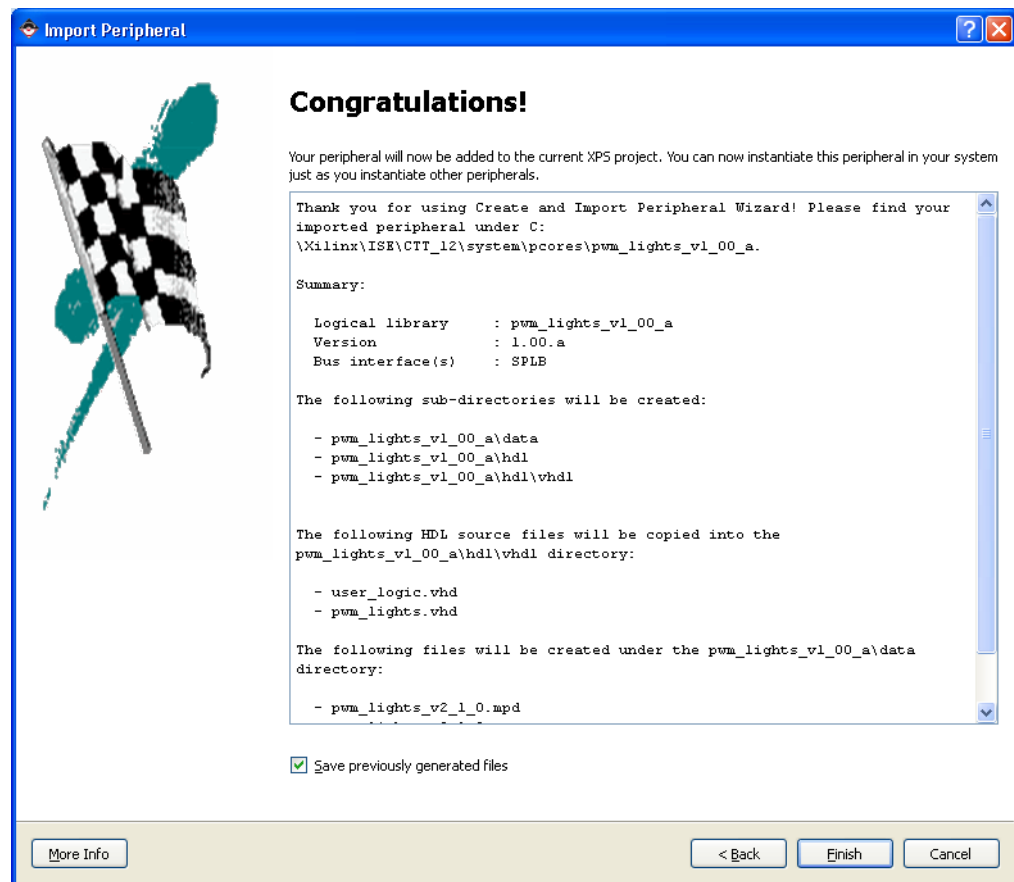
Figure 7-21: **Project Details Page**

20. Click **Finish**. The Import function completes.

Adding the pwm_lights Pcore to Your Project

You can see your custom peripheral listed in the IP Catalog under `Project Local pcores/USER`.

Before adding `pwm_lights` to your design, you must make one change to the existing design. The four LEDs on the evaluation board are currently connected to GPIO outputs. Now that `pwm_lights` is driving these LEDs, the `LEDs_4Bit` pcore must be removed from the design.

1. In the System Assembly View, right-click `LEDs_4Bit` and select **Delete Instance**. The Delete IP Instance dialog box appears:
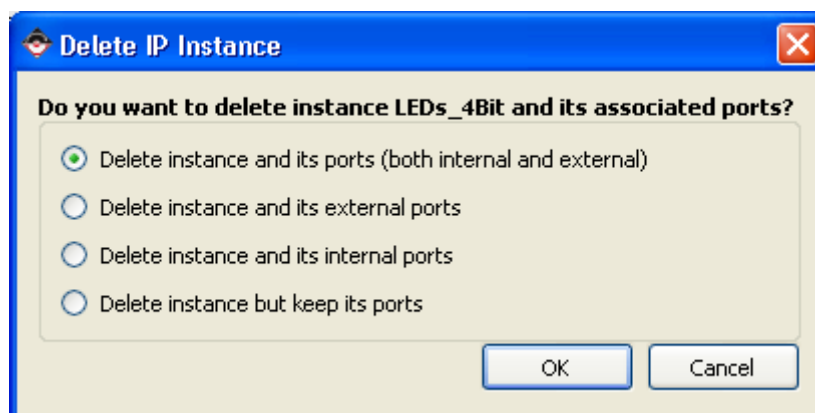
Figure 7-22: **Delete IP Instance Dialog Box**

2. Accept the default setting. You'll add the external ports back into the design manually.

3. Locate the `pwm_lights` pcore in the IP Catalog, right-click the pcore, and select **Add IP**.

   XPS adds the IP to the System Assembly View. You can see it in the Bus Interfaces tab.

4. In the Connectivity Panel, click the PLB bus to add the bus connection.
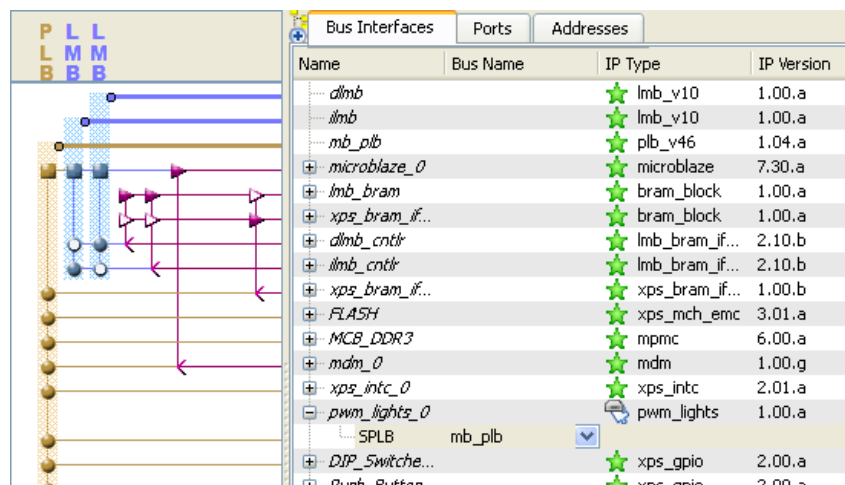
Figure 7-23: **Connecting Your New IP in the Bus Interfaces Tab**

The `pwm_lights` core is now added to the embedded system. However, you must make the external connections between `pwm_lights` and the LEDs on the evaluation board.

5. Click the Ports tab, expand `pwm_lights_0`, and select **Make External** from the drop-down menu in the **Net** column.

   A default name of `pwm_lights_0_LEDs_pin` was assigned as the net name. You can view this name by expanding the **Name** column.

6. To change the assigned net and pin names, click in the **Name** and **Net** columns, respectively. Alternatively, you can manually edit the MHS file. For now, don't change the assigned names.

   The next step is to generate or assign an address for the `pwm_lights` pcore.

7. Click the Addresses tab. If you don't see `pwm_lights_0` under **Unmapped Addresses**, select **Project > Rescan User Repositories**.

8. Click **Generate Addresses**. The `pwm_lights_0` pcore is assigned to an address range of $0xC4600000 - 0xC460FFFF$.

9. Verify that the address range for `MCB_DDR3` is $0x88000000 - 0x8FFFFFFF$. If this address has changed, change it back to the original value.

   If it seems strange for a simple peripheral to be assigned a 64Kbyte address space, don't worry. A wider address space requires decoding of fewer address lines. In an FPGA, a decoder with many inputs is implemented as a cascade of lookup tables.

   The deeper the cascade, the slower the operating frequency. By assigning wide peripheral address ranges, the resulting FPGA implementation will run faster.

   The final step is to update the UCF constraints file to assign the LED outputs to the proper FPGA pins.

10. Select the Project tab and double-click the `system.ucf` file to open it in the XPS main window.

11. Look for `fpga_0_LEDs_4Bit_GPIO_IO_O`. These pin assignments were left in the UCF file even though you earlier deleted the GPIO pcore. It is important to note that removing a pcore does not automatically trigger an update to the UCF file.

