EDK Concepts, Tools, and Techniques

A Hands-On Guide to Effective Embedded System Design

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/13/2011	13.1	Book release for ISE® Design Suite 13.1.
07/06/2011	13.2	Book release for Design Suite 13.2.
10/19/2011	13.3	Book release for Design Suite 13.3. • Updated examples to match 13.3 software. • Removed the "Dual Processor Design and Debug" chapter. • Updated images of icons to match new icon graphics in the software.
01/18/2012	13.4	Revalidated for the 13.4 release. Editorial updates only; no technical content updates.
04/24/2012	14.1	Book release for Design Suite 14.1 release. • Updated version numbers of simulators in "Simulation Installation Requirements," page 7. • Edited Chapter 4, "Working with Your Embedded Platform"to use the PlanAhead™ design tool for project creation instead of Project Navigator. • Updated figures. • Removed Appendix regarding custom DSP designs. • Removed Appendix regarding Video IP.
07/25/2012	14.2	Book release for Design Suite 14.2 release. • Modified Test Drives and figures as needed to match new software version. • Added Chapter 8, "Creating a New Project Using the Vivado Design Suite." • Updated links in Appendix A, "Additional Resources."
10/16/2012	14.3	Book release for Design Suite 14.3 release. • Minor updates to match GUI terminology changes. • Updated links in Appendix A, "Additional Resources."

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!

The best way to learn a software tool is to use it, so this guide provides opportunities for you to work with the tools under discussion. 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

Links to more detailed documentation on EDK is available in Appendix A, "Additional Resources."

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 complicated 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. You can find EDK-specific terms in the Xilinx Glossary. A link to this document is located in Appendix A, "Additional Resources."

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) tools
- PlanAhead™ design analysis software
- 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. A link to this page is located in Appendix A, "Additional Resources."

The PlanAhead Design Tool

The PlanAhead design tool is the new tool offering by Xilinx. It targets complete system design, simulation, implementation, and debug in a single interface. It supersedes Project Navigator and streamlines the entire hardware design flow.

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 in batch mode or using the GUI, which is what we will be demonstrating in this guide. XPS is now integrated with PlanAhead tools, making development easier.

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

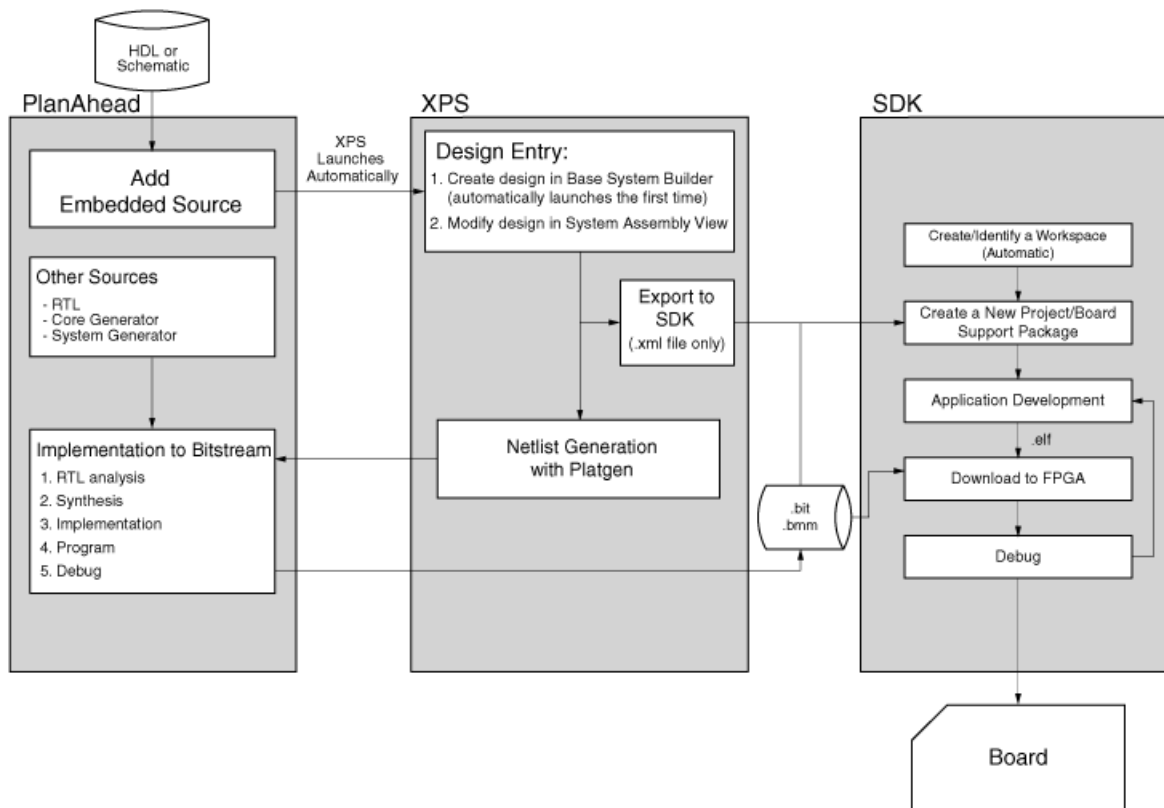EDK is designed to assist in all phases of the embedded design process.

Figure 1-1: **Basic Embedded Design Process Flow**

How the EDK Tools Expedite the Design Process

Figure 1-1, page 5 shows the simplified flow for an embedded design.

Typically, the ISE Design Tools development software is used to add an Embedded Processor source, which is then created in XPS using the Base System Builder.

- You use XPS for embedded processor hardware system development. Specification of the microprocessor, peripherals, and the interconnection of these components, along with their respective detailed configuration, 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.

  Three types of simulation are supported for embedded systems:

  - Behavioral
  - Structural
  - Timing-accurate

  You can simulate your project in either XPS or Project Navigator. When you start your design in Project Navigator, it automatically sets up the verification process structure.

After 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*. A link to this document is provided in Appendix A, "Additional Resources."

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 Vivado

EDK installation requirements

ISE design tools and Vivado™ 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.

Whether you use ISE or Vivado for your design, you will use the EDK software for your embedded design. EDK includes both XPS and SDK.

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 *Xilinx Design Tools: Installation and Licensing Guide*. A link to this document is located in Appendix A, "Additional Resources."

Simulation Installation Requirements

To perform simulation using the EDK tools, you must have an appropriate Secure-IP capable mixed-language simulator installed and simulation libraries compiled.

Note: If you're using ISim, the simulation libraries are already compiled.

Supported simulators include:

- For Linux Enterprise Edition:
  - ModelSim v10.1a
  - Incisive Enterprise Simulator (IES) v11.1 or later.
  - Synopsys Verilog Compiler Simulator (VCS) and VCS yMX 2011.12
  - ISim simulator (used in this tutorial)
- For Windows:
  - ModelSim v10.1a
  - ISim simulator (used in this tutorial)

You can optionally use AXI Bus Functional Model (BFMs) to run BFM Simulation. You must have an AXI BFM license to use this utility. For more information about using AXI BFMs for embedded designs with XPS, refer to the *AXI Bus Functional Models v1.9* data sheet *(DS824)*.

Simulation installation requirements

For information about the installation process, refer to the *Xilinx Design Tools: Installation and Licensing Guide*.

Links to these resources are available in Appendix A, "Additional Resources."

Hardware Requirements for this Guide

This tutorial is based on the Kintex™-7 KC705 FPGA Evaluation Board and cables.

If you have an older board, refer to the appropriate version of this manual by going to http://www.xilinx.com/support/documentation/dt_edk.htm and selecting a software release.

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 select and configure a processor and I/O interfaces, add internal peripherals, 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 requires the selection of an available development board, or a custom board. Supported development boards can be selected in Project Navigator, or if starting in XPS, in the BSB introduction screens.

Supported Boards

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 UCF file contains functional default values that are pre-selected in Xilinx Platform Studio (XPS). You can further enhance this base-level project in XPS and implement it with utilities provided by the ISE® tools.

Selecting a board type

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 an Interconnect Type

You can create an Advanced eXtensible Interface (AXI) system or a PLB system in the Base System Builder.

Selecting and Configuring a Processor

You can select the following for your MicroBlaze or PowerPC processor.

Note: PowerPC processors are available only for version 4 and 5 devices.

- Single-processor system or dual-processor system (for MicroBlaze processors)
- External reference clock frequency
- Reset polarity
- Cache setup (this is configurable if DDR is selected)
- 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 customize commonly used parameters for each peripheral.

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

Adding Internal Peripherals

You can add additional peripherals in the BSB wizard. 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.

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 a Kintex™-7-based KC705 Starter Board and targets a MicroBlaze processor. The options you select are listed in "Take a Test Drive! Creating a New Embedded Project," page 11.

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

Setting Up Software

The Software Development Kit (SDK) is required 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.

The BSB Wizard and the PlanAhead Tool

The following test drive walks you through starting your new project in the PlanAhead™ design tool and using the New Project wizard to create your project.

Take a Test Drive! Creating a New Embedded Project

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

1. Start the PlanAhead design tool.
2. In the Welcome window, select **Create New Project**.
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
Project Name	Project name	Choose a name and location for your project.
	Create Project Subdirectory	Leave this option checked.
Project Type	Select the type of source to import.	**RTL Project**
Add Sources	This is optional; no changes needed for this example.	
Add Existing IP	Optional; no changes needed.	
Add Constraints	Optional; no changes needed.	
Default Part	Specify	Select **Boards**.
	Board list	Select **Kintex-7 KC705 Evaluation Platform**.
New Project Summary	Review the project settings.	

When you click **Finish**, the New Project Wizard closes and a blank KC705 PlanAhead design tool project opens.

4. Verify the FPGA part, target language, and simulator settings by clicking **Project Settings** in the Flow Navigator panel.

5. To add an EDK project, in the Sources panel, right-click **Design Sources** and select **Add Sources**.

   Note: You can also use shortcut **Alt + A**.

   The Add Sources wizard opens.

6. Click **Add or Create Embedded Sources** and click **Next**.

   Note: You can also click **Alt + M** to select this option.

7. Click **Create Sub-Design**.

8. In the dialog box that opens, type `system` for the module name and click **OK**.

9. Click **Finish**.

   The PlanAhead design tool creates your embedded design source project. It recognizes that you have an embedded processor system and starts XPS.

Next, you'll use the BSB wizard to create a Base System in XPS.

1. In the dialog box opens to ask if you want to create a Base System using the BSB wizard, click **Yes**.

   The first window of the BSB asks you to elect whether to create an AXI-based or PLB-based system.

2. Select **AXI system** and click **OK**.

3. In the Base System Builder wizard, create a project using the settings described in the following table. If no setting or command is indicated in the table, accept the default values.

Wizard Screens	System Property	Setting or Command to Use
Board and System Selection	Board	Use the default option to create a system for the Kintex-7 KC705 Evaluation Platform. *Note:* This is pre-populated because you selected this board in the PlanAhead tool.
	Board Configuration	This information is pre-populated based on your board selection. *Note:* If you had selected to create a system for a custom board, these fields would be editable.
	Select a System	**Single MicroBlaze Processor System**
	Optimization Strategy	**Area**

Wizard Screens	System Property	Setting or Command to Use
Processor, Cache, and Peripheral Configuration	Processor Frequency	**100 MHz** (default)
	Select a Processor	microblaze_0
	Enable Floating Point Unit	Do not enable this setting.
	Local Memory Size	**64 KB**
	Instruction Cache Size	**8 KB**
	Data Cache Size	**8 KB**
	Select and Configure Peripherals	• Remove QSPI_FLASH from the Included Peripherals list. • Add the `axi_timer` peripheral and select the **Use Interrupt** check box.

4. To generate your design, click **Finish**.

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

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 (*.xbd2) file. The .xbd2 file:

• Defines the supported interfaces of a given board, system, or sub-system

• Enables you to create a system-level design through the Base System Builder

For more information about using .xbd2 files, refer to the "Microprocessor Peripheral Definition Translation tool (MPDX) chapter in the *Embedded System Tools Reference Manual (UG111)*. A link to this document is provided in Appendix A, "Additional Resources."

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.

