EDK Concepts, Tools, and Techniques

A Hands-On Guide to Effective Embedded System Design

UG683 EDK 11
Xilinx is disclosing this user guide, manual, release note, and/or specification (the “Documentation”) to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU “AS-IS” WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

© 2009 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/07</td>
<td>9.1i</td>
<td>Book release for EDK 9.1i.</td>
</tr>
<tr>
<td>09/05/07</td>
<td>9.2i</td>
<td>Book release for EDK 9.2i.</td>
</tr>
<tr>
<td>11/05/07</td>
<td>10.1</td>
<td>Book release for ISE Unified 10.1.</td>
</tr>
<tr>
<td>09/18/08</td>
<td>10.1</td>
<td>Book release for ISE v10.1 SP3.</td>
</tr>
<tr>
<td>05/11/09</td>
<td>11</td>
<td>Book release for ISE Design Suite 11.</td>
</tr>
<tr>
<td>06/24/09</td>
<td>11.2</td>
<td>Book release for ISE Design Suite 11.2.</td>
</tr>
<tr>
<td>12/02/09</td>
<td>11.4</td>
<td>Book release for ISE Design Suite 11.4.</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

About This Guide .................................................. 5
Additional Documentation ..................................... 5
Attachments to this Guide ...................................... 6
How EDK Simplifies Embedded Processor Design .......... 6
The Integrated Design Suite, Embedded Edition ............ 6
The Embedded Development Kit (EDK) ....................... 6
How the EDK Tools Expedite the Design Process .......... 7
What You Need to Set Up Before Starting .................. 8

Chapter 2: Creating a New Project

The Base System Builder ....................................... 11
Why Use the BSB? ................................................. 11
What You Can Do in the BSB Wizard ......................... 11
The BSB Wizard and the ISE Design Suite ................... 13
A Note on the BSB and Custom Boards ...................... 16
What’s Next? ...................................................... 16

Chapter 3: Using Xilinx Platform Studio

What is XPS? ...................................................... 17
The XPS Software ............................................... 17
Project Information Area ....................................... 19
System Assembly View ........................................ 21
Console Window ................................................ 23
Start Up Page .................................................... 23
XPS Tools .......................................................... 23
XPS Directory Structure ...................................... 24
Directory View ................................................... 25
What’s Next? ...................................................... 26

Chapter 4: Working with Your Embedded Platform

What’s in a Hardware Platform? ............................. 27
Hardware Platform Development in Xilinx Platform Studio 27
The Hardware Platform in System Assembly View ........ 28
Generating the Hardware Platform .......................... 28
Exporting Your Hardware Platform ......................... 28
What’s Next? ...................................................... 31

Chapter 5: Introducing the Software Development Kit

About SDK ...................................................... 33
What Just Happened? .......................................... 36
What’s Next? ...................................................... 40

Chapter 6: Editing, Debugging, and Releasing in SDK

More on Drivers ................................................ 41
Chapter 7: Creating Your Own Intellectual Property

Using the CIP Wizard ................................................................. 53
Overview of IP Creation ......................................................... 54
Using the CIP Wizard for Creating Custom IP ......................... 54
What You Need to Know Before Running the CIP Wizard ............. 55
Example Design Description .................................................. 63
Reviewing the File Contents ................................................... 66
Adding Your Custom IP to Your Processor System ...................... 67
What’s Next? ........................................................................... 76

Chapter 8: Dual Processor Design and Debug

Using the BSB to Create a Dual-Processor Design ....................... 77

Appendix A: Simulating in Project Navigator with ModelSim

Introducing Simulation ............................................................. 85
Examining the Output .............................................................. 88
What Just Happened? .............................................................. 88

Appendix B: Intellectual Property Bus Functional Model Simulation

What are BFMs and Why Should I Use Them? ............................ 89

Appendix C: Glossary
Chapter 1

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 document 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 based on Windows operating system behavior. Linux behavior or the graphical user interface (GUI) display might vary slightly.

Take a Test Drive!

Because the best way to learn a software tool is to use it, this document 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 we need to do it. We will also cover what happens when you run automated functions.

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

Additional Documentation

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

Attachments to this Guide

Some examples in this guide require you to access example project files. These files are included in the Zip file for this guide.

The following files are attached:

- bus_transaction_bfl_code.txt
- leds.c
- pn.do file
- pwm_light.vhd
- user_logic.vhd

How EDK Simplifies Embedded Processor Design

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

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

The Integrated Design Suite, Embedded Edition

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

- Integrated Software Environment (ISE)
- PlanAhead design analysis tool, ChipScope™ Pro (which is useful for debugging FPGA designs)
- Embedded Development Kit (EDK)

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

The Embedded Development Kit (EDK)

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)

To run XPS, ISE must be installed as well. Think of ISE as an umbrella that covers all things related to embedded processor systems and their design.

XPS is the development environment used for designing the hardware portion of your embedded processor system. You can run XPS using a bash shell command line, in batch mode, or using the GUI, which is what we will be demonstrating in this guide.
Software Development Kit (SDK)

SDK is an integrated development environment, complimentary to XPS, that is used for C/C++ embedded software application creation and verification. SDK is built on the Eclipse™ open-source framework, so this software development tool might appear familiar to you or members of your design team.

Other EDK Components

EDK includes other elements such as:

- 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

The utilities provided with EDK are designed to assist in all phases of the embedded design process.

How the EDK Tools Expedite the Design Process

The diagram below shows the simplified flow for an embedded design.
Typically, the ISE development software is used first to create an “Embedded Processor” source, which is then added to the ISE project.

- You use XPS primarily for embedded processor hardware system development. Specification of the microprocessor, peripherals, and the interconnection of these components, along with their respective property assignments, takes place in XPS.
- You use SDK for software development. Starting with this release of the Embedded Edition of the ISE Design Suite, SDK is also available as a standalone application, which means it is a separate executable and can be run without any other tools or application suite.
- You can verify the correct functionality of your hardware platform by running the design through a Hardware Description Language (HDL) simulator. XPS facilitates three types of simulation:
  - Behavioral
  - Structural
  - Timing-accurate
  XPS sets up the verification process structure automatically, including HDL files for simulation. You only have to enter clock timing and reset stimulus information, along with any application code.
- After completing your design in XPS, you return to ISE to generate the FPGA configuration file used to program the target device.
- Once your FPGA is configured with the bitstream containing the embedded design, you can download and debug the Executable and Linkable Format (ELF) file from your software project from within SDK.


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 have set up match those you need to follow the “Test Drive” sections in this document.

Installation Requirements: What You Need to Run EDK Tools

ISE and EDK

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

Bash Shell for Linux

When you use EDK on a Linux platform, you need a bash shell. Also, be sure to check out the supported platforms covered in the Xilinx release notes: ISE Design Suite 11: Installation, Licensing, and Release Notes.
Software Licensing

Beginning with the ISE 11.1 release, Xilinx software now 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 them. If you are just trying out the software, you can also generate an evaluation license.

For more information about licensing Xilinx software, refer to the ISE Design Suite 11: Installation, Licensing, and Release Notes.

Simulation Installation Requirements

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

1. A Secure-IP capable mixed language simulator (ModelSim PE/SE v6.4b or NCSim 8.1-s009 or later) must be installed for the simulation steps. MXE is not supported for embedded designs; it doesn't have mixed language or SecureIP support.

   Optionally, install the CoreConnect™ toolkit. The CoreConnect toolkit is only required if you intend to perform Bus Functional Model (BFM) Simulation. If you do not intend to run BFM simulations, you may skip installation of the CoreConnect Toolkit.

   CoreConnect is a free utility provided by IBM. You can download CoreConnect from the Xilinx website at:

   After you make the appropriate selections on the web page to order and register, you will have access to the download.

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

   a. To open the help from XPS, select Help > Help Topics. Alternately, the XPS Help is available on the web at

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

   For additional details on the installation process refer to the ISE Design Suite 11: Installation, Licensing, and Release Notes.
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. It might be all you need to create your design, but if you require more customization, the BSB saves you a lot of time because it automates basic hardware and software platform configuration tasks common to most processor designs. After running the wizard, you have a working project that contains all the basic elements needed to build a more customized or complex system, should that be necessary.

What You Can Do in the BSB Wizard

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

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

File creation includes the option to apply settings from another project you have created with the BSB.

Selecting a Board Type

The BSB provides the ability to select a board type from a list or to create a custom board.
Supported Boards
If you are targeting one of the supported embedded processor development boards available from Xilinx or from one of our partners, the BSB lets you choose from the peripherals available on that board, automatically match the FPGA pinout to the board, and create a completed platform and test application that is ready to download and run on the board. Each option has functional default values that are pre-selected in Xilinx Platform Studio (XPS). This base-level project can be further enhanced in XPS or implemented with utilities provided by ISE®.

Upon initial installation of EDK, only Xilinx board files are installed. If you want to target a third party board, you must add the necessary board support files. The BSB Board Selection screen contains a link that helps you find third party board support files. After the files are installed, the BSB drop-down menus display those boards as well.

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

Selecting and Configuring a Processor
You can choose a MicroBlaze or PowerPC processor and select:

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

Selecting and Configuring Multiple I/O Interfaces
The BSB wizard understands the external memory and I/O devices available on your predefined board and allows you to select the following, as appropriate to a given device:

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

For your convenience, data sheets for external memory and I/O devices can be opened from within the BSB wizard.
Adding Internal Peripherals

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

Setting Up Software

Standard input and output devices can be specified in the BSB, and sample C applications are generated. The sample C applications generated by the BSB are used in the simulation examples shown in Appendix A, “Simulating in Project Navigator with ModelSim.” The Software Development Kit (SDK) is recommended for software development, and you’ll have the chance to try it as you work through this guide. Sample C applications used in software Debug Test Drives are generated in SDK.

Viewing a System Summary Page

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

Device and Board Selections used in Test Drives

This guide uses the Spartan®-3A DSP 1800A Starter Board and targeting a MicroBlaze processor. The options you will select are listed in “Take a Test Drive! Creating a New Embedded Project,” page 14.

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

The BSB Wizard and the ISE Design Suite

You will start your new project in the ISE software. In Project Navigator, you will use the New Project wizard to create your project. When your project is created, ISE recognizes your embedded project, automatically starts Xilinx Platform Studio (XPS), and opens the BSB to complete your design. It takes a minute or so for XPS to launch. It has many things to do!