12. Replace `fpga_0_LEDs_4Bit_GPIO_IO_O_pin` with `pwm_lights_0_LEDs_pin` in all four locations and save the UCF file.

   Congratulations, you have created a custom pcore!

Exporting the Design and Generating a New Bitstream

The next steps are to export the hardware design and generate a new bitstream and then test this new pcore in hardware.

1. In XPS, select **Project > Export Hardware Design to SDK**. Accept the default directory and click **Export Only**.

2. When the export process completes, close XPS, return to ISE, and double-click **Generate Programming File**.

3. After ISE finishes generating the programming file, launch SDK.and create a new workspace. Put it anywhere you like.

   Once SDK opens, we need to point to the exported system.xml file.

4. Select **File > New > Xilinx Hardware Platform Specification** to create a new hardware project.

5. Call the Project **LEDS** and point to the `system.xml` file. It should be in the `/system/SDK/SDK_Export/HW` folder of your project.

6. Click **Finish**.

   SDK opens to the C/C++ Perspective with a table showing all the IP in your design. Confirm that `pwm_lights_0` is listed.

7. Create a new Xilinx C project. Use the **Hello World** project template, and use the default name that SDK provides.

   Note: Recall that you can create a new Xilinx C project by selecting **File > New > Xilinx C Project**.

   Using the project template is useful for now, because it will do a lot of the driver setup for you. As you become more proficient with SDK, you can do much of this manually.

8. Click **Next**.

   The default selection on this page is to have SDK create a new BSP package. Don't change the default settings.

9. Click **Finish** to generate the project.

10. Right-click the new project in the Project Explorer and select **New > Source File**.

    Note the "C" in the graphic for this menu selection, indicating that the source file is a C file.

11. Make the following selections for the new source file:

    – Source Folder: `hello_world_0/src`

    – Source File: `leds.c`

    – Template: Default C source template

12. Click **Finish**.

    The leds.c file opens in the SDK window. This is a blank file for now, but you'll be adding content to it next.

13. From the Zip file that accompanies this guide, open the `leds.c` file that is supplied for you. Copy the C code from the file and paste it into the `leds.c` file that you just created in SDK.

    Note: For information about the Zip file and where to get it, refer to "Attachments to this Guide," page 6.

14. Save and close leds.c. Because you have the Build Automatically feature enabled, SDK automatically compiles the project.

    Note: Recall that you can toggle the Build Automatically feature on and off by selecting **Project > Build Automatically**.

    The compile fails, because there are two `main()` functions in the project.

15. Delete hello_world.c and platform.c by right-clicking them in the SDK Project Explorer and selecting **Delete**.

    The project should now compile successfully.

16. Download the bitstream using Bootloop as the initialization file.

    Note: Recall that you learned how to do this in "Take a Test Drive! Debugging in SDK," page 39.

17. Debug `leds.elf`.

18. Run `leds.elf` with a terminal window open. When the application code begins to run, the HyperTerminal window displays the debug options:
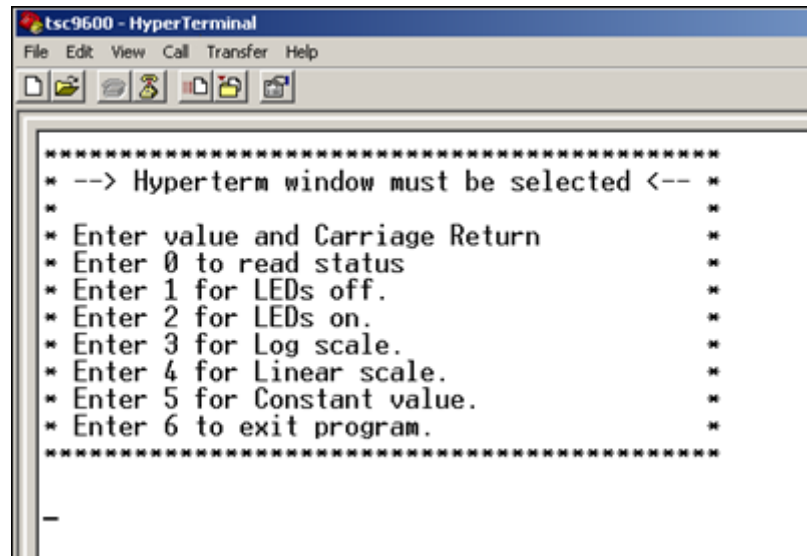
Figure 7-24: **HyperTerminal with Debug Options**

Select the terminal window and type input values.

19. Experiment with the program, and verify that the LED behaves as expected.

What Just Happened?

You used the CIP wizard to create custom IP. While there are many steps required to complete the task, you should now be familiar enough with the steps that you should be able to use the CIP wizard efficiently in the future.

What's Next?

In the next chapter you are going to create a dual processor design, then debug the design in EDK.

Chapter 8

Dual Processor Design and Debug

MicroBlaze™ processors are soft microprocessors. There can be as many MicroBlaze processors in an FPGA as will fit. Building a dual-processing system with the Xilinx® Platform Studio (XPS) Base System Builder (BSB) is virtually identical to building a system with one MicroBlaze processor. You can also debug an embedded system easily with more than one MicroBlaze processor using the Software Development Kit (SDK).

Note: Dual processing is not supported for AXI-based designs.

Using the BSB to Create a Dual-Processor Design

If you are using an FPGA with a PowerPC® processor, these concepts also apply. You can use the BSB to build dual-PowerPC designs (if your FPGA contains two PowerPC processors), or a design with one PowerPC and one MicroBlaze processor. Using the Spartan®-3A DSP board, you can create a dual-MicroBlaze processor design and then extend it in XPS by adding more MicroBlaze processors to the design.

Take a Test Drive! Creating an Embedded System with Two MicroBlaze Embedded Processors

In this Test Drive, you will use the Base System Builder and ISE® Project Navigator to create and implement a dual-processor system using a method similar to that used in "Creating a New Project," page 11.

Note: A single embedded project can have multiple processors. In this Test Drive, the embedded project has two MicroBlaze processors.

1. Create a new ISE Project with a single embedded processor source.

    Wait for XPS to open automatically.

2. When XPS starts, select the option to create a new project using the BSB wizard.

3. Accept all the defaults in the BSB *except* for those listed below.

Wizard Screens	System Property	Setting or Command to Use
System Configuration	Type of system: single-processor or dual-processor	Select a dual-processor system.
Processor Configuration	Processor type and settings	Change the Local Memory to 8 KB for both processors.
Peripheral Configuration	Processor 1 Peripherals	Remove the following peripherals from the default list: - DIP Switches - Ethernet MAC - LEDs The remaining peripherals for Processor 1 are DDR2, RS232, and the DLMB and ILMB BRAM controllers.
	Shared Peripherals	Remove XPS mutex. The remaining peripheral is XPS mailbox.
	Processor 2 Peripherals	Remove the following peripherals from the default list: - Push buttons - SPI FLASH The remaining peripherals for Processor 2 are the ILMB and DLMB BRAM controllers.

4. When you complete your design, examine your new system in the System Assembly View in XPS.

    The two embedded processor systems are completely independent, each with its own memory map. The exception is the mailbox peripheral, which is essentially a dual-port RAM with one port connecting to the PLBv46 bus on one processor and the other port connecting to the PLBv46 bus on the other processor. To learn more about the mailbox peripheral, right-click it in the System Assembly View and select **View PDF Datasheet**.

5. Export the `system.xml` file to SDK by selecting **Project > Export Hardware Design to SDK**.

6. Use the default location and click **Export Only**.

7. Navigate back to ISE Project Navigator and add the UCF to your ISE Project.

    Note: For information about adding a UCF to your ISE project, refer to Chapter 4, "Working with Your Embedded Platform."

8. With your .xmp file selected in the Design panel, double-click **Generate Programming File** to implement the design and generate a bitstream.

You now have a bitstream ready for downloading to the target hardware and a `system.xml` file for use in SDK.

Take a Test Drive! Using SDK to Develop Software for a Dual-Processor System

In this Test Drive, you will create an SDK project that can be used to develop and debug software for both embedded processors.

