Platform Studio User Guide

Embedded Development Kit EDK 7.1i

UG113 (v4.0) February 15, 2005

XILINX®

© 2005 Xilinx, Inc. All Rights Reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

Platform Studio User Guide
UG113 (v4.0) February 15, 2005

The following table shows the revision history for this document.

	Version	Revision
01/31/04	1.0	Initial Xilinx release for EDK 6.2i.
03/12/04		Updated for service pack release.
03/19/04	2.0	Updated for service pack release.
08/20/04	3.0	EDK 6.3i.
02/15/05	4.0	Updated for EDK 7.1i.

About This Guide

This document describes how to use the Xilinx® Embedded Development Kit (EDK). EDK is a series of software tools for designing embedded programmable systems, and it supports designs of processor sub-systems using the IBM PowerPC™ hard processor core and the Xilinx® MicroBlaze™ soft processor core.

Guide Contents

This manual contains the following chapters.

- Chapter 1, "Overview," gives an overview of Platform Studio™ technology.
- Chapter 2, "Creating a Basic Hardware System in XPS," contains a step-by-step procedure to generate a simple hardware system for EDK-based designs.
- Chapter 3, "Writing Applications for a Platform Studio Design," contains a step-by-step procedure to generate software for EDK-based designs using Xilinx® EDK 7.1i and Xilinx® ISE™ 7.1i software.
- Chapter 4, "Address Management," describes the embedded processor program address management techniques. For advanced address space management, a discussion on linker scripts is also included in this chapter
- Chapter 5, "Interrupt Management," outlines interrupt management in both MicroBlaze and PowerPC. It details the interrupt handling in MicroBlaze and PowerPC, and the role of Libgen for MicroBlaze and PowerPC.
- Chapter 6, "Using Xilkernel," describes Xilkernel, a set of interfaces and functions that allow context switching and resource sharing between applications.
- Chapter 7, "Using XilMFS," describes XilMFS, a memory-based file system library.
- Chapter 8, "Simulation in EDK," describes the HDL simulation flow using EDK and third party software.
- Chapter 9, "Debugging in EDK," describes the basics of debugging a system designed using EDK.
- Chapter 10, "Profiling Embedded Designs," describes the steps to profile a program on hardware using LibXil profile library provided with EDK.
- Chapter 11, "System Initialization and Download," describes the basics of system initialization and download using the Platform Studio tools.

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

Table 1-1: **Additional Resources**

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging. http://support.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records. http://support.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches. http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes
Data Sheets	Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging. http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues. http://support.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment. http://www.support.xilinx.com/xlnx/xil_tt_home.jsp

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Table 1-2: **Typographical Conventions**

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *`design_name`*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**

Table 1-2: **Typographical Conventions *(Continued)***

Convention	Meaning or Use	Example
Italic font	Variables in a syntax statement for which you must supply values	**ngdbuild** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr ={on|off}**
Vertical bar |	Separates items in a list of choices	**lowpwr ={on|off}**
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Repetitive material that has been omitted	**allow block** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Table 1-3: **Online Document Conventions**

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Handbook*.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Table of Contents

Chapter 4: Address Management

Chapter 5: Interrupt Management

Chapter 6: Using Xilkernel

Chapter 9: Debugging in EDK

Chapter 10: Profiling Embedded Designs

XILINX®

Chapter 1

Overview

The *Platform Studio User Guide* is for users of the Embedded Development Kit (EDK). EDK is a series of software tools for designing embedded processor systems on programmable logic, and supports the IBM PowerPC™ hard processor core and the Xilinx® MicroBlaze™ soft processor core. Platform Studio™ is the graphical user interface technology that integrates all of the processes from design entry to design debug and verification. This document describes both simple and complex design tasks that you might do. This chapter gives an overview of Platform Studio technology.

This chapter contains the following sections.

- "Creating an Embedded Hardware System"
- "Creating Software for the Embedded System"
- "Software Libraries"
- "System Simulation"
- "System Debug and Verification"
- "System Initialization and Download to the Board"

Note: Platform Studio tutorials are located online at http://www.xilinx.com/ise/embedded/edk_examples.htm.

Creating an Embedded Hardware System

In order to design an embedded processor system, you must first create a hardware platform. A hardware platform consists of one or more processors, buses, and peripherals connected to the processors. Chapter 2, "Creating a Basic Hardware System in XPS," describes the steps needed to create a hardware platform.

Creating Software for the Embedded System

After a hardware processor system is created, you can run application software on the processor. Chapter 3, "Writing Applications for a Platform Studio Design" guides you through creating and compiling application software using Platform Studio. The chapter also details options for downloading and debugging the application.

Software Libraries

Platform Studio supports various software libraries that are included in the installation. These libraries can be used to enhance application software, because they provide specific functions that are more coarsely grained tasks than driver routines. Chapter 6, "Using Xilkernel," describes the XilKernel library that provides functions for context switching

and resource sharing for multiple tasks. Chapter 7, "Using XilMFS," describes XilMFS, a memory-based file system library for creating and managing files and directories in memory (RAM). Chapter 10, "Profiling Embedded Designs" describes the XilProfile library, a software-intrusive profiling technology to create Call Graph and Histogram information for program optimization. This library can be used to find hotspots in the application software and tune the application for better performance.

System Simulation

After you have created a hardware embedded design and written the software to run on the processors, you must do design simulation before verifying the design. Chapter 8, "Simulation in EDK," describes all of the simulation options that are a part of Platform Studio technology, and illustrates the steps required to simulate the hardware and the software running on the hardware.

System Debug and Verification

Once the processor system is created and application software written for the system, you can either simulate the system using Platform Studio's simulation options, or debug your system as detailed in Chapter 9, "Debugging in EDK." This chapter describes the Xilinx® Microprocessor Debug (XMD) tool that forms the debug interface for hardware system debug and software running on hardware. It explains the GNU debugger (GDB) tool and how to use GDB with XMD. The chapter also discusses steps to using ChipScope™ with Platform Studio Technology.

System Initialization and Download to the Board

Once you have created the processor system, and written the application software for the system, you can download the system can be downloaded to the FPGA development board. Chapter 11, "System Initialization and Download," describes system initialization and download, with an emphasis on Fast Download in MicroBlaze systems and System ACE™ file Generation capabilities.

Fast Download

The Fast Download section describes an innovative "fast" download for MicroBlaze systems. This technique uses a unidirectional Fast Simplex Link (FSL) from the MicroBlaze Debug Module to the processor.

Generating an ACE File

This section illustrates the steps required to generate a System ACE configuration file that is useful in production systems. The System ACE file is used to completely configure the FPGA on a board with hardware bitstream, program, and data information, and to start processor execution. This is the final step in design production.

Chapter 2

Creating a Basic Hardware System in XPS

This chapter provides a step-by-step procedure to generate a simple hardware system for EDK-based designs using Xilinx® EDK 7.1 and Xilinx® ISE™ 7.1i EDK hardware. This involves assembling a system that contains a processor along with buses and peripherals, generating an HDL netlist, and implementing the design using ISE implementation tools to generate a bitstream.

This chapter assumes that you are using the Xilinx® Platform Studio™ (XPS), an integrated development environment included with the EDK. The chapter contains the following sections.

Overview

XPS is an integrated design environment (IDE) used to develop EDK-based system designs. A simple Hello World system is used to demonstrate the flow involved in building a processor hardware system. The hardware flow explained here is for the PowerPC™ 405 processor embedded in Xilinx® Virtex™-II Pro devices, but the flow for the Xilinx® MicroBlaze™ soft-core processor is similar. The differences for creating a MicroBlaze system are included.

Assumptions

This chapter assumes that you:

- Have a basic understanding of processor and bus based systems
- Have a board on which to test the generated hardware

Steps

The steps involved in creating a hardware system for EDK using XPS are as follows:

1. Create a New XPS Project
2. Select a Target Board
3. Select the Processor to be Used
4. Configure the Processor

5. Configure IO Interfaces
6. Specify Internal Peripheral Settings
7. Specify Software Configuration
8. View System Summary and Generate
9. View Peripherals and Bus Settings
10. Generate Bitstream
11. Download Bitstream and Execute

Create a New XPS Project

1. Open XPS from **Start → Programs → Xilinx Embedded Development Kit 7.1 → Xilinx Platform Studio.**

   When XPS opens, a dialog box opens.

2. Select the **Base System Builder Wizard** radio button to create a new XPS project using the Base System Builder (BSB) Wizard, and then click **OK**.

   You can also begin a new project in the BSB by selecting **File → New Project → Base System Builder**. This opens the Create New Project Using Base System Builder Wizard dialog box.

3. Specify the location at which you want to create your XPS project. An XPS project file has a `.xmp` extension, and your XPS project resides in the directory where the XMP file resides.
4. After you specify the XMP file location, click **OK.**

Select a Target Board

After you click **OK** in the previous step, XPS opens the BSB Wizard.

1. Select the **I would like to create a new design** radio button.
2. Click **Next**.

   The next dialog box that opens allows you to select a target board.

3. Select the **I would like to create a system for the following development board** dialog box.
4. Select **Xilinx** for the Board Vendor and **AFX Virtex-II Pro fg456 Board** for the board name.

   The wizard also has an option to create a design for a custom board.

5. After you make all the selections, click **Next** to continue.

Select the Processor to be Used

After you select a board, the next dialog box prompts you to select a processor. You can choose either PowerPC or MicroBlaze.

1. Select **POWERPC**.
2. Click **Next**.

Configure the Processor

The software displays the Configure Processor dialog box that corresponds to the processor that you selected in the previous step.

1. Specify the reference clock frequency that is available on the board.

   Based on the selected reference clock, the choice of clock frequencies available for processor and bus displays.

2. For this example, accept the default frequency.

   The configuration page allows you to select the type of debug interface, on-chip memory, and choice to enable cache. Default selections are already made.

3. Click **Next**.

Note: For MicroBlaze, the configuration page allows you to select the debug interface type, whether you want any local data and instruction memory, and whether you want to enable cache for MicroBlaze. Accept all the default values, and click **Next**.

Configure IO Interfaces

Based on the board you selected, the BSB Wizard presents a list of external devices present on the board. Each page in the wizard displays up to three devices. If there are more than three, they are spread across multiple pages. By default, the wizard selects all devices to be included in the design. For the purpose of creating a simple hardware system, keep only one device for this example, the RS232.

1. Deselect all devices other than RS232.
2. Click **Next**.

   Additional Configure IO Interfaces pages open.

3. Deselect all the devices shown on other pages and click **Next** until you reach the Add Internal Peripherals dialog box.

Specify Internal Peripheral Settings

When you are done with all of the Configure IO Interfaces pages, the Add Internal Peripherals dialog box opens. For a PowerPC design, an instance of internal memory (PLB BRAM IF CNTLR) is added by default.

1. Select 32 kB for the memory size.

   You can add additional peripherals by clicking **Add Peripheral**. For this simple hardware system, however, do not add any other internal peripherals.

2. Click **Next**.

Note: For a MicroBlaze system, no internal peripheral is added by default. Do not add any additional peripherals. Click **Next**.

Specify Software Configuration

Now you have created and configured your hardware system. However, since this is a processor system, in order to test the hardware, you need some software that will execute on the processor and exercise the bus, the memories, and the peripherals. The BSB Wizard creates a simple test application for the software. The Software Settings dialog box in the BSB Wizard allows you to change memories used for different sections of the software.

However, the default software created by BSB is sufficient to test the hardware you just created.

Click **Next**.

View System Summary and Generate

Now you have a hardware system and the software to test it. In the System Created dialog box, the wizard displays a hardware system summary, the processor settings, and various peripherals arranged in terms of the buses to which they are connected. Review the settings and click **Generate**. The wizard generates various XPS design files that capture the system you have just created. Once design generation is done, the wizard displays a list of files generated for the design. Click **Finish** to close the wizard.

View Peripherals and Bus Settings

When you complete the BSB Wizard, you return to the XPS main window. The BSB allows you to create a simple design. Once you create the design, XPS provides various other tools to view and modify the design. The System tab on the left displays a list of components that your design contains. It also displays various project files and project options. The Applications tab contains software file information.

Some of the most useful tools are located in the Add/Edit Hardware Platform Specifications dialog box. To access this dialog box, select **Project → Add/Edit Cores**. This dialog box contains five tabs: Peripherals, Bus Connections, Addresses, Ports, and Parameters.

The Peripherals tab displays all of the non-bus components in your design. You can add or delete new components on this page. It also provides a catalog of all IPs available for your design. You can filter the IP catalog based on the processor or bus, and based on various category of IPs.

The Bus Connections tab displays a matrix of buses and various peripherals connected to that bus.

The Addresses tab displays addresses of all of the peripherals. You can change the address map. If you add new peripherals or change your design, click **Generate Addresses** and XPS generates new addresses. You can also lock certain addresses and let XPS generate the rest.

For further details on this dialog box, refer to the "Xilinx Platform Studio (XPS)" chapter in the *Embedded System Tools Guide*. For this simple hardware system, do not make any modifications in the design.

Click **Cancel**.

Generate Bitstream

This brings you back into the main XPS environment. Now you are ready to implement the design for the board. Select **Tools → Generate Bitstream.** This runs various tools which take the hardware design and generate a bitstream for the FPGA. The location of this bitstream is `implementation/system.bit`.

The bitstream only contains the hardware information. To populate the bitstream with the software for the processor, select **Tools → Update Bitstream**. This option generates the software and updates the bitstream with the software information. The resulting bitstream is ready to be downloaded. It is located at `implementation/download.bit`.

Download Bitstream and Execute

To download the application, set up the board and the parallel cable as required.

1. Open the HyperTerminal application from **Start** → **Programs** → **Accessories** → **Communications** → **HyperTerminal** on your Windows desktop.
2. Open a new connection with Baud Rate Setting of 9600.
3. Connect the appropriate COM port the board.
4. In XPS, select **Tools** → **Download** to download the bitstream to the board.

   The processor begins executing. If you see meaningful text in your HyperTerminal window, the design is running successfully on the board.

Chapter 3

Writing Applications for a Platform Studio Design

This chapter provides a step-by-step procedure to generate software for EDK-based designs using Xilinx® EDK7.1i and Xilinx® ISE™ 7.1i software. EDK software involves building libraries, compiling C applications, initializing bitstreams with the application, downloading applications onto external memories, and debugging applications using the open source GNU debugger (GDB) running along with the EDK custom on-chip debugger.

This chapter assumes that you are using the Xilinx® Platform Studio™ (XPS), an integrated development environment included with EDK. This chapter contains the following sections.

Overview

XPS is an integrated design environment (IDE) used to develop EDK-based system designs. A simple "Hello World" example is used to demonstrate the flow involved in building EDK libraries, creating and compiling applications, and debugging the application using a debugger. The software flow is processor-independent and is applicable to both MicroBlaze™ and PowerPC™.

Assumptions

This chapter assumes that you:

- Have created a valid EDK project for the Hello World example in `C:\Data\HelloWorld_System`
- Have created a valid hardware platform in the Hello World example project (`C:\Data\HelloWorld_System\`) using the procedure described in Chapter 2, "Creating a Basic Hardware System in XPS"
- Are familiar with C programming language and using GNU tools

Steps

The following are the steps involved in generating software for EDK designs using XPS:

1. Configure Software Settings
2. View and Set Project Options
3. Create EDK Software Libraries
4. Open / Create Your Application(s)
5. View and Set Application Options
6. Build Applications
7. Initialize Bitstreams with Applications
8. Download and Execute Your Application
9. Download and Debug Applications Using XMD

These steps are explained in detail in the following sections.

Configure Software Settings

1. Start XPS from **Start** → **Programs** → **Xilinx Embedded Development Kit 7.1** → **Xilinx Platform Studio**.
2. Open the Hello World example project by either clicking the **Open Project** toolbar button or by selecting **File** → **Recent Projects**.

   The tree view of the project, displayed on the left side of the XPS main window, includes references to a variety of project-related files. These are grouped together in the following general categories:

   - **System BSP** — This category defines the hardware platform used in the project. The hardware platform includes processors, buses, and peripherals. Double-click anywhere inside the category to open the Software Settings dialog box
   - **Project Files** — This category includes all project-specific files. The file types are MHS, MSS, PBD, and UCF. For more information, refer to the "Xilinx Platform Studio (XPS)" chapter in the *Embedded System Tools Reference Manual*.
   - **Project Options** — This category includes all project-specific options. These options include Device, Netlist, Implementation, HDL, and Sim Model. For more information on these options, refer to the "Xilinx Platform Studio (XPS)" chapter in the *Embedded System Tools Reference Manual*.

   You can configure software settings for an EDK project using XPS or by editing the MSS file.

3. Right-click on the processor name in the tree view and select **S/W settings**.

   This opens the Software Platform Settings dialog box. The top half of the window displays various devices in the hardware platform and the drivers assigned to them. In the bottom-right side of the window, you can select an operating system (OS). By default, **standalone** is selected. This means that there is not an operating system between the application software and the hardware platform. The application software can still use the device drivers and some basic libraries. Default drivers are assigned to the processor and each peripheral present in the hardware platform. The pull-down menus in the **Driver** and **Version** columns allow you to select other applicable drivers and driver versions.

4. To configure processor and peripheral drivers, click the Processor and Driver Parameters tab. Similarly, you can use the Library / OS Parameters tab to configure libraries and OS parameters. After you configure the software settings, click **OK**. The wizard creates the software settings MSS file as shown in the right-hand side of the XPS main window.

View and Set Project Options

1. In XPS, select **Options** → **Project Options**.

   The Project Options dialog box opens.

2. Set the appropriate Xilinx architecture, device size, package, and speed grades for which the current project is targeted.

3. If custom drivers are used in the system, specify the appropriate path in the **My Peripheral Repository Directory** field.

   By default the Hello World project does not need an entry for this section.

4. Click **OK**.

Create EDK Software Libraries

In this step, you generate software libraries for the Hello World project.

In XPS, select **Tools** → **Generate Libraries and BSPs**. This invokes the tool for generating software libraries, Libgen.

The software library for the Hello World project is created in this project area:

```
C:\Data\HelloWorld_Syste\microblaze_0\lib\libxil.a
```

The address map of the system is created in this header file:

```
C:\Data\HelloWorld_Syste\microblaze_0\include\xparameters.h
```

Open/Create Your Application(s)

1. In XPS, click the Applications tab in the left panel.

   A tree view opens with Software Projects on top and all software applications for the processor below it.

2. Right-click **Software Projects** and select **Add New Project** to create a new software project for the processor in the system.

   This allows you to enter a name for this empty project. The tree view updates to display the new project.

3. Add the following C file, `hello.c`, as one of the sources to the hello_world_app.

```
/*-----------------------------------------------------------------
  *  Filename : hello.c
  *  Desc : Prints "Hello World" on the STDOUT device
  * ----------------------------------------------------------------*/
#include "xparameters.h"
int main() {
  xil_printf("Hello World \n");
}
```

4. Create the file `hello.c` using a text editor.
5. Right-click **Sources** for the `hello_world_app` project and select **Add File**. This opens the file browser to select the appropriate file. Select `hello.c` from the appropriate location.

   Caution! Adding a source to your project does not cause any file to be copied. This action only adds the path and file name to the project data.

View and Set Application Options

Now you must set the compiler options for building the hello_world_app. To do this, right-click on the project name in the tree view and select **Set Compiler Options**. A window opens with multiple tabbed panels for setting various compiler options. Select the **Executable** mode.

Build Applications

When you are done creating the application and setting the options, right-click on the project name in the tree view and select **Build Project** to create the executable.

Alternatively, select **Tools → Compile Program Sources** in XPS to build all of the applications.

Initialize Bitstreams with Applications

After building the applications, XPS allows you to select the application that must be initialized in the generated bitstream. To initialize the hello_world_app executable in the bitstream, right-click on the project name `hello_world_app` in the tree view and select **Mark to Initialize BRAMs**. From the XPS main window, select **Tools → Update Bitstream** to initialize the BRAMs with the hello_world_app executable information.

Download and Execute Your Application

After the bitstream is initialized with the hello_world_app executable, download the bitstream to the board. To download the application:

1. Set up the board and the parallel cable as required.
2. Set up the STDOUT device for display.

   For the sample HelloWorld_System, RS232 is used as the STDOUT device. Therefore, a Hyper terminal application opens with the appropriate Baud rate settings and is connected to the appropriate COM port.

To download and execute the `hello_world_app` application, select **Tools → Download** in the XPS main window. This downloads the bitstream onto the board. After downloading, `Hello World` displays on the Hyper terminal.

Download and Debug Applications Using XMD

To debug your application using the Xilinx® Microprocessor Debugger (XMD), you must set the appropriate compiler options in the application. To debug `hello_world_app`:

1. Right-click on the `hello_world_app` project name in the tree view of the Applications tab and select **Set Compiler Options**.

   This opens a window with multiple tabs for setting various compiler options.

2. Select **XmdStub** mode in order to debug the application.

Next, you must set the debug options.

1. In the Compiler Settings dialog box, click the Optimization tab.
2. From the Optimization panel, under Debug Options, select **Create symbols for debugging (-g option)**.
3. After selecting the debug option, build the hello_world_app application as detailed in "Build Applications" on page 28.
4. Download the bitstream and start XMD by selecting **Tools → XMD**.

   The XMD debugger starts. Connect XMD to the board.

   Refer to the "Xilinx Microprocessor Debugger (XMD)" chapter in the *Embedded System Tools Guide* for more information.

5. When XMD is connected to the board, select **Tools → Software Debugger** to start the GNU debugger.
6. The debugger window opens.

You can now download and debug the hello_world application using XMD and GDB. Refer to the "Xilinx Microprocessor Debugger (XMD)" chapter in the *Embedded System Tools Guide* for more information.

Chapter 4

Address Management

This chapter describes address management techniques for the embedded processor program. For advanced address space management, this chapter also discusses linker scripts.

This chapter contains the following sections.

MicroBlaze Processor

Programs and Memory

In MicroBlaze™, you can write either C, C++, or Assembly programs, and use the Embedded Development Kit (EDK) to transform your source code into bit patterns stored in the physical memory of a EDK System. Your programs typically access local and on-chip memory, external memory, and memory-mapped peripherals. Memory requirements for your programs are specified in terms of how much memory is required for storing the instructions and how much memory is required for storing the data associated with the program.

MicroBlaze address space is divided between the system address space and user address space. In certain examples, you might need advanced address space management, which you can do with the help of linker a script, as described in "Linker Script" on page 38.

Current Address Space Restrictions

Memory and Peripherals Overview

MicroBlaze uses 32-bit addresses, and as a result it can address memory in the range 0 through 0xFFFFFFFF. MicroBlaze can access memory either through its Local Memory Bus (LMB) port or through the On-chip Peripheral Bus (OPB). The LMB is designed to be a fast access, on-chip block RAM (BRAM) memories only bus. The OPB represents a general purpose bus interface to on-chip or off-chip memories as well as other non-memory peripherals.

BRAM Size Limits

The amount of BRAM memory that can be assigned to the LMB address space or to each instance of an OPB mapped BRAM peripheral is limited. The largest supported BRAM memory size for Virtex™ / Virtex™E is 16 kB and for Virtex™-II it is 64 kB. It is important

to understand that these limits apply to each separately decoded on-chip memory region only. The total amount of on-chip memory available to a MicroBlaze system might exceed these limits. The total amount of memory available in the form of BRAMs is also FPGA device specific. Smaller devices of a given device family provide less BRAM than larger devices in the same device family.

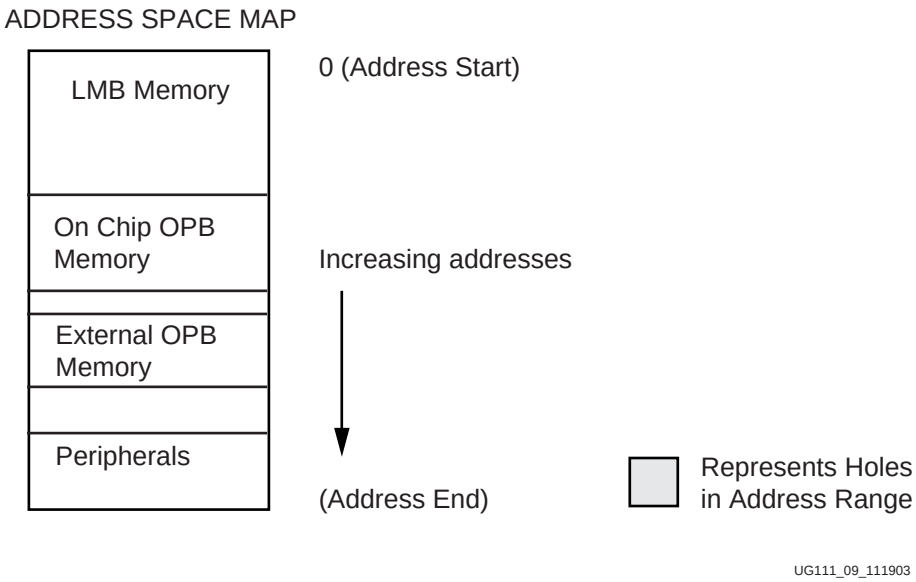

Figure 4-1: **Sample Address Map for a MicroBlaze System**

Special Addresses

Every MicroBlaze system must have user-writable memory present in addresses 0x00000000 through 0x00000028. These memory locations contain the addresses that MicroBlaze jumps to after a reset, interrupt, or exception event occurs. This memory can be part of the LMB or the OPB BRAM address space. Refer to the "MicroBlaze Application Binary Interface (ABI)" chapter in the *MicroBlaze Processor Reference Guide* for further details.

OPB Address Range Details

Within the OPB address space, you can arbitrarily assign address space to on / off-chip memory peripherals and to on / off-chip non-memory peripherals. The OPB address space might contain holes representing regions that are not associated with any OPB peripheral. Special linker scripts and directives might be required to control the assignment of object file sections to address space regions.

Address Map

Figure 4-1 shows a possible address map for a MicroBlaze System. The MicroBlaze Hardware Specification (MHS) file contains an address map specifying the addresses of LMB memory, OPB memory, external memory, and peripherals.

The address range grows from 0. At the lowest range is the LMB memory. This is followed by the OPB memory, external memory and the Peripherals. Some addresses in this address space have predefined meaning. The processor jumps to address 0x0 on reset, to address 0x8 on exception, and to address 0x10 on interrupt.

Memory Speeds and Latencies

MicroBlaze requires two clock cycles to access on-chip Block RAM connected to the LMB for write and two clock cycles for read. On chip memory connected to the OPB bus requires three cycles for write and four cycles for read. External memory access is further limited by off-chip memory access delays for read access, resulting in five to seven clock cycles for read. Furthermore, memory accesses over the OPB bus might incur further latencies due to bus arbitration overheads. As a result, instructions or data that must be accessed quickly should be stored in LMB memory when possible.

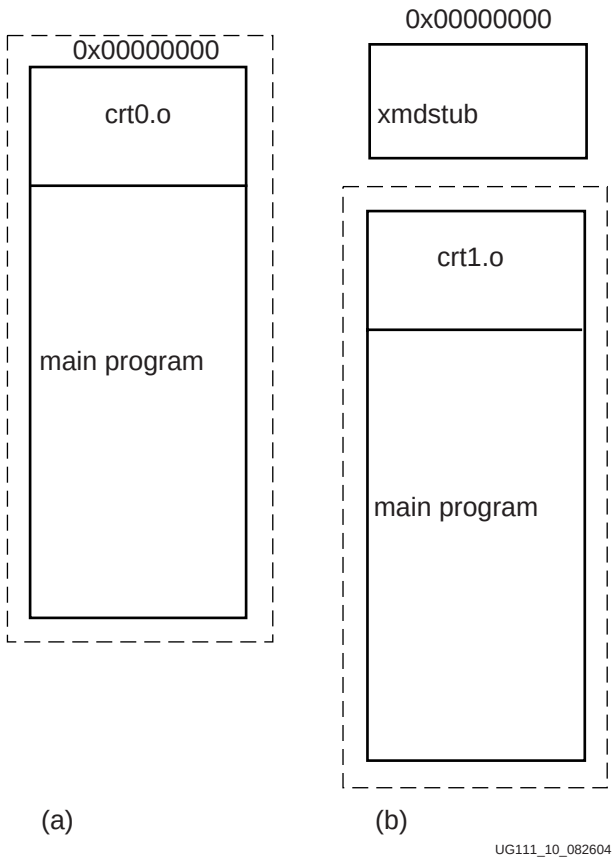

Figure 4-2: **Execution Scenarios**

For more information on memory access times, refer to the *MicroBlaze Reference Guide*.

System Address Space

MicroBlaze programs can be executed in different scenarios. Each scenario needs a different set of system address space. The system address space might be occupied by the XMDStub when software intrusive debug is required. System address space is also needed by the C-runtime routines.

System with Only an Executable (No Software Intrusive Debug)

This scenario is depicted in Figure 4-2(a). The C-runtime file `crt0.o` is linked with your program. The system file `crt0.o` starts at address location 0x0, immediately followed by your program.

System with Software Intrusive Debugging Support

With systems requiring debug support, XMDStub must be downloaded at address location 0x0. The C-runtime file `crt1.o` is bundled with your program and is placed at a default location. This scenario is shown in Figure 4-2(b).

Default User Address Space

The default usage of the compiler mb-gcc places your program immediately after the system address space. You do not need to give any additional options in order to make space for the system files. The default start address for your program is described in Table 4-1.

Table 4-1: **Start Address for Compilation Switches**

Compile Option	Start Address
-xl-mode-executable	0x0
-xl-mode-xmdstub	0x800

If you need to start the program at a location other than the default start address or if you need non-contiguous address space, you must use advanced address space management, as described in the next section.

Advanced User Address Space

Different Base Address, Contiguous User Address Space

Your program can run from any memory; that is, LMB memory or OPB memory. By default, the compiler places your program at a location defined in Table 4-1. To run a program from any address location other than default, you must provide the compiler mb-gcc with an additional option.

The option required is:

`-Wl,defsym -Wl,_TEXT_START_ADDR=`*`start_address`*

where *start_address* is the new base address required for your program.

Different Base Address, Non-Contiguous User Address Space

You can place different components of your program on different memories. For example, on MicroBlaze systems with non-contiguous LMB and OPB memories, you can keep your code on LMB memory and the data on OPB memory. You can also create systems which have contiguous address space for LMB and OPB memory, but with holes in the OPB address space.

For all such programs, you must create non-contiguous executable files. To facilitate your creation of non-contiguous executable files, you must modify linker scripts. The default linker script provided with the MicroBlaze Distribution Kit places all of your code and data in one contiguous address space.

Linker scripts are defined in later sections in this document.

For more details on linker options, refer to the "GNU Compiler Tools" chapter in the *Embedded System Tools Reference Manual*.

Object-File Sections

The sections of an executable file are created by concatenating the corresponding sections in an object (`.o`) file. The various sections of the object file are displayed in Figure 4-3.

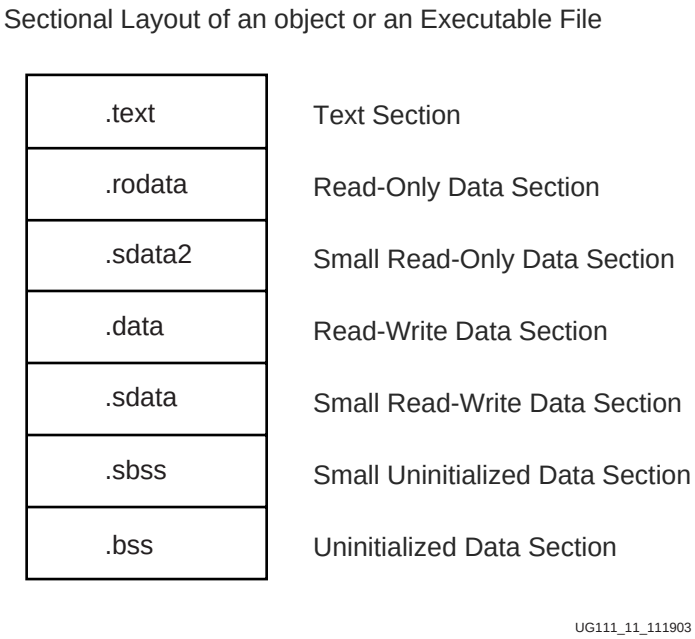

Figure 4-3: **Sectional Layout of an Object or Executable File**

.text

This section of the object file contains executable code. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags.

.rodata

This section contains read-only data of a size more than a specified size; the default is 8 bytes. You can change the size of the data put into this section with the **`mb-gcc -G`** option. All data in this section is accessed using absolute addresses. This section has the `r` (read-only) and the `i` (initialized) flags. For more details, refer to the "MicroBlaze Application Binary Interface (ABI)" chapter in the *MicroBlaze Processor Reference Guide*.

.sdata2

This section contains small read-only data of size less than 8 bytes. You can change the size of the data going into this section with the **`mb-gcc -G`** option. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all data in the `.sdata2` section can be accessed using a single instruction; therefore, a preceding imm instruction is never necessary. This section has the `r` (read-only) and the `i` (initialized) flags. For more details refer to the "MicroBlaze Application Binary Interface (ABI)" chapter in the *MicroBlaze Processor Reference Guide*.

.data

This section contains read-write data of a size more than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`mb-gcc -G`** option. All data in this section is accessed using absolute addresses. This section has the `w` (read-write) and the `i` (initialized) flags.

.sdata

This section contains small read-write data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`mb-gcc -G`** option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all data in the `.sdata` section uses a single instruction; therefore, a preceding imm instruction is never necessary. This section has the `w` (read-write) and the `i` (initialized) flags.

.sbss

This section contains small un-initialized data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`mb-gcc -G`** option. This section has the `w` (read-write) flag.

.bss

This section contains un-initialized data of a size more than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`mb-gcc -G`** option. All data in this section is accessed using absolute addresses. The stack and the heap are also allocated to this section. This section has the `w` (read-write) flag.

The linker script describes the mapping between all of the sections in all of the input object files, and the output executable file.

Note: If your address map specifies that the LMB, OPB, and external memory occupy contiguous areas of memory, you can use the default built-in linker script to generate your executable. To do this, you invoke mb-gcc as follows:

```
mb-gcc file1.c file2.c
```

Using the built-in linker script implies that you have no control over which parts of your program are mapped to the different kinds of memory. The default scripts used by the linker are located at:

<installation directory>/gnu/microblaze/*<platform>*/microblaze/lib/ldscripts where *<installation directory>* points to the installation root of EDK. *<platform>* = nt, sol, or lin These scripts are imbibed into the linker; therefore, any changes to these scripts are not reflected. To customize linker scripts, you must write your own linker script or select **Tools → Generate Linker Script**.

Minimal Linker Script

If your LMB, OPB, and external memory do not occupy contiguous areas of memory, you can use a minimal linker script to define your memory layout. Here is a minimal linker script that describes the memory regions only, and uses the default built-in linker script for everything else.

```
/*
* Define the memory layout, specifying the start address and size of the
* different memory regions. The instruction-side local memory bus
(ILMB) will contain only executable code (x),
* the data-side local memory bus (DLMB) will contain only initialized
data (i), and the data-side on-chip peripheral bus (DOPB) will contain
* all other writable data (w). All sections of all your input
* object files must map into one of these memory regions. Other memory
types
* that can be specified are "r" for read-only data.
*/
MEMORY
  {
    ILMB (x) : ORIGIN = 0x0, LENGTH = 0x1000
    DLMB (i) : ORIGIN = 0x2000, LENGTH = 0x1000
    DOPB (w) : ORIGIN = 0x8000, LENGTH = 0x30000
  }
```

This script specifies that the ILMB memory contains all object file sections that have the x flag, the DLMB contains all object file sections that have the i flag, and the DOPB contains all object file sections that have the w flag. An object file section that has both the x and the i flags, such as the .text section, is loaded into ILMB memory because this is specified first in the linker script. Refer to "Object-File Sections" on page 35 for more information on object file sections and the flags that are set in each.

You can now compile your source files by specifying the minimal linker script as though it were a regular file. For example:

```
mb-gcc minimal linker script file1.c file2.c
```

Remember to specify the minimal linker script as the first source file.

If you want more control over the layout of your memory, you must write a full-fledged linker script. For example, you might want to split up your .text section between ILMB and instruction-side on-chip peripheral bus (IOPB) or you might want your stack and heap in DLMB and the rest of the .bss section in DOPB.

Linker Script

You must use a linker script if you want to control how your program is targeted to LMB, OPB, or external memory. LMB memory is faster than both OPB and external memory. You might want to keep the portion of your code that is accessed the most frequently in LMB memory, and that which is accessed the least frequently in external memory.

You must provide a linker script to mb-gcc using the following command:

```
mb-gcc -Wl,-T -Wl,linker_script file1.c file2.c -save-temps
```

This tells mb-gcc to use your linker script only, and to not use the default built-in linker script.

The linker script defines the layout and the start address of each of the sections for the output executable file. Here is a sample linker script.

```
/*
* Define the memory layout, specifying the start address and size of the
* different memory regions.
*/
MEMORY
  {
    LMB : ORIGIN = 0x0, LENGTH = 0x1000
    OPB : ORIGIN = 0x8000, LENGTH = 0x5000
  }


/*
* Specify the default entry point to the program
*/
ENTRY(_start)

/*
* Define the sections, and where they are mapped in memory
*/
SECTIONS
{

/*
* Specify that the .text section from all input object files will be
* placed in LMB memory into the output file section .text
* mb-gdb expects the executable to have a section called .text
*/
.text : {
/* Uncomment the following line to add specific files in the opb_text */
/* region */
   /*    *(EXCLUDE_FILE(file1.o).text) */
   /* Comment out the following line to have multiple text sections */

   *(.text)
  } >LMB


  /* Define space for the stack and heap */
  /* variables _heap must be set to the beginning of this area */
  /* and _stack set to the end of this area */

  . = ALIGN(4);
  _heap = .;
```

```
  .bss : {
_HEAP_SIZE = 0x400;

. += _HEAP_SIZE;
_heap_end = .;
    _STACK_SIZE = 0x400;
    . += _STACK_SIZE;
    . = ALIGN(4);
  } >LMB
  _stack = .;

  /*                      */
  /* Start of OPB memory */
  /*                      */

  .opb_text : {
   /* Uncomment the following line to add an executable section into */
    /* opb memory */
    /*    file1.o(.text) */
  } >OPB

  . = ALIGN(4);
  .rodata : {
    *(.rodata)
  } >OPB

 /* Alignments by 8 to ensure that _SDA2_BASE_ on a word boundary */
  . = ALIGN(8);
  _ssrw = .;
  .sdata2 : {
    *(.sdata2)
  } >OPB
  . = ALIGN(8);
  _essrw = .;
  _ssrw_size = _essrw - _ssrw;
 _SDA2_BASE_ = _ssrw + (_ssrw_size / 2 );

  . = ALIGN(4);
  .data : {
    *(.data)
  } >OPB

  /* Alignments by 8 to ensure that _SDA_BASE_ on a word boundary */
  /* .sdata and .sbss must be contiguous */

  . = ALIGN(8);
  _ssro = .;
  .sdata : {
    *(.sdata)
  } >OPB
  . = ALIGN(4);
  .sbss : {
 __sbss_start = .;
    *(.sbss)
 __sbss_end = .;
  } >OPB
  . = ALIGN(8);
  _essro = .;
  _ssro_size = _essro - _ssro;
```

```
  _SDA_BASE_ = _ssro + (_ssro_size / 2 );

  . = ALIGN(4);
.opb_bss : {
  __bss_start = .;
    *(.bss) *(COMMON)
  . = ALIGN(4);
  __bss_end = .;
  } > OPB

  _end = .;
}
```

If you choose to write a linker script, you must do the following to ensure that your program will work correctly. This example linker script incorporates these restrictions. Each restriction is highlighted in the example linker script.

- Allocate space in the `.bss` section for stack and heap. Set the `_heap` variable to the beginning of this area, _heap_end to the end of heap area (_heap + _HEAP_SIZE), and the `_stack` variable to the end of this area. See the `.bss` section in the preceding script for an example. Stack and Heap can have their own spaces in terms of _STACK_SIZE and _HEAP_SIZE size variables, but the main restriction is that both stack and heap should be assigned to same memory contiguously as shown in the previous example.
- Ensure that the _SDA2_BASE_ variable points to the center of the `.sdata2` area, and that _SDA2_BASE_ is aligned on a word boundary. See the previous example to view how this is done.
- Ensure that the `.sdata` and the `.sbss` sections are contiguous, that the _SDA_BASE_ variable points to the center of this section, and that _SDA_BASE_ is aligned on a word boundary. See the previous example to view how this is done.
- If you are not using the XMDStub, ensure that `crt0` is always loaded into memory address 0. The compiler mb-gcc ensures that this is the first file specified to the loader, but the loader script needs to ensure that it gets loaded at address 0. See the `.text` section in the previous example to view how this is done.
- Ensure that the `__sbss_start`, `_sbss_end`, `__bss_start`, and `__bss_end` variables are defined to the start and end of the `.sbss` and `.bss` sections, respectively. See the `.bss` and `.sbss` sections in the previous example to view how this is done.
- Ensure that the `.bss` and `.common` sections from input files are contiguous. ANSI C requires that all uninitialized memory be initialized to startup. This is not required for stack and heap. The standard `crt0.s` that Xilinx provides assumes a single `.bss` section that is initialized to 0. If there are multiple `.bss` sections, this crt will not work. You should write your own crt that initializes all of the `.bss` sections.
- To minimize your simulation time, make sure to point your `__bss_end` immediately after your declarations of all the `.bss`, and `.common` sections from input files. See the `.opb_bss` section in the previous example to view how this is done.

For more details on the linker scripts, refer to the GNU loader documentation in the binutil online manual, located at http://www.gnu.org/manual.

PowerPC Processor

Programs and Memory

In PowerPC™, you can write either C, C++, or Assembly programs, and use EDK to transform your source code into bit patterns stored in the physical memory of the EDK System. Your programs typically access local / on-chip memory, external memory, and memory-mapped peripherals. Memory requirements for your programs are specified in terms of how much memory is required for storing the instructions, and how much memory is required for storing the data associated with the program.

Figure 4-4 shows a sample address map for a PowerPC-based EDK system. It shows that there can be various memories in the system. Here you must use advanced address space management, which you can do with the help of a linker script, as described in "Linker Script" on page 38.

Current Address Space Restrictions

Special Addresses

Every PowerPC system should have the boot section starting at 0xFFFFFFFC.

Default Linker Options

By default, the linker assumes that the program can occupy contiguous address space from 0xFFFF0000 to 0xFFFFFFFF. It also assumes a default stack size of 4 kB, and a default heap size of 4 kB.

To change the size of the allocated stack space, provide the following option to the compiler powerpc-eabi-gcc:

`-Wl,defsym -Wl,_STACK_SIZE=`*`stack_size`*

where *stack_size* is the required stack size in bytes.

To change the size of the allocated heap space, provide the following option to the compiler powerpc-eabi-gcc:

`-Wl,defsym -Wl,_HEAP_SIZE=`*`heap_size`*

where *heap_size* is the required heap size in bytes.

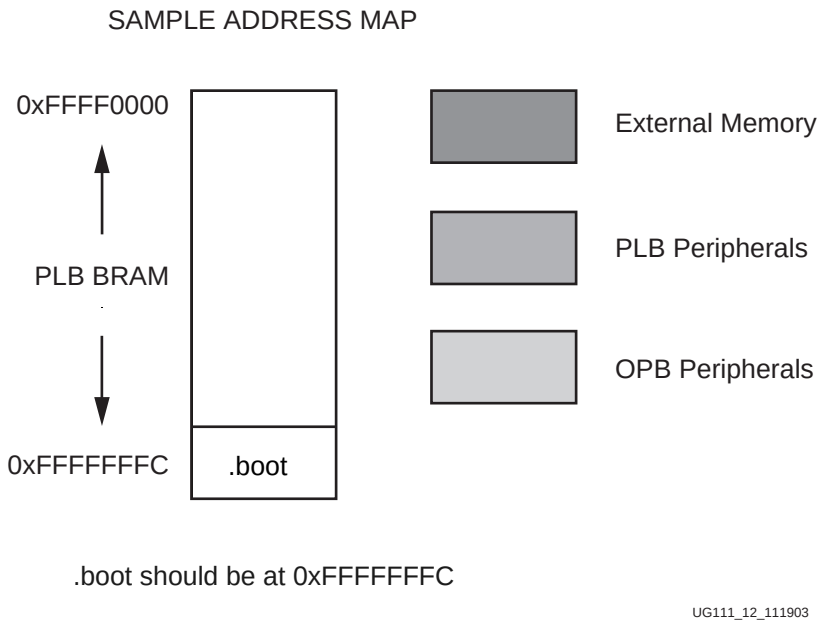

Figure 4-4: **Sample Address Map for a PowerPC System**

Advanced User Address Space

Different Base Address, Contiguous User Address Space

Your program can run from any memory. By default, the compiler places your program at location 0xFFFF0000. To run the program from any address location other than the default, you must provide the compiler powerpc-eabi-gcc with an additional option.

The option required is:

`-Wl,-defsym -Wl,_START_ADDR=`*`start_address`*

where *start_address* is the new base address required for your program.

Different Base Address, Non-Contiguous User Address Space

You can place different components of your program in different memories. For example, on PowerPC systems, you can keep your code in instruction cache memory and the data in Zero Bus Turnaround™ (ZBT) memory.

For all such programs, you must create non-contiguous executable files. To facilitate creation of non-contiguous executable files, you must modify linker scripts. The default linker script provided with EDK places all of your code and data in one contiguous address space.

Linker scripts are defined in the following section.

For more details on linker options, refer to the "GNU Compiler Tools" chapter in the *Embedded System Tools Reference Manual*.

Linker Script

PowerPC Linker is built with default linker scripts. These scripts assume a contiguous memory starting from address 0xFFFF0000. The script defines `boot.o` as the first file to be linked. The `boot.o` file is present in the `libxil.a` library, which is created by the library generator tool, Libgen. The script defines the start address as 0xFFFF000 if you have given the start address through the linker option as:

```
-Wl, -defsym -Wl,_START_ADDRESS=0xFFFF8000
```

In this case, the start address is 0xFFFF8000. The script starts assigning addresses to different sections of the final executable: `.vectors`, `.text`, `.rodata`, `.sdata2`, `.sbss2`, `.data`, `.got1`, `.got2`, `.fixup`, `.sdata`, `.sbss`, `.bss`, `.boot0`, and `.boot` in that order. As it assigns the addresses, the script defines the following start and end of sections variables: `__SDATA2_START__`, `__SDATA2_END__`, `__SBSS2_START__`, `__SBSS2_END__`, `__SDATA_START__`, `__SDATA_END__`, `__sbss_start`, `___sbss_start`, `__sbss_end`, `___sbss_end`, `__SDATA_START__`, `__SDATA_END__`, `__bss_start`, and `__bss_end`. These variables define the sectional boundaries for each of the sections. Stack and heap are allocated from the `.bss` section. They are defined through `__stack`, `__heap_start`, and `__heap_end`. The bss section boundary does not include either of stack or heap. The variable `_end` is defined after the `.boot0` section definition.

The `.boot` section is fixed to start at location 0xFFFFFFFC. This section is a jump to the start of the `.boot0` section. The jump is defined to be 24 bits; therefore, the `boot` and `boot0` sections should not have a difference of more than 24 bits. The `.boot` section is at 0xFFFFFFFC because of the fact that the PowerPC405 processor, on powerup, starts running from the location 0xFFFFFFFC.

You can review the default linker scripts used by the linker at:

$XILINX_EDK*`/gnu/powerpc-eabi/nt(or sol)/powerpc-eabi/lib/ldscripts`, where *$XILINX_EDK is the EDK installed directory. These scripts are imbibed into the linker; therefore, any changes to these scripts will not be reflected.

The choices of the default script that will be used by the linker from the *$XILINX_EDK*`/gnu/powerpc-eabi/nt(orsol)/powerpc-eabi/lib/ldscripts` area are described as below:

- **`elf32ppc.x`** is used by default when none of the following cases apply
- **`elf32ppc.xn`** is used when the linker is invoked with the **`-n`** option.
- **`elf32ppc.xbn`** is used when the linker is invoked with the **`-N`** option.
- **`elf32ppc.xr`** is used when the linker is invoked with the **`-r`** option.
- **`elf32ppc.xu`** is used when the linker is invoked with the **`-Ur`** option.
- **`elf32ppc.x`** is used when the linker is invoked with the **`-n`** option.

For a more detailed explanation of the linker options, refer to the GNU linker documentation, available online at http://www.gnu.org/manual.

Minimal Linker Script

The Need for a Linker Script

You must write a linker script if you want to control how your program is targeted to instruction cache, ZBT, or external memory.

You must provide a linker script to powerpc-eabi-gcc using the following command:

`powerpc-eabi-gcc -Wl,-T -Wl,`*`linker script file1.c file2.c`*` -save-temps`

This command tells powerpc-eabi-gcc to use your linker script only, and to not use the default built-in one. The Linker Script defines the layout and the start address of each of the sections for the output executable file.

Restrictions

If you choose to write a linker script, you *must* do the following to ensure that your program will work correctly. The following example linker script incorporates these restrictions. Each restriction is highlighted in the example linker script.

- Allocate space in the `.bss` section for stack and heap. Set the `_stack` variable to the location after `_STACK_SIZE` locations of this area, and the `_heap_start` variable to the next location after the `_STACK_SIZE` location. Since the stack and heap need not be initialized for hardware as well as simulation, define the `__bss_end` variable after the `.bss` and `.common` definitions. Refer to the `.bss` section in the following example script to see how this is done.
- Ensure that the variables `__SDATA_START__`, `__SDATA_END__`, `SDATA2_START`, `__SDATA2_END__`, `__SBSS2_START__`, `__SBSS2_END__`, `__bss_start`, `__bss_end`, `__sbss_start`, and `__sbss_end` are defined to the beginning and end of the sections `sdata`, `sdata2`, `sbss2`, `bss`, and `sbss` respectively. Refer to the following example to see how this is done.
- Ensure that the `.sdata` and the `.sbss` sections are contiguous.
- Ensure that the `.sdata2` and the `.sbss2` sections are contiguous.
- Ensure that the `.boot` section starts at 0xFFFFFFFC.
- Ensure that `boot.o` is the first file to be linked. Check the STARTUP `boot.o` in the following script, which achieves this.
- Ensure that the `.vectors` section is aligned on a 64 k boundary. In order to ensure this, make `.vectors` the first section definition in the linker script. The memory where `.vectors` is assigned to should start on a 64 k boundary. Include this section definition only when your program uses interrupts or exceptions. See the following example script to view how this is done.
- Each physical region of memory must use a separate program header. Two discontinuous regions of memory cannot share a program header.
- Put all uninitialized sections (`.bss`, `.sbss`, `.sbss2`, `stack`, `heap`) at the end of a memory region. If this is impossible, such as if `.sdata` and `.sbss` are in the same physical memory as `.sdata2` and `.sbss2`, start a new program header for the first initialized section after all uninitialized sections.
- ANSI C requires that all uninitialized memory be initialized to startup. This is not required for stack and heap. The standard `crt0.s` that Xilinx provides assumes a single `.bss` section that is initialized to 0. If there are multiple `.bss` sections, this crt will not work. You should write your own crt that initializes all of the `.bss` sections.

For more details on the linker scripts, refer to the GNU loader documentation in the binutil online manual, available at http://www.gnu.org/manual.

The following is a sample linker script.

```
/*
 * Define default stack and heap sizes
 */

STACKSIZE       = 1k;
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 4k;

/*
 * Define boot.o to be the first file for linking.
 * This statement is mandatory.
 */

STARTUP(boot.o)

/* Specify the default entry point to the program */
ENTRY(_boot)

/*
 * Define the Memory layout, specifying the start address
 * and size of the different memory locations
 */

MEMORY
{
  bram  : ORIGIN = 0xffff8000, LENGTH = 0x7fff
  boot  : ORIGIN = 0xfffffffc, LENGTH = 4
}

/*
 * Define the sections and where they are mapped in memory
 * Here .boot sections goes into boot memory. Other sections
 * are mapped to bram memory.
 */

SECTIONS
{
/*
 * .vectors section must be aligned on a 64k boundary
 * Hence should be the first section definition as bram start location
is 64k aligned
*/

  .vectors :
  {
    *(.vectors)
  } > bram

  .boot0   : { *(.boot0)} > bram
  .text    : { *(.text) } > bram
  .boot    : { *(.boot) } > boot
  .data :
  {
    *(.data)
    *(.got2)
```

```
    *(.rodata)
    *(.fixup)
  } > bram


  /* small data area (read/write): keep together! */
.sdata : { *(.sdata) } > bram
.sbss :
  {
    . = ALIGN(4);
    *(.sbss)
    . = ALIGN(4);
  } > bram
  __sbss_start = ADDR(.sbss);
  __sbss_end   = ADDR(.sbss) + SIZEOF(.sbss);

/* small data area 2 (read only) */
  .sdata2 :  { *(.sdata2) } > bram
__SDATA2_START__ = ADDR(.sdata2);
__SDATA2_END__ = ADDR(.sdata2) + SIZEOF(.sdata2);

.sbss2 :  { *(.sbss2) } > bram
  __SBSS2_START__ = ADDR(.sbss2);
  __SBSS2_END__ = ADDR(.sbss2) + SIZEOF(.sbss2);

.bss     :
  {
    . = ALIGN(4);
    *(.bss)
    *(COMMON)
/* stack and heap need not be initialized and hence bss end is declared
here */
. = ALIGN(4);

__bss_end   = .;

    /* add stack and heap and align to 16 byte boundary */
    . = . + STACKSIZE;
    . = ALIGN(16);
    __stack = .;
    _heap_start = .;
    . = . + _HEAP_SIZE;
    . = ALIGN(16);
    _heap_end = .;
  } > bram
  __bss_start = ADDR(.bss);
}
```

Chapter 5

Interrupt Management

This chapter details the interrupt handling in MicroBlaze™ and PowerPC™, and the role of Libgen for MicroBlaze and PowerPC. The chapter contains the following sections.

Interrupt Management

Prior to the EDK 6.2 release, there were two levels of interrupt management possible using EDK based on the levels of drivers. Interrupt management in EDK 6.2 unifies the different level-based interrupt management into a single flow. This interrupt handling mechanism works only for interrupt controller driver **intc v1.00.c.** The interface functions of the interrupt management remains unchanged; therefore, your code remains unchanged. For any interrupt controller driver prior to version v1.00.c, refer to the *Interrupt Management* chapter in EDK 6.1 release.

Interrupt handling explained in this document is specific to the interrupt controller driver **intc v1.00.c**.

MicroBlaze Interrupt Management

Overview

This section describes interrupt management for MicroBlaze. Interrupt Management involves writing interrupt handler routines for peripherals and setting up the MHS and MSS files appropriately. MicroBlaze has one interrupt port. An interrupt controller peripheral is required for handling more than one interrupt signal.

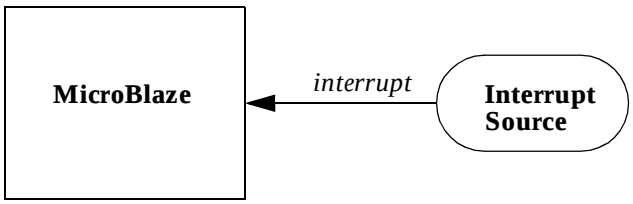

Figure 5-1: **MicroBlaze Connected to an Interrupt Source**

Figure 5-1 shows MicroBlaze connected to an interrupt source. The *interrupt* port is connected to the interrupt port of MicroBlaze. On interrupts, MicroBlaze jumps to address location 0x10. This is part of the C Runtime library and contains a jump to the default interrupt handler (_interrupt_handler). This function is part of the MicroBlaze Board Support Package (BSP), which is provided by Xilinx. It accesses an interrupt vector table to determine the name of the interrupt handler for the Interrupt Source. The interrupt vector table is a single entry table. The entry is a combination of the interrupt service routine (ISR) and an argument that should be used with the ISR. This entry can be programmed in your code. Functions are provided in the MicroBlaze BSP to change the handler of the Interrupt Source at run time. The Interrupt Source could be any of the following:

- Interrupt Controller Peripheral
- Peripheral with an Interrupt Port
- External Interrupt Port
- Interrupt Handlers
- Interrupt Vector Table in MicroBlaze
- Interrupt Routines in MicroBlaze

Each of these cases is explained in detail in the following sections.

Interrupt Controller Peripheral

An interrupt controller peripheral should be used for handling multiple interrupts. In this case, you are responsible for writing interrupt handlers for the peripheral interrupt signals only. The interrupt handler for the interrupt controller peripheral is automatically generated by Libgen. This handler ensures that interrupts from the peripherals are handled by individual interrupt handlers in the order of their priority. Figure 5-2 shows peripheral interrupt signals with priorities 1 through 4 connected to the interrupt controller input.

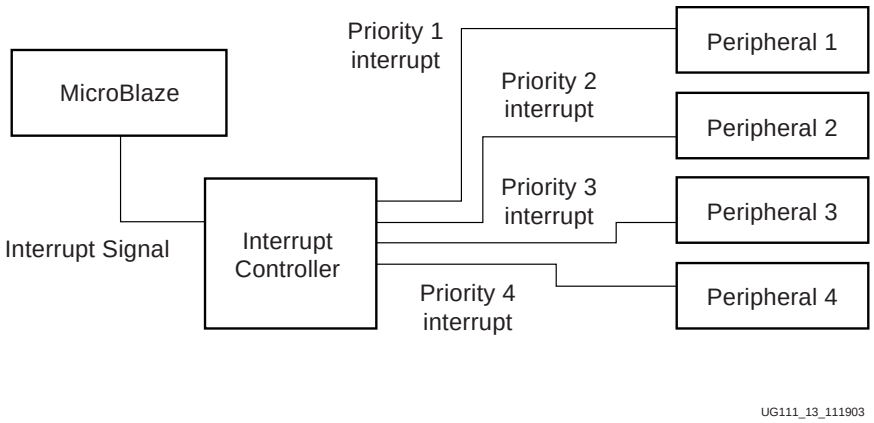

Figure 5-2: **Interrupt Controller and Peripherals**

The corresponding MHS snippet is as follows:

```
BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = Priority4_interrupt & Priority3_interrupt &
Priority2_interrupt & Priority1_interrupt
END

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

The interrupt signal output of the controller is connected to the interrupt input of MicroBlaze. The order of priority for each of the interrupt signals is defined from right to left, with the right most signal having the highest priority and the left most signal having the least priority as defined in the *Intr* port entry for interrupt controller in the MHS file snippet shown above.

On interrupts, MicroBlaze jumps to the `XIntc_DeviceInterruptHandler` handler of the interrupt controller peripheral using the interrupt vector table as defined in "MicroBlaze Interrupt Management" on page 47. The handler of the interrupt controller peripheral is automatically registered in the interrupt vector table by Libgen. The interrupt controller handler services each interrupt signal that is active starting from the highest priority signal. Each of the peripheral interrupt signal needs to be associated with an ISR. The interrupt controller handler uses a vector table to store these routines corresponding to each interrupt signal. If an interrupt is active, the interrupt controller handler calls the corresponding routine. An argument is passed when the routine is called. The vector table used by the interrupt controller handler is automatically generated by Libgen.

The association of an ISR for a peripheral interrupt signal can be done either in the MSS file or registered at run time using the function provided by the interrupt controller driver `XIntc_Connect, XIntc_RegisterHandler`. These functions work on the vector table generated by Libgen. For more information on the exact prototype of these functions, refer to the *Xilinx Device Drivers Documentation*. If the ISRs are specified in the MSS file, Libgen automatically registers these routines with the vector table of the interrupt controller driver listed in the order of priority. The base address of the peripherals are registered as the arguments to be passed to the ISR in the vector table. The following MSS snippet shows how to register the ISR for a peripheral interrupt signal:

```
BEGIN DRIVER
parameter HW_INSTANCE = Peripheral_1
parameter DRIVER_NAME = Peripheral_1_driver
parameter DRIVER_VER = 1.00.a
parameter INT_HANDLER = peripheral_1_int_handler, INT_PORT =
Priority1_Interrupt
END
```

Limitations

The following are the limitations for interrupt management using an interrupt controller peripheral:

- The priorities associated with the interrupt sources connected the interrupt controller peripheral are fixed statically at the time of definition in the MHS file. The priorities cannot be changed dynamically in your code.
- There cannot be any holes in the range of interrupt sources defined in the MHS file. For example, in the previous MHS file snippet, a definition such as the following is not acceptable:

```
port Intr = Priority4_interrupt & 0b0 & Priority2_interrupt ...
```

Peripheral with an Interrupt Port

A peripheral with an interrupt port can be directly connected to MicroBlaze as shown in Figure 5-2 on page 49. In this case, you are responsible for writing interrupt handler for the peripheral interrupt signal. The following MHS snippet describes the connectivity between MicroBlaze and a peripheral instance. In this example, the peripheral instance is called opb_timer.

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = interrupt
port CaptureTrig0 = net_gnd
END

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

On interrupts, MicroBlaze jumps to the handler of the timer peripheral using the interrupt vector table as defined in "MicroBlaze Interrupt Management" on page 47. If the timer peripheral's handler is specified in the MSS file, this routine is automatically registered in the interrupt vector table by Libgen. The MSS snippet for the timer peripheral is then similar to the following:

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

The base address of the timer peripheral is registered as the argument for the routine in the interrupt vector table. Alternately, this routine can be registered at run time in your code using the `microblaze_register_handler` function provided in the MicroBlaze BSP. Refer to the "Standalone Board Support Package" chapter in the *EDK OS and Libraries Reference Manual* for more specifics on the function prototype.

External Interrupt Port

An external interrupt pin can connect to the interrupt port of MicroBlaze. In this situation, the interrupt source is a global external interrupt. The following MHS snippet describes the connectivity between MicroBlaze and the global interrupt signal:

```
PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = LOW, SIGIS =
INTERRUPT

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt_in1
end
```

On interrupts, MicroBlaze jumps to the handler of the global external interrupt signal, using the interrupt vector table as defined in "MicroBlaze Interrupt Management" on page 47. If the global interrupt signal's handler is specified in the MSS file, this routine is automatically registered in the interrupt vector table by Libgen. In this case, the MSS snippet is similar to the following:

```
PARAMETER int_handler = globint_handler, int_port = interrupt_in1
```

A null value is registered as the argument for the routine in the interrupt vector table. Alternately, you can use the `microblaze_register_handler` function, provided in the MicroBlaze BSP, in your code to register the routine at run time. Refer to the "Standalone Board Support Package" chapter in the *EDK OS and Libraries Reference Manual* for more specifics on the function prototype.

Interrupt Handlers

You can write your own interrupt handlers, or ISRs, for any peripherals that raise interrupts. You write these routines in C as with other functions. You can give the interrupt handler function any name with the signature **void *func* (void *)**. Alternately, you can elect to register the handlers defined as a part of the drivers of the interrupt sources.

Interrupt Vector Table in MicroBlaze

On interrupts, MicroBlaze jumps to address location 0x10. This is part of the C Runtime library and contains a jump to the default interrupt handler `_interrupt_handler`. This function is part of the MicroBlaze BSP and is provided by Xilinx. It accesses an interrupt vector table to figure out the name of the interrupt handler for the Interrupt Source. The interrupt vector table is a single entry table. The entry is a combination of the ISR and an argument that should be used with the ISR. This entry can be programmed in your code using the interrupt routines in MicroBlaze BSP. The interrupt vector table is defined in the file `microblaze_interrupts_g.c` in the MicroBlaze BSP.

Interrupt Routines in MicroBlaze

The following are the interrupt-related routines defined in the MicroBlaze BSP.

MicroBlaze Enable and Disable Interrupts

The functions `microblaze_enable_interrupts` and `microblaze_disable_interrupts` are used to enable and disable interrupts on MicroBlaze. These functions are part of the MicroBlaze BSP and are described in the "Standalone Board Support Package" chapter in the *EDK OS and Libraries Reference Manual*.

MicroBlaze Interrupt Handler

The function `_interrupt_handler` is called whenever interrupt input of MicroBlaze becomes active. This function uses the interrupt vector table `MB_InterruptVectorTable` to jump to the interrupt handler registered in the table. This function is a part of the MicroBlaze BSP and is described in the "Standalone Board Support Package" chapter in the *EDK OS and Libraries Reference Manual*.

MicroBlaze Register Handler

The function `microblaze_register_handler` is used to register an interrupt handler with the MicroBlaze interrupt vector table. This function is a part of the MicroBlaze BSP and is described in the "Standalone Board Support Package" chapter in the *EDK OS and Libraries Reference Manual*.

PowerPC Interrupt Management

This section describes interrupt management for PowerPC.

Interrupt management involves writing interrupt handler routines for peripherals and setting up the MHS and MSS files appropriately. PowerPC has two interrupt ports: critical and non-critical interrupt ports. An interrupt controller peripheral is required for handling more than one interrupt signal.

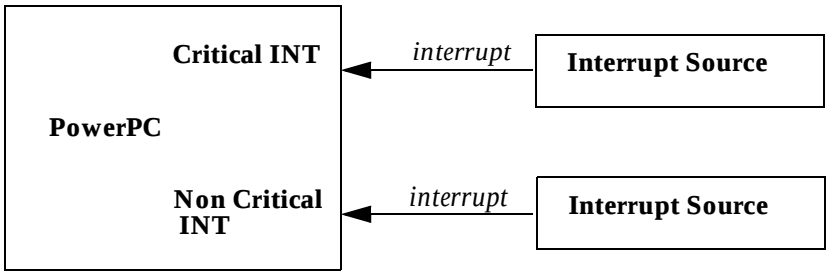

Figure 5-3: **PowerPC with Critical and Non-Critical Interrupts Connected to Interrupt Sources**

Figure 5-3 shows PowerPC connected to interrupt sources. The *interrupt* ports of the interrupt sources are connected to the critical and non-critical interrupt ports of PowerPC. On interrupts, PowerPC jumps to the handler registered in the exception table. You must register the handler of the interrupt source with the PowerPC exception table using the function `XExc_RegisterHandler` in the PowerPC BSP. This function is provided by Xilinx. The Interrupt Source connected to PowerPC could be any of the following:

- Interrupt Controller Peripheral
- Peripheral with an Interrupt Port
- External Interrupt Port

For PowerPC, the interrupt handler of either an interrupt controller (in systems using interrupt controller), or a peripheral/global port handler should be registered with the exception table. Registering such handler should be part of your code. The handler can be for either a non-critical interrupt port or a critical interrupt port based on the connection defined in the MHS file. The following example snippet shows how to register a handler for non-critical interrupts for PowerPC:

```
/* Initialize the ppc405 exception table*/
  XExc_Init();

/* Register the interrupt controller handler with the exception table*/
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
                       (XExceptionHandler)XIntc_DeviceInterruptHandler,
                       XPAR_OPB_INTC_0_DEVICE_ID);
```

Apart from the registering of the handler with the exception table, the rest of the processing for all the three interrupt sources previously listed are similar to the explanation in MicroBlaze sections.

Libgen Customization

Purpose of the Libgen Tool

The Libgen tool generates the address map of the hardware system defined. The address map defines the base and high addresses of each of the peripherals connected to the processor. It also generates interrupt priorities for each of the peripherals connected to an

interrupt controller peripheral. The information is generated in the header file `xparameters.h`. Based on the MSS file, Libgen does the following for interrupt management:

- Register a handler with the exception table for MicroBlaze
- If an interrupt controller peripheral is used, generate the vector table for the interrupt controller peripheral.
- Register handlers of each of the peripheral interrupt signal connected to the interrupt controller peripheral in the vector table, if defined in the MSS file.

Introducing xparameters.h

The `xparameters.h` file defines the hardware system that is used by the software. The file includes an address map of the hardware system which includes the base and high addresses of each of the peripherals connected to a processor. The naming conventions used by the tool for generating base and high addresses are:

```
XPAR_<PERIPHERAL_INSTANCE_NAME>_BASE_ADDR

XPAR_<PERIPHERAL_INSTANCE_NAME>_HIGH_ADDR
```

The interrupt controller driver uses the priorities and the maximum number of interrupt sources in a system using the definitions in `xparameters.h`. Libgen generates priorities for each of the interrupt signals as #defines in `xparameters.h` using the following naming conventions:

```
XPAR_<INTC_INSTANCE_NAME>_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_INTR

XPAR_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_MASK
```

For example, the `priority 1` interrupt is defined as

```
XPAR_OPB_INTC_0_PERIPHERAL_1_PRIORITY_1_INTERRUPT_INTR

XPAR_PERIPHERAL_1_PRIORITY_1_INTERRUPT_MASK
```

in `xparameters.h`, where `opb_intc_0` is the instance name of the interrupt controller peripheral.

Libgen also generates `XPAR_<INTC_INSTANCE_NAME>_MAX_NUM_INTR_INPUTS` to define the total number of interrupting sources connected to the interrupt controller peripheral, as displayed in Figure 5-2. The `INTR` definitions define the identification of the interrupting sources and should be in the range to `XPAR_<INTC_INSTANCE_NAME>_MAX_NUM_INTR_INPUTS - 1`, with 0 being the highest priority interrupt.

Example Systems for MicroBlaze

System *Without* Interrupt Controller (Single Interrupt Signal)

An interrupt controller is not required if there is a single interrupting peripheral or an external interrupting pin. A single peripheral might raise multiple interrupts. In this case, an interrupt controller is required.

Procedure

To set up a system without an interrupt controller that handles only one level sensitive interrupt signal, you must do the following:

1. Set up the MHS and MSS file as follows:
   - Connect the interrupt signal of the peripheral, or the external interrupt signal, to the interrupt input of the MicroBlaze in the MHS file.
   - Give the peripheral an instance name using the `INSTANCE` keyword in the MHS file.
     Libgen creates a definition in `xparameters.h (OUTPUT_DIR/PROC INST NAME/include)` for `XPAR_<INSTANCE_NAME>_BASEADDR` mapped to the base address of this peripheral.
2. Write the interrupt handler routine that handles the signal. The base address of the peripheral instance is accessed as `XPAR_<INSTANCE_NAME>_BASEADDR`.
3. Designate the handler function to be an interrupt handler for the signal using the `INT_HANDLER` keyword in the MSS file. Refer to the "Microprocessor Software Specification (MSS)" chapter in the *Platform Specification Format Reference Manual* for more information.
   The peripheral instance is first selected in the MSS file, and then the `INT_HANDLER` attribute is given the function name. In case of an external interrupt signal, the `INT_HANDLER` attribute is given as a global parameter in the MSS file. The attribute is not part of any block in the MSS.
4. Libgen and mb-gcc are executed. This operation has the following implications:
   - The function automatically registers with the exception table. This ensures that MicroBlaze calls the function on interrupts. By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.
   - On interrupts, MicroBlaze jumps to the handler function using the exception table.

Example MHS File Snippet

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = interrupt
port CaptureTrig0 = net_gnd
END

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr_p, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }

    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(baseaddr_p, 0, csr);

  }
}

void
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

/* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);
```

```
  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

  /* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

  /* Wait for interrupts to occur */
  while (1)
      ;

}
```

Example MHS File Snippet (For an External Interrupt Signal)

```
PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = LOW, SIGIS =
INTERRUPT

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt_in1
end
```

Example MSS File Snippet

```
PARAMETER int_handler = global_int_handler, int_port = interrupt_in1
```

Example C Program

```
#include <xparameters.h>

/* global interrupt service routine */
void global_int_handler(void * arg) {
/* Handle the global interrupts here */

}

void
main() {

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();
/* Wait for interrupts to occur */
  while (1)
      ;

}
```

System *With* an Interrupt Controller (One or More Interrupt Signals)

An Interrupt Controller peripheral (INTC) should be present if more than one interrupt can be raised. When an interrupt is raised, the interrupt handler for the INTC, `XIntc_DeviceInterruptHandler`, is called. This function then accesses the interrupt controller to find the highest priority device that raised an interrupt. This is done via the vector table created automatically by Libgen. On return from the peripheral interrupt handler, INTC acknowledges the interrupt. It then handles any lower priority interrupts, if they exist.

Procedure

To set up a system with one or more interrupting devices and an interrupt controller, you must complete the following steps:

1. Set up the MHS and MSS files as follows:
   - Assign the interrupt signals of all the peripherals to the interrupt port (Intr in most cases) of the interrupt controller in the MHS file.
     The interrupt signal output of INTC is then connected to the interrupt input of MicroBlaze.
   - Give the peripherals instance names using the INSTANCE keyword in the MHS file. Libgen creates a definition in `xparameters.h` for `XPAR_<INTC_INSTANCE_NAME>_BASEADDR` mapped to the base address of each peripheral for use in the user program. Libgen also creates an interrupt mask and interrupt ID for each interrupt signal with the priority settings `XPAR_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_MASK` and `XPAR_<INTC_INSTANCE_NAME>_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_INTR`. You can use this to enable or disable interrupts.
2. Write the interrupt handler functions for each interruptible peripheral.
3. Designate each handler function to be the handler for an interrupt signal using the INT_HANDLER keyword in the MSS file. Alternately, you can register the INTC interrupt vector table in your code. For this example, we showcase both of these use cases by setting the routine for timer in the MSS file and setting up the `uart` interrupt port handler in your code. Do not give the INTC interrupt signal an `INT_HANDLER` keyword. If the `INT_HANDLER` keyword is not present for a particular peripheral, a default dummy interrupt handler is used.
4. Run Libgen and mb-gcc to do the following:
   - Register the `XIntc_DeviceInterruptHandler` function as the main interrupt handler with the MicroBlaze exception table by Libgen.
     By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding `rtid` instruction is executed.
   - Automatically generate and compile an interrupt vector table in Libgen. Each of the peripherals connected to INTC can also register its interrupt handlers with the INTC interrupt handler.
   - Have `XIntc_DevicenterruptHandler` call the peripheral interrupt handler using the updated interrupt vector table to identify the handler in order of priority.

   Note: MicroBlaze jumps to `XIntc_DeviceInterruptHandler` using the exception table when an interrupt occurs.

Example MHS File Snippet

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = timer1
port CaptureTrig0 = net_gnd
END

BEGIN opb_uartlite
parameter INSTANCE = myuart
parameter HW_VER = 1.00.b
parameter C_BASEADDR  = 0xFFFF8000
parameter C_HIGHADDR = 0xFFFF80FF
parameter C_DATA_BITS  = 8
parameter C_CLK_FREQ   = 30000000
parameter C_BAUDRATE   = 19200
parameter C_USE_PARITY = 0
bus_interface SOPB = opb_bus
port RX = rx
port TX = tx
port Interrupt = uart1
END

BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = timer1 & uart1
END

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 1.00.c
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

```
BEGIN DRIVER
parameter HW_INSTANCE = myuart
parameter DRIVER_NAME = uartlite
parameter DRIVER_VER = 1.00.b
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xuartlite_l.h>
#include <xintc_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* uartlite interrupt service routine */
void uart_int_handler(void *baseaddr_p) {
  char c;
  /* till uart FIFOs are empty */
  while (!XUartLite_mIsReceiveEmpty(XPAR_MYUART_BASEADDR)) {
    /* read a character */
    c = XUartLite_RecvByte(XPAR_MYUART_BASEADDR);
    /* if the character is between "0" and "9" */
    if ((c>47) && (c<58)) {
      timer_count = c-48;
      /* print character on hyperterminal (STDOUT) */
      putnum(timer_count);
      /* Set timer with new value of timer_count */
      XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0, (timer_count*tim
er_count+1) * 1000000);
    }
  }
}


/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }
```

```
    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

/* Connect uart interrupt handler that will be called when an interrupt
* for the uart occurs
*/
  XIntc_RegisterHandler(XPAR_MYINTC_BASEADDR,
XPAR_MYINTC_MYUART_INTERRUPT_INTR,
(XInterruptHandler)uart_int_handler,
(void *)XPAR_MYUART_BASEADDR);


  /* Start the interrupt controller */
  XIntc_mMasterEnable(XPAR_MYINTC_BASEADDR);

  /* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

  /* Enable timer and uart interrupts in the interrupt controller */
  XIntc_mEnableIntr(XPAR_MYINTC_BASEADDR, XPAR_MYTIMER_INTERRUPT_MASK
| XPAR_MYUART_INTERRUPT_MASK);

/* Enable Uartlite interrupt */
XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);

  /* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);


  /* Wait for interrupts to occur */
  while (1)
      ;
```

```
}
```

Example Systems for PowerPC

System *Without* Interrupt Controller (Single Interrupt Signal)

An interrupt controller is not required if there is a single interrupting peripheral or an external interrupting pin and its interrupt signal is level sensitive. A single peripheral might raise multiple interrupts. In this case, an interrupt controller is required.

Procedure

To set up a system without an interrupt controller that handles only one level sensitive interrupt signal, you must do the following:

1. Set up the MHS and MSS files as follows:
   - Connect the interrupt signal of the peripheral, or the external interrupt signal, to one of the interrupt inputs of the PowerPC in the MHS file. The interrupt inputs can be critical or non-critical.
   - Give the peripheral an instance name using the `INSTANCE` keyword in the MHS file. Libgen creates a definition in `xparameters.h (OUTPUT_DIR/PROC INST NAME/include)` for `XPAR_<PERIHPERAL_INSTANCE_NAME>_BASEADDR` mapped to the base address of this peripheral.
2. Write the interrupt handler routine that handles the signal. The base address of the peripheral instance is accessed as `XPAR_<PERIPHERAL_INSTANCE_NAME_BASEADDR`.
3. Designate the handler function to be an interrupt handler for the signal using the `INT_HANDLER` keyword in the MSS file. Refer to the "Microprocessor Software Specification (MSS)" chapter in the *Platform Specification Format Reference Manual* for more information.

   The peripheral instance is first selected in the MSS file, and then the INT_HANDLER attribute is given the function name. In case of an external interrupt signal, the INT_HANDLER attribute is given as a global parameter in the MSS file. The attribute is not part of any block in the MSS.
4. Run Libgen and powerpc-eabi-gcc.

Example MHS File Snippet

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = interrupt
port CaptureTrig0 = net_gnd
END

BEGIN ppc405
 PARAMETER INSTANCE = PPC405_i
 PARAMETER HW_VER = 1.00.a
```

```
BUS_INTERFACE DPLB = myplb
 BUS_INTERFACE IPLB = myplb
 PORT CPMC405CLOCK = sys_clk
 PORT PLBCLK = sys_clk
 PORT CPMC405CORECLKINACTIVE = net_gnd
 PORT CPMC405CPUCLKEN = net_vcc
 PORT CPMC405JTAGCLKEN = net_vcc
 PORT CPMC405TIMERTICK = net_vcc
 PORT CPMC405TIMERCLKEN = net_vcc
 PORT MCPPCRST = net_vcc
 PORT TIEC405DISOPERANDFWD = net_vcc
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT TIEC405MMUEN = net_gnd
 PORT EICC405EXTINPUTIRQ = interrupt
 PORT EICC405CRITINPUTIRQ = net_gnd
 PORT JTGC405TCK = JTGC405TCK
 PORT JTGC405TDI = JTGC405TDI
 PORT JTGC405TMS = JTGC405TMS
 PORT JTGC405TRSTNEG = JTGC405TRSTNEG
 PORT C405JTGTDO = C405JTGTDO
 PORT C405JTGTDOEN = C