The Xilinx Microprocessor Project (*.xmp) File

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

The XMP file is created and handled in ISE like any other non-embedded source, such as HDL code and constraints files. You’ll learn all about that process in the next test drive.
**Take a Test Drive! Creating a New Embedded Project**

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

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

<table>
<thead>
<tr>
<th>Wizard Screen</th>
<th>System Property</th>
<th>Setting or Command to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create New Project</td>
<td>• Project Name</td>
<td>• Choose a name for your project (do not use spaces).</td>
</tr>
<tr>
<td></td>
<td>• Project Location and Description</td>
<td>• Choose a location for your project (again, no spaces).</td>
</tr>
<tr>
<td></td>
<td>• Top-level source type</td>
<td>• You can also add a description.</td>
</tr>
<tr>
<td></td>
<td>• Choose a name for your project (do not use spaces).</td>
<td>• Select <strong>HDL</strong> (default).</td>
</tr>
<tr>
<td>Device Properties</td>
<td>• Product Category</td>
<td>• All</td>
</tr>
<tr>
<td></td>
<td>• Family</td>
<td>• Spartan-3A DSP</td>
</tr>
<tr>
<td></td>
<td>• Device</td>
<td>• XC3SD1800A</td>
</tr>
<tr>
<td></td>
<td>• Package</td>
<td>• FG676</td>
</tr>
<tr>
<td></td>
<td>• Speed</td>
<td>• -4</td>
</tr>
<tr>
<td></td>
<td>• Synthesis Tool</td>
<td>• XST (VHDL/Verilog)</td>
</tr>
<tr>
<td></td>
<td>• Simulator</td>
<td>• User-specific*</td>
</tr>
<tr>
<td></td>
<td>• Preferred Language</td>
<td>• <strong>VHDL</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accept all other defaults.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*Supported simulators are listed in “Installation Requirements: What You Need to Run EDK Tools,” page 8.</td>
</tr>
<tr>
<td>Create New Source</td>
<td></td>
<td>Click <strong>New Source</strong>. This opens the “New Source Wizard.”</td>
</tr>
<tr>
<td>Select Source Type</td>
<td>Select the <strong>Embedded Processor</strong> source type, then specify the following values.</td>
<td>• <strong>system</strong></td>
</tr>
<tr>
<td></td>
<td>• File name</td>
<td>• Type <strong>system</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Location</td>
<td>• Accept the default location.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leave <strong>Add to project</strong> checked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Click <strong>Next</strong>, then <strong>Finish</strong>.</td>
</tr>
<tr>
<td>Create New Source</td>
<td>You will not be adding any more new sources.</td>
<td>Click <strong>Next</strong>.</td>
</tr>
<tr>
<td>Add existing sources</td>
<td>Do not add anything.</td>
<td>Click <strong>Next</strong>.</td>
</tr>
<tr>
<td>Project Summary</td>
<td></td>
<td>Click <strong>Finish</strong>.</td>
</tr>
</tbody>
</table>

After you complete the New Project wizard, ISE recognizes that you have an embedded processor system and starts XPS.
4. A dialog box appears, asking if you want to create a Base System using the BSB wizard. (This can take a few moments.) Click **Yes**.

Now that the BSB wizard has started, you can create a project using the settings described in the following table.

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

<table>
<thead>
<tr>
<th>Wizard Screens</th>
<th>System Property</th>
<th>Setting or Command to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome to the Base System Builder</td>
<td>Project type options</td>
<td>Select the option to create a new design.</td>
</tr>
</tbody>
</table>
| Board Selection | • Board Vendor  
• Board Name  
• Board Revision | • Select **Xilinx**.  
• Select the **Spartan-3A DSP 1800A Starter Board**.  
The 1800A board contains a Spartan-3A DSP device, which means that the BSB allows you to configure one or more MicroBlaze processors.  
• Select board revision 1 (default). |
| System Configuration | Type of system | Select **Single-Processor System**. |
| Processor Configuration | • Processor Type  
• System Clock Frequency  
• Local Memory  
• Enable Floating Point Unit | • Select a **MicroBlaze** processor.  
• Don’t change the default system clock frequency of **62.5 MHz**.  
• Change the local memory to **16 KB**.  
• Do not enable the floating point unit. |
| Peripheral Configuration | Processor 1 (MicroBlaze) Peripherals list | Remove the following peripherals from the “Processor 1 (MicroBlaze) Peripherals” list of default values:  
• Ethernet_MAC  
• SPI_FLASH  
Leave the remaining cores as is. |
| Cache Configuration | Instruction/Data caches | Do not enable caches. |
| Application Configuration | Example Application Options | Do not change the default values for the example applications. |
| Summary | System Summary page | After you have selected and configured all your system components, the BSB displays an overview of the system, allowing you to verify your selections.  
You should have a MicroBlaze processor with the following components:  
• Multi-Port Memory Controller (mpmc)  
• XPS GPIO (three instances)  
• XPS UartLite (xps_uartlite)  
• LMB BRAM IF Controller (two instances)  
You can go back to any previous wizard page and make revisions.  
The BSB creates a default memory map. The memory map cannot be modified inside the BSB, but it can be changed after the BSB is finished. |

5. After reviewing the system summary, click **Finish**.
A Note on the BSB and Custom Boards

If you plan to create a project that includes a custom board, you can create a Xilinx Board Description file (*.xbd) for your custom board library and place it in the $XILINX_EDK/board location. For more information, see “Xilinx Board Description (XBD)” in the Platform Specification Format Reference Manual, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

What’s Next?

The upcoming sections address Hardware Fundamentals.

- In Chapter 3, “Using Xilinx Platform Studio,” you will use the Xilinx Platform Studio (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 book and other documentation on XPS more easily.

What is XPS?

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

The XPS Software

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

Optional Test Drives are provided in this chapter so you can explore the information and tools available in each of the XPS main window areas.
Chapter 3: Using Xilinx Platform Studio

Using the XPS User Interface

The XPS main window is divided into these three areas:

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

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

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

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

Project Tab

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

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

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

- **Design Summary**
  The Design Summary is a graphical display of the state of your embedded design and gives you easy access to system files.

![Figure 3-2: Project Information Area, Project Tab](image)
Applications Tab

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

- Create and add a software application project, build the project, and load it to the block RAM.
- Set compiler options.
- Add source and header files to the project.

**Note:** You can create and manage software projects in XPS; however, SDK is the recommended tool for software development.

![Applications Tab](image)

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

IP Catalog Tab

The IP catalog lists information about the IP cores, including:

- Core name and licensing status (not licensed, locked, or unlocked)
- Release version and status (active, early access, or deprecated)
- Supported processors
- Classification

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

**Note:** You might have to click and drag to expand the pane to view all details of the IP.
Take a Test Drive! Reviewing the Project Information Area

1. With your project open in XPS, click the Project tab.
2. Right-click any item under Project Files and select Open. In future Test Drives, you will edit some of these files.
3. Close the file by selecting File > Close.
4. Right-click any item in the Project Options category to open the Project Options dialog box. Alternatively, you can select Project > Project Options.
5. Close the Project Options dialog box.
6. Click the IP Catalog tab.
7. At the top left of the IP Catalog window, note the two buttons. Click them and observe how they change how the IP catalog displays.
8. Right-click any item in the IP Catalog to see what options are available. Note a few of these in particular:
   - Add IP, which adds the selected IP to your design
   - View PDF Datasheet, which brings up the datasheet for the IP
   - View IP Modifications (Change Log), which lists the revision history for the selected IP.
9. Find and expand the Communication Low-Speed IP category.
10. Right-click the XPS_UART (Lite) peripheral and select View PDF Datasheet to view the related PDF datasheet.

System Assembly View

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

Bus Interface, Ports, and Addresses Tabs

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

- The Bus Interface tab displays the buses in your design. You can use this view to modify the information and connections for each bus.
- The Ports tab displays ports in your design. You can use this view to modify the details for each port.
- The Addresses tab displays the address range for each IP instance in your design. You can automatically generate the system address map by clicking Generate Addresses.
Connectivity Panel

With the Bus Interfaces tab selected, you’ll see the Connectivity Panel (label 4 in Figure 3-1, page 18), which is a graphical representation of the hardware platform connections.

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

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

Filters Pane

XPS provides filters that you can use to change how you view the Bus Interfaces and Ports in the System Assembly View. The filters are listed in the Filters pane (label 6 in Figure 3-1, page 18) when the Bus Interfaces or Ports tabs are selected. Using these filters can unclutter your connectivity panel when doing a design with a large number different buses.

View Buttons

So you can sort information and revise your design more easily, the System Assembly View provides two buttons that change how the data is arranged (label 5 in Figure 3-1, page 18):

- **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](image)

1. Click the **Ports** tab located at the top of the screen.
2. Expand the **External Ports** category to view the signals that are present outside the FPGA device.
3. Note the signal names in the **Net** column and find the signals related to the RS232_Uart_1 port. (You might need to drag the right side of the **Net** column header to see its entire contents.) These signals are referenced in the next step. Collapse this category when finished.
4. Scroll down, locate, and expand the RS232_Uart_1 peripheral.

   Note the net names and how they correspond to the names of external signals. The **RX** and **TX** net from the UART are name-associated with the external ports.
5. Right-click the RS232_Uart_1 peripheral and select **Configure IP** to launch the associated IP Configuration dialog box. You can open a similar configuration dialog box for any peripheral in your system.
   a. Observe what happens when you hold the mouse cursor over a parameter name.
   b. Note the three top tabs and buttons available for this core.
   c. Close this dialog when finished.

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

### Console Window

The Console window (label 3 in Figure 3-1, page 18) provides feedback from the tools invoked during runtime. Notice the three tabs: Console, Warnings, and Errors.

### Start Up Page

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

**Take a Test Drive! Familiarizing Yourself with the Start Up Page**

In this Test Drive, you will look at the contents of the Start Up page.

1. In the XPS main window, select the **Start Up Page** tab. If you don’t see this tab, select Help > View Start Up Page.

2. On the New This Release page, click the **Redesigned interface improves project hand-off and portability between hardware and software teams** link. If you have used previous versions of XPS, you'll learn about some important new features. (Most of the new features are covered in upcoming Test Drives.)

### XPS Tools

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

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

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

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

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

- The Library Generation tool, Libgen, configures libraries, device drivers, file systems, and interrupt handlers for the embedded processor system. Click **Software > Generate Libraries and BSPs** to start Libgen.
Take a Test Drive! Reviewing the XPS Structure

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

XPS Directory Structure

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

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

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

__xps
Contains intermediate files generated by XPS and other tools for internal project management. You will not use this directory.

blockdiagram
Contains files related to the block diagram.

data
Contains the user constraints file (UCF). For more information on this file and how to use it, see the ISE® UCF help topics at: [http://www.xilinx.com/support/documentation/sw_man uals/xilinx11/manuals.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/manuals.pdf).

etc
Contains files that capture the options used to run various tools. This directory is empty because no actions outside of the BSB have been performed.

pcores
Used for including custom hardware peripherals.

Two other directories contain the files generated by the BSB:

- **TestApp_Memory_microblaze_0**
- **TestApp_Peripheral_microblaze_1**

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

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

**system.xmp**
This is the top-level project design file. XPS reads this file and graphically displays its contents in the XPS user interface.

**system.mhs**
The system MHS file, captures the system elements, their parameters, and connectivity in a text format. The MHS file is the hardware foundation for your project.