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 software 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.
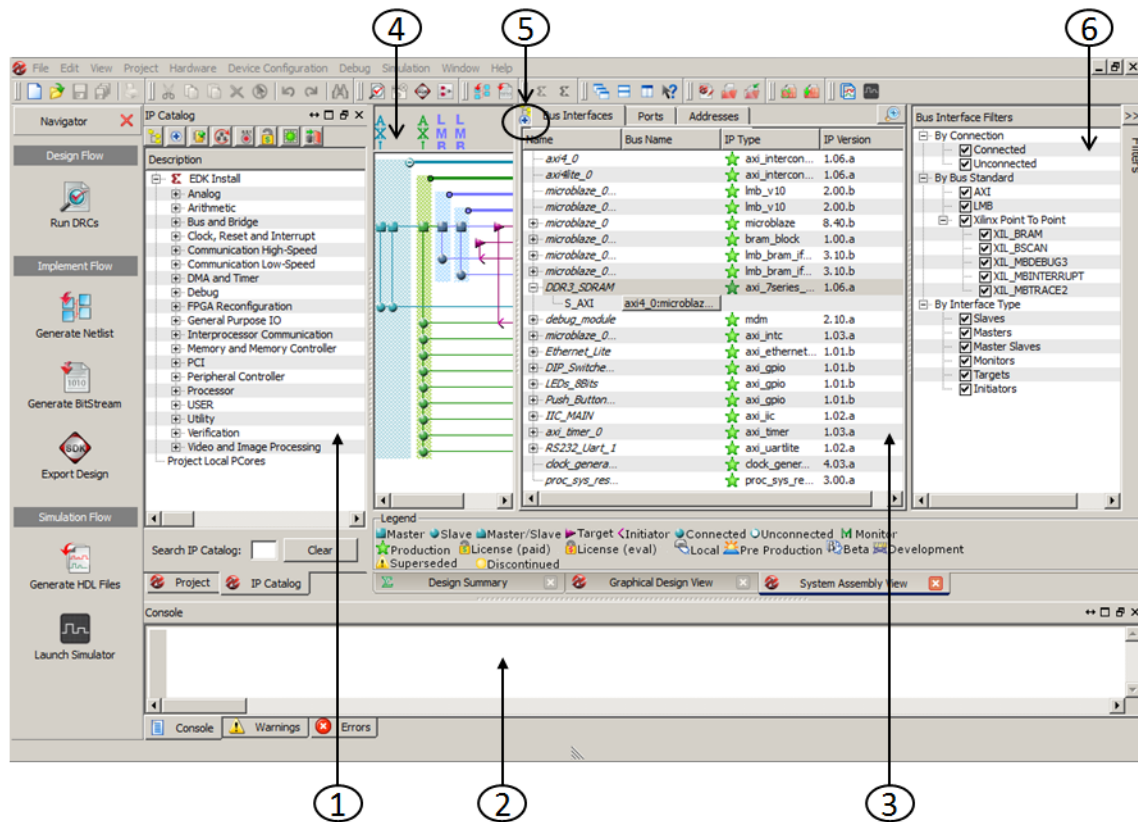
Figure 3-1: **XPS Project Window**

Using the XPS user interface

The XPS main window is divided into these three areas:

- Project Information Area (1)
- Console Window (2)
- System Assembly View (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 and IP Catalog tabs.

Project Tab

The Project Tab, shown in Figure 3-2, contains information on the current project, including important project files and implementation settings.

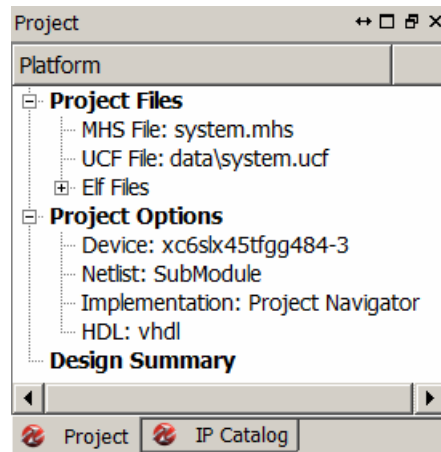*Figure 3-2:* **Project Information Area, Project Tab**

IP Catalog Tab

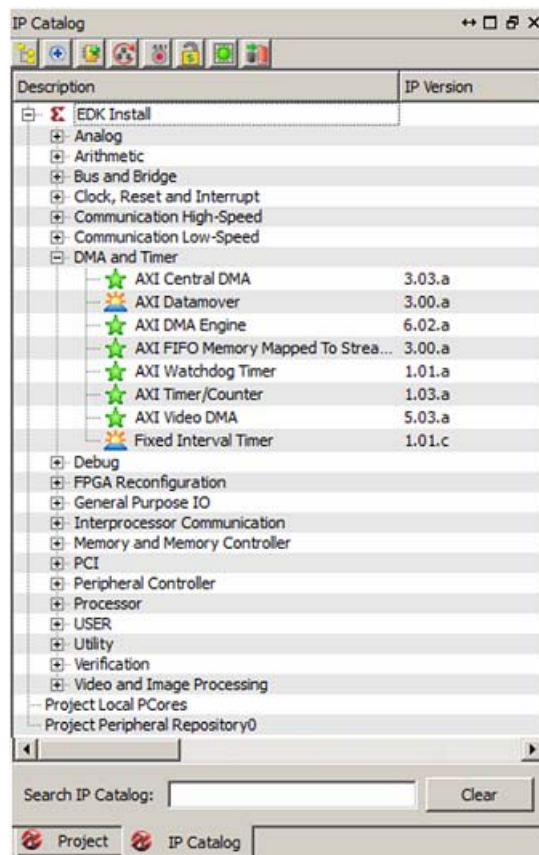The IP catalog tab is shown in Figure 3-3.

Figure 3-3: **IP Catalog Tab**

The IP Catalog tab 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. In the XPS software, at the top of the IP Catalog, note the following buttons:

   Change to Flat/Hierarchical View

   Expand/Collapse All Tree Nodes

   Add/Remove IP Repository Search Path

   Rescan User IP Repository

   Hide PLB IP Only

   Display Licensed Cores Only

   Display Cores Without License

   Create and Import Peripheral

   Click these buttons and observe changes to the IP Catalog.

2. 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 data sheet for the IP
   - **View IP Modifications (Change Log)/View Helper IP Modifications (Change Log)**, which lists the revision history for the selected IP and its dependencies.

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

4. Right-click the AXI UART(Lite) 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.

Console Window

The Console window (label 2 in Figure 3-1, page 16) provides feedback from the tools invoked during run time. Notice the three tabs: Console, Warnings, and Errors.

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 Interfaces** tab displays the connectivity in your design. Use this view to modify parameters of peripherals and interconnects.

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

- The **Addresses** tab displays the address range for each IP instance in your design.

Connectivity Panel

With the Bus Interfaces tab selected, you see the Connectivity Panel (label 4 in Figure 3-1, page 16), which is a graphical representation of the hardware platform connections. You can hover your mouse over the Connectivity Panel to view available bus connections.

AXI interconnect blocks are displayed vertically, and a horizontal line represents an interface to an IP core. If a compatible connection can be made, a connector is displayed at the intersection between the interconnect block and the IP core interface.

The lines and connectors are color-coded to show bus compatibility:

- Connection symbols indicate whether IP blocks are masters or slaves.

- A hollow connector represents a connection that you can make.

- A filled connector represents an existing connection.

Click the connector symbol to create or disable a connection.

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 16) 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.

Note: The Filters pane is collapsible along the right side of the System Assembly View.

View Buttons

The System Assembly View has two buttons to change how the data is arranged (label 5 in Figure 3-1, page 16). You can use these buttons to sort information and revise your design.

- **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 **Bus Interfaces** tab located at the top of the System Assembly View.

2. 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. Browse the tabs and settings available for this core. Do not make changes now.

   c. Close this dialog box when finished.

3. Click the **Ports** tab to switch to the Ports view.

4. Expand the **External Ports** category to view signals that leave the embedded system.

5. Note the signal names in the **Connected Ports** column and find the connected ports related to the RS232_Uart_1 ports. (You might need to drag the right side of the **Connected Ports** column header to see its entire contents.) These ports are referenced in the next step.

6. Scroll down, locate, and expand the RS232_Uart_1 peripheral.

   Note the port names and how they correspond to the names of external signals. The **Rx (Receive)** and **Tx (Transmit)** suffixes on the UART name are name-associated with the external ports.

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

Design Rule Check

The Design Rule Check (DRC) performs system-level design rule checks in XPS. When this command is performed, the Warnings and Errors tabs in the console are cleared to display the most recent design rule check messages.

To check design rules, select **Project > Design Rule Checks**, or click the **Project DRC** button .

Note: You might need to click the **Console** tab to view the Design Rule Check messages.

XPS Tools

XPS includes the underlying tools you need to develop the hardware components of an embedded processor system.

Platgen

XPS includes the Hardware Platform Generation tool, Platgen, which generates HDL and lower level netlists for the embedded processor system.

When you implement the FPGA design in the PlanAhead™ design tool, Platgen automatically runs if necessary.

Simgen

XPS includes the Simulation Model Generation tool, Simgen, for generating simulation models of your embedded hardware system based on your original embedded hardware design (behavioral) or finished FPGA implementation (timing-accurate).

Simgen also provides simulation models for external memory and has automated support to instantiate memory models in the simulation testbench.

Refer to the *Embedded System Tools Reference Manual* for information about external memory simulation. You can find a link to this document in Appendix A, "Additional Resources."

When you implement the FPGA design in the PlanAhead™ design tool, simulation is not available in XPS. Use the PlanAhead tool for simulation; it automatically invokes Simgen when necessary.

Create and Import Peripheral Wizard

XPS includes the Create and Import Peripheral (CIP) wizard to help 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 Create and Import Peripheral wizard is described in more detail in Chapter 6, "Creating Your Own Intellectual Property."

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 XPS created and see how it could be useful as the project development progresses.

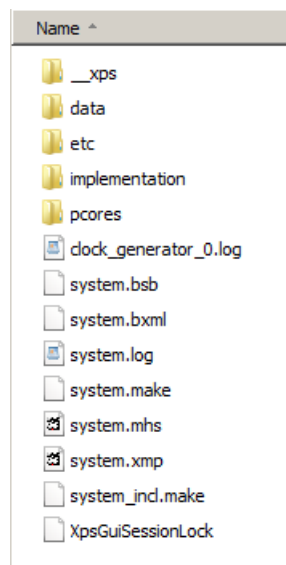**Note:** The files are stored in the location where you created your project file.

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.
`data`	Contains the user constraints (UCF) and PRJ files, which have information related to DDR part properties and pinouts.
	For more information on the UCF file and how to use it, see the UCF help topics in the ISE® help.
`etc`	Contains files that capture the options used to run various tools.
`implementation`	Contains the netlist and HDL wrapper files generated by Platgen.
`pcores`	Used for including custom hardware peripherals. The `pcores` directory is described in more detail in Chapter 6, "Creating Your Own Intellectual Property."

In the main project directory, you will also find numerous 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.
	Project Navigator uses this file as the main source file for the embedded processor system.
`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.

Your entire embedded hardware system is represented by the MHS file.

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*. A link to this document is available in Appendix A, "Additional Resources."