This section should seem familiar after completing Chapter 5 and Chapter 6 because there is very little difference between debugging a single-processor or dual-processor system when using SDK.

Preparing an SDK project consists of creating software platforms and then creating C or C++ application projects for them. This process is the same regardless of how many processors are in your system. When you create a software platform, you must identify the processor with which the platform will be used.

1. Launch SDK.

2. When the Workspace Launcher dialog box opens, create a new workspace called `Dual_Processor_Workspace.` Save it to a directory of your choice.

3. In the New Hardware Specification File dialog box, point to the `system.xml` file that you exported earlier. If you used the default project locations, the file is located at *<ISE Project Name>*`\system\SDK\SDK_Export\hw\system.xml`.

    When the `system.xml` file is imported, the C/C++ Perspective opens. SDK recognizes that there are two MicroBlaze processors in the embedded system, as shown below.
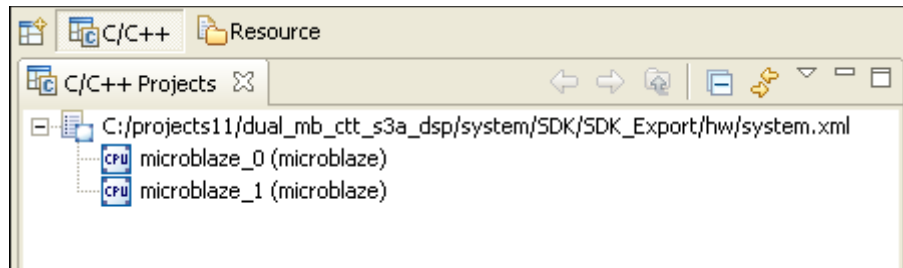

Figure 8-1: **Dual MicroBlaze Processors Displayed in the Embedded System**

4. Select **File > New > Software Platform** and create a new software platform project with the following settings:

    –   Project Name: **MicroBlaze_Platform_0**

    –   Processor: **microblaze_0 (microblaze)**

    –   Platform Type: **standalone**

    –   Project Location: **Use default**

5. Use the same procedure to create a second software platform project with the following settings:

   – Project Name: **MicroBlaze_Platform_1**

   – Processor: **microblaze_1 (microblaze)**

   – Platform Type: **standalone**

   – Project Location: **Use default**

   Each MicroBlaze processor has a single associated software platform, as shown in the following figure.
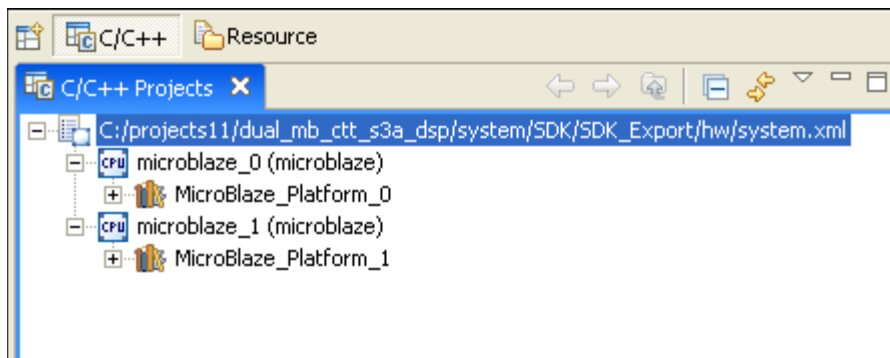
Figure 8-2:   **MicroBlaze Processors and Associated Software Platforms**

The next step is to create a Managed C Application Project for each processor. In this example, you will create and modify a "hello world" project for each processor.

6. Select **File > New > Managed Make C Application Project** and create a new Managed Make C application project with the following settings:

   – Project Name: **hello_world_0**

   – Software Platform: **MicroBlaze_Platform_0**

   – Project Location: **Use Default Location for Project**

   – Sample Applications: **Hello World**

7. Use the same procedure to create a second Managed Make C application project with the following settings:

   – Project Name: **hello_world_1**

   – Software Platform: **MicroBlaze_Platform_1**

   – Project Location: **Use Default Location for Project**

   – Sample Applications: **Hello World**

You now have two sample C applications, one for each processor, as shown in the following figure.
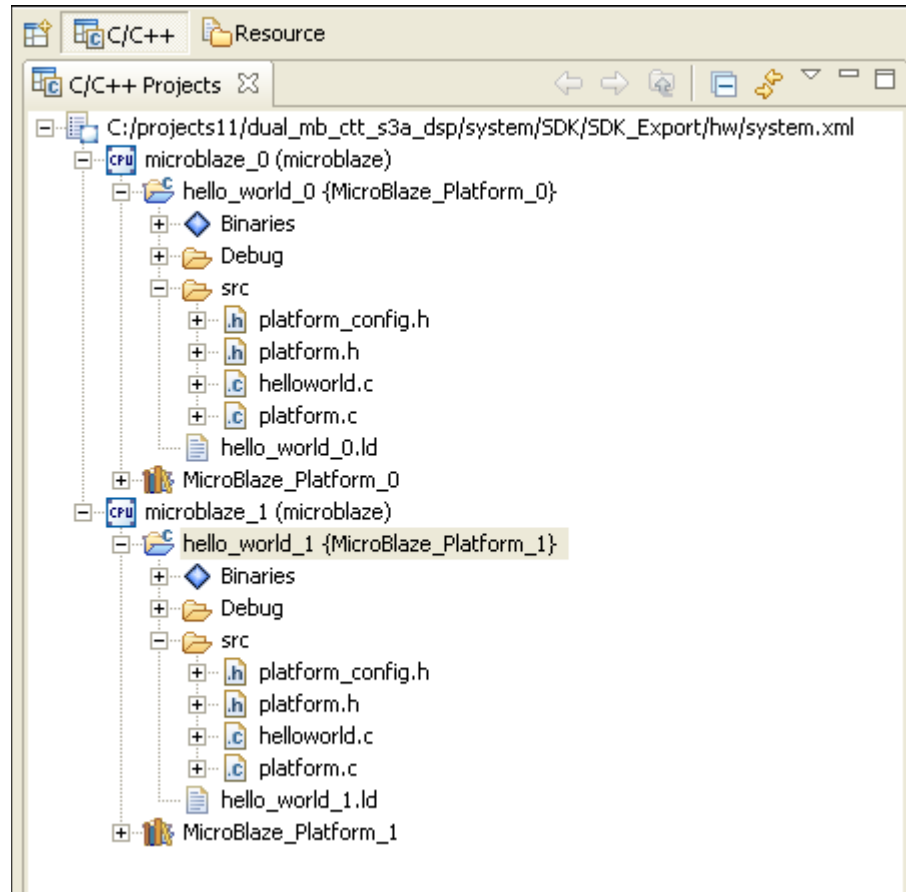
Figure 8-3: **MicroBlaze Processors and Associated Managed C Application Projects**

8. Modify the `src\helloworld.c` file in each MicroBlaze platform to indicate which processor it is running on. For example, open the `helloworld.c` file in the `MicroBlaze_Platform_0` project and change the code

   ```
   print("Hello World\n\r");
   ```

   to

   ```
   print("Hello From Processor 0!\n\r");
   ```

9. Modify the `helloworld.c` file in the `MicroBlaze_Platform_1` project in the same way.

10. Save each file. SDK automatically builds the files while saving. Note the output in the console window:

    ```
    ************** Determining Size of ELF File **************

    mb-size hello_world_1.elf
     text data bss dec hexfilename
     1958 296 2090 4344 10f8hello_world_1.elf

    Build complete for project hello_world_1
    ```

Programs must have enough memory on which to run the application.
In this sample design, only `MicroBlaze_Platform_0` has access to the external DDR2
memory, and `MicroBlaze_Platform_1` only has access to 8 KB of on-chip block RAM. As
you can see in the console output displayed above, `hello_world_1.elf` is 4.344 KB, less
than 8 KB, so there is sufficient memory.

Take a Test Drive! Modifying the Software Platform Settings

The `MicroBlaze_Platform_0` processor uses its UART for the `stdin` and `stdout`
peripherals. However, the `MicroBlaze_Platform_1` processor does not have a UART and
must use XMD with MDM-UART for the `stdin` and `stdout` peripherals. Consequently,
the software platform settings for `MicroBlaze_Platform_1` might need to be modified.