**system.mss**
The system microprocessor software specification, or MSS file, captures the software portion of the design, describing the system elements and various software parameters associated with the peripheral in a text format. The MSS file is the software foundation for your project.

The MHS and MSS files are the main products of your XPS design. Your entire hardware and software system are represented by these two files.
Take a Test Drive! Exploring the Directory Structure

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

1. Using a file explorer utility, such as Window Explorer, navigate to the top-level directory for your project.
2. Open the various subdirectories and become familiar with the basic file set.

What’s Next?

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

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.” The behavior of each processor or peripheral core can be customized. Implementation parameters control optional features and specify what is ultimately implemented in the FPGA. The implementation parameters also define the addresses for your system.

Hardware Platform Development in Xilinx Platform Studio

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.

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 and 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 Format Specification Reference Manual, available at http://www.xilinx.com/ise/embedded/edk_docs.htm.

Take a Test Drive! Examining the MHS File

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

1. Select the Project tab in the Project Information Area of the XPS software.
2. Look under the Project Files heading to find MHS File:system.mhs. Double-click the file to open it.
3. Search for xps_uartlite in the system.mhs file by selecting Edit > Find and using the Find tool that appears below the main window area.
   Notice how the peripherals and their ports and parameters are configured in this file.
4. Take some time to review other IP cores in your design. When you are finished, close the system.mhs file.
Chapter 4: Working with Your Embedded Platform

The Hardware Platform in System Assembly View

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

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

Editing a port name or setting a parameter takes effect when you press **Enter** or click **OK**. XPS automatically writes the system modification to the hardware database, which is contained in the MHS file. The recommended method for editing the MHS file is to open it in the System Assembly View.

*Note:* Adding, deleting, and customizing IP are described in Chapter 7, “Creating Your Own Intellectual Property.”

Generating the Hardware Platform

Generating the hardware platform is a three-step process. First, XPS generates a netlist. Next, the design is implemented (mapped into FPGA logic) in the ISE® Design Suite tools. In the final step, the implemented design is converted to a bitstream that can be then downloaded to the FPGA.

Generating the Netlist

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

- Reads the design platform configuration MHS file.
- Generates a Hardware Description Language (HDL) representation of the MHS file written to **system.[vhd|v]** along with a **system_stub.[vhd|v]** file. The system file is your MHS description written in HDL format. The system_stub file is a top-level HDL template file that is useful should you want to instantiate your processor system as a component in a larger, HDL-based design.
- Synthesizes the design using Xilinx Synthesis Technology (XST).
- Produces a netlist file.


Exporting Your Hardware Platform

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

*The system.xml File* The **system.xml** file has the information SDK requires for you to do software development and debug work on the hardware platform that you designed.
**Take a Test Drive! Exporting Your Hardware Platform to SDK**

1. In the XPS software, select Project > Export Hardware Design to SDK.
2. We recommend that you accept the default location. The ultimate repository search path to the `system.xml` file is then `/system/SDK/SDK_Export/hw/..` in your project directory.

   **Note:** There are other XML files used in XPS, so it is important to know the location of the XML file that you will be using.
3. Select **Export Only**. You’ll run several Test Drives of SDK in upcoming chapters.

**What Just Happened?**

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

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

The other option, **Export & Launch SDK**, automatically overwrites any existing XML files that already exist in the export directory. Any existing bitstream (BIT) and Block Memory Map (BMM) files in the export directory are erased. If the export includes BIT and BMM files, XPS saves them in the export directory. This process prevents the export directory from containing hardware files that are out of synchronization.

---

**Take a Test Drive! Generating the Bitstream**

Upon successful completion of the XPS process, you can use the ISE Project Navigator software to implement the design and generate the bitstream.

The Project Navigator software reads the netlist that was created, and in conjunction with the User Constraints File (UCF), produces a BIT file containing the hardware design. Compiled C code is not part of that bitstream; it is added later in SDK.
1. Review the Project Navigator main window. The Design panel on the left side should look like this:

![Design Panel](image)

**Figure 4-1:** ISE Project Navigator Design Area

You are about to run the design through to where a bitstream is generated. But before you can do that, you need to add some information so that the ISE Place and Route (PAR) tool has information about your design, such as the pinout and how fast it needs to run.

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

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

This step is only necessary when the XMP source is the top-level design in an ISE project. If the XMP is instantiated in a VHDL or Verilog file, ISE manages the EDK UCF file.
3. The Adding Source Files dialog box appears to show the progress of processing the UCF file. When the file processing completes, click **OK**.

   Your Sources window should now look like this:

   ![Image of Sources window]

   **Figure 4-2:** The system.xmp and the system.ucf Files

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

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

5. In the Processes pane of the Design panel, double-click on **Generate Programming File** to create your bitstream. It takes a few minutes and should conclude with the message “Process ‘Generate Programming File’ completed successfully.”

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

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

**What’s Next?**

Now you can start developing the software for your project using SDK. The next two chapters explain embedded software design fundamentals.
Chapter 5

Introducing the Software Development Kit

The Xilinx® Software Development Kit (SDK) facilitates the development of embedded software application projects. SDK has its own software that is based on the Eclipse open source tool suite. The SDK is a complementary program to XPS. From SDK, you develop the software that is used by the peripherals and processor(s) elements built in XPS.

About SDK

It’s important to understand some of the terminology used in SDK: hardware platform, software platform, software project, perspectives, and views.

SDK Terminology

The hardware platform is the embedded hardware design that is created in XPS and exported in the form of an XML file. When you import the XML file into SDK, you import the hardware platform. There can be only one hardware platform in an SDK project.

Once the hardware platform is identified and imported, you create the software platform. A software platform is a collection of libraries and drivers that form the lowest layer of your application software stack. Your software applications must link against or run on top of a given software platform, using the provided Application Program Interfaces (APIs). Therefore, before you can create and use software applications in SDK, you must create a software platform project. SDK includes the following two software platform types:

- **Standalone** - A simple, semi-hosted and single-threaded environment that provides basic features like 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.

A single SDK project can contain more than one software platform. For instance, you can have one software platform that is set up for standalone, and another set up for Xilkernel.

Software platforms contain software projects. The software project is simply your application. A given software platform can contain multiple software projects.

Perspectives and Views

When you start using SDK, you will notice that the software 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, it has a third appearance (not covered in this book). For more information about Profiling in EDK, see the EDK Profiling User Guide. These different displays of windows are called perspectives, and each window in the perspective is called a view.

You will see that switching perspectives is as easy as clicking the large tab on the top left of your workspace. And it is one of the things that gives SDK its power and flexibility.
Take a Test Drive! Importing a Hardware Platform

1. Double-click the Xilinx Software Development Kit desktop icon, or select Programs > Xilinx ISE Design Suite 11 > EDK > Xilinx Software Development Kit from the Windows Start menu.

2. Identify a workspace. You can put this in any location. For now, point to the home folder of your project, or the location where you want to do some SDK work.
   Caution! Make sure the path name does not have spaces.

3. Locate the hardware design that you exported in the previous chapter. This is an XML file (system.xml), and it contains a representation of the embedded platform on which your code is going to execute.

4. The following message displays. Read it carefully before you click OK because it’s a good description of SDK project structure.

   ![Next Steps Dialog Box](image)

   Figure 5-1: Next Steps Dialog Box
SDK opens with the C/C++ Projects tab displayed, as shown below. The microblaze_0 item represents the MicroBlaze™ processor in your new hardware platform.

![Figure 5-2: C/C++ Projects Tab with Hardware Platform](image)

**Take a Test Drive! Creating a Software Platform**

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

1. Select **File > New > Project** and select **Software Platform**.

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

*Note:* If you had a hardware platform that contained more than one microprocessor, each would be listed in the **Processor** drop-down list.

2. Populate the new software platform project with the following selections:
   - Project name: **SW_Platform_1**
   - Processor: **microblaze_0 (microblaze)**
   - Platform type: **standalone**
   - Project Location: Make sure that the **Use default** check box is selected

3. Click **Finish**.
What Just Happened?

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

To view the list of drivers and libraries used in the software platform, right-click **Software Platform** and select **Software Platform Settings**. Select the pages to view the software platform, OS and libraries, and drivers used in the software platform. You can also use this dialog box to change the configuration.

Expand the **microblaze_0** section of **SW_Platform_1** in the C/C++ Projects tab as shown. The **code**, **include**, **lib**, and **libsrc** folders contain the libraries for all of the hardware in your embedded design.

![Software Directory Tree](Image)

*Figure 5-3: Software Directory Tree*

You can double-click any of the files in this view, and they will appear in the SDK Editor area.
Take a Test Drive! Setting Up the Software Environment

You have set up your environment so you can begin writing code that targets the embedded processor design which you built in the previous chapters.

Now it’s time to create a software project that is a new Managed C Application Project for Software_Platform_1.

1. Select File > New > Managed Make C Application Project.
   Notice that there are a number of sample applications available. You are going to start with a simple “Hello World” application.

2. Populate the New Managed Make C Application Project with the following selections:
   - Project Name: hello_1
   - Software Platform: SW_Platform_1
   - Project Location: Make sure that the Use Default Location for Project check box is selected
   - Sample Applications: Select the Hello World application

3. Click Next.

4. Populate the Build Configuration screen with the following selections:
   - Project Type: Xilinx MicroBlaze Executable.
   - Configuration: Leave all check boxes selected (Debug, Release, and Profile).

5. Click Finish.
   The hello_1 sample application builds automatically, producing an ELF file suitable for downloading onto the target hardware.
The C/C++ Projects tab now contains information related to the software platform and the software project. Take a minute to get familiar with it. The relevant project management information is displayed in this window.

6. Expand the src folder in the hello_1 software project. Notice that it has the heading `hello_1 {SW_Platform_1}`, indicating that it's a software project built for the SW_Platform_1 software platform.

7. Double-click the helloworld.c file. The file opens in the SDK Editor window. You can modify the sample code or create your own, as needed.

You can also see the hello_1.ld linker script that was generated for this project. A linker script is required to specify where your software code is loaded in the hardware system memory.

You now have a complete framework for editing, compiling, and building a software project. The last 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. Because this is an FPGA, you must perform an extra step that might seem new to you. It involves configuring the FPGA with a bitstream that loads a design into the FPGA. In this case, the design is an embedded processor system.

1. Set your system up for debugging:
   a. Connect a USB Programming cable, and an RS232 cable between the demo board and the computer. The LED on the USB cable should be glowing green because the demo board is turned on.
   b. Open a Hyperterminal (or another terminal emulation program) on your PC and set the display to 9600 baud, 8 bit data, 1 stop bit.

2. Select **Tools > Program FPGA**.

3. Populate the dialog box with the following selections:
   - Bit File: `system.bit` in your project folder
   - Bmm File: `edkBmmFile_bd.bmm` in your project folder
   - ELF File: `microblaze_0` processor of type `microblaze` with a `BootLoop` initialization ELF file

**Note:** Your filenames might be different depending on how you set up your hardware project.

![Program FPGA Dialog Box](image)
4. Click **Save and Program**. SDK confirms that the FPGA was successfully programmed with the bitstream.