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 `axi_uartlite` 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 RX = RS232_Uart_1_sin
   ```

4. Search the file for another instance of the port name `RS232_Uart_1_sin`. 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 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 6: "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 the following three-step process:

1. XPS generates a netlist that is representative of your embedded hardware platform.

2. The design is implemented (mapped into FPGA logic) in the PlanAhead™ tool.

3. The implemented design is converted to the bitstream that can then be 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 generate the netlist, it invokes the platform building tool, Platgen, which does the following:

- Reads the design platform configuration MHS file and runs all necessary design rule checks to validate the correctness of the design.

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

- Produces netlist files (with an `.ngc` extension) for each peripheral, as well as the overall embedded system.

- Generates Hardware Description Language (HDL) wrapper files for each peripheral and the overall system. To see the created HDL files, look in the `<project_name>\system\hdl` directory.

More information about Platgen is provided in the "Platform Generator (Platgen)" chapter of the *Embedded System Tools Reference Manual*. A link to this document is available in Appendix A, "Additional Resources."

You can control netlist generation using the PlanAhead design tools. In the sections ahead, we will be doing the actual netlist generation using the PlanAhead tools.

Take a Test Drive! Generating the Bitstream

Now that you've described your Hardware Platform in XPS, you'll use the PlanAhead tool to implement the design and generate the bitstream.

Implementing the design using the PlanAhead tool

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

1. Close XPS.

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

Generating a bitstream and creating a UCF file

   A project in the PlanAhead tool has one top-level module that is the root of the design hierarchy for the purpose of implementation. When you create a new project, the highest level module is automatically assigned as the top module.

   An XMP file cannot be a top-level module. If an XMP file is the only source in your project, then you must generate a VHDL or Verilog wrapper file.

Generating a top HDL source

2. In the PlanAhead tool, make sure that your `system.xmp` file is selected in the Sources pane.

3. Right-click the .xmp file and select **Create Top HDL** to generate the wrapper file for the `system.xmp` file. This is now the top module for your project.
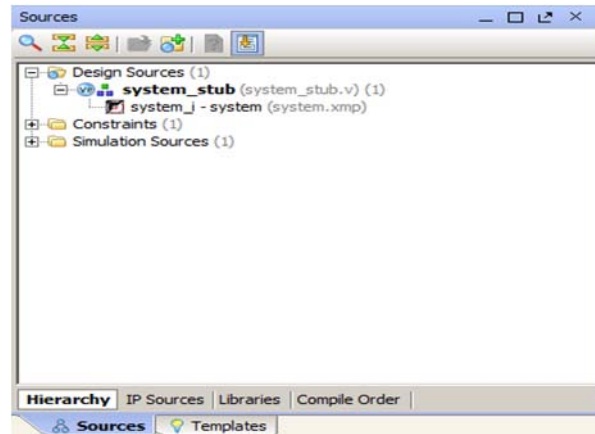
Figure 4-1: **Viewing the New Verilog Wrapper File**

4. In the **Flow Navigator** pane, expand **Program and Debug** and then click **Generate Bitstream**.

The Bitstream generation concludes with the Bitstream Generation Completed dialog box.
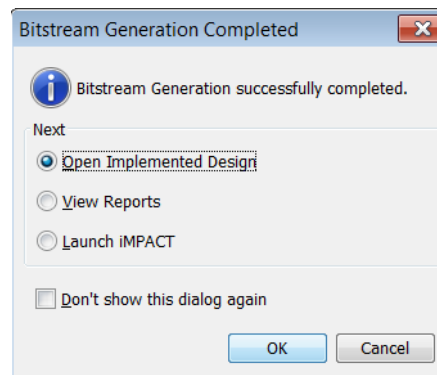
Figure 4-2: **Bitstream Generation Completed Dialog Box**

5. Select **Open Implemented Design** and click **OK**.

Generated bitstream files

The generated bitstream file is located in the `\<Project-Name.runs>\impl_1` folder of your project and is called `system_stub.bit`. There is another file generated called `system_stub_bd.bmm`, which SDK uses for loading memory onto your target board.

It is not necessary to add the constraint file (.ucf) generated by XPS to your project. The PlanAhead design tool automatically locates and uses the constraints during implementation.

Exporting Your Hardware Platform

You created your project in the PlanAhead tool and added an embedded processor source, then designed your hardware platform in XPS using the Base System Builder, and finally generated a bitstream for the FPGA. Now, you will export your hardware platform description to the Software Development Kit (SDK).

The exported 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

You can export your hardware platform from XPS or from the PlanAhead design environment. In this tutorial, you'll use the PlanAhead tool to export your hardware platform. You'll work with SDK in the next chapter.

1. In the PlanAhead tool, select **File > Export > Export Hardware for SDK**.

2. In the Export Hardware dialog box that opens, select the **Include Bitstream** and **Export Hardware** options.

3. Click **OK**.

What Just Happened?

The PlanAhead design tool exported your hardware design to SDK. It is important to understand the export operation, especially if you are managing multiple hardware versions.

When you export your hardware design to SDK, 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.

What's Next?

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

Software Development Kit

The Xilinx® Software Development Kit (SDK) facilitates the development of embedded software application projects. SDK is a complementary program to XPS. You use SDK to 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

Some common terminology used when describing SDK operation includes:

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

SDK terminology
When you open SDK, you create a *workspace*. A workspace is a directory location used by SDK to store project data and metadata. You must provide an initial workspace location when SDK is launched. You can create multiple workspaces to more easily manage multiple software versions.

A *software project* contains one or more source files, along with the necessary header files, to allow compilation and generation of a binary output (.elf) file. A workspace can contain multiple software projects. Each software project must have a corresponding *board support package*.

You must have a *hardware platform* for your design. 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. Multiple hardware platforms can exist in a single workspace.

A *board support package* (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 board support package 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 with 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.

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*.

Take a Test Drive! Creating a Software Project

Creating a software project using SDK requires the following steps:

1. Creating an SDK Workspace
2. Creating a Hardware Platform for the System
3. Creating an Application Project and Corresponding Board Support Package

Creating an SDK Workspace

1. Open SDK by selecting **Start > Programs > Xilinx Design Suite > EDK > Xilinx Software Development Kit**.

2. When SDK opens, it prompts you to create a workspace. This is the folder in which your software projects are stored. For this example, create a new workspace called **SDK_Workspace**.

    Caution! Make sure the path name does not include 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**.

Creating a Hardware Platform for the System

Next, you'll create a the hardware platform for your system.

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

    The New Project wizard opens.

2. Expand **Xilinx** and select **Hardware Platform Specification**.

3. Click **Next**.

    SDK opens the New Hardware Project wizard.

4. Type the project name **hw_platform_0** and use the default location.

5. In the **Target Hardware Specification** field, select the system.xml file in the <project name>\<project_name>.sdk\SDK\SDK_Export\hw folder of your project.

Figure 5-1 shows the New Hardware Project dialog box.
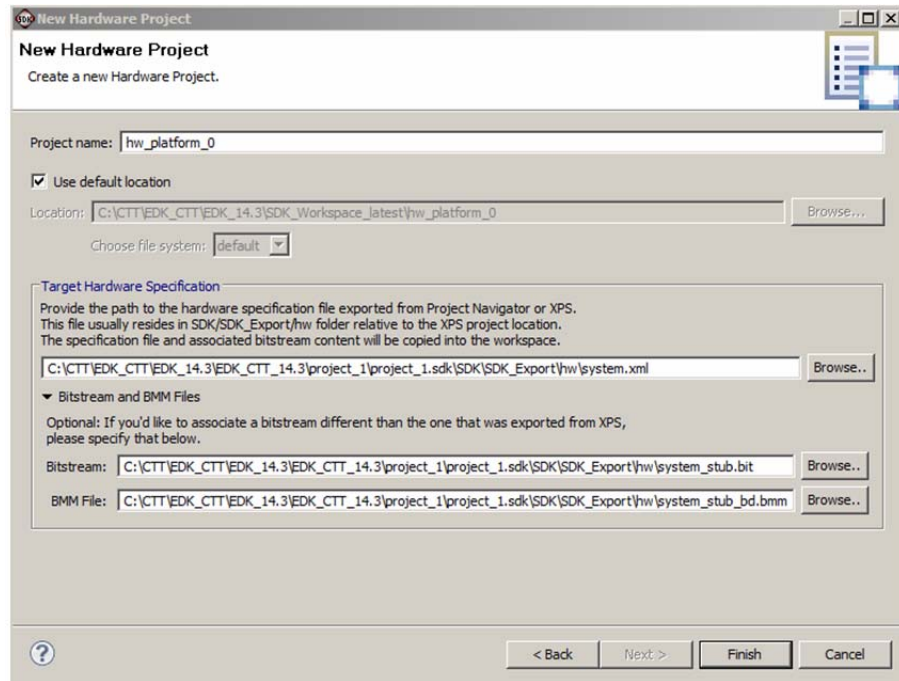
Figure 5-1:   **New Hardware Project Dialog Box**

6.   Click **Finish**.

SDK creates the hardware platform.

Creating an Application Project and Corresponding Board Support Package

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

The New Project wizard opens.

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

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	hello_world_0
	Use Default Location	Select this option
	Hardware Platform	hw_platform_0
	Processor	microblaze_0
	OS Platform	standalone
	Language	C
	Board Support Package	Select Create New and provide the name of **`hello_world_0_bsp`**.
Templates	Available Templates	Hello World

3.   Click **Finish**.

XPS creates the hello_world_0 application project and hello_world_0_bsp BSP project under the project explorer. It also builds both the application project and the BSP.

SDK displays your new hardware project with several panels. The most notable of these panels 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.

What Just Happened?

SDK examined your hardware specification file (`system.xml`) 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.

SDK also created the new Board Support Package `hello_world_bsp_0`.

The Project Explorer tab now contains information related to the hardware platform, the software project, and the BSP. The relevant project management information is displayed here.
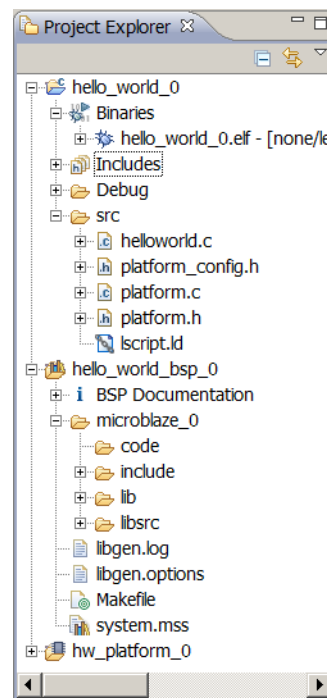
Figure 5-2: **Project Files Displayed in the Project Explorer Tab**

Let's explore the new project components.

- Expand the **microblaze_0** section under `hello_world_0_bsp` 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.

- Expand the `Binaries` item in the `hello_world_0` software project. The `hello_world_0.elf` file listed there is the ELF file that will be downloaded to the target board.

- Expand the `src` folder in the `hello_world_0` software project. 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.

Double-click the `lscript.ld` file to view the linker script contents in the text editor. If you are familiar with the intricacies of linker scripts, you can make changes by selecting **Xilinx Tools > Generate Linker Script**.

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 Kintex™ KC705 board as follows:

1. Connect two mini-USB cables between your computer and the two mini-USB jacks on the KC705 board.

   One of the USB connections connects to a JTAG download and debug interface built into the KC705 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. Turn on the power to your KC705 board.

3. If you haven't already installed the drivers for your KC705 board, you'll need to do it now.

   - 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.

   - Install the CP210x VCP drivers that came with your KC705 board. You can also find these drivers on the Silicon Labs website:
     http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx

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**.

   SDK automatically populates the bitstream (BIT) and block memory map (BMM) files for you.

2. Click **Program**.

   When the Programming completes, your FPGA is configured with your design.

   You've now 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 `hello_world_0 > Binaries`, right-click `hello_world_0.elf` and select **Debug As > Debug Configurations**.

   The Debug Configurations dialog box opens.

4. Double-click **Xilinx C/C++ ELF**.

   SDK creates a new debug configuration.

5. In Debug Configurations, click the **STDIO Connection** tab.

6. Enable **Connect STDIO to Console**.

7. Type the port name.

8. Type the baud rate of `9600`.

9. Click **Apply**.

    The PC serial terminal COM port connects to the SDK console window with the display set to 9600 baud, 8 bit data, and 1 stop bit. Be sure to set the COM port to correspond to the COM port that the Silicon Labs driver is using.

10. Click **Debug**.

    The Debug Perspective window opens.

The Debugd perspective

11. 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[1]` the `main()` function is currently sitting at line 30 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 left margin of the helloworld.c window and selecting **Show Line Numbers**.

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

    The output in the terminal window displays `Hello World`.

13. Terminate the debug session by clicking the **Terminate** button or pressing **Ctrl + F2**.

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.

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

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.

You can also view information about Application Programming Interface (API) documentation for each device driver in your install directory: `< install area>\EDK\doc\usenglish\xilinx_drivers.htm`.

SDK Windows

As demonstrated in the previous chapter, SDK has different predefined sets of display windows, called *perspectives*.

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 development and debug environment 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

So far, you have 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.

Changing Your Workspace

1. Select **File > Switch Workspace > Other**.

2. When prompted, create a new workspace and save it anywhere on your system.

   Note that SDK briefly closes and then reopens to the new workspace.

Creating New Application Projects

Now that the SDK project space is set up correctly, you can create a new hardware platform and application project.

1. Create a new hardware platform as described in "Take a Test Drive! Creating a Software Project," page 30.

2. Create an application project and corresponding board support package as described in "Creating an Application Project and Corresponding Board Support Package," page 31.

   In the next few steps, you will create two more Application 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 to create larger, real-life projects.

3. Create two more Application Projects for memory tests and peripheral test.

4. For the memory test project, make the selections indicated in the table below.

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	memory_test_0
	Use Default Location	Select this option
	Hardware Platform	hw_platform_0
	Processor	microblaze_0
	OS Platform	standalone
	Language	C
	Board Support Package	Select **Use Existing** and provide the name of **hello_world_0_bsp**.
Templates	Available Templates	Memory Tests

5. For the peripheral test project, make the selections indicated in the table below.

Wizard Screen	System Property	Setting or Command to Use
Application Project	Project Name	peripheral_test_0
	Use Default Location	Select this option
	Hardware Platform	hw_platform_0
	Processor	microblaze_0
	OS Platform	standalone
	Language	C
	Board Support Package	Select **Use Existing** and provide the name of **hello_world_0_bsp**.
Templates	Available Templates	Peripheral Tests

Running Your Applications

Before you can run these two applications, download the FPGA's bitstream to the board.

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

2. Click **Program**.

Let's observe what the two sample programs do. You'll run the memory_test application and then the peripheral_tests application.

Running the memory_test Application

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

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

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

4. Select **Run > Resume** to run the program. The program output displays on your terminal window. When the test runs successfully, it returns "--Memory Test Application Complete--."

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

Running the peripheral_tests Application

1. Right-click the `peripheral_tests_0.elf` file and select **Debug As > Launch on Hardware**.

2. Select **Run > Resume** to run the program. The program output displays on your terminal window. When the test runs successfully, it returns `---Exiting Main---`.

3. 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, open `memorytest.c` and `testperiph.c`.

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

2. In `memorytest.c`, change `main()` to `memorytest_main()`. This should be around line number 59.

   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 `main()` to `peripheraltest_main()`. This should be around line 51.

4. Save and close both files.

   The files build automatically. They will fail because 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. The `helloworld.c` file is in the `/src` folder in the C Project called `hello_world_0`. Open `helloworld.c` and modify it as shown in Figure 5-3.