1. Select **Tools > Software Platform Settings** to modify the settings for the
   `MicroBlaze_Platform_1` processor.

2. In the OS and Libraries settings page, review the `stdin` and `stdout` settings. In this
   case, the hardware on which XMD runs needs to be set to `MDM`. The default value is
   `mdm_0`, so this is already correctly set.

If you run through the same exercise to view the settings for the `MicroBlaze_Platform_0`
processor, you'll notice that `stdin` and `stdout` are set to use `xps_uartlite`.

Take a Test Drive! Debugging Multiple Processors in a Single SDK Debug Perspective

Each embedded processor must have a separate binary ELF file. SDK names the file
automatically based on the processor name. For example, the `MicroBlaze_Platform_0`
processor binary file is called `hello_world_0.elf`, and the `MicroBlaze_Platform_1`
processor binary file is called `hello_world_1.elf`.

Each binary file is individually downloaded. Before that happens, you must download the
bitstream for the design with the dual processor system to the target hardware.

1. Select **Tools > Program FPGA** and select your bitstream and block memory map files
   from the following locations:

   – `<ISE Project Name>\system.bit`

   – `<ISE Project Name>\edkBmmFile_bd.bmm`

2. Confirm that the initialization file for each processor is set to **BootLoop**, then click
   Save and Program.

   The target hardware has now been programmed with the bitstream for the dual
   processor design.

   The next step is to download an ELF file for each processor.

3. Right-click the binary file `hello_world_0.elf` in the `hello_world_0`
   `{MicroBlaze_Platform_0}` folder of the C/C++ Projects tree, and select **Debug As >
   Debug on Hardware**.

   The Debug Perspective opens for the `MicroBlaze_Platform_0` processor.

4. Go back to the C/C++ Perspective, and use the same procedure to debug the `hello_world_1.elf` file and to download this file to the `MicroBlaze_Platform_1` processor.

The Debug Perspective opens again. There are two debugging tasks in the Debug window.
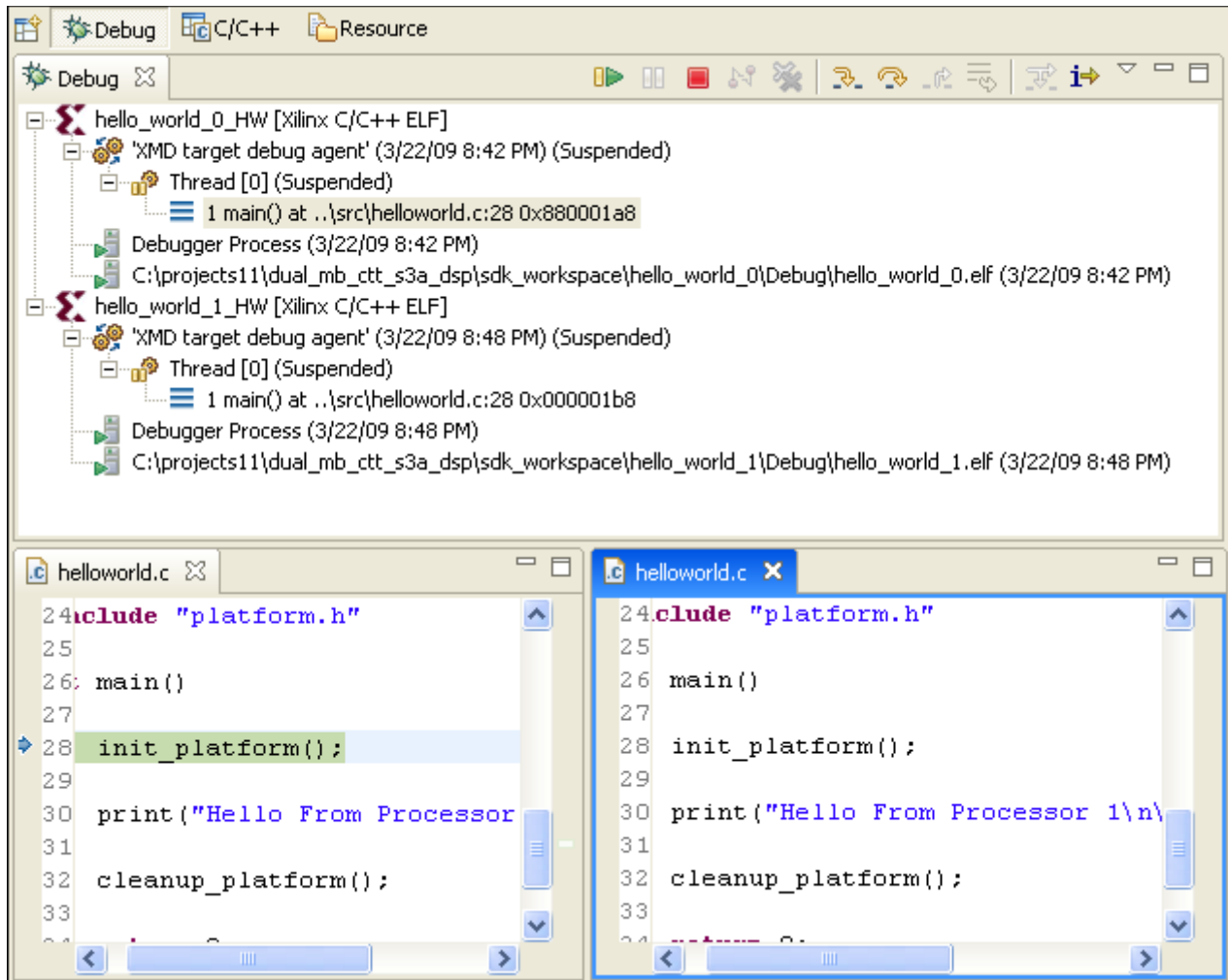
Figure 8-4: **The SDK Debug Perspective**

5. Before debugging these programs, connect an RS232 cable between your computer and the target board to observe the console I/O in a terminal window.

6. In the XMD window, type **terminal** to stream terminal I/O over the MDM. You can use this window to monitor the output of `MicroBlaze_Platform_1`.

   Note: If the XMD window isn't available, select **Window > Show View > Other** and select **Xilinx > XMD Console**.

7. Highlight the call stack in the Debug window for either `hello_world_0` or `hello_world_1`, and select **Run > Resume**.

   Note: Both call stacks should read "1 main() at..."

The processor output displays in the terminal window or XMD console, as shown in the following figure.

Figure 8-5: **Processor Output in the Terminal and XMD Console Windows**

Debugging more than one processor design in SDK is similar to debugging a single processor. This was a simple example. You can perform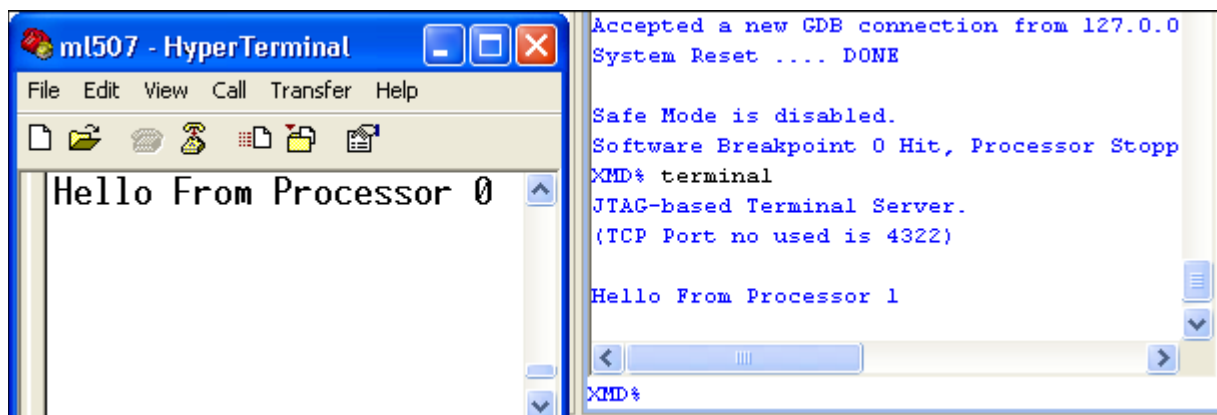 other software development tasks with SDK as well, such as stepping, setting breakpoints, and examining registers and memory.