At this point, you are both downloading the bitstream to the FPGA and initializing 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.

For future designs, if you want to include the ELF file in the FPGA’s on-chip RAM, you can specify that ELF file in the **Initialization ELF** field of the Program FPGA dialog box. For now, you’ll be downloading the ELF file in a separate step instead.

5. Right-click **hello_1.elf** and select **Debug As > Debug on Hardware**.

Notice that the perspective changes to the Debug Perspective.

6. In the Debug Perspective, observe that the C code is now highlighted at the first executable line of code, and the debug window shows that for `thread[0]` the `main()` function is currently sitting at line 28 because there is an automatically-inserted breakpoint.

7. Execute the code by clicking the **Resume** button.

8. Terminate the debug session by clicking the **Terminate** button.

9. Observe the output in the terminal window, as shown. When you are finished, close SDK.

![Hello World](image)

**Figure 5-6: Terminal Display**

**What Just Happened?**

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

**What’s Next?**

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

In the next chapter, you’ll be digging deeper into SDK as you create a new software project, use the source code management, and explore debugging.
Chapter 6

Editing, Debugging, and Releasing in SDK

The Xilinx® Software Development Kit (SDK) can be used for the entire lifecycle of the software development process. All software development activities can be done in SDK, including creating, editing, and building your software projects; debugging your software on target hardware; profiling your software on your target hardware; releasing your software; and programming the software into flash memory.

More on Drivers

Before getting started with SDK, you need to know about the low-level software drivers that Xilinx provides. These drivers are located in `<Xilinx Install>\EDK\sw\XilinxProcessorIPLib\drivers`.

There is a directory for each peripheral driver, and drivers corresponding to each piece of hardware available to you in EDK. This directory contains, at a minimum, the following for each driver:

- The driver source code
- HTML documentation about the drivers
- Examples of how the drivers could be used

Before continuing, take a minute to familiarize yourself with this important information.

SDK Perspectives and Window Types

As demonstrated in the previous chapter, SDK has different screen organizations, called perspectives.

So far you have worked in the C/C++ Perspective and the Debug Perspective. The other perspective built into SDK is the Profiling perspective. Whether you are working in the C/C++ or Debug Perspective, you’ll find the SDK windowing system very powerful.

There are two kinds of windows within perspectives: editing windows and informational windows.

Editing windows, which contain C or C++ source code, are language-specific and syntax-aware. Right-click an item in an editing window to view a comprehensive list of actions that can be done for that item.

Informational windows are particularly flexible. You can have as many informational windows as you like. An information window can have any number of views, each view indicated by a tab on the top of the window.
Views in the Debug Perspective are: 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 and drag it to another window. You will see that its contents are displayed in the new window. To see what views are available for the selected perspective, select Window > Show View.

Experiment with moving windows around. The ability to easily customize your workspace is one of the more powerful SDK features.

**Take a Test Drive! Editing Software**

In the previous chapter, you compiled and debugged a sample software module. In this Test Drive, you’ll run two more sample modules and create a third software module from scratch to call the first two routines. The goal is to show how to manage software projects consisting of multiple source files.

1. If you closed SDK, restart it by selecting Programs > Xilinx ISE Design Suite 11 > EDK > Xilinx Software Development Kit from the Windows Start menu.
2. Create a new workspace.
3. Import the same hardware platform using the same Hardware Specification File and procedure you used in “Take a Test Drive! Importing a Hardware Platform” in Chapter 5.
4. Create a new standalone software platform in the same way as in “Take a Test Drive! Creating a Software Platform” in Chapter 5. Name the software platform editor_exercise_platform.

**Take a Test Drive! Creating Managed Make C Application Projects**

In the next few steps, you will create two Managed Make-C Application Projects, each with a different sample application. You’ll then make a blank C Application Project and learn how to include two files as part of a third.

This fundamental type of file management is required for doing larger projects. If at any time this gets confusing, review the project management steps in the Test Drives in Chapter 5.

1. Create a Managed Make C Application Project using same technique as in Chapter 5. Use the Memory Tests sample application.
2. Create another Managed Make C Application Project. This time, use the Peripheral Tests sample application.
3. Your display now looks similar to the figure below. Notice how both C Projects (memory_tests and peripheral_tests) are built for the editor_exercise_platform.

![C Project Comparison](image)

*Figure 6-1: C Project Comparison*

Before you can run these two applications, you must download the FPGA bitstream to the board, as you did in the previous chapter.

4. Select **Tools > Program FPGA**

5. Populate the dialog box with the following selections:
   - Bit File: `system.bit` in your project folder
   - Bmm File: `edkBmmFile_bd.bmm` in your project folder
   - ELF File: MicroBlaze™ processor `microblaze_0` with a BootLoop initialization ELF file

*Note:* Your file names might be different depending on how you set up your hardware project.
To see what the two sample programs do, you’ll run the `memory_tests` project first, then the `peripheral_tests` project.

6. Run the `memory_tests` project using the following procedure:
   a. In the project management area, locate the `memory_tests.elf` file, which is located in `memory_tests/binaries`.
   b. Right-click `memory_tests.elf` and select **Debug As > Debug on Hardware**.
   c. When the Debug Perspective opens, run the program by selecting **Run > Resume** from the pull-down menu. Observe the output of the program on your terminal window.
   d. End your debug session by selecting **Run > Terminate**.
7. Click on the C/C++ Perspective and repeat the procedure in step 6 to run the `peripheral_tests` C project.

After running the `memory_tests` and the `peripheral_tests` applications, the output in the terminal window looks like this:

```
--Starting Memory Test Application--
Testing memory region: DDR2_SDRAM
  Memory Controller: mpmc
  Base Address: 0x88000000
  Size: 0x08000000 bytes
  32-bit test: PASSED!
  16-bit test: PASSED!
  8-bit test: PASSED!
--Memory Test Application Complete--
---Entering main---
```

Running `GpioInputExample()` for DIP_Switches_8Bit...
`GpioInputExample` PASSED. Read data: 0x0

Running `GpioOutputExample()` for LEDs_8Bit...
`GpioOutputExample` PASSED.

Running `GpioInputExample()` for Push_Butons...
`GpioInputExample` PASSED. Read data: 0x0

Running `VartLiteSelfTestExample()` for mdm_0...
`VartLiteSelfTestExample` PASSED
---Exiting main---

![Figure 6-3: Steps Shown on the Terminal Display](image)

8. End your debug session by selecting **Run > Terminate**.

Now that the two applications have run successfully, you create a third application that calls each of the other two applications individually. In SDK, that is done by importing existing applications. The next Test Drive shows you how to import an application.
### Take a Test Drive! Working with Multiple Source Files

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

1. In the C/C++ Perspective, double-click the `memorytest.c` and `testperiph.c` filenames to open the files.

   ![C/C++ Projects](image)

   **Figure 6-4:** Projects Highlighted in the C/C++ Project Window

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

3. In `testperiph.c`, change the name of `main()` to `peripheraltest_main()`. This should be near line 46.
4. Save both files.
   The files build automatically. They will fail because there is no longer a main function,
   which the build is trying to locate. If you were to change either function name back to
   main(), that build would proceed error-free.

   Now you need to create a new module that calls the two renamed functions:
memorytest_main() and peripheraltest_main().

5. Create a new Managed Make C Application Project using the technique described in
   “Take a Test Drive! Setting Up the Software Environment” in Chapter 5. Use the
   Empty Application sample application.

6. Select File > New > File and select top_test to create a new source file called
   top_test.c.

7. Open top_test.c and add the code shown in the following figure.

   ![top_test.c](image)

   Figure 6-5: top_test.c

8. Save the file, and observe that SDK builds it automatically. (Automatic building can be
   controlled by selecting or deselecting the Project > Build Automatically menu item.)

   SDK errors out, because it cannot locate the peripheral test or memory test functions.

   The next task is to import the memorytest_main() and peripheraltest_main() functions into the top_test project so that the top_test software project can access them.
1. Select **File > Import** and select **File system**.
2. Populate the File system window as shown in the following figure.

   ![Import File System](image)

   **Figure 6-6: Import File System**

   The window shown in the figure above tells SDK how to import the `memory_tests` file system. In this example, you have a single C file to import in each directory, so you don’t have to import an entire hierarchy of files. If you had a more substantial project to import, you would select the **Create Complete Folder Structure** option. You can do this if you want to see how it would look in your project; it works the same way.

3. Repeat the previous step for the `peripheral_tests` file system.
After you import the two file systems, the errors in `top_test.c` disappear and the SDK workspace looks like the workspace shown in the following figure. If the errors persist, make sure that you have imported the files that were created in step 2 of the Test Drive on page 42.

![Project Directory Tree](image-url)

*Figure 6-7: Project Directory Tree*
4. Right-click test_top and select **Generate Linker Script** to create a linker script for the application.

5. Right-click the top_test.elf file and select **Debug As > Debug on Hardware** to download and run the application and confirm that it runs correctly. A display on your terminal window shows that both the memory test and the peripheral test are working.

---

**Take a Test Drive! Working with the Debugger**

Now that you have done some file manipulation in the C/C++ Perspective, it’s time to look at some of the features of the Debugger. SDK provides full source-level debugging capabilities. If you have used other debuggers you can see that the SDK debugger has most, if not all, of the usual features.

The Debug window, a key part of the Debug Perspective, contains a wealth of information about the state of your debug session.

![Debug Window](image)

*Figure 6-8: Debug Window*

In the figure above, you can see that our call stack is three deep. Specifically, main(), at address 0x000001c8, called peripheraltest_main(), at address 0x000005ac, which then went on to call GpioInputExample() at address 0x0000085c.

In addition, the program state is currently suspended, which means a breakpoint was encountered. Each item on the call stack also shows on what line of code the calling routine is located.

Software execution can also be controlled from the Debug window. In the Debug Perspective, hover your mouse over the buttons at the top of the Debug window to see a description of what each one controls.
Take a Test Drive! Examining the Debug Output

To begin this test drive, be sure that you have completed the Test Drive under “Take a Test Drive! Working with the Debugger,” page 50.

1. Be sure the top_test.elf file is selected in the Projects pane. With the C/C++ Perspective open, select Run > Debug As > Debug On Hardware to download the top_test.elf file to your target board. Once successfully downloaded, the Debug Perspective opens automatically.

2. If some of the views such as Disassembly or Memory are not visible, select Window > Show View and select the view that you want to see.

   If the view doesn’t appear in the window that you intended, click and drag it into place.

   As you can see in the previous figure, the processor code is positioned at the beginning of main() and program execution is suspended at line 26.

   You can correlate that with the Disassembly View, which shows the assembly level program execution also suspended at 0x000001b8.

   Open the Registers View and observe that the RPC register (the program counter) contains 0x000001b8. If the Registers View tab isn’t visible, select Window > Show View > Registers.