```
25
26 void print(char *str);
27
28 int main()
29 {
30     init_platform();
31
32     print("Hello World\n\r");
33
34     memorytest_main();
35     peripheraltest_main();
36
37     cleanup_platform();
38
39     return 0;
40 }
41
```

Figure 5-3: **Modified Version of helloworld.c File**

6. Save and close 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, because 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 `hello_world_0` 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 `hello_world_0` folder. Figure 5-4 shows the source files that the directory should contain.

   Note: Do not move over the `platform_config.h`, `platform.c`, `platform.h`, or `lscript.ld` files. These files are already part of `hello_world_0`.
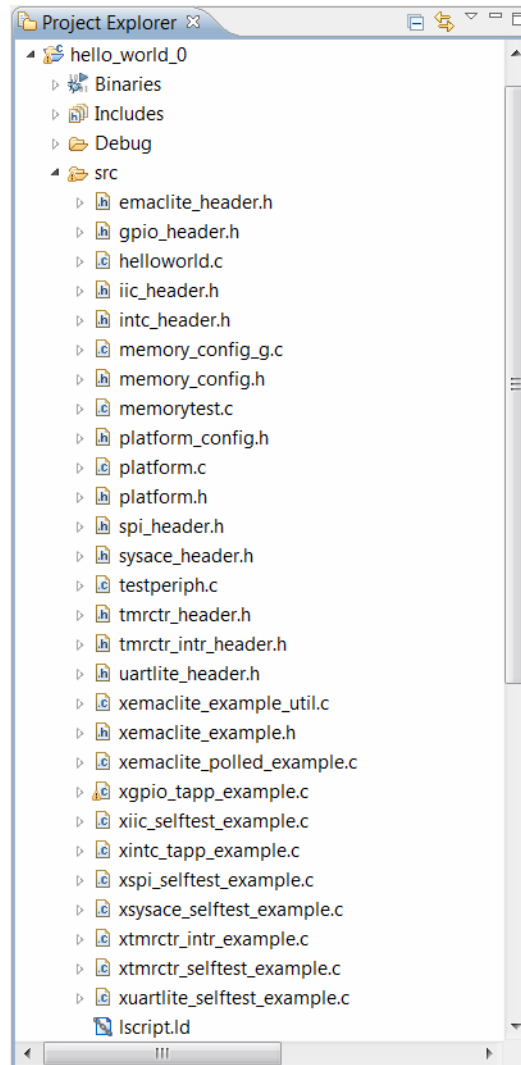
Figure 5-4: **Source Files in hello_world_0**

As you drag and drop the files, `hello_world_0` builds after each file. After you've dropped the last file and the ELF file successfully builds, the following message displays in the Console View. Your values might vary slightly.

```
Invoking: MicroBlaze Print Size
mb-size hello_world_0.elf  |tee "hello_world_0.elf.size"
   text    data    bss     dec     hexfilename
  35750     488    5480   41718    a2f6hello_world_0.elf
Finished building: hello_world_0.elf.size
```

Note: If you don't see this message, click one of the source files you just moved.

Note the size: 41718 (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 32K of instruction memory local to the MicroBlaze™ processor. Our application has grown to 46K, 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. Click on `memory_ranges` and then right-click and select **Open Declaration** to open the `memory_config_g.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_g.c` as follows:

```
{
"ddr3_sdram",
"axi_7series_ddrx",
0xC1000000, /Change from 0xC0000000 to 0xC1000000*/
134217728,
},
```

12. Save and close the file. It recompiles without errors.

13. Run the `hello_world_0.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.

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.

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 37 and have a binary file called `hello_world_0.elf`.

1. In the C/C++ Perspective, right-click on the executable file and select **Debug As > Launch on Hardware** to download `hello_world_0.elf` to your target board. The Debug Perspective automatically opens.

   When the Debug Perspective opens, it should look similar to Figure 5-5. 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.
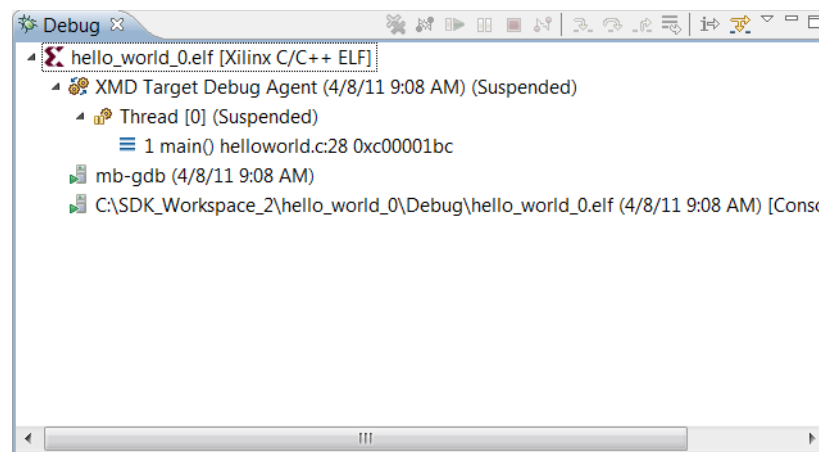

Figure 5-5: **Debug Perspective**

   As you can see, the MicroBlaze processor is currently sitting at the beginning of `main()` with program execution suspended at line `0xc00001bc`. You can correlate that with the Disassembly view, which shows the assembly-level program execution also suspended at `0xc00001bc`. 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 `0xc00001bc`.

   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 `0xc00001dc`.

4. Select **Run > Step Into** to step into the `peripheraltest_main()` routine. Program execution is suspended at location `0xc0000634`. 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.

7. Close SDK.

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 and either PlanAhead™ or Project Navigator. 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 AXI 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 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 XPS System Assembly View (shown in Figure 3-1, page 16) shows connections among buses, AXI devices, processors, and IP. Any piece of IP you create must be compliant with the system you design. To ensure compliance, 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:

   - AXI4-Lite: Simpler, non-burst control register style interface. Use AXI4-Lite for most simple, register based peripherals that only require single beat transfers.

   - AXI4: Burst Capable, high-throughput memory mapped interface. AXI4 provides high performance burst transfers, and you should use it when high speed access to and from memory systems is required.

- Processor Local Bus (PLB) version 4.6. The PLBv46 provides a high-speed interface between a PowerPC® processor and high-performance peripherals.

- 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.

A common design case is the need to connect your custom logic directly to an AXI interconnect block. With the CIP wizard, you can make that connection even without understanding AXI or AXI-Lite details. Both slave and master connections are available.

Creating 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 your 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.

CIP Wizard 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, download the Xilinx Documentation Navigator, available online at http://www.xilinx.com/support/documentation/navigator/xilinxdocnav_win.zip.

If you plan to create an AXI4 or AXI4-Lite slave peripheral, examine one of the appropriate data sheets for your custom peripheral.

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, "Software Development Kit."

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

   a. Open the Xilinx ISE® PlanAhead™ design tool and load your project.

      Note: If you are using Project Navigator, open it and load the project.

   b. In the PlanAhead tool Sources pane, double-click `system_i-system(system.xmp)` to invoke XPS.

      Note: If you are using Project Navigator, select `system.xmp` and double-click the **Manage Processor Design (XPS)** process (located under **Design Utilities**) to launch XPS.

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

   Click **Next** in the Welcome page to open the Peripheral Flow page. On this page, you can either create a new peripheral or import an existing peripheral.
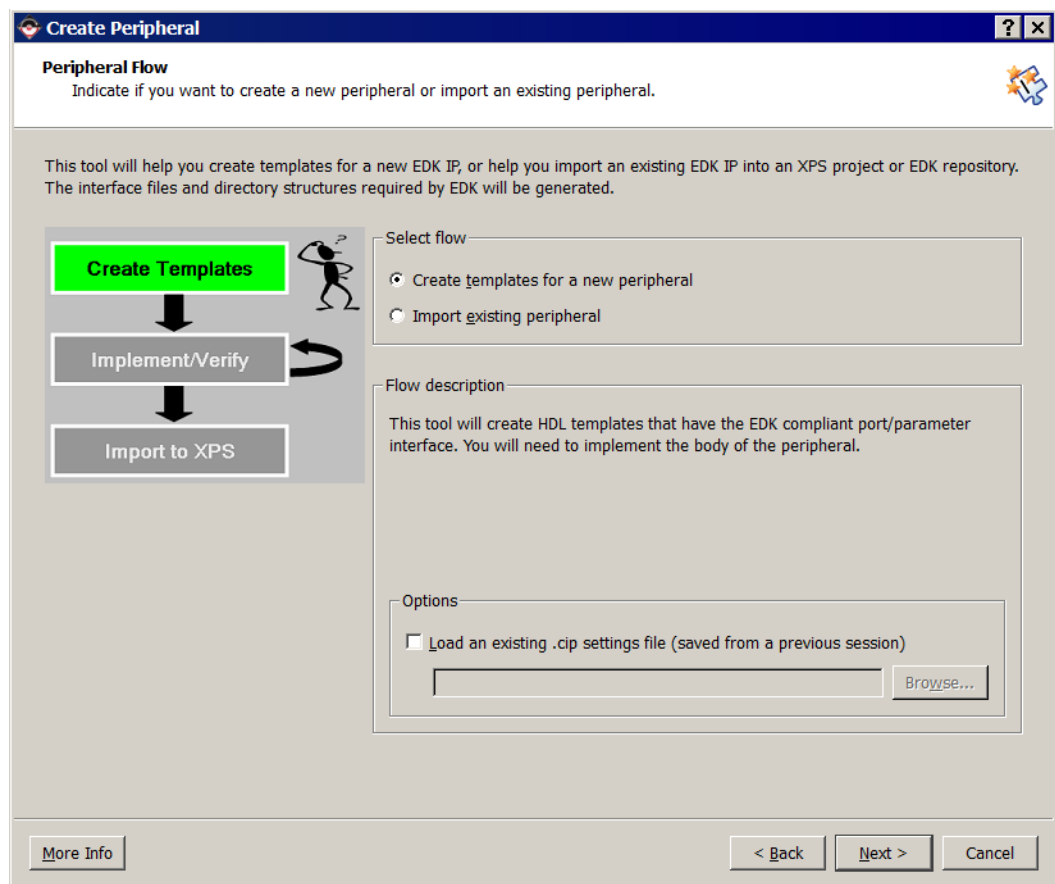

Figure 6-1: **Peripheral Flow Page**

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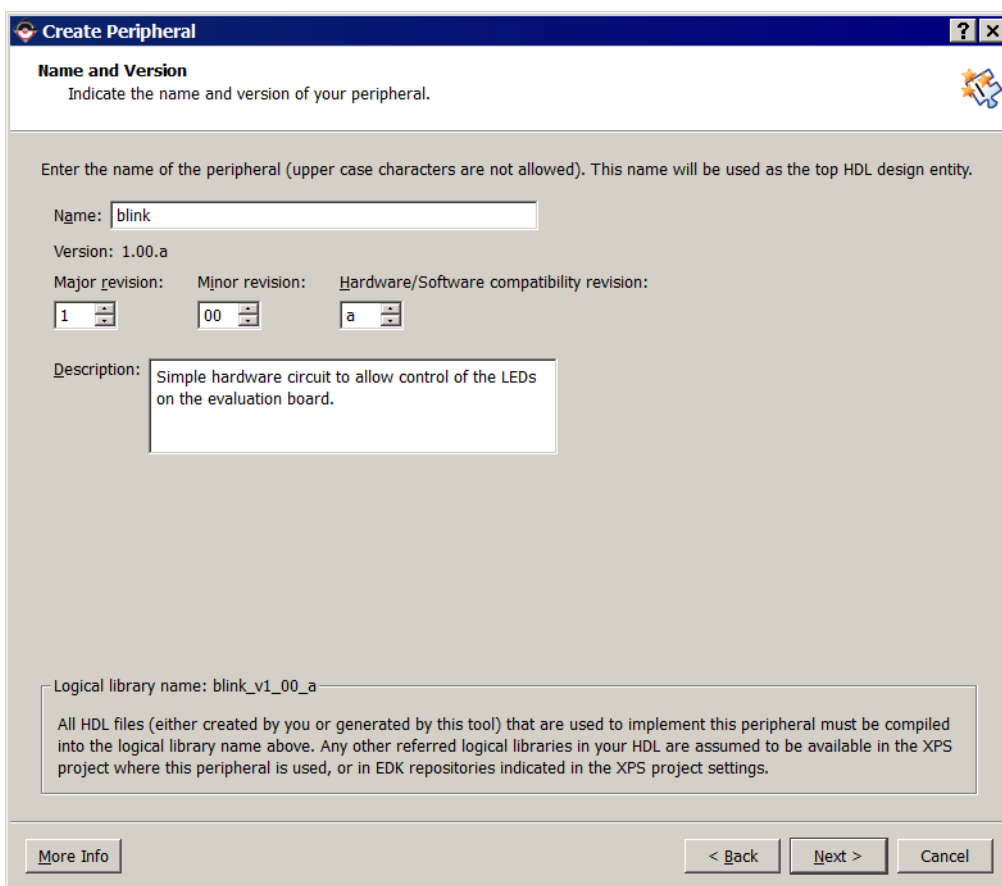
3. On the Repository or Project page, specify where to store the custom peripheral files.

   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 automatically filled in.

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

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

Figure 6-2: **Name and Version Page**

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

6. On the Bus Interface page, select the interconnection or bus type that connects your peripheral to your embedded design. For this example, select AXI4-Lite.

Note: You can access related data sheets from the Bus Interface page.
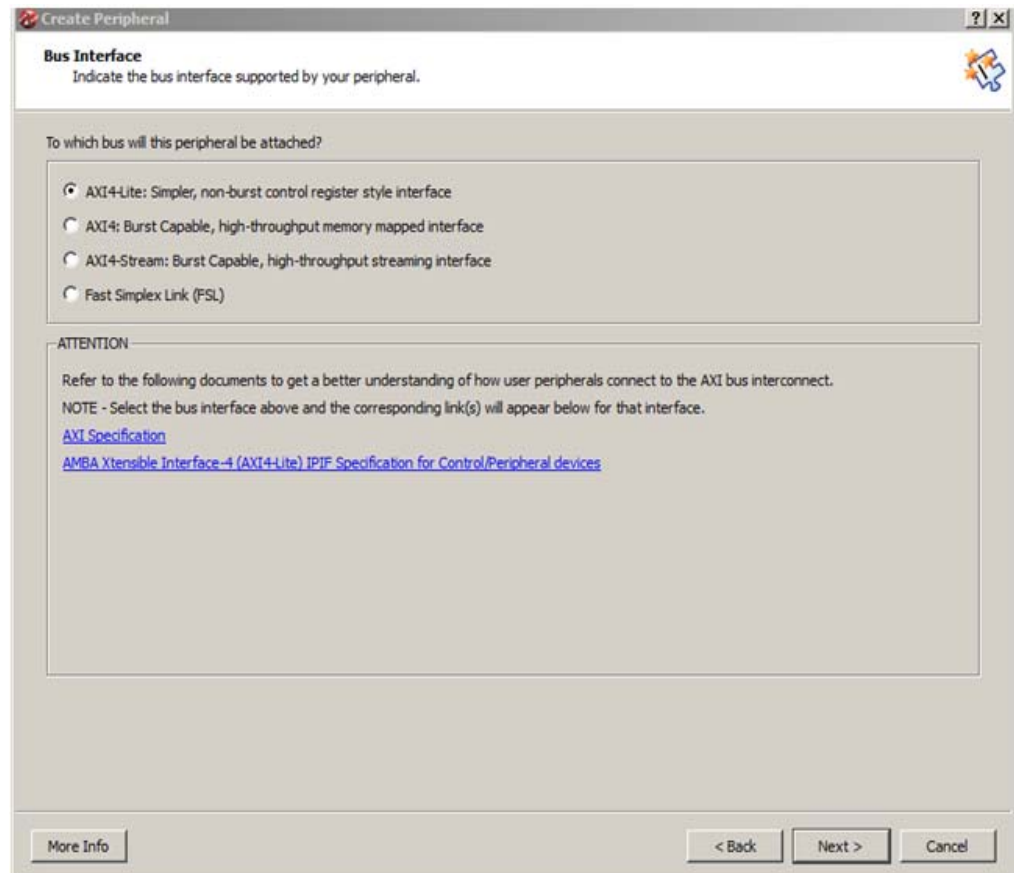
Figure 6-3: **Bus Interface Page**

7. On the IPIF (IP Interface) Services page, indicate the IPIF services for your peripheral.
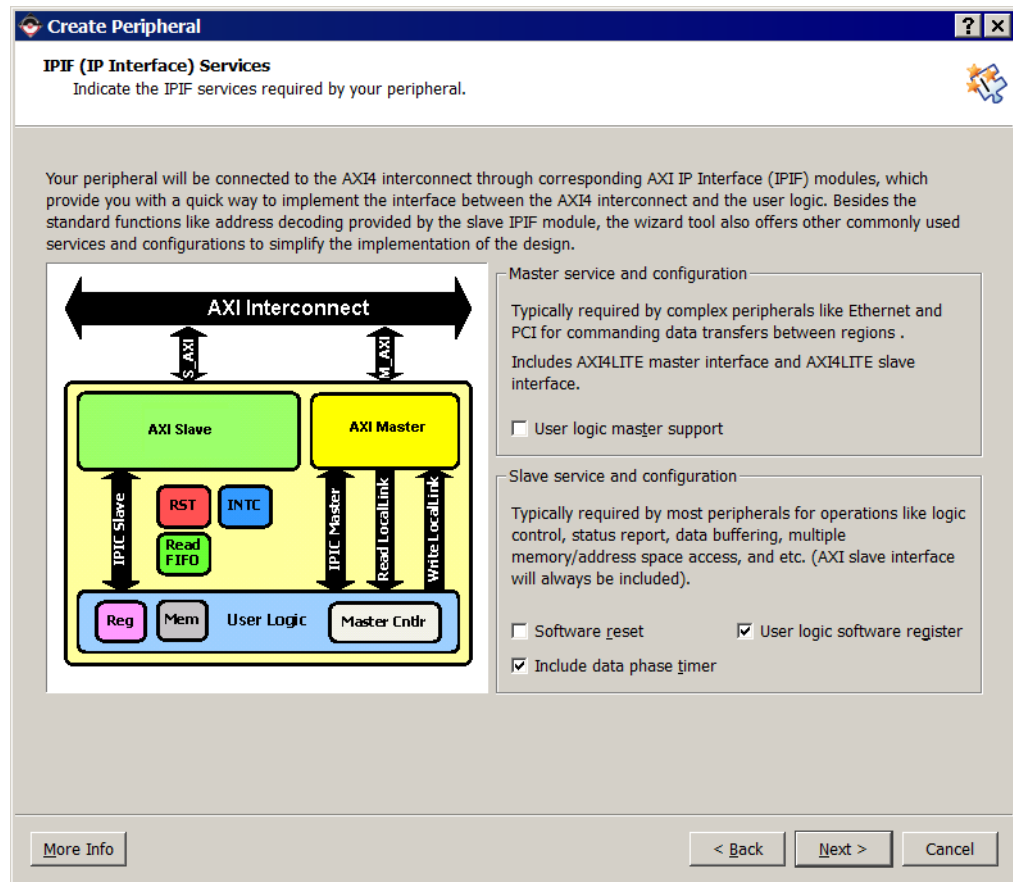
Figure 6-4: **IP Interface Services Page**

The CIP wizard automatically creates the following:

-   Slave connections to the AXI device
-   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**. You can read details on each of these services to help you determine whether the features are necessary for your IP.

Because User Logic Software Register was selected in the IPIF Services page, the User Software Accessible Registers page opens.

8.   Leave the default value of **1** selected.

Figure 6-5:   **Software Accessible Registers**

9. On the IP Interconnect (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.
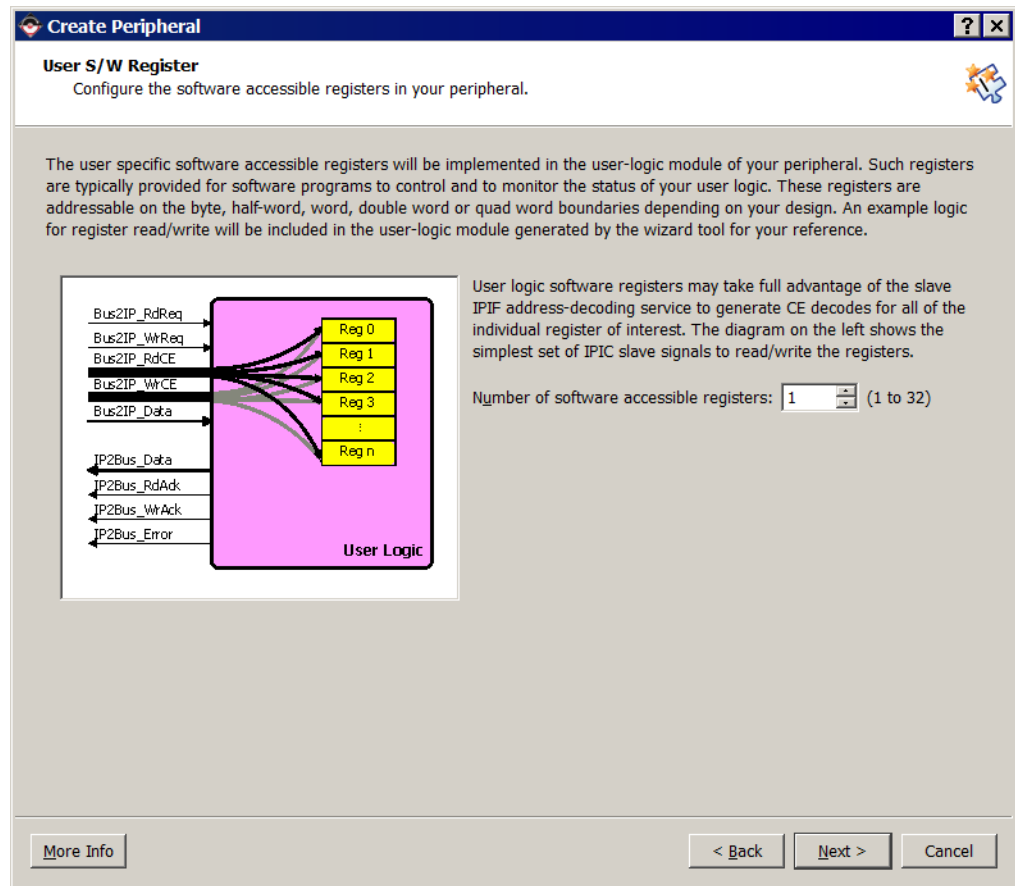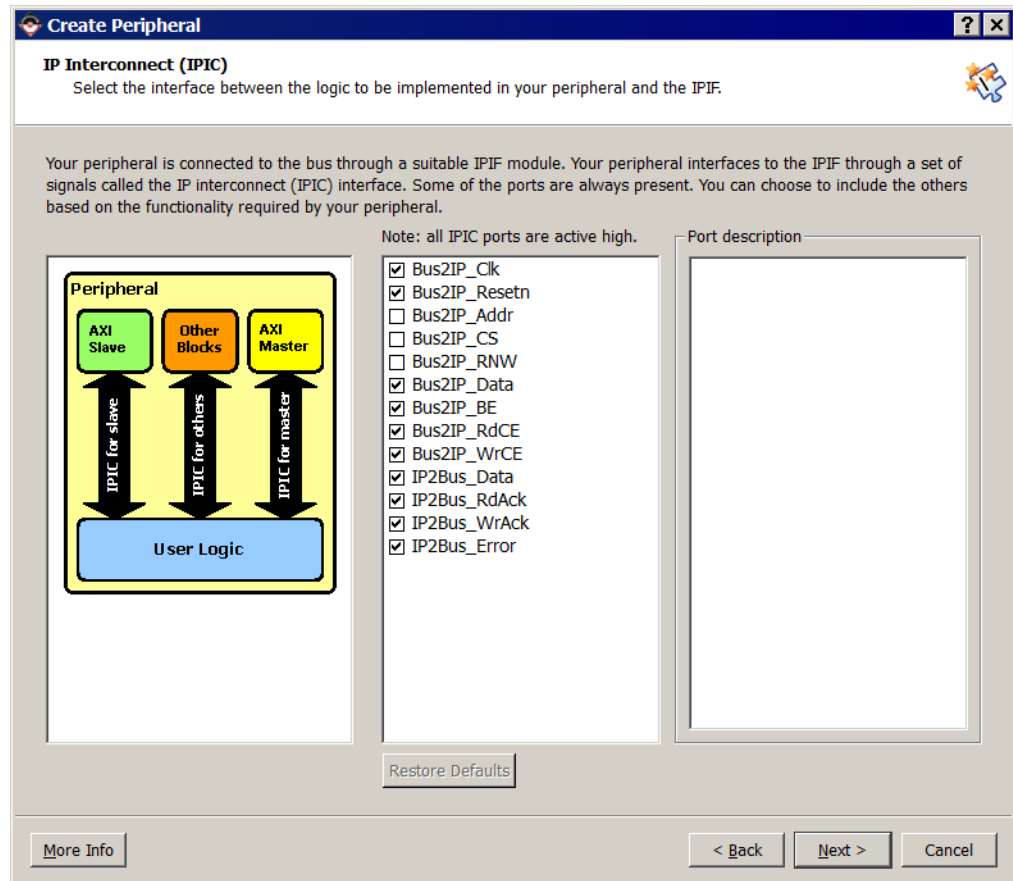

Figure 6-6: **IP Interconnect Page**

On the Peripheral Simulation Support page, you can elect to have the CIP generate a BFM simulation platform for your project.
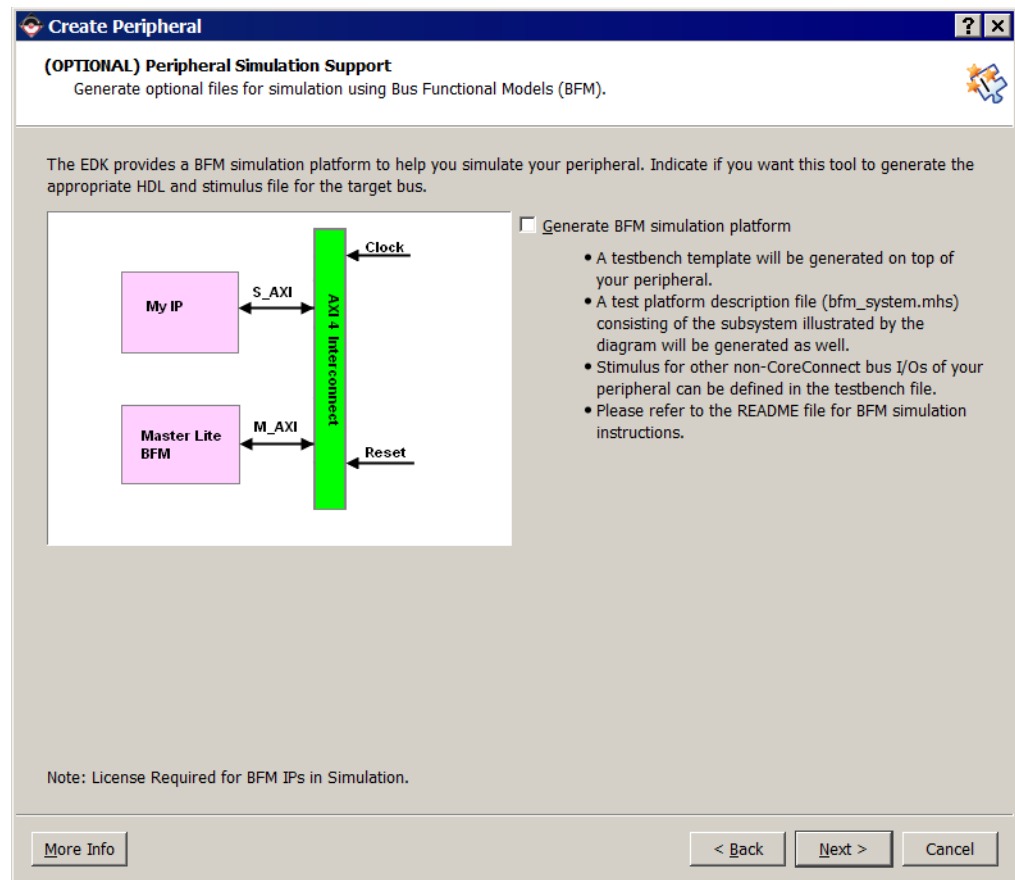
Figure 6-7: **Peripheral Simulation Support Page**

Generating BFM simulation platform

A BFM simulation requires a license for the AXI BFM Simulation model. For more information about using AXI BFMs for embedded designs with XPS, refer to refer to the *AXI Bus Functional Models v1.9* data sheet *(DS824)*. A link to this document is provided in Appendix A, "Additional Resources." AXI BFM simulation also requires a supported simulator: ISim, ModelSim-SE/PE, Questa Simulator, or IES.

If you think you might want to run a BFM simulation on this IP example, generate the BFM platform now.

Note: AXI BFM simulation must be licensed. An AXI BFM license is not included with the Design Suite installation. You can purchase the license through the Xilinx Sales Channel, then obtain it at http://www.xilinx.com/getlicense.

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

- The `blink.vhd` file - the file that contains the AXI 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 - the template file where you add the custom Register Transfer Level (RTL) that defines your peripheral. Although you can create additional source files, the simple design example you are using requires only the `user_logic.vhd` file.

The Peripheral Implementation Support page lists three options for creating optional files for hardware and software implementation.
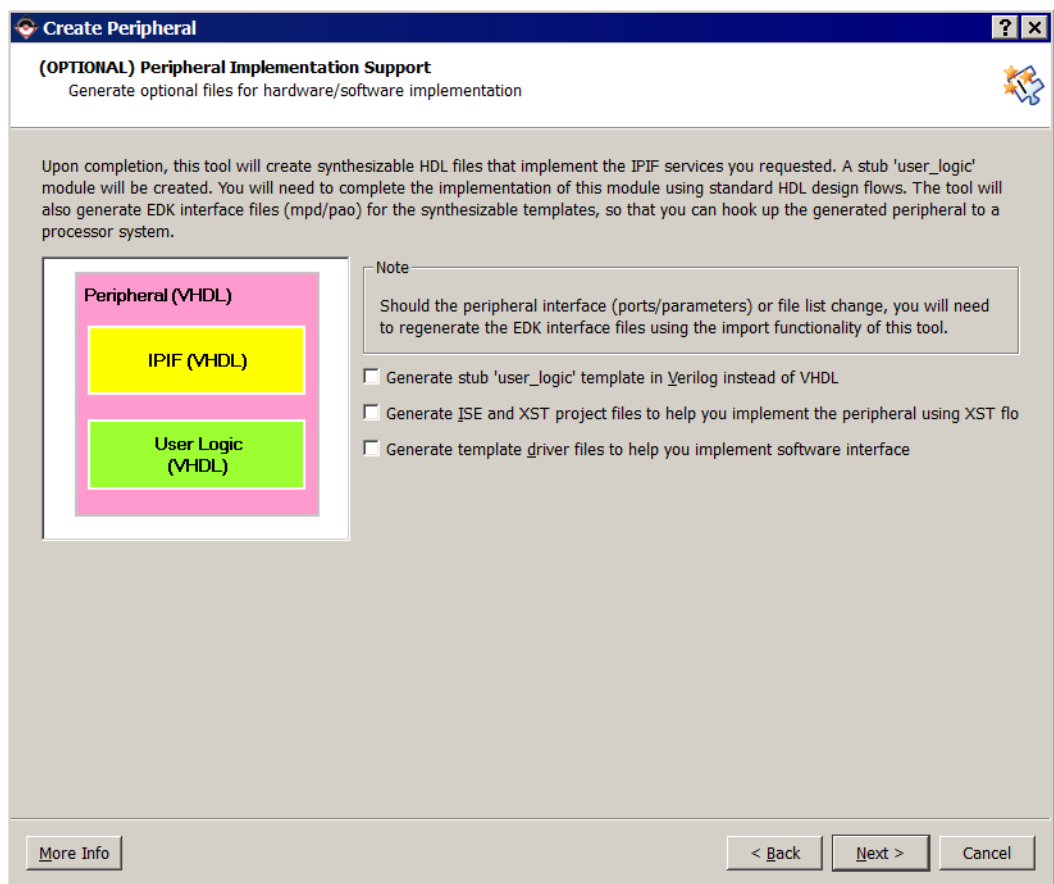
Figure 6-8: **Peripheral Implementation Support Page**

Verilog support

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 and XST project files to help you implement the peripheral using XST flow** check box. The CIP wizard creates the necessary project files for the tools. 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, including the files created and their locations.

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

Figure 6-9: **Create and Import Peripheral Wizard Summary Page**

What Just Happened?

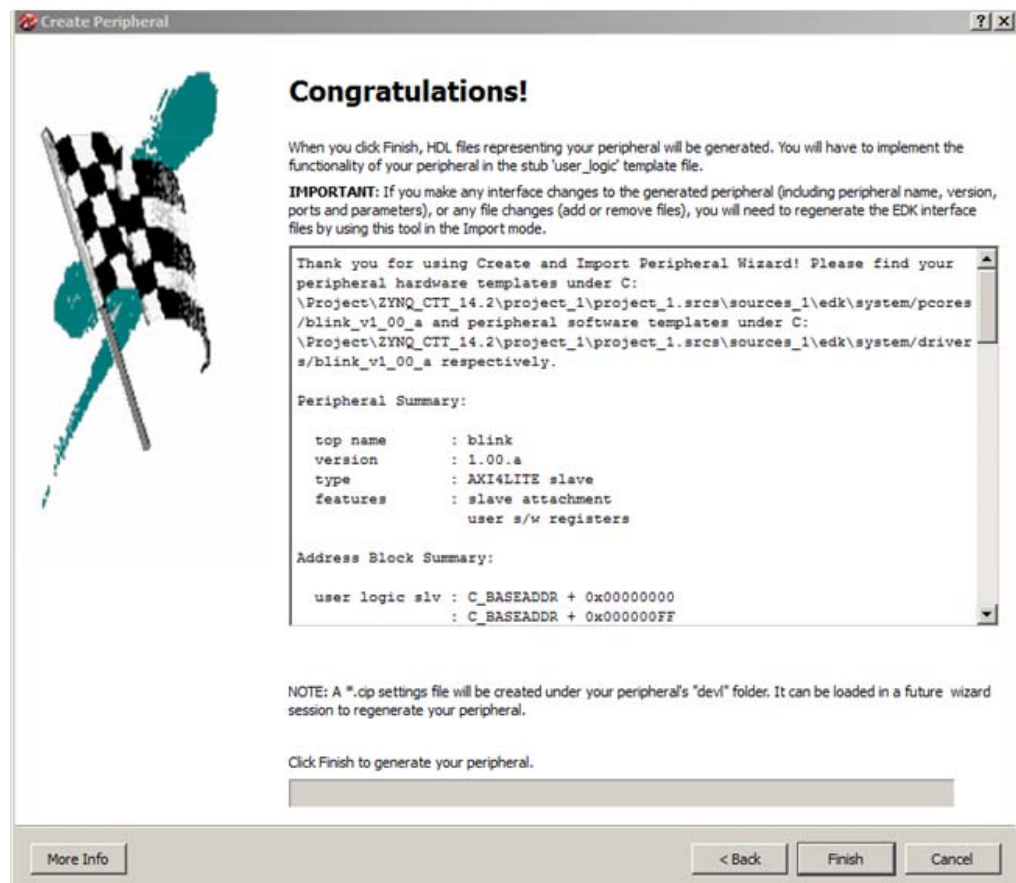*Important summary information*

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

EDK uses AXI slave and burst peripherals to implement common functionality among various processor peripherals. The AXI 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 AXI slave and burst peripheral elements your IP requires.

AXI slave and burst peripherals

The AXI 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 directly operating on the AXI or FSL protocols. Using the AXI slave and burst peripherals with parameterization that suits your needs greatly reduces your design and test effort.

Figure 6-6, page 50 illustrates the relationship between the bus, a simple AXI slave peripheral, IPIC, and your user logic.

Figure 6-10 shows the directory structure and the key files that the CIP wizard created. These file reside in the `/pcores` subdirectory of your project directory.
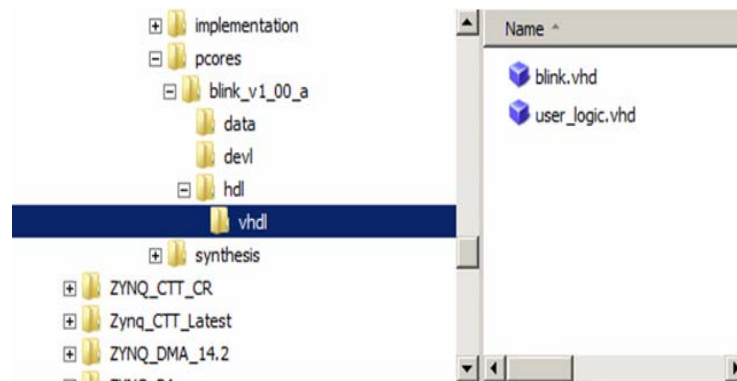
Figure 6-10: **Directory Structure Generated by the CIP Wizard**

Information about the files generated by the CIP wizard:

- The wizard created two HDL template files: `blink.vhd` and `user_logic.vhd`. These files are located in the `hdl/vhdl` folder.
- The `user_logic` file connects to the AXI device using the AXI slave core configured in `blink.vhd`.
  - The `user_logic.vhd` file is equivalent to the "Custom Functionality" block.
  - The `blink.vhd` file is equivalent to the "AXI slave" block.
- 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 AXI 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.

The custom peripheral blinks the four LEDs on the evaluation board.

Modifying the Template Files

In this section, you'll modify the template files, review the files, and then add the pcore to your project.

Take a Test Drive! Modifying the CIP Wizard Template Files

In the next Test Drive, you will modify the code generated by the CIP wizard to implement the new blink peripheral.

The peripheral is very simple. A single control register is used to enable or disable a counter. This counter divides down the bus clock and blinks the LEDs in a binary pattern.

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

2. Navigate to the `pcores\blink_v1_00_a\hdl\vhdl` directory and locate the `blink.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 `blink.vhd` file.

   In the next two steps, you'll add the external port names in two places in this file:

   - The top level entity port declaration (step 4)
   - The port map for the instantiation of the `user_logic` (step 5)

4. Scroll down to approximately line 140. In the code segment shown here, 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.