Intellectual Property Bus Functional Model Simulation

This chapter uses the `pwm_lights` design and the bus functional model (BFM) simulation platform from Chapter 7, "Creating Your Own Intellectual Property." Before proceeding, you must complete the design described in Chapter 7, and you must have also compiled the EDK simulation libraries.

What are BFMs and Why Should I Use Them?

Bus functional models are simulation models used to model the behavior of bus transactions. In this case, the PLBv46 bus. BFMs are typically used only to model the behavior of custom IP. A secondary use of BFMs is to speed up the simulation of complex transactions, because BFM simulation runs much faster than post-synthesis or timing simulation.

The specification for PLBv46 is long and complex. You might need to attach custom IP to the bus and then verify that the IP meets the bus specification. If you had to create the testbenches to create the proper stimulus and checking necessary to confirm proper operation, few if any designers would ever simulate the bus behavior. In addition, you could never be certain that the testbenches were written correctly.

To assist with this, a set of Bus Functional Models (BFMs) were created as known good stimulus models. In addition to containing models for PLBv46 master and slave devices, a monitor module captures and verifies the correctness of the transactions.

The final piece involves the BFM compiler (BFC). Using a specific design language, you can write a series of read and write bus transactions along with expected values.

While conceptually simple, manually setting up an entire simulation environment to run bus functional simulation is difficult. The CIP wizard automates most of this for you, because it can automatically connect the necessary BFM simulation models to your IP under test.
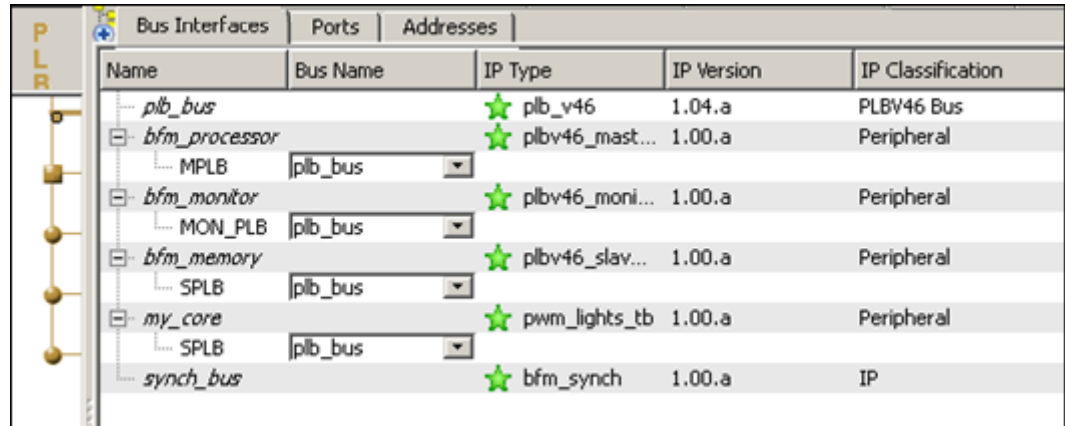
Note: If the Version Management wizard opens when you launch the `bfm_system` project, use the it to update any required cores.

Take a Test Drive! Running the BFM

Before starting this Test Drive, close any open XPS projects. If you elected to create the BFMs, the CIP wizard created the `bfmsim` sub-directory in your `\pcores\pwm_lights_v1_00_a\devl` directory, in which it saved the XPS BFM simulation project called `bfm_system.xmp`.

1. Open the `bfm_system.xmp` project in XPS. The Bus Interfaces window appears:

Figure A-1: **XPS BFM User PCORE Simulation Project**

2. Select **Project > Project Options** and click the HDL and Simulation tab.

3. Select the HDL format in which to simulate. For this example, use the default, VHDL.

4. Because BFM offers Behavioral Simulation only, leave the Simulation Model selection set to its default.

5. Select **OK** when you have finished setting up the simulation options.

6. Select **Simulation > Generate Simulation HDL Files** to run the Simulation Model Generator (Simgen) for this test project.

   Simgen creates a simulation directory structure under the `/bfmsim` directory. The simulation directory contains the HDL wrapper files along with the `DO` script files needed to run a behavioral simulation.

7. Click **Custom Button 1** in the XPS GUI tool bar. The CIP wizard configures this tool bar button when it creates the BFM simulation project.

   Custom Button 1 initiates the following:

   – Launches a Bash shell to run a `make` file.

   – Calls the IBM CoreConnect™ ToolKit Bus Functional Compiler (BFC) to operate on a `sample.bfl` file using the simulation options that were previously set. See `<project name>\pcores\pwm_lights_v1_00_a\devl\bfmsim\scripts\ sample.bfl` for more information.

   – Invokes the simulator with the BFC output command files (`INCLUDE` or `DO` files) depending on the simulator to execute the commands in the `sample.bfl` file.

The simulator waveform result is similar to the one shown below.
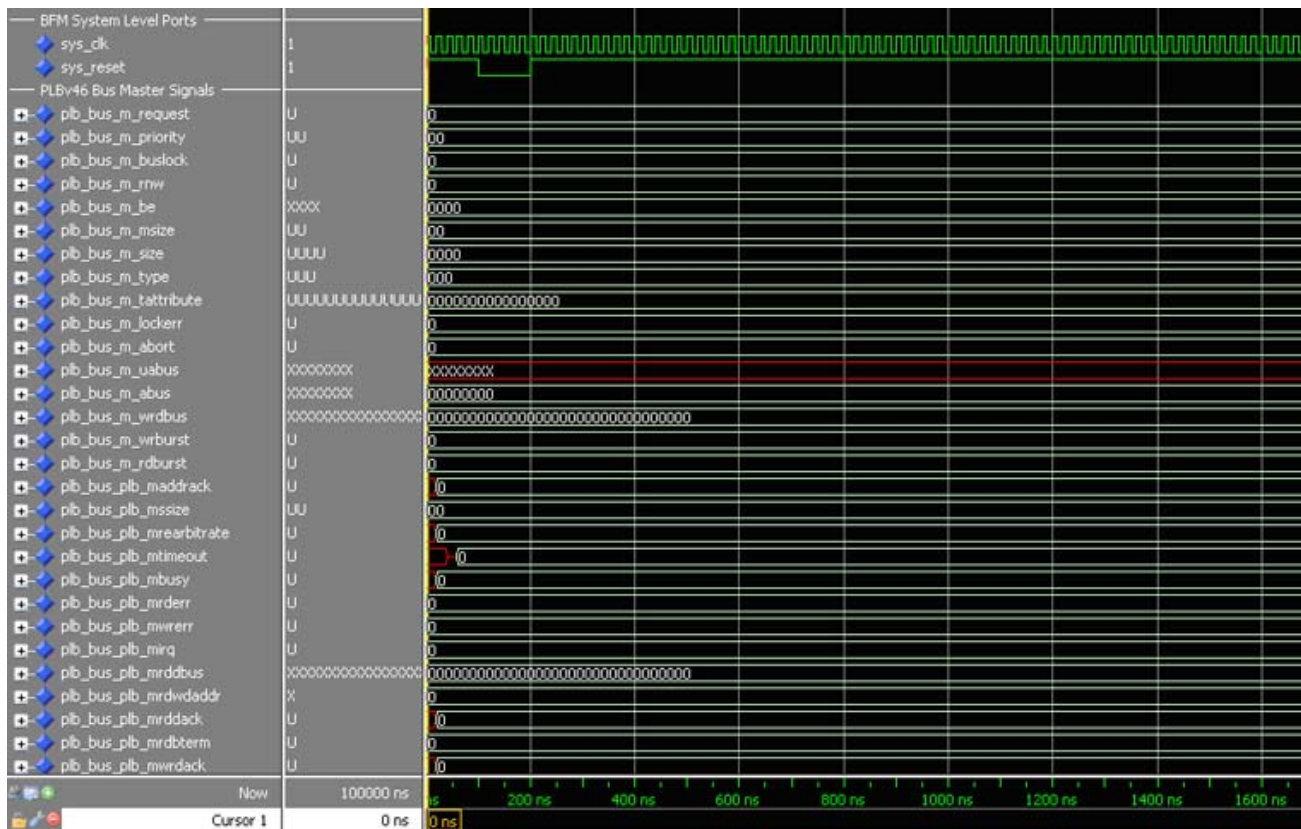
Figure A-2: **BFM Waveform Simulation Results for sample.bfl**

What Just Happened?

Before running the simulation, the CIP wizard created the following:

CIP Wizard Creates a Test Project

- A set of HDL templates files, which you modified to become a working pcore
- A test project, which isolates your pcore and allows you to verify its functionality with the bus before hooking it to a larger system

This project resides in the `<project_name>\pcores\pwm_lights_v1_00_a\devl\bfmsim` directory.

This test project uses several BFMs supplied by the CoreConnect ToolKit. In this case, there is a model of the processor, bus, memory, and bus monitor, all connected to your core.

Benefits of XPS Tools By using the XPS tools, you avoided having to create these models yourself and XPS automatically made all the correct connections.

After generating the simulation platform, you created your own custom button (**Custom Button 1**) to automate several steps in the simulation process. These steps run the `sample.bfl` through the CoreConnect Bus Functional compiler, and must be performed to generate the command file the simulator uses.

To see more information associated with these buttons, select **Project > Customize Buttons** and press **F1** to view the related help topic. The location of the `make` file you will use is provided in the following Test Drive.

In addition to compiling the BFL, the `make` file executed by **Custom Button 1** called the simulator with the command files to start simulation, simplifying the simulation launch and compilation process to a single button click.

Take a Test Drive! Writing a Script to Perform Bus Transactions

1. In XPS, select **File > Open** and navigate to the `<project name>\pcores\pwm_lights_v1_00_a\devl\bfmsim\scripts` directory.

2. Open the `sample.bfl` file.

   Replace this file with the attached `sample.bfl` file if you want to see what happens in this Test Drive without walking through all the steps. If you elect to replace the file, you can skip ahead to step 5.

   Approximately the first 160 lines of code set command aliases to make it easier to use and read the command lines. These commands automatically populate source and destination memory, and test the various core features. You can add or subtract commands to various sections as your core requires or create a completely new BFL command file.

BFL Command Information

Note: If you create a new BFL file, you must also adjust the `bfm_sim_xps.make` file in the /bfmsim directory to reflect your command file. For more information about the BFL commands, look in your `$XILINX_EDK\third_party\doc` directory for the `PLBToolkit.pdf` file.

Next, you'll modify the BFM code and run an actual simulation of `pwm_lights` to demonstrate the power of bus functional simulation.

The completed `sample.bfl` file is included in the .zip file for this guide. You can use it for comparison once you complete this Test Drive or to replace the `sample.bfl` file that you will generate. The following commands that you about to add to your `sample.bfl` file are saved in the attached text file, `bus_transaction_bfl_code.txt`.

Adding Commands to sample.bfl

3. In the `sample.bfl` template, at approximately line 175, change the line starting with `configure` to read:

   `configure(msize = 01)`

4. After the `start testing` section near the end of the file, add the following code from the `bus_transaction_bfl_code.txt`. These commands generate bus transactions.

```
--
-- Define several bus transactions for pwm_lights
-- Memory updates are 64 bits write, bus transactions are 32 bits wide.
--
--
-- Write value of 22 hex to LED register
--
mem_update(addr=30000010,data=22222222_22222222)
write (addr=30000010,size=0000,be=11110000)
-- Read status register, expect to get F0F02207
read (addr=30000000,size=0000,be=11110000)
-- Write to offset 0, then read status, expect to get F0F02208
mem_update(addr=30000000,data=00000000_00000000)
write (addr=30000000,size=0000,be=11110000)
read (addr=30000000,size=0000,be=11110000)
-- Write to offset 4, then read status, expect to get F0F02200
--mem_update(addr=30000000,data=00000000_00000000)
write (addr=30000004,size=0000,be=00001111)
read (addr=30000000,size=0000,be=00001111)
-- Write to offset 8, then read status, expect to get F0F02208
mem_update(addr=30000008,data=00000000_00000000)
write (addr=30000008,size=0000,be=11110000)
read (addr=30000000,size=0000,be=11110000)
-- Write to offset C, then read status, expect to get F0F02200
--mem_update(addr=30000000,data=00000000_00000000)
write (addr=3000000C,size=0000,be=00001111)
read (addr=30000000,size=0000,be=00001111)
001111)
```

Modifying the code results in the following:

– The `mem_update` sets the write data value at the address to which you want to write.

– The write command initiates the PLB write.

– The read command initiates the PLB read.

– A size setting of 0000 implies a single transaction.

– The `be` (byte enable) settings correspond to a 64 bit bus.

– Addresses aligned to 0, 8, and so on, set the byte enables to `11110000`.

– Addresses aligned to 4, c, and so on, set the byte enables to `00001111`.

5. Open the attached `sample.bfl` file.

6. Make sure your `sample.bfl` file matches the attached file and then save your file.

7. Click **Custom Button 1**.

8. In ModelSim, scroll down to the end of the signal listing in the Waveform window. The last signals listed correspond to `user_logic` signals, which are the custom signals in the `pwm_logic` pcore.

What Just Happened?

The BFM script performed a series of five writes followed by reads in the `pcore`. Read the BFM script to see the order in which the writes were done:

- When `ip2bus_rdack` is high, the `ip2bus_data` signal contains the value read back from `pwm_lights`.

- For the order of writes to the core, the simulation will show (in order) `F0F02207`, `F0F02208`, `FF002200`, `FF002204`, and `FF002205`.

Verify this information for your simulation. The first value read back occurs at 450ns. This exercise helps you understand why these specific values were read back.

In addition to the BFL file, the CIP wizard created a corresponding `/pcores` directory under the `/BFMSIM` project that contains the template for the BFM test bench.

You can add to the template test bench as your core logic requires.

You have now seen the power of bus functional simulation, and how you might take advantage of how XPS does all the hard work (except, of course, writing the correct stimulus). As you create custom IP in the future, BFM simulation can both reduce your testing time and provide assurance that your IP functions as expected.

Creating an AXI-Based Design in EDK

Introduction

Beginning with the ISE® Design Suite release 12.3, Xilinx introduces support for the Advanced eXtensible Interface (AXI) in software tools and IP.

AXI is part of the ARM® Adavanced Microcontroller Bus Architecture (AMBA®) family of interfaces. For more information on AXI and AMBA, consult www.amba.com and the Xilinx AXI Reference Guide, available at www.xilinx.com/ipcenter/axi4.htm.

For embedded systems, the effect of the move to AXI is that the PLB interfaces on PLB-based peripherals will be replaced with AXI4 and AXI4-Lite interfaces. The way AXI4 and AXI4-Lite systems are built is also different from PLB. This appendix illustrates the changes in both of these areas.

The following table describes AXI4 and AXI4-Lite in more detail.

Table B-1: **AXI4 and AXI4-Lite Details**

Interface	Features	Replaces
AXI4	• Traditional memory mapped address/data interface. • Data burst support.	PLB v4.6 NPI XCL
AXI4-Lite	• Traditional memory mapped address/data interface. • Single data cycle only.	PLBv4.6 (singles only)

Additional Resources

More detailed documentation about AXI is available at:
http://www.xilinx.com/ipcenter/axi4.htm.

Creating an AXI Project

This guide describes, in detail, how to create a new PLB project using ISE, EDK, and SDK. When you use these tools to create an AXI project, however, some settings will be different. Follow this outline for creating an AXI project.

Hardware and Software Requirements

AXI-based designs are only supported in Spartan®-6 and Virtex®-6 Devices.

To create an AXI-based design using this guide, review the hardware and software requirements for this guide listed in Chapter 1, "Introduction."

Creating a New AXI Project

Start by creating your design in ISE as described in Chapter 2, "Creating a New Project."

When XPS opens, create your BSB project using the settings listed in the following table.

Note: If no setting or command is indicated in the table, accept the default values.