3. Double-click in the margin of the top_test.c window next to the line of code that says peripheraltest_main(). This sets a breakpoint at peripheraltest_main().

   Figure 6-9: Debug Perspective

   As you can see in the previous figure, the processor code is positioned at the beginning of main() and program execution is suspended at line 26.
4. Confirm that the breakpoint is set by checking the Breakpoints View. If the Breakpoints View tab is not visible, select Window > Show View > Breakpoints.

5. Select Run > Resume to resume program execution and run to the breakpoint. Note that program execution stops at the line of code, with execution stopped at 0x000001c0, as displayed in the peripheraltest_main() disassembly and the Debug window.

6. Select Run > Step Into to move to the peripheraltest_main() routine. Observe that your program is now in the peripheraltest_main() routine, program execution is suspended at location 0x00000580, and the call stack is now two deep.

7. Select Run > Resume again to run the program to conclusion. When the program is finished executing, the Debug window shows that the program has suspended in an exit routine, which occurs when you are running under the control of the debugger.

8. Observe the relevant text on your terminal output, indicating that both peripheraltest_main() and memorytest_main() have run.

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

Having run the Test Drives in this chapter, you now have a C project with multiple files to work with. You have also gained enough exposure to the debugger to experiment with and customize SDK, so it works the way you want it to.

What’s Next?

In the next chapter, you will create your own IP. The following chapters cover the advanced topics of creating and debugging dual processor designs.
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 mouse clicks.

Benefits of XPS and BSB

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

Benefits of CIP Wizard

While you are the expert regarding the functionality of the required custom IP, you might not fully understand CoreConnect™ bus protocols, the /pcores directory structure required by XPS, or the creation of Bus Function Model simulation frameworks. This chapter clarifies these important system distinctions and walks 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 walks you through the initial process. It also includes completed pcore design for study and analysis.
Overview of IP Creation

The Bus Interface tab in the XPS System Assembly View (shown in Figure 3-1, page 18) shows connections among buses, processors, and IP. Any piece of IP you create must be compliant with the system you design.

To ensure compliance, you must follow these steps:

1. Determine the interface required by your IP. The bus to which you want to attach your custom peripheral must be identified. For example, you could select one of the following interfaces:
   - Processor Local Bus (PLB) version 4.6. The PLBv4.6 provides a high-speed interface between the processor and high-performance peripherals. PLBv4.6 is used in both the PowerPC® and the MicroBlaze™ processor systems.
   - Fast Simplex Link (FSL). The FSL is a point-to-point “FIFO-like” interface. It can be used in designs using MicroBlaze processors, but generally is not used with PowerPC processor-based systems.

2. Implement and verify your functionality. As you do, keep in mind 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, and it supports the same buses that are supported by XPS.

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

The CIP wizard assists with implementation and verification of your design by walking you through IP creation. It sets up a number of templates that you can populate with proprietary logic.

Besides creating HDL templates, the CIP wizard can create a pcore verification project for Bus Functional Model (BFM) verification. The templates and the BFM project creation are great for jump-starting your IP development as well as ensuring that your IP will comply with the system you create. For details of BFM simulation, refer to Appendix B, “Intellectual Property Bus Functional Model Simulation.”
What You Need to Know Before Running the CIP Wizard

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

Supported Peripherals in the CIP Wizard

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

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

The CIP wizard supports creation of Fast Simplex Link (FSL) peripherals. You can also enable legacy PLB v3.4 and OPB buses by selecting the Enable OPB and PLB v3.4 bus interfaces check box in the Bus Interface page of the wizard.

Note: Support for OPB and PLBv34 IP will be removed in XPS 12.

Documentation

Before launching the CIP wizard, you should review the documentation specific to the bus interface you intend to use. Reviewing this information can help eliminate much of the confusion often associated to bus system interfaces. Also, review the XPS Help topics related to the CIP wizard by selecting Help > Help Topics and navigate to Procedures for Embedded Processor Design > Creating and Importing Peripherals. This documentation provides you with enough background to answer many of the questions that arise as you run the CIP wizard.

XPS provides datasheets related to the IP in your system. To access them, open the Start Up page by selecting Help > View Start Up Page. In the Start Up page, select the Documentation tab, expand IP Reference and Device Drivers Documentation, and click the Processor IP Catalog link.

If you plan to create a PLBv46 peripheral, examine one of the following datasheets, depending on whether your custom peripheral is a slave or master, single data access, or burst data access:

- plbv46_slave_single
- plbv46_master_single
- plbv46_slave_burst
- plbv46_master_burst

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

Note: Normally the CIP wizard is launched from within XPS, as described in the next Test Drive, but the CIP wizard can also run outside of XPS.
Take a Test Drive! Generating and Saving Templates

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

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

1. Start the CIP Wizard and determine the location in which to store the custom peripheral files:
   a. Open Xilinx ISE® Project Navigator, load your project, select system.xmp, and double-click the Manage Processor Design (XPS) process to launch XPS.
   b. In XPS, select Hardware > Create or Import Peripheral.
      After clicking past the initial Welcome page, the Peripheral Flow page opens, on which you can either create a new peripheral or import an existing peripheral.

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, so select To an XPS project.
   Because you launched the CIP wizard from within XPS, the directory location is filled in automatically.
   **Note:** If the custom pcore will be used for multiple embedded projects, you can save the file in an EDK repository.
4. Click **Next** to open the Name and Version page.

![Name and Version Page](image)

**Figure 7-1: Name and Version Page**

On this page, you will indicate the name and version of your peripheral.

- For this example design, use the name `pwm_lights`.
- A version number is supplied automatically, but you can change this number to fit your revision scheme. You can also add a description of your project.

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

**Note:** You can access related datasheets from the Bus Interface page.
6. Click **Next** to open the IPIF (IP Interface) Services page.

![IPIF Services Page](image)

**Figure 7-2: IPIF Services Page**

On this page, note that the CIP wizard automatically creates the following:
- Master or slave connections to the PLB bus
- Necessary bus protocol logic
- Signal sets used to attach your custom HDL code

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

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

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

The next page is the Slave Interface page, on which you can set up burst and cache-line support. Although you won’t use this support for this example, take a moment to review the content about slave peripherals and data width, then move on to the next page of the wizard.
8. Click **Next** to open the IP Interconnect (IPIC) page.

![IP Interconnect (IPIC) Page]

On this 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, you can go back and review the appropriate specification. The signals selected should be adequate to connect most custom peripherals.

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

For future reference, you could also select the **User logic memory space** option on the IPIF Services page, which opens a wizard page specific to managing user memory space.

9. Click **Bus2IP_Addr** and leave the other boxes unchanged.
10. Click **Next** to open the Peripheral Simulation Support page.

On this page, you can elect to have the CIP generate a BFM simulation platform for your project. Note that generating a BFM simulation platform requires the following:

- You must have already downloaded and installed the BFM simulation package for EDK.
- You must have ModelSim-SE or ModelSim-PE installed.
If you'd like, you can stop here and click the **BFM Package Installation Instructions** to go through the steps necessary to license, download, and install the BFMs. BFM simulation is described in **Appendix B, “Intellectual Property Bus Functional Model Simulation.”** If you think you might want to run a BFM simulation on this IP example, go ahead and generate the BFM platform now.

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

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

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

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

![Peripheral Implementation Support Page](image)

*Figure 7-4: Peripheral Implementation Support Page*
The Peripheral Implementation Support page lists three options for creating optional files for hardware and software implementation.

- The CIP wizard can create the user_logic template in Verilog instead of VHDL. To set this up, 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), you can click the Generate ISE and XST project files to help you implement the peripheral using XST flow check box. The CIP wizard will create the necessary ISE project files. However, if your peripheral is low-speed or very simple, this step is not necessary.

- If your peripheral is complex enough to require more complex software drivers, you can click the Generate template driver files to help you implement software interface check box to have the CIP wizard create the necessary driver structure and some prototype drivers based on the services selected.

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

**What Just Happened?**

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

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

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

**PLBv46 Slave and Burst Peripherals**

The PLB slave and burst peripherals are verified, optimized, and highly parameterizable interfaces. They also give you a set of simplified bus protocols. Your custom RTL interfaces to the IPIC signals, which are much easier to work with when compared to operating on the PLB or FSL bus protocols directly. Using the PLB slave and burst peripherals with parameterization that suits your needs greatly reduces your design and test effort because you don’t have to reinvent the wheel.
Figure 7-5 illustrates the relationship between the bus, a simple PLB slave peripheral, IPIC, and your user logic.

![Diagram of PLB Slave/Burst Module in a Custom Peripheral]

**Figure 7-5:** **PLB Slave/Burst Module in a Custom Peripheral**

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

*Note:* This figure shows only a partial list of files. It does not represent the full directory listing.

![Directory Structure Generated by the CIP Wizard]

**Figure 7-6:** **Directory Structure Generated by the CIP Wizard**
Note the following points about the files generated by the CIP wizard:

- The wizard created two HDL template files: `pwm_lights.vhd` and `user_logic.vhd`.
- The `user_logic` file makes the connection to the PLB v4.6 bus using the PLB slave/burst cores configured in `pwm_lights.vhd`.
  - The `user_logic.vhd` file is equivalent to the “Custom Functionality” block.
  - The `pwm_lights.vhd` file is equivalent to the “PLBv46 slave/burst” blocks.
- Your custom logic interfaces using the IPIC signals.

The following figure illustrates the relationship between the block diagram shown in Figure 7-5 and the generated files shown in Figure 7-6.

![Figure 7-7: Relationship of IP Module to Generated Files](image)

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

**Example Design Description**

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

You’ll open and modify the source code files for this peripheral in the next Test Drive. These files are located in the `/pcores` directory on your system. You’ll also use some example files that are included in the Zip file for this guide. Find and review the files listed there. You’ll open them in the Test Drive.
The custom peripheral controls the 8 LEDs on the evaluation board. To make the design interesting, the pwm_lights circuit will:

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

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

**Take a Test Drive! Modifying the CIP Wizard Template Files**

In this Test Drive you’ll open and modify the template files that the CIP wizard created for your project.

Conceptually, what you are going to do in this final Test Drive is simple: you’re going to load and run some C code that controls the pcore created by the CIP wizard. Before starting, notice a few of the software features that also come into play:

- SDK tracks the `system.xml` file that is used in its workspace. Recall that you introduced this file in Chapter 4, “Working with Your Embedded Platform.” If and when that file changes for any reason, SDK flags the change. You’ll see an example of that feature in this Test Drive when you add hardware to the file.
- By default, SDK maps all of your C code to block RAM. In this Test Drive, the piece of C code you will be using is larger (in terms of memory use) than that used previously. Consequently, the available block RAM to which SDK mapped is too small. Therefore, you must modify the Linker script. You can see that SDK has a built in GUI that simplifies modifying your Linker Script.