```
136    );
137    port
138    (
139      -- ADD USER PORTS BELOW THIS LINE ------------------
140      LEDs                            : out std_logic_vector (3 downto 0);
141      -- ADD USER PORTS ABOVE THIS LINE ------------------
```

Figure 6-11: **Add User Ports**

5. Scroll down to approximately line 304. In the code segment shown here, 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.

```
300       )
301       port map
302       (
303          -- MAP USER PORTS BELOW THIS LINE ------------------
304          LEDs                            => LEDs,
305          -- MAP USER PORTS ABOVE THIS LINE ------------------
```

Figure 6-12: **Add Port Mapping**

6. Save and close 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. Open and examine the `user_logic.vhd` file.

8. Scroll down to approximately line 100. In the code segment shown here, the user port LEDs are displayed in the appropriate location. Add the LEDs port declaration.

```
96    );
97    port
98    (
99       -- ADD USER PORTS BELOW THIS LINE ------------------
100      LEDs                            : out std_logic_vector (3 downto 0);
101      -- ADD USER PORTS ABOVE THIS LINE ------------------
```

Figure 6-13: **Adding the LEDs Port Declaration**

9. Scroll down to approximately line 133, and add this signal declaration.

```
130   architecture IMP of user_logic is
131
132      --USER signal declarations added here, as needed for user logic
133      signal count                           : std_logic_vector (27 downto 0);
134
135      ----------------------------------------
```

Figure 6-14: **Adding the Signal Declaration**

10. Scroll down to approximately line 214 and add the following code, beginning with **-- Create counter**. This code implements the counter logic, and connects the register bit to control counter operation.