Wizard Screens	System Property	Setting or Command to Use
Interconnect Type	Interconnect type	**AXI system**
Welcome to the Base System Builder	Project type options	**I would like to create a new design**.
Board Selection	Board Vendor	**Xilinx**
	Board Name	**Spartan-6 SP605 Evaluation Platform**
	Board Revision	**1.0**
System Configuration	Type of system	**AXI System with Single MicroBlaze Processor**
Processor Configuration	Reference Clock Frequency	**200 MHz**
	Processor Frequencey	**100 MHz**
	Local Memory Size	**16 KB**
	Debugging Interface	**MDM**
	Enable Floating Point Unit	Do not enable the floating point unit.
Peripheral Configuration	Processor 1 (MicroBlaze) Peripherals list	**Remove** the following peripherals from the "Processor 1 (MicroBlaze) Peripherals" list of default values: • IIC_DVI • IIC_SFP **Add** the axi_timer peripheral and select the **Use Interrupt** check box.
Cache Configuration	Instruction Cache Size	**8 KB**
	Data Cache Size	**8 KB**

Wizard Screens	System Property	Setting or Command to Use
Summary	System Summary page	After you've selected and configured all of your system components, the BSB displays an overview of the system for you to verify your selections.
		You should have a processor system with the following components:
		• MicroBlaze™ processor
		• DIP Switch interface
		• Ethernet lite
		• IIC EEPROM interface
		• 4-bit LED interface
		• DDR3 interface
		• 4-bit pushbutton interface
		• UARTlite
		• Compact Flash Interface
		• Timer
		• One Debug module
		• Two LMB Block RAM interfaces
		• AXI Interconnect
		You can go back to any previous wizard page and make revisions.
		The BSB creates a default memory map. The memory map cannot be modified inside the BSB, but it can be changed after the BSB is finished.

Using Xilinx Platform Studio

Explore the XPS software as described in Chapter 3, "Using Xilinx Platform Studio."

When you open an AXI design in XPS, the Bus Interfaces tab allows all masters to talk to all slaves. You can disable unnecessary connections by clicking the connection point in the Patch Panel. This reduces the overall size of the AXI Interconnect.

The Bus Connectivity panel and the System Assembly view are color coded to assist in rapid recognition of the bus standard type. The same color coding is used in the System Block Diagram.

In both the Bus Interfaces and Ports tabs, you can reorder the top-to-bottom list of IPs.

XPS automatically applies the connectivity rule for associated bus standards and adjusts related connections as needed. For example, each bus interface can have one connector connected to a point-to-point bus, and a slave AXI bus interface can be connected to multiple AXI master interconnections within the same AXI interconnect. Switching connection from one AXI interconnect to another automatically disconnects all the connections from the previous AXI interconnect.

As you make connectivity changes in the Connectivity panel, the corresponding Bus Connection column in the System Assembly view is also updated. When you move your mouse pointer over a connector, XPS displays connection information, such as the bus standard and instance names and the connected bus interface.

Right-click on one of the AXI Interconnects and note the **View PDF Datasheet** option. The datasheet for the AXI Interconnect has a great deal more detail regarding the configuration possibilities of AXI-based embedded systems. Perhaps the most powerful feature that AXI provides is the ability to have finer granularity in clock domain partitioning over what PLB allows. This flexibility enables higher performance in embedded systems.
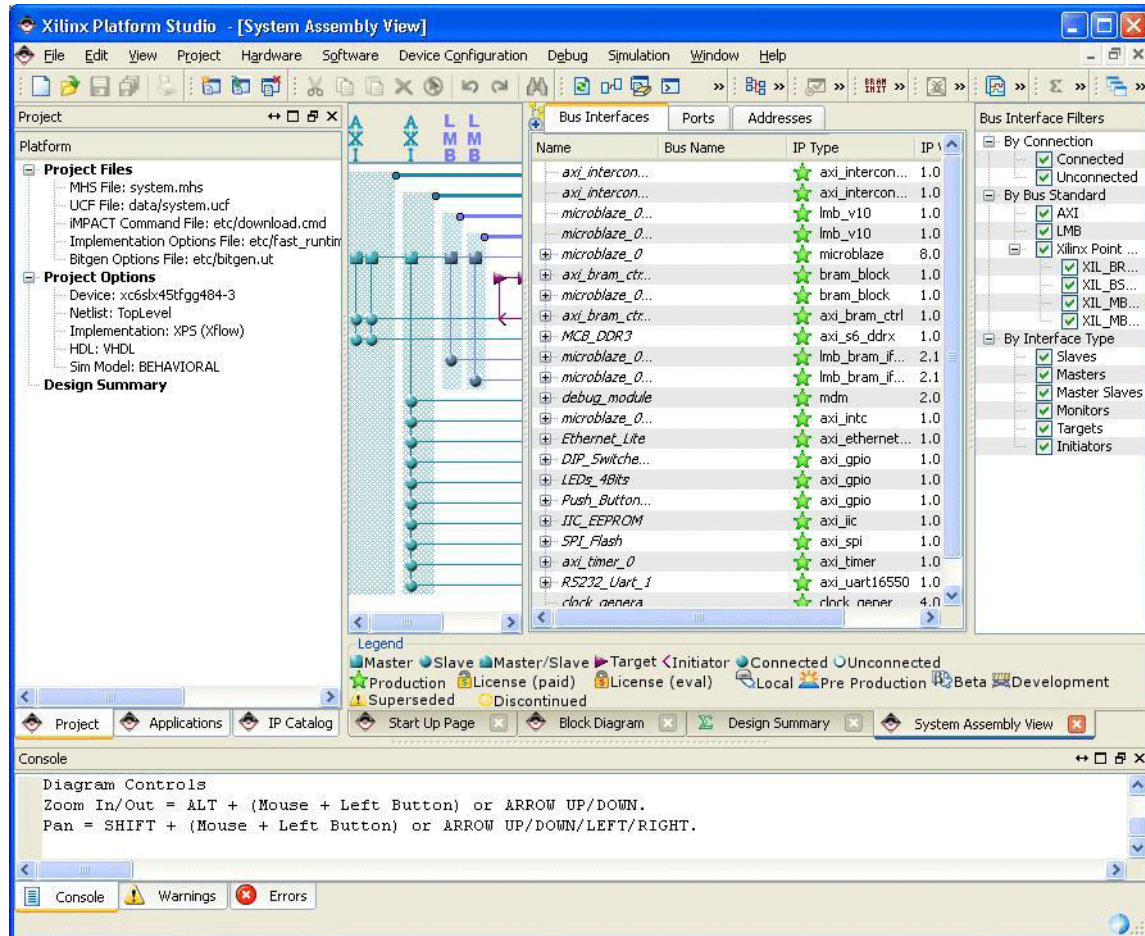
Figure B-1: **EDK Main Window**

Working with Your Embedded Platform

Follow the procedures listed in Chapter 4, "Working with Your Embedded Platform" to learn how to work with your embedded hardware platform.

Working with your AXI Design in SDK

Export your design to SDK and continue working with it as described in Chapter 5, "Introducing the Software Development Kit."

Continue use SDK to edit, debug, and release your design as described in Chapter 6, "More on the Software Development Kit: Edit, Debug, and Release."

Note: Dual Processor design is not supported for AXI designs.

Using the Create and Import Peripheral Wizard with an AXI Design

XPS contains a Create and Import IP (CIP) wizard that automates adding your IP to the IP repository in Platform Studio. To aid in creating AXI IP, refer to solution record, 37425, available at http://www.xilinx.com/support/answers/37425.htm, which contains simple templates that help create AXI-compliant IP. Creation of AXI-based IP is not covered in this guide.

Migrating to Xilinx AXI Protocols

Migrating an existing core is a process of mapping your core's existing I/O signals to corresponding AXI protocol signals. In some cases, additional logic might be needed. The MicroBlaze™ embedded processor changed its endianan-ness to go from having a Big-endian orientation, (which aligned with the PLB interfaces of the PowerPC® embedded processors), to Little-endian (which aligns with ARM processor requirements and the AXI protocol).

For more information on the implications of the move to Litle Endian, consult the Xilinx AXI Reference Guide.

AXI to PLBv46 Bridge

The Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI4) to Processor Local Bus (PLB v4.6) Bridge translates AXI transactions into PLBv.46 transactions. It functions as 32- or 64-bit Slave on AXI4 and a 32- or 64-bit Master on the PLBv.46.

For more information about the AXI to PLB v.46 Bridge, refer to the Xilinx AXI Reference Guide.

PLBv46 to AXI Bridge

The PLBv46 to AXI Bridge translates PLB v.46 transactions into AXI transactions. This IP has the Slave on a PLBv46 bus and a Master for an AXI Interconnect Bus.

For more information about the AXI to PLBv46 Bridge, refer to the Xilinx AXI Reference Guide. You can also examine the AXI to PLBv46 Bridge data sheet, accessible from within Xilinx Platform Studio.

BFM

Bus functional models (BFMs) are not supported for AXI designs.

Glossary

A

AXI

Advanced eXtensible Interface (AXI).

B

BBD file

Black Box Definition file. The BBD file lists the netlist files used by a peripheral.

BFL

Bus Functional Language.

BFM

Bus Functional Model.

BIT File

Xilinx® Integrated Software Environment (ISE™) Bitstream file.

BitInit

The Bitstream Initializer tool. It initializes the instruction memory of processors on the FPGA and stores the instruction memory in blockRAMs in the FPGA.

block RAM (BRAM)

A block of random access memory built into a device, as distinguished from distributed, LUT based random access memory.

BMM file

Block Memory Map file. A BMM file is a text file that has syntactic descriptions of how individual block RAMs constitute a contiguous logical data space. Data2MEM uses BMM files to direct the translation of data into the proper initialization form. Since a BMM file is a text file, it is directly editable.

BSB

Base System Builder. A wizard for creating a complete design in Xilinx Platform Studio (XPS). BSB is also the file type used in the Base System Builder.

BSP

See Standalone BSP.

C

CFI

Common Flash Interface

D

DCM

Digital Clock Manager

DCR

Device Control Register.

DLMB

Data-side Local Memory Bus. See also: LMB.

DMA

Direct Memory Access.

DOPB

Data-side On-chip Peripheral Bus. See also: OPB.

DRC

Design Rule Check.

DSPLB

Data-side Processor Local Bus. See also: ISPLB.

E

EDIF file

Electronic Data Interchange Format file. An industry standard file format for specifying a design netlist.

EDK

Xilinx Embedded Development Kit.

ELF file

Executable and Linkable Format file.

EMC

External Memory Controller.

EST

Embedded System Tools.

F

FATfs (XilFATfs)

LibXil FATFile System. The XilFATfs file system access library provides read/write access to files stored on a Xilinx SystemACE CompactFlash or IBM microdrive device.

Flat View

Flat view provides information in the Name column of the IP Catalog and System Assembly Panel as directly visible and not organized in expandable lists.

FPGA

Field Programmable Gate Array.

FSL

MicroBlaze Fast Simplex Link. Unidirectional point-to-point data streaming interfaces ideal for hardware acceleration. The MicroBlaze processor has FSL interfaces directly to the processor.

G

GDB

GNU Debugger.

GPIO

General Purpose Input and Output. A 32-bit peripheral that attaches to the on-chip peripheral bus.

H

Hardware Platform

Xilinx FPGA technology allows you to customize the hardware logic in your processor subsystem. Such customization is not possible using standard off-the-shelf microprocessor or controller chips. Hardware platform is a term that describes the flexible, embedded processing subsystem you are creating with Xilinx technology for your application needs.

HDL

Hardware Description Language.

Hierarchical View

This is the default view for both the IP Catalog and System Assembly panel, grouped by IP instance. The IP instance ordering is based on classification (from top to bottom: processor, bus, bus bridge, peripheral, and general IP). IP instances of the same classification are ordered alphabetically by instance name. When grouped by IP, it is easier to identify all data relevant to an IP instance. This is especially useful when you add IP instances to your hardware platform.

I

IBA

Integrated Bus Analyzer.

IDE

Integrated Design Environment.

ILA

Integrated Logic Analyzer.

ILMB

Instruction-side Local Memory Bus. See also: LMB.

IOPB

Instruction-side On-chip Peripheral Bus. See also: OPB.

IPIC

Intellectual Property Interconnect.

IPIF

Intellectual Property Interface.

ISA

Instruction Set Architecture. The ISA describes how aspects of the processor (including the instruction set, registers, interrupts, exceptions, and addresses) are visible to the programmer.

ISC

Interrupt Source Controller.

ISE

Xilinx ISE Project Navigator project file.

ISOCM

Instruction-side On-Chip Memory.

ISPLB

Instruction-side Peripheral Logical Bus. See also: DSPLB.

ISS

Instruction Set Simulator.

J

JTAG

Joint Test Action Group.

L

Libgen

Library Generator sub-component of the Xilinx Platform Studio technology.

LibXil Standard C Libraries

EDK libraries and device drivers provide standard C library functions, as well as functions to access peripherals. Libgen automatically configures the EDK libraries for every project based on the MSS file.

LMB

Local Memory Bus. A low latency synchronous bus primarily used to access on-chip block RAM. The MicroBlaze processor contains an instruction LMB bus and a data LMB bus.

M

MDD File

Microprocessor Driver Description file.

MDM

Microprocessor Debug Module.

MFS File

LibXil Memory File System. The MFS provides user capability to manage program memory in the form of file handles.

MHS file

Microprocessor Hardware Specification file. The MHS file defines the configuration of the embedded processor system including buses,peripherals, processors, connectivity, and address space.

MLD file

Microprocessor Library Definition file.

MVS file

Microprocessor Verification Specification file.

MOST®

Media Oriented Systems Transport. A developing standard in automotive network devices.

MPD file

Microprocessor Peripheral Definition file. The MPD file contains all of the available ports and hardware parameters for a peripheral.

MSS file

Microprocessor Software Specification file.

N

NCF file

Netlist Constraints file.

NGC file

The NGC file is a netlist file that contains both logical design data and constraints. This file replaces both EDIF and NCF files.

NGD file

Native Generic Database file. The NGD file is a netlist file that represents the entire design.

NGO File

A Xilinx-specific format binary file containing a logical description of the design in terms of its original components and hierarchy.

NPI

Native Port Interface.

NPL File

Xilinx® Integrated Software Environment (ISE®) Project Navigator project file.

O

OCM

On Chip Memory.

OPB

On-chip Peripheral Bus.

P

PACE

Pinout and Area Constraints Editor.

PAO file

Peripheral Analyze Order file. The PAO file defines the ordered list of HDL files needed for synthesis and simulation.

PBD file

Processor Block Diagram file.

Platgen

Hardware Platform Generator sub-component of the Platform Studio technology.

PLB

Processor Local Bus.

PROM

Programmable ROM.

PSF

Platform Specification Format. The specification for the set of data files that drive the EDK tools.

S

SDF file

Standard Data Format file. A data format that uses fields of fixed length to transfer data between multiple programs.

SDK

Software Development Kit.

SDMA

Soft Direct Memory Access

Simgen

The Simulation Generator sub-component of the Platform Studio technology.

Software Platform

A software platform is a collection of software drivers and, optionally, the operating system on which to build your application. Because of the fluid nature of the hardware platform and the rich Xilinx and Xilinx third-party partner support, you may create several software platforms for each of your hardware platforms.

SPI

Serial Peripheral Interface.

Standalone BSP

Standalone Board Support Package. A set of software modules that access processor-specific functions. The Standalone BSP is designed for use when an application accesses board or processor features directly (without an intervening OS layer).

SVF File

Serial Vector Format file.

U

UART

Universal Asynchronous Receiver-Transmitter.

UCF

User Constraints File.

V

VHDL

VHSIC Hardware Description Language.

X

XBD File

Xilinx Board Definition file.

XCL

Xilinx CacheLink. A high performance external memory cache interface available on the MicroBlaze processor.

Xilkernel

The Xilinx Embedded Kernel, shipped with EDK. A small, extremely modular and configurable RTOS for the Xilinx embedded software platform.

XMD

Xilinx Microprocessor Debugger.

XMP File

Xilinx Microprocessor Project file. This is the top-level project file for an EDK design.

XPS

Xilinx Platform Studio. The GUI environment in which you can develop your embedded design.

XST

Xilinx Synthesis Technology.

Z

ZBT

Zero Bus Turnaround™.