1. In XPS, select **File > Open** and navigate to the `pcores\pwm_lights_v1_00_a\hdl\vhdl` directory. This is where you find the `pwm_lights.vhd` file and the `user_logic.vhd` file. (See Figure 7-6, page 62.)
   **Note:** You might have to change the **Files of type** drop-down list to view and open these files.

2. Open the `pwm_lights.vhd` file.
   You need to add the external port names in this file. The external port names must be added in two places:
   - The top level entity port declaration
   - The port map for the instantiation of the `user_logic`
3. Scroll down to approximately line 160. In the code segment shown in the figure below, the user port LEDs are displayed in the appropriate location. Add the LEDs port declaration for the top-level entity in your file as shown here.

```vhdl
port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
  LEDs : out std_logic_vector(0 to 7);
  -- ADD USER PORTS ABOVE THIS LINE ------------------
  -- DO NOT EDIT BELOW THIS LINE ---------------------
  -- Bus protocol ports, do not add to or delete
  SPLB_Clk : in std_logic;
  SPLB_Rst : in std_logic;
)

Figure 7-8: LED Port Code
```

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

```vhdl
  C_SLV_DWIDTH => USER_SLV_DWIDTH, 
  C_NUM_REG => USER_NUM_REG 
| port map 
(
  -- MAP USER PORTS BELOW THIS LINE ------------------
  LEDs => LEDs,
  -- MAP USER PORTS ABOVE THIS LINE ------------------

  Bus2IP_Clk => ipif_Bus2IP_Clk,
  Bus2IP_Reset => ipif_Bus2IP_Reset,

Figure 7-9: Add Port Declarations
```

5. Save the file.

   Where user information is required in the two template files (`<ip core name>.vhd` and `user_logic.vhd`), you’ll find comments within the file that 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`.

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

7. Open and examine the `user_logic.vhd` file.

8. A completed `user_logic.vhd` file is provided in the Zip file for this guide. Replace the contents of the currently opened `user_logic.vhd` file with the contents of `user_logic.vhd` file and save the file.
Chapter 7: Creating Your Own Intellectual Property

Reviewing the File Contents

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

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

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

At approximately line 100, notice that the user port LEDs (0 to 7) were added. This output vector drives the eight LEDs on the evaluation board. Anytime you add signals specific to your design, you must add these ports in this location. You also need to add these ports in the top-level file and map them through to `user_logic`.

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

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

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

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

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

- Write to offset `0x00`: All LEDs turned off
- Write to offset `0x04`: All LEDs turned on
- Write to offset `0x08`: LED brightness varies, using a square function drive signal
- Write to offset `0x0C`: LED brightness varies, using a linear function drive signal
- Write to offset `0x10`: LED brightness is set to a constant value (the range is 0 to `0xFF`)

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

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

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

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

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

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

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

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

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

The final signals, **IP2Bus_WrAck** and **Bus2IP_WrCE(0)**, are also critical. **IP2Bus_WrAck** is a write acknowledge that must be returned by the custom logic. **IP2Bus_WrAck** must only be driven high 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, and connecting **IP2Bus_WrAck** directly to **Bus2IP_WrCE(0)** provides a simple, zero wait state response. The logic for the read acknowledge signal is identical. Again, the peripheral can add wait states if necessary.

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

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

Finally, 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 could connect logic to drive **IP2Bus_Error** to terminate the bus transfer.

**Adding Your Custom IP to Your Processor System**

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

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

- The first time you ran the CIP wizard, you created the **pwm_lights** peripheral, set up the bus interface, and generated the required template files.
- Next you will add **pwm_lights** to your project, again using the CIP wizard. In the process, **pwm_lights** is imported to an XPS-appropriate directory and the CIP wizard creates the MPD and PAO files. For more information about PSF files, see the **Platform Specification Format Reference Manual**, available at [http://www.xilinx.com/ise/embedded/edk_docs.htm](http://www.xilinx.com/ise/embedded/edk_docs.htm).
Take a Test Drive! Using the CIP Wizard to Re-Import the Modified File into Your XPS Project

1. Open the CIP wizard and specify that you want to import an existing peripheral to an XPS project.
2. On the Name and Version page, select `pwm_lights` from the Name drop-down list. A version is not required, but for this example click Use version and use either the default setting or a custom version number.
3. If the CIP wizard asks if you want to overwrite an existing peripheral with this name, select Yes.
4. Select HDL source files as shown in Figure 7-10. You can create pcores using RTL or existing netlists. It is also possible that documentation exists for the custom peripheral. Up to this point, the import flow has been easy to understand. In this procedure, however, import flow becomes more complex, so read carefully.

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

5. Select Use existing Peripheral Analysis Order file (*.pao).
6. Browse to the PAO file. By default, the browse function opens at the top-level `pcores` directory. The PAO file is located in the `/pwm_lights/data` subdirectory. When you use the PAO file to locate the necessary source files (including lower level libraries), it is not necessary to add any additional files or libraries.

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

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

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

---

**Figure 7-11:** HDL Analysis Information Page
8. The Current Logical Library dialog box might appear to indicate that the two VHDL files that you added have not yet been compiled. Click **Next** to have the wizard compile them.

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

10. In the Port and Parameter pages, accept the default values.

11. In the Identify Interrupt Signals page, uncheck **Select and configure interrupts**.

   The Identify Interrupt Signals page shows a complete listing of the PLBv46 bus signals that are used in this design. The CIP wizard automatically deals with all these signals and the associated bus protocols.

   In most cases, there is no need to spend any time analyzing whether all the necessary signals are there, especially if you initially used the CIP wizard to create the core.

   If a core was created using another means, or if it contains a complex or custom bus interface, this page is a useful tool for analyzing the bus signals.

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

   The `pwm_lights` peripheral contains no interrupt sources.

12. In the Parameter Attributes page, you can view and control available settings.

   Note that the pcore parameters were generated on an earlier page of the CIP wizard. The bus interface parameters were automatically generated. For this simple peripheral, no changes are required.
Complex peripherals might require a large number of parameters and require careful control of PLB bus behavior. You can use the Parameter Attributes page to view and control available settings.

Using the drop-down lists, you can view parameters for your custom pcore, parameters for the selected bus interface, or a combination of the two.

Note that the pcore parameters were generated on an earlier page of the CIP wizard. The bus interface parameters were automatically generated.

For this simple peripheral, no changes are required.

13. On the Port Attributes page, click on LEDs, then click **Display Advanced Attributes** to view the full display. This gives you more control over the port attributes because the attributes are written to the MPD file.

In the drop-down list, select List All Ports to view all ports that are used by the interface between your custom pcore and your embedded processor subsystem. In this case, the list of PLB signals is very long, but the CIP wizard did this part of your design for you.
14. Look around this screen, then move to the final screen. When you click **Finish**, the Import function completes.

If you haven’t noticed yet, you can see your custom peripheral listed in the IP Catalog under **Project Local pcores**.

Before adding `pwm_lights` to your design, you must make one change to the existing design. The eight LEDs on the evaluation board are currently connected to GPIO outputs. Now that `pwm_lights` is driving these LEDs, the `LEDs_8Bit` pcore must be removed from the design.

15. In the System Assembly View, right-click `LEDs_8Bit` and select **Delete Instance**. The Delete IP Instance dialog box appears:

![Delete IP Instance Dialog Box](image)

**Figure 7-13: Delete IP Instance Dialog Box**

16. For now, accept the default setting. You’ll add the external ports back into the design manually.

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

   XPS adds the IP to the System Assembly View. You can see it in the Bus Interfaces tab.
18. In the Connectivity Panel, click on the PLB bus to add the bus connection.

19. Click the Ports tab, expand pwm_lights_0, and select Make External from the Net drop-down menu.

A default name of pwm_lights_0_LEDs[0:7] was assigned as the net name. By expanding External Ports at the top of this view, you see that pwm_lights_0_LEDs_pin[0:7] is the port name.

20. You can change the assigned net and pin names by clicking in the drop-down box for pwm_lights_0. Alternatively, you can manually edit the MHS file. For now, leave the assigned names.

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

21. Select the Addresses tab. If you don’t see pwm_lights_0 under Unmapped Addresses, select Project > Rescan User Repositories.

22. Under the Address tab, click Generate Addresses. The pwm_lights pcore is assigned to an address range of 0xC4600000 – 0xC460FFFF.

23. Verify that the address range for the DDR2_SDRAM is 0x88000000 – 0x8fffffff. If this address has changed, change it back to the original value.

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

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

24. The final step is to update the UCF constraints file to assign the LED outputs to the proper FPGA pins.
25. Select the Project tab and double-click the UCF file to open it in the XPS main window.
26. Look for \texttt{fpga\_0\_LEDs\_8Bit\_GPIO\_IO\_0}. These pin assignments were left in the UCF file even though you earlier deleted the GPIO pcore. It is important to note that removing a pcore does not automatically trigger an update to the UCF file.
27. Replace \texttt{fpga\_0\_LEDs\_8Bit\_GPIO\_IO\_0\_pin} with \texttt{pwm\_lights\_0\_LEDs\_pin} in all 8 locations and save the UCF file.

Congratulations, you have created a custom pcore!

**Exporting the Design and Generating a New Bitstream**

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

1. In XPS, select **Project > Export Hardware Design to SDK**. Accept the default directory and click **Export Only**.
2. When the export process completes, close XPS, return to ISE, and double-click **Generate Programming File**.
3. Launch SDK. It could take a few moments to start up. If it recognizes any changes to the hardware platform, the Hardware Design Changes Detected dialog box appears:
4. Click the **Show Hardware Changes** button.

The file shown in the following figure opens in your browser.

**Figure 7-15: Hardware Change Window**

<table>
<thead>
<tr>
<th>Module Level Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROCESSOR:</strong> microblaze_0</td>
</tr>
<tr>
<td><strong>PROCESSOR PARAMETERS</strong></td>
</tr>
<tr>
<td>\texttt{VERSION}</td>
</tr>
<tr>
<td>\texttt{C_SCO}</td>
</tr>
<tr>
<td>\texttt{C_DATA_SIZE}</td>
</tr>
<tr>
<td>\texttt{C_DYNAMIC_BUS_SIZING}</td>
</tr>
<tr>
<td>\texttt{...}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>PROCESSOR PERIPHERALS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INSTANCE</strong></td>
</tr>
<tr>
<td>dinoe_cntic</td>
</tr>
<tr>
<td>linoe_cntic</td>
</tr>
<tr>
<td>\texttt{DS232_Dart_1}</td>
</tr>
<tr>
<td>\texttt{Push_Buttons}</td>
</tr>
<tr>
<td>\texttt{DIP_Switches_0Bit}</td>
</tr>
<tr>
<td>\texttt{ED_NORAM}</td>
</tr>
<tr>
<td>\texttt{mem_0}</td>
</tr>
<tr>
<td>\texttt{LEDs_8Bit}</td>
</tr>
<tr>
<td>\texttt{+pwm_lights_0}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>PROCESSOR MEMORY MAP</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PERIPHERAL</strong></td>
</tr>
<tr>
<td>dinoe_cntic</td>
</tr>
<tr>
<td>linoe_cntic</td>
</tr>
<tr>
<td>\texttt{DS232_Dart_1}</td>
</tr>
<tr>
<td>\texttt{Push_Buttons}</td>
</tr>
<tr>
<td>\texttt{DIP_Switches_0Bit}</td>
</tr>
<tr>
<td>\texttt{ED_NORAM}</td>
</tr>
<tr>
<td>\texttt{mem_0}</td>
</tr>
<tr>
<td>\texttt{-LEDs_8Bit}</td>
</tr>
<tr>
<td>\texttt{+pwm_lights_0}</td>
</tr>
</tbody>
</table>
The `xps_gpio` pcore driving the LEDs is now removed, and `pwm_lights` is included. SDK recognized and displayed the changes to the hardware platform that you made in the `system.xml` file.

5. Close the browser window.
   If you click the SDK actions tab at the bottom of the Hardware Design Changes Detected dialog box, information about the automatic removal of the `gpio` driver and inclusion of the `pwm_lights` driver display.

6. Close the Hardware Design Changes Detected dialog box.
   SDK opens to the C/C++ Perspective.

7. If you used the same SDK workspace that you used in the SDK chapters, a standalone platform already exists.
   If you started SDK with a new workspace, create a new standalone Software Platform called `cip_wizard_platform`.

8. Create a new Managed Make C Application project called `LEDS` using the Empty Application sample application.

9. Add a new source file with the name `leds.c`. Paste the C code from the attached `LEDs.c` file, save it, and see if the file compiles correctly (it won’t). The compilation error is a result of the object code size being larger than the selected memory space. Let’s investigate.

10. Right-click `leds.c` in the Projects pane and select **Generate Linker Script**.
    The Linker Script Editor opens. Notice that everything, including text, heap, and stack, is assigned to block RAM (specifically called out as `ilmb_cntlr_dlm_b_cntlr` in the Linker Script Generator). You must change it to use DDR2 RAM instead, because it provides a larger memory space.

11. In the Linker Script Generator, assign all code sections to `DDR2_SDRAM_MPMC_BASEADDR` using the drop-down list. Verify that the heap and stack sizes are set to `0x1000`.
    The code successfully recompiles. The ELF file generated is about 93K, much larger than the 16K of block RAM that you have available.

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

14. Run `leds.elf` with a terminal window open. When the application code begins to run, the HyperTerminal window displays the debug options:

```
- Hyperterm window must be selected  
- Enter value and Carriage Return  
- Enter 0 to read status  
- Enter 1 for LEDs off.  
- Enter 2 for LEDs on.  
- Enter 3 for Log scale.  
- Enter 4 for Linear scale.  
- Enter 5 for Constant value.  
- Enter 6 to exit program.  
```

Select the terminal window and type input values.

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

**What Just Happened?**

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

**What's Next?**

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

Using the BSB to Create a Dual-Processor Design

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

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

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

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

1. Create a new ISE Project with a single embedded processor source.
   Wait for XPS to open automatically. This could take a minute.
2. When XPS starts, select the option to create a new project using the BSB wizard.
3. Accept all the defaults in the BSB except for those listed below.

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

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

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

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

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

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

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

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

You now have a bitstream ready for downloading to the target hardware and a system.xml file for use in SDK.
Take a Test Drive! Using SDK to Develop Software for a Dual-Processor System

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

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

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

1. Launch SDK.
2. When the Workspace Launcher dialog box appears, create a new workspace called Dual_Processor_Workspace and save it to a directory of your choice.
3. In the New Hardware Specification File dialog box, point to the system.xml file that you exported earlier. If you used the default project locations, the file is located at <ISE Project Name>/system/SDK/SDK_Export/hw/system.xml.

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

![C/C++ Projects](image)

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

4. Select File > New > Software Platform and create a new software platform project with the following settings:
   - Project Name: MicroBlaze_Platform_0
   - Processor: microblaze_0 (microblaze)
   - Platform Type: standalone
   - Project Location: Make sure the Use default check box is selected
5. Use the same procedure to create a second software platform project with the following settings:
   - Project Name: MicroBlaze_Platform_1
   - Processor: microblaze_1 (microblaze)
   - Platform Type: standalone
   - Project Location: Make sure the Use default check box is selected

You can now see that each MicroBlaze processor has a single associated software platform, as shown in the following figure.
The next step is to create a Managed C Application Project for each processor. In this example, you will create and modify a “hello world” project for each processor.

6. Select File > New > Managed Make C Application Project and create a new Managed Make C application project with the following settings:
   - Project Name: hello_world_0
   - Software Platform: MicroBlaze_Platform_0
   - Project Location: Make sure the Use Default Location for Project check box is selected
   - Sample Applications: Select Hello World

7. Use the same procedure to create a second Managed Make C application project with the following settings:
   - Project Name: hello_world_1
   - Software Platform: MicroBlaze_Platform_1
   - Project Location: Make sure the Use Default Location for Project check box is selected
   - Sample Applications: Select Hello World

You now have two sample C applications, one for each processor, as shown in the following figure.
8. Modify the `src\helloworld.c` file in each MicroBlaze platform to indicate which processor it is running on. For example, open the `helloworld.c` file in the `MicroBlaze_Platform_0` project and change the code

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

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

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

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

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

mb-size hello_world_1.elf
  text data bss dec hexfilename
  1958 296 2090 4344 10f8 hello_world_1.elf

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

**Take a Test Drive! Modifying the Software Platform Settings**

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

1. Select **Tools > Software Platform Settings** to modify the settings for the `MicroBlaze_Platform_1` processor.
2. In the OS and Libraries settings page, review the stdin and stdout settings. In this case, the hardware on which XMD runs needs to be set to `MDM`. The default value is `mdm_0`, so this is already correctly set.

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

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

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

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

1. Select **Tools > Program FPGA** and select your bitstream and block memory map files from the following locations:
   - `<ISE Project Name>\system.bit`
   - `<ISE Project Name>\edkBmmFile_bd.bmm`
2. Confirm that the initialization file for each processor is set to `BootLoop`, then click **Save and Program**.
   - The target hardware has now been programmed with the bitstream for the dual processor design.
   - The next step is to download an ELF file for each processor.
3. Right-click the binary file `hello_world_0.elf` in the `hello_world_0` (`MicroBlaze_Platform_0`) folder of the C/C++ Projects tree, and select **Debug As > Debug on Hardware**.
   - The Debug Perspective opens for the `MicroBlaze_Platform_0` processor.
4. Go back to the C/C++ Perspective, and use the same procedure to debug the `hello_world_1.elf` file and to download this file to the `MicroBlaze_Platform_1` processor.

The Debug Perspective opens again. Look at the contents of the Debug window and note that there are two debugging tasks:

![Debug Perspective with two debugging tasks](image)

**Figure 8-4: The SDK Debug Perspective**

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

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

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

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

   *Note:* Both call stacks should read “1 main() at...”
The processor output displays in the terminal window or XMD console, as shown in the following figure.

![Processor Output in the Terminal and XMD Console Windows](image-url)

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

As you can see, debugging more than one processor design in SDK is similar to debugging a single processor. This was a simple example. You can perform other software development tasks with SDK as well, such as stepping, setting breakpoints, and examining registers and memory.
For the majority of the Test Drive sections in this document, you used ISE® Project Navigator as the starting point for all activities. You can also simulate an embedded design in Project Navigator.

**Introducing Simulation**

You’ll use the same design you created in Chapter 7, “Creating Your Own Intellectual Property,” in the Test Drives.

**Take a Test Drive! Simulating an Embedded Design**

In this Test Drive you’ll add some simple modifications to a VHDL file that XPS automatically generates. You use it as the new top-level file in which you instantiate the embedded system (contained in the `system.xmp` file). Also, you add a configuration block that loads the block RAMs in the design with an example ELF file.

In addition to demonstrating how to simulate an embedded design in Project Navigator, this exercise shows a simple technique for creating a top-level VHDL file that instantiates your embedded system. This example is for VHDL only.

The first part of simulating an embedded design in Project Navigator is setting up the simulation model generation.

1. Open the ISE project you used in Chapter 7, “Creating Your Own Intellectual Property.”
2. Launch XPS by double-clicking the XMP source.
3. In XPS, select **Project > Project Options**.
4. Click the **HDL and Simulation** tab.
5. Set up HDL and Simulation with the following options:
   - HDL: **VHDL**
   - Simulation Test Bench: Make sure that the **Generate test bench template** check box is not selected
   - Simulation Models: **Behavioral**

   Next, you must select an executable file to simulate.

6. In the Applications tab, right-click the `TestApp_Peripheral_microblaze_0` and select **Mark to Initialize BRAMs**.
7. Right-click on the `TestApp_Memory_microblaze_0` project and select **Build Project**.
8. In XPS, select **Simulation > Compile Simulation Libraries** and follow the on-screen instructions in the wizard to compile or locate your simulation libraries. This will take several minutes to complete.

9. Select **Simulation > Generate Simulation HDL Files**.
   You now have a VHDL representation of your entire embedded system, including some C code that executes out of block RAM.

Now that the simulation model generation is set up, you create the top-level VHDL file, which instantiates the processor subsystem.

1. In ISE, open the system_stub.vhd source file in the `<project name>/system/hdl` directory. This file was created automatically when the embedded system netlist was created in Chapter 7.

2. Select **File > Save As** and save the file using the name system_top.vhd.
   It is important to save this file with a different name because it is overwritten in future instances of running the tool.

3. Select **Project > Add Source**.

4. Open the system_top.vhd file that you just created in the `system/hdl` subdirectory of your project. Associate it with **All**.

5. At the top of the Design view, select **Behavioral Simulation** from the **Sources for** drop-down list.

6. Add the following code at the end of system_top.vhd (after the `#end architecture STRUCTURE` statement):
   ```vhdl
   -- synthesis translate_off
   configuration system_conf_top of system_stub is
     for STRUCTURE
       for system_i : system
         use configuration work.system_conf;
       end for;
     end for;
   end system_conf_top;
   -- synthesis translate_on
   ``
   This code is included in any VHDL design where you have block RAM that needs to be initialized with data (in this case it is an ELF file).

7. Save the file.
Next, set up Project Navigator to run your simulation.

1. Copy the pn.do file (included in the Zip file for this guide) to the top-level directory of your ISE project.

2. In the Design view, right-click the **Simulate Behavioral Model** process for the ModelSim Simulator and select **Process Properties** to open the ModelSim Simulation Properties.

3. Browse to the saved pn.do file and edit the Process Properties as shown here. Make sure to do the following:
   - Make sure the Use Custom Do File check box is selected.
   - Uncheck the Use Automatic Do File check box.
   - Browse to and select a custom Do file.

4. Click **OK**.

5. Double-click the **Simulate Behavioral Model** process to run ModelSim.
Examining the Output

After the simulation loads and runs, the ModelSim Wave window displays, showing all of
the top-level signals for your embedded design.

The MicroBlaze™ processor code runs from block RAM starting at location 0x00000000.
To see this, scroll down in the Wave window to the signal/system/ilmb_lmb_abus
(the MicroBlaze Local Memory Address bus) at time 1400000ps. You should see the
addresses 0, 4, 8, and beyond. The instructions fetched at those addresses are in the
/system/ilmb_lmb_readdbus file.

What Just Happened?

You just simulated the TestApp_Memory_Microblaze software project that is built and
managed in XPS and runs in SDK. The simulation generator, Simgen, determines what
program is marked to initialize block RAMs in the Applications Tab of XPS and builds a
simulation model based on that information.

The simulated ELF file, executable.elf, is located in the
/system/TestApp_Memory_Microblaze_0 subdirectory of your project directory.

If you elect to simulate one of the applications that was developed using SDK, you can
copy its ELF file into \system\TestApp_Memory_Microblaze_0, regenerate the
simulation HDL files, and run the simulation again.

The technique illustrated here is limited to software that executes from on-chip block
RAM. Simulating code executing from external DDR2 memory is beyond the scope of this
document. For more information on simulating from external DDR2 memory, consult the
Synthesis and Simulation Design Guide for ISE, available at

There are a number of required steps to building the embedded system simulation
environment. Once you have it set up, you have a very powerful environment that you can
use for simulating both your embedded system and the rest of your FPGA.
This chapter uses the `pwm_lights` design and the bus functional model (BFM) simulation platform from Chapter 7, “Creating Your Own Intellectual Property.” If you skipped the CIP wizard chapter and have arrived here, you will need to go back and complete the design described in Chapter 7.

You must have also compiled the EDK simulation libraries.

**What are BFMs and Why Should I Use Them?**

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

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

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

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

While conceptually simple, manually setting up an entire simulation environment to run bus functional simulation is difficult. The CIP wizard automates most of this for you, because it can automatically connect the necessary BFM simulation models to your IP under test.

**Note:** If the Version Management wizard opens when you launch the `bfm_system` project, use the it to update any required cores.
Take a Test Drive! Running the BFM

Prior to starting this Test Drive, close any XPS project that is open. If you elected to create the BFM s, the CIP wizard created the \bfmsim\ sub-directory in your \pcorespwm\lights\v1.00\a\devl directory, in which it saved the XPS BFM simulation project called bfm_system.xmp.

1. Open the bfm_system.xmp project in XPS. The Bus Interfaces window looks like this:

![Bus Interfaces Window](image)

2. Select Project > Project Options and click the HDL and Simulation tab.
3. Select the HDL format in which you would like to simulate. We will use the default, VHDL.
4. BFM offers Behavioral Simulation only, so leave the Simulation Model selection set to its default.
5. Select OK when your have finished setting up the simulation options.
6. Select Simulation > Generate Simulation HDL Files to run the Simulation Model Generator (Simgen) for this test project.
   
   Simgen creates a simulation directory structure under the /bfmsim directory. The simulation directory contains the HDL wrapper files along with the DO script files needed to run a behavioral simulation.

7. Click Custom Button 1 in the XPS GUI tool bar. The CIP wizard configures this tool bar button when it creates the BFM simulation project.
   
   Custom Button 1 initiates the following:
   
   - Launches a Bash shell to run a make file.
   - Calls the IBM CoreConnect™ ToolKit Bus Functional Compiler (BFC) to operate on a sample.bfl file using the simulation options that were previously set (see \pcorespwm\lights\v1.00\a\devl\bfmsim\scripts\sample.bfl for more detail).
   - Invokes the simulator with the BFC output command files (INCLUDE or DO files) depending on the simulator to execute the commands in the sample.bfl file.
What are BFMs and Why Should I Use Them?

The simulator waveform result will be similar that shown here.

**Figure B-2:** BFM Waveform Simulation Results for sample.bfl

What Just Happened?

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

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

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

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

Benefits of XPS Tools

The clear benefit is that you not only avoided having to create these models yourself, but XPS also automatically made all the correct connections.

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

To find more information associated with these buttons, select **Project > Customize Buttons** and press F1 to view the related help topic. The location of the make file you will use is provided in the following Test Drive.
In addition to compiling the BFL, the make file executed by **Custom Button 1** called the simulator with the command files to start simulation, simplifying the simulation launch and compilation process to a single button click.

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

The syntax of BFM scripts is not very intuitive; however, by looking at a completed script, it will be easy to extend it to have more functionality.

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

2. Open the `sample.bfl` file.

   Remember that you have the option of replacing this file with the attached `sample.bfl` file if you want to see what happens in this Test Drive without walking through all the steps. If you elect to do this, you can skip ahead to step 5.

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

   **BFL Command Information**

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

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

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

   **Adding Commands to sample.bfl**

   3. Starting with the `sample.bfl` template, at approximately line 175, find the line starting with `configure` and change it to read:

      ```
class configure{
      msize = 01
      };
      ```
4. After the start testing section near the end of the file, add the following code from the bus_transaction_bfl_code.txt. These commands generate bus transactions.

```plaintext
-- Define several bus transactions for pwm_lights
-- Memory updates are 64 bits write, bus transactions are 32 bits wide.

-- Write value of 22 hex to LED register
mem_update(addr=30000010,data=22222222_22222222)
write (addr=30000010,size=0000,be=11110000)
-- Read status register, expect to get F0F02207
read (addr=30000000,size=0000,be=11110000)
-- Write to offset 0, then read status, expect to get F0F02208
mem_update(addr=30000000,data=00000000_00000000)
write (addr=30000000,size=0000,be=11110000)
read (addr=30000000,size=0000,be=11110000)
-- Write to offset 4, then read status, expect to get F0F02200
--mem_update(addr=30000004,data=00000000_00000000)
write (addr=30000004,size=0000,be=00001111)
read (addr=30000000,size=0000,be=00001111)
-- Write to offset 8, then read status, expect to get F0F02208
mem_update(addr=30000008,data=00000000_00000000)
write (addr=30000008,size=0000,be=11110000)
read (addr=30000008,size=0000,be=11110000)
-- Write to offset C, then read status, expect to get F0F02200
--mem_update(addr=3000000C,data=00000000_00000000)
write (addr=3000000C,size=0000,be=00001111)
read (addr=30000000,size=0000,be=00001111)
```

Modifying the code results in the following:

- The `mem_update` is used to set the write data value at the address to which you want to write.
- The write command initiates the PLB write, and the read command initiates the PLB read.
- Size setting of 0000 implies a single transaction.
- The be (byte enable) settings correspond to a 64 bit bus.
- Addresses aligned to 0, 8, and so on, set the byte enables to 11110000.
- Addresses aligned to 4, c, and so on, set the byte enables to 00001111.

5. Open the attached sample.bfl file.
6. Make sure your sample.bfl file matches the attached file and then save your file.
7. Click **Custom Button 1**.
8. In ModelSim, scroll down to the end of the signal listing in the Waveform window. The last signals listed correspond to `user_logic` signals, which are the custom signals in the `pwm_logic` pcore.
What Just Happened?

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

- When ip2bus_rdack is high, the ip2bus_data signal contains the value read back from pwm_lights.
- For the order of writes to the core, the simulation will show (in order) F0F02207, F0F02208, FF002200, FF002204, and FF002205.

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

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

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

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

Glossary

B

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

BFL
Bus Functional Language.

BFM
Bus Functional Model.

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

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

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

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSB</td>
<td>Base System Builder. A wizard for creating a complete design in Xilinx Platform Studio (XPS). BSB is also the file type used in the Base System Builder.</td>
</tr>
<tr>
<td>BSP</td>
<td>See Standalone BSP.</td>
</tr>
<tr>
<td>CFI</td>
<td>Common Flash Interface</td>
</tr>
<tr>
<td>DCM</td>
<td>Digital Clock Manager</td>
</tr>
<tr>
<td>DCR</td>
<td>Device Control Register.</td>
</tr>
<tr>
<td>DLMB</td>
<td>Data-side Local Memory Bus. See also: LMB.</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access.</td>
</tr>
<tr>
<td>DOPB</td>
<td>Data-side On-chip Peripheral Bus. See also: OPB.</td>
</tr>
<tr>
<td>DRC</td>
<td>Design Rule Check.</td>
</tr>
<tr>
<td>DSPLB</td>
<td>Data-side Processor Local Bus. See also: ISPLB.</td>
</tr>
</tbody>
</table>
EDIF file
Electronic Data Interchange Format file. An industry standard file format for specifying a design netlist.

EDK
Xilinx Embedded Development Kit.

ELF file
Executable and Linkable Format file.

EMC
External Memory Controller.

EST
Embedded System Tools.

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

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

FPGA
Field Programmable Gate Array.

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

G

GDB

GNU Debugger.

GPIO

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

H

Hardware Platform

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

HDL

Hardware Description Language.

Hierarchical View

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

I

IBA

Integrated Bus Analyzer.

IDE

Integrated Design Environment.

ILA

Integrated Logic Analyzer.

ILMB

Instruction-side Local Memory Bus. See also: LMB.
IOPB
Instruction-side On-chip Peripheral Bus. See also: OPB.

IPIC
Intellectual Property Interconnect.

IPIF
Intellectual Property Interface.

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

ISC
Interrupt Source Controller.

ISE
Xilinx ISE Project Navigator project file.

ISOCM
Instruction-side On-Chip Memory.

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

ISS
Instruction Set Simulator.

J

JTAG
Joint Test Action Group.
L

Libgen

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

LibXil Standard C Libraries

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

LMB

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

M

MDD File

Microprocessor Driver Description file.

MDM

Microprocessor Debug Module.

MFS File

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

MHS file

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

MLD file

Microprocessor Library Definition file.

MVS file

Microprocessor Verification Specification file.

MOST®

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

MSS file
Microprocessor Software Specification file.

N

NCF file
Netlist Constraints file.

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

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

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

NPI
Native Port Interface.

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

O

OCM
On Chip Memory.

OPB
On-chip Peripheral Bus.
Appendix C: Glossary

P

PACE
Pinout and Area Constraints Editor.

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

PBD file
Processor Block Diagram file.

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

PLB
Processor Local Bus.

PROM
Programmable ROM.

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

S

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

SDK
Software Development Kit.

SDMA
Soft Direct Memory Access

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

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

SPI

Serial Peripheral Interface.

Standalone BSP

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

SVF File

Serial Vector Format file.

UART

Universal Asynchronous Receiver-Transmitter.

UCF

User Constraints File.

VHDL

VHSIC Hardware Description Language.

XBD File

Xilinx Board Definition file.

XCL


Xilkernel

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

XMD

Xilinx Microprocessor Debugger.

XMP File

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

XPS

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

XST

Xilinx Synthesis Technology.

Z

ZBT

Zero Bus Turnaround™.