```
205    ------------------------------------------
206    -- Example code to drive IP to Bus signals
207    ------------------------------------------
208    IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
209                    (others => '0');
210
211    IP2Bus_WrAck <= slv_write_ack;
212    IP2Bus_RdAck <= slv_read_ack;
213    IP2Bus_Error <= '0';
214
215    -- Create counter
216    -- Use slv_reg0 value to enable counter (LSB = '1' to run, LSB = '0' to stop)
217    counter:  process (Bus2IP_Clk)
218      begin
219        if (Bus2IP_Resetn = '0') then
220          count <= (others => '0');
221        elsif (Bus2IP_Clk 'event and Bus2IP_Clk = '1') then
222          if slv_reg0(0) = '1' then
223            count <= count + 1;
224          else
225            count <= count;
226          end if;
227        end if;
228      end process counter;
229
230      -- Attach slowest bits to LEDs
231      LEDs(3 downto 0) <= count(27 downto 24);
232
233  end IMP;
```

Figure 6-15: **Adding the Counter Logic**

Reviewing the File Contents

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

The `user_logic.vhd` file is similar to the top-level `blink.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 (3 downto 0) 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 internal signal count, the VHDL code that blinks the LEDs starts at line 212.

The CIP wizard created a single user register that is connected to the counter and used to control the counter. Writing a one to the least significant bit will enable the counter, and writing a zero will stop the counter.

All the code described here is simple and can be modified if you want to experiment later. However, the interface signals in lines 208-210 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 read by the processor during a read operation. For this simple peripheral, the data last written to the peripheral control register can also be read back.

The final signal, `IP2Bus_WrAck`, is 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 `slv_write_ack` 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 `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.

Take a Test Drive! Adding the Pcore to Your Project

To add the `blink` pcore to your project, you'll first update the MPD file and add the pcore. Then, you'll export the design and generate a new bitstream and test the pcore in hardware.

Adding the Pcore to Your Project

When you modified `blink.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, the MPD file must be updated to reflect these changes.

1. Open the MPD file for the blink pcore from the `pcores\blink_v1_00_a\data` directory.

2. Under the comment `##Ports`, add this line:

   `PORT LEDs = "", DIR = O, VEC = [3:0]`

3. Save the file.

4. In XPS, select **Project > Rescan User Repositories** to force XPS to recognize the changes made to the blink pcore.

   Note: Xilinx recommends that you rescan the IP repositories any time you make a change to a custom peripheral.

   Your custom pcore is now ready to add to the embedded design.

   For more information about PSF files, refer to the *Platform Specification Format Reference Manual*. A link to this document is available in Appendix A, "Additional Resources."

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

   Note: If the IP Catalog isn't visible, select **View > Tabs > IP Catalog**.

   Before adding `blink` 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 `blink` is driving these LEDs, the `LEDs_4Bit` pcore must be removed from the design.

5. In the System Assembly View, right-click `LEDs_4Bits` and select **Delete Instance**. The Delete IP Instance dialog box appears:
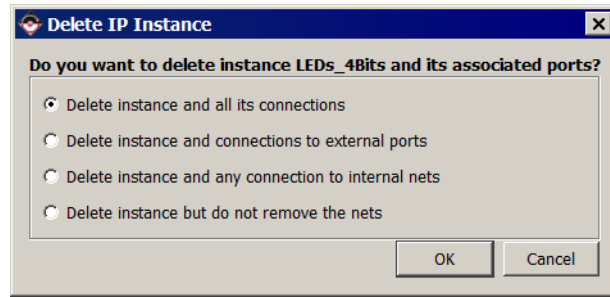

Figure 6-16: **Delete IP Instance Dialog Box**

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

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

   The XPS Core Config dialog box opens automatically.

8. Accept all defaults and click **OK** to close the dialog box.

   The Instantiate and Connect IP window opens.

9. Accept the defaults and click **OK**.

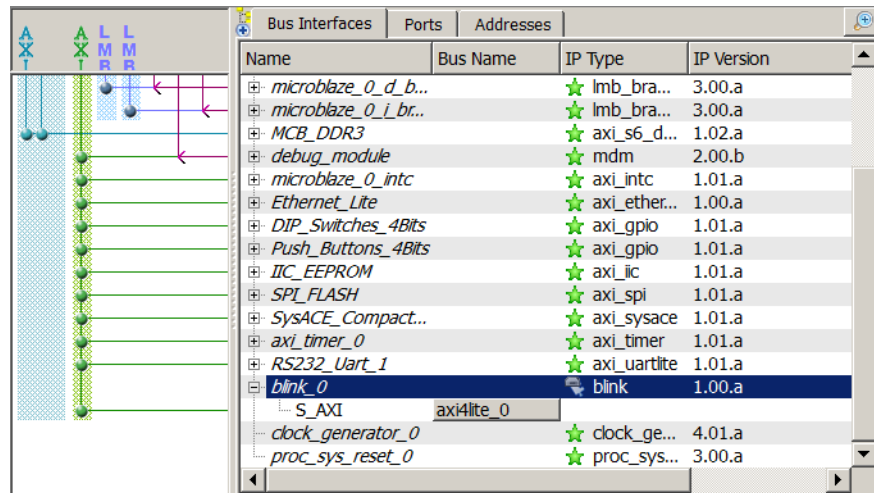   XPS adds the IP to the System Assembly View. You can see it in the Bus Interfaces tab.

Figure 6-17: **Connecting Your New IP in the Bus Interfaces Tab**

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

10. Click the Ports tab, expand `blink_0`, right-click **LEDs**, and select **Make External**.

    A default name of `blink_0_LEDs_pin` was assigned as the External Ports name.

    To change the assigned net and pin names, click in the **Connected Port** column, respectively. Alternatively, you can manually edit the MHS file. For now, don't change the assigned names.

Chapter 6: **Creating Your Own Intellectual Property**

11. Click the **Addresses** tab and change the address range for `blink_0` to `0x7C600000 – 0x7C60FFFF`.

    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 runs faster.

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

12. Click the **Project** tab and double-click the `system.ucf` file to open it in the XPS main window.

13. Look for `LEDs_4Bits_TRI_O`. These pin assignments were left in the UCF 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.

14. Replace `LEDs_4Bits_TRI_O` with `blink_0_LEDs_pin` in all four locations and save the UCF file.

    Congratulations, you have created and added a custom pcore!

15. Close XPS.

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. Go back to the tools and re-run the Generate Top Level HDL process as you did in "Take a Test Drive! Generating the Bitstream," page 25.

    It is necessary to rerun the Generate Top Level HDL process because when the blink peripheral was added, four of the top level port names were changed.

2. In the design suite, right-click the **Export Hardware Design to SDK with Bitstream** process and select **Process Properties**.

3. Click the **Launch SDK after Export** check box to check it.

4. Run the Export Hardware Design to SDK with Bitstream process.

5. When SDK launches, create a new workspace. The new hardware platform is automatically imported.

    SDK opens to the C/C++ Perspective with a table showing all the IP in your design. Confirm that `blink_0` is listed in the Address Map section of the `system.xml` file.

6. Create a new Hello World software project and BSP by selecting **File > New > Application Project**.

7. Download the bitstream to the board by selecting **Xilinx Tools > Program FPGA**.

8. Run the Hello World project to confirm that the new hardware design runs correctly.

9. Select **Xilinx Tools > XMD Console** to open an XMD console.

    We will verify the correct behavior of the blink IP by directly writing to and reading the control register.

10. At the XMD prompt, type the following:

    a. **mb**. XMD connects to the MicroBlaze processor.

    b. **stop**. The processor stops so that you can read and write to the registers.

    c. **mwr 0x7c600000 0x1**. The LEDs on the board begin to blink.

    d. **mrd 0x7c600000**. Confirm that you read back the value 0x00000001.

    e. **mwr 0x7c600000 0x0**. The LEDs stop blinking.

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.

For more information about XMD commands, refer to the "Xilinx Microprocessor Debugger" chapter of the *Embedded System Tools Reference Guide* (UG111). A link to this document is available in Appendix A, "Additional Resources."

Working with Project Navigator

Project Navigator Overview

Note: The test drives in this chapter are based on the 14.1 version of ISE® Design Suite Project Navigator. Use with later versions of the tools has not been tested and therefore might behave differently than described.

Project Navigator organizes your design files and runs processes to move the design from design entry through implementation to programming the targeted Xilinx device. Project Navigator is the high-level manager for your Xilinx® FPGA and CPLD designs, which allows you to do the following:

• Add and create design source files, which appear in the Sources window

• Modify your source files in the Workspace

• Run processes on your source files in the Processes window

• View output from the processes in the Transcript window

Note: Optionally, you can run processes from a script you create or from a command line prompt. However, it is recommended that you first become familiar with the basic use of the Xilinx Integrated Software Environment (ISE®) tools and with project management, as described in the following sections.

Take a Test Drive! Creating a New Embedded Project

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

1. Start the ISE Project Navigator software.

2. Select **File > New Project**.

3. In the New Project wizard, 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.
	Location	Select a location in which to save your project.
	Working Directory	Select a working directory for your project. This defaults to the same location you selected in the previous field.
	Description	Type a description for your project (optional).
	Top-level source type	Keep the default setting, **HDL**.
Project Settings	Evaluation Development Board	Select Kintex™-7 KC705 Evaluation Platform. All the other fields are populated for you based on your board choice. Keep these settings.

4. When you click **Finish**, the New Project wizard closes and a blank KC705 project opens.

   Next you must add an embedded project using the New Source wizard.

5. Select **Project > New Source**.

6. In the New Source wizard, 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	File name	Name the source file. For example, `system`. **Caution!** Do not use spaces in the file name or path.
	Location	Select a location in which to save your source. By default, your project directory is selected.
	Add to Project	Leave this option selected.
Summary		Review the source summary details.

7. Click **Finish** to create the embedded source.

   A dialog box opens, asking if you want to create a base system using the BSB wizard.

8. Click **Yes**.

   The first window of the Base System Builder asks you to decide whether to create an AXI-based or PLB-based system.

9. Select **AXI System** and click **OK**.

10. In the Base System Builder wizard, create a project using the settings in the following table. If no setting or command is indicated in the table, accept the default values.

Wizard Screens	System Property	Setting or Command to Use
Board and System Selection	Board	Use the default option to create a system for the Kintex-7 KC705 Platform. ***Note:*** This is pre-populated because you selected this board in Project Navigator.
	Board Configuration	This information is pre-populated based on your board selection. ***Note:*** If you had selected to create a system for a custom board, these fields would be editable.
	Select a System	**Single MicroBlaze Processor System**
	Optimization Strategy	**Area**
Processor, Cache, and Peripheral Configuration	Processor Frequency	**100 MHz** (default)
	Select a Processor	microblaze_0
	Enable Floating Point Unit	Do not enable this setting.
	Local Memory Size	**64 KB**
	Instruction Cache Size	**8 KB**
	Data Cache Size	**8 KB**
	Select and Configure Peripherals	Add the `axi_timer` peripheral and select the **Use Interrupt** check box.

11. To generate your design, click **Finish**.

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

Generating the Netlist

Select **Hardware > Generate Netlist**.This action invokes the platform building tool, Platgen, which does the following:

- Reads the design platform configuration MHS file and runs all necessary design rule checks to validate the correctness of the design.

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

- Produces netlist files (with an .ngc extension) for each peripheral, as well as the overall embedded system.

- Generates Hardware Description Language (HDL) wrapper files for each peripheral and the overall system. To see the created HDL files, look in the `<project_name>\system\hdl` directory.

More information about Platgen is provided in the "Platform Generator (Platgen)" chapter of the *Embedded System Tools Reference Manual*. A link to this document is available in Appendix A, "Additional Resources."

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

Take a Test Drive! Generating the Bitstream

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

Implementing the design using Project Navigator

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

1. Close XPS.

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. Before you can do that, you need to add some information so that the Place and Route (PAR) tool has information about your design, such as the pinout.

A Project Navigator project has one top-level module that is the root of the design hierarchy for the purpose of implementation. When you create a new project, the highest level module is automatically assigned as the top module.

An XMP file cannot be a top-level module. If an XMP file is the only source in your project, then you must generate a VHDL or Verilog wrapper file.

Generating a top HDL source

2. In Project Navigator, make sure your `<project>.xmp` file is selected in the Source pane.

3. Right-click the .xmp file and select **Create Top HDL Source** to generate the wrapper file for the `<project>.xmp` file. This is now the top module for your project.
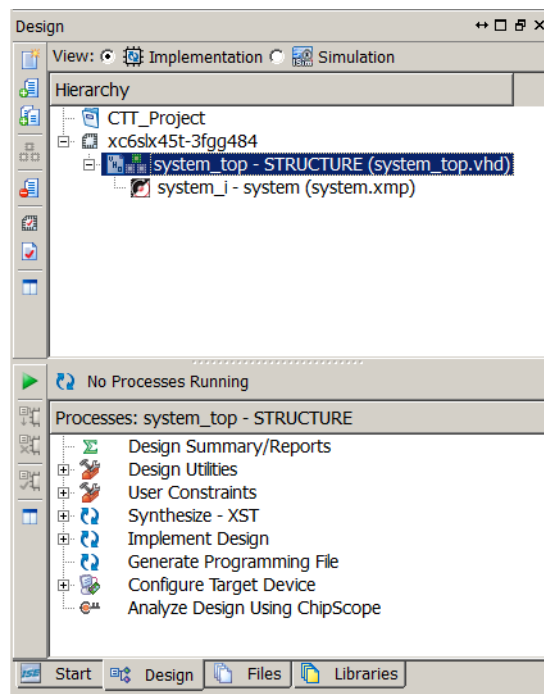

Figure 7-1: **Viewing the Top HDL Wrapper File**

4. With the system_top file selected, double-click **Generate Programming File** in the Processes pane to create your bitstream. It takes a few minutes and concludes with the message:

```
Programming File" completed successfully
```

Generated bitstream files

The generated bitstream file is located in the `\implementation` folder of your project and is called `system.bit`. There is another file generated called `system_bd.bmm`, which SDK uses for loading memory onto your target board.

It is not necessary to add the constraint file (`.ucf`) generated by XPS to your project. Project Navigator automatically locates and uses the constraints during implementation.

Exporting Your Hardware Platform

You created your project in Project Navigator and added an embedded processor source, then designed your hardware platform in XPS using the Base System Builder, and finally generated a bitstream for the FPGA. Now, you will export your hardware platform description to the Software Development Kit (SDK).

The exported 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

You can export your hardware platform from XPS or from Project Navigator. In this tutorial, you'll use Project Navigator to export your hardware platform, but you'll modify the process properties so that SDK doesn't open. You'll work with SDK in the next chapter.

1. In Project Navigator, expand `system_top` and select your `system.xmp` file.

2. In the Processes pane, right-click **Export Hardware Design to SDK with Bitstream** and select **Process Properties**.

3. Uncheck the **Launch SDK after Export** option and click **OK**.

4. Double-click **Export Hardware Design to SDK with Bitstream**.

What Just Happened?

Project Navigator exported your hardware design to SDK. It is important to understand the export operation, especially if you are managing multiple hardware versions.

When you export your hardware design to SDK, 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.

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.

Notice that the Launch SDK after Export option was selected by default. When this is selected, SDK launches after Project Navigator exports the design. When SDK launches this way, it automatically imports the hardware platform for your design.

Creating a New Project Using the Vivado Design Suite

Vivado Design Suite Overview

The Vivado™ Integrated Design Environment (IDE) provides an easy-to-use design flow that can be run using a graphical user interface (GUI) or using Tool Command Language (Tcl) commands and scripts.

The design flow includes the following high-level steps:

1. RTL synthesis
2. Netlist linking and constraint application
3. Placement, packing, and optimization
4. Routing and optimization
5. Static timing analysis
6. Bitstream generation
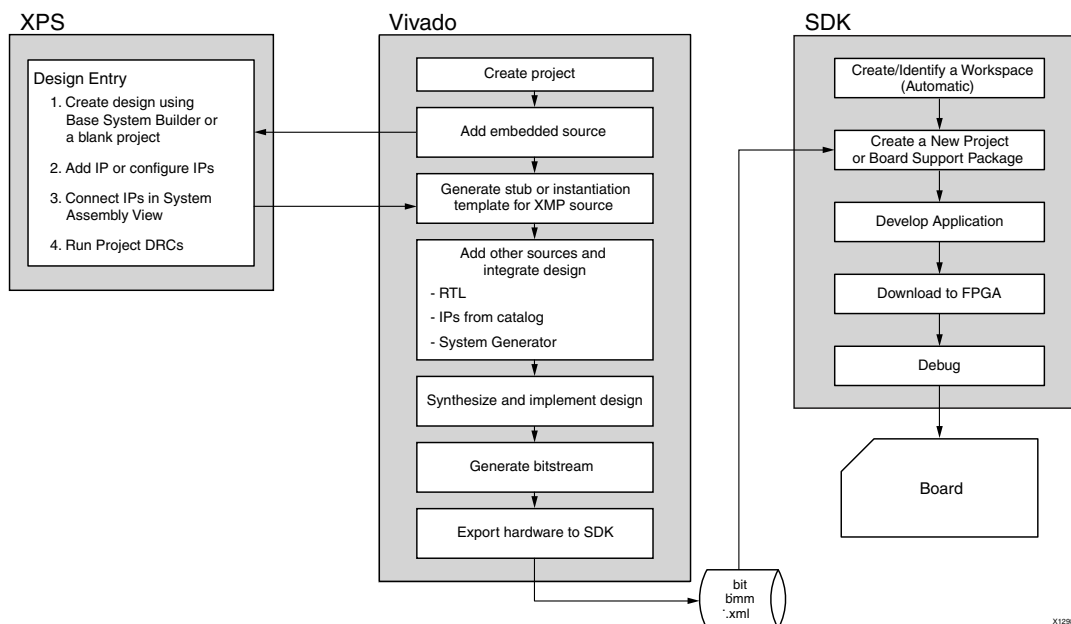7. Exporting Hardware details to SDK

Figure 8-1: **Basic Vivado Embedded Design Flow**

In this chapter, you will create a project with an embedded processor system using the Vivado™ Design Suite.

You will use Base System Builder (BSB) wizard to create a MicroBlaze™ processor based default system for the Kintex™-7-based KC705 Board.

Note: For information about the BSB wizard, refer to Chapter 2.

The following Test Drive walks you through starting your new project in the Vivado™ Design Suite and using the New Project wizard to create your project.

Take a Test Drive! Creating a New Embedded Project

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

1. Start the Vivado design tool.

2. In the Welcome window, select **Create New Project**.

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
Project Name	Project name	Choose a name and location for your project.
	Create Project Subdirectory	Leave this option checked.
Project Type	Specify the type of project to create.	**RTL Project**
Add Sources	This is optional; no changes needed for this example.	
Add Existing IP	Optional; no changes needed.	
Add Constraints	Optional; no changes needed.	
Default Part	Specify	Select **Boards**.
	Board list	Select **Kintex-7 KC705 Evaluation Platform**.
New Project Summary	Review the project settings.	

When you click **Finish**, the New Project Wizard closes and a blank KC705 Vivado design tool project opens.

4. Verify the FPGA part, target language, and simulator settings by clicking **Project Settings** in the Flow Navigator panel.

5. To add an EDK project, click **Add Sources** in the Flow Navigator panel.

Note: You can also add a source using the shortcut **Alt + A**.

The Add Sources wizard opens.

6. Click **Add or Create Embedded Sources** and click **Next**.

7. Click **Create Sub-Design**.

8. In the dialog box that opens, type `system` as a name for the module and click **OK**.

9. Click **Finish**.

The Vivado design tool creates your embedded design source project. It recognizes that you have an embedded processor system and starts Xilinx® Platform Studio (XPS).

Next, you'll use the BSB wizard to create a Base System in XPS.

1. A dialog box opens with the following message:

   ```
   This project appears to be a blank project. Do you want to create
   a Base system using the BSB wizard?
   ```

2. Click **Yes**.

   The first window of the BSB asks you to select an Interconnect Type. By default, an AXI System is selected.

   Note: PLB system is disabled in the Wizard.

3. Click **OK**.

4. In the Base System Builder wizard, create a project using the settings described in the following table. If no setting or command is indicated in the table, accept the default values.

Wizard Screens	System Property	Setting or Command to Use
Board and System Selection	Board	Use the default option to create a system for the Kintex-7 KC705 Evaluation Platform. **Note:** This is pre-populated because you selected this board in the Vivado tool.
	Board Configuration	This information is pre-populated based on your board selection. **Note:** If you had selected to create a system for a custom board, these fields would be editable.
	Select a System	**Single MicroBlaze Processor System**
	Optimization Strategy	**Area**
Processor, Cache, and Peripheral Configuration	Processor Frequency	**100 MHz** (default)
	Select a Processor	microblaze_0
	Enable Floating Point Unit	Do not enable this setting.
	Local Memory Size	**64 KB**
	Instruction Cache Size	**64 KB**
	Data Cache Size	**64 KB**
	Select and Configure Peripherals	Add the `axi_timer` peripheral and select the **Use Interrupt** check box.

5. To generate your design, click **Finish**.

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

   Note: For more information about XPS, refer to Chapter 3, "Using Xilinx Platform Studio."

6. Close XPS.

Next, you'll run through the design to the point at which a bitstream is generated.

First, you need to provide some information about your design, such as the pinout, to the Vivado design tool.

A project in the Vivado Design tool has one top-level module that is the root of the design hierarchy for the purpose of implementation. When you create a new project, the highest level module is automatically assigned as the top module.

An XMP file cannot be a top-level module. If an XMP file is the only source in your project, then you must generate a VHDL or Verilog wrapper file.

1. In the Vivado Design tool, make sure your system.xmp file is selected in the Source pane.

2. Right-click the `system.xmp` file and select **Create Top HDL Source** to generate the wrapper file for the `system.xmp` file. This is now the top module for your project.
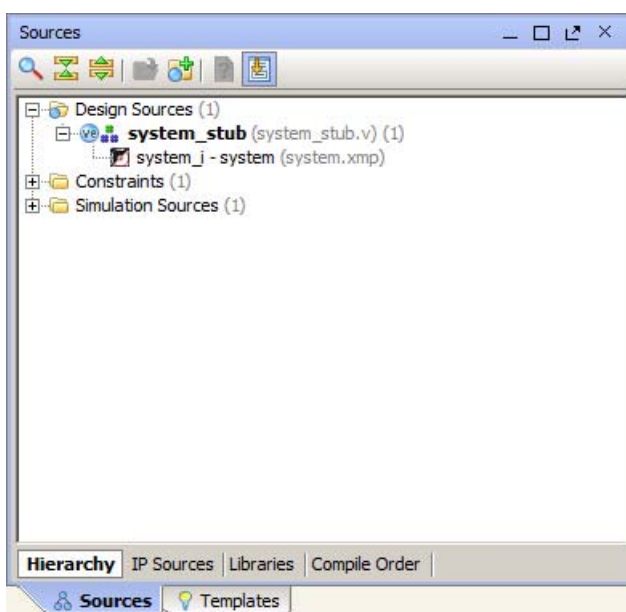
Figure 8-2: **Viewing the New Verilog Wrapper File**

3. In the Flow Navigator pane, expand **Program and Debug** and then click **Generate Bitstream**.

   If a critical warning appears, you can safely ignore it.

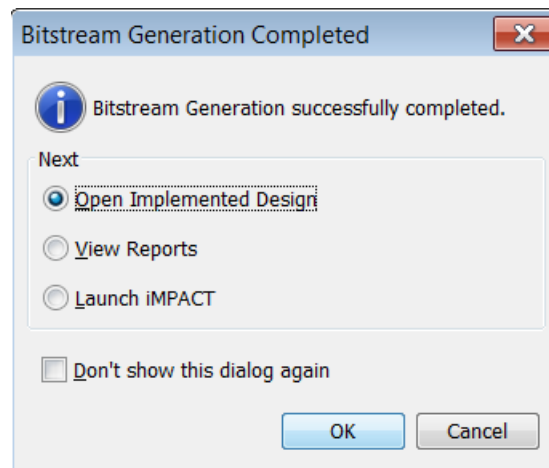Bitstream generation takes some time and concludes with below message window:

Figure 8-3: **Bitstream Generation Completed Dialog Box**

4. Select View Reports and click **OK**.

   It shows reports related to Synthesis, Place and Route, and Bitstream.

Generated bitstream files

The generated bitstream file is located in the `<Project_1>\<Project_1.runs>\impl_1 folder` of your project and is called `system_stub.bit`. There is another file generated called `system_bd.bmm`, which SDK uses for loading memory onto your target board.

It is not necessary to add the Xilinx Design Constraint file (`.xdc`) file that was generated by XPS to your project. The Vivado design tool automatically locates and uses the constraints during implementation.

Exporting Your Hardware Platform

You created your project in the Vivado Design tool and added an embedded processor source, then designed your hardware platform in XPS using the Base System Builder, and finally generated a bitstream for the FPGA. Now, you will export your hardware platform description to the Software Development Kit (SDK).

The exported XML file has the information that 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

You can export your hardware platform from XPS or from the Vivado design environment. In this tutorial, you'll use the Vivado tool to export your hardware platform.

1. In the Vivado tool, select **File > Export > Export Hardware**.

2. In the Export Hardware dialog box that opens, select the Include Bitstream and Export Hardware options.

3. Click **OK**.

What Just Happened?

The Vivado design tool exported your hardware design to SDK. It is important to understand the export operation, especially if you are managing multiple hardware versions.

When you export your hardware design to SDK, 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.

What's Next?

Now you can start developing the software for your project using SDK. Go to Chapter 5, "Software Development Kit," which explains embedded software design fundamentals.

Additional Resources

Xilinx Resources

- **Xilinx® Device User Guides**:
  http://www.xilinx.com/support/documentation/user_guides.htm

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

- *Xilinx Design Tools: Installation and Licensing Guide (UG798)*:
  http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/iil.pdf

- *Xilinx Design Tools: Release Notes Guide (UG631)*:
  http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/irn.pdf

- **Product Support and Documentation:** http://www.xilinx.com/support

EDK Documentation

You can access the entire documentation set online at:
http://www.xilinx.com/support/documentation/dt_edk_edk14-3.htm

Individual documents are linked below.

- *EDK Profiling Guide (UG448)*:
  http://www.xilinx.com/support/documentation/xilinx14_3/edk_prof.pdf

- *Embedded System Tools Reference Manual (UG111)*:
  http://www.xilinx.com/support/documentation/xilinx14_3/est_rm.pdf

- **MicroBlaze™ Processor User Guide** (UG081):
  http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/mb_ref_guide.pdf

- *Platform Specification Format Reference Manual (UG642)*:
  http://www.xilinx.com/support/documentation/xilinx14_3/psf_rm.pdf

- *AXI Bus Functional Models v1.9 Data Sheet (DS824)*:
  http://www.xilinx.com/support/documentation/ip_documentation/cdn_axi_bfm/v1_9/ds824_axi_bfm.pdf

EDK Additional Resources

- **EDK Tutorials website**:
  http://www.xilinx.com/support/documentation/dt_edk_edk14-3_tutorials.htm

- **Platform Studio and EDK website**:
  http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm

- **XPS/EDK Supported IP website**:
  http://www.xilinx.com/ise/embedded/edk_ip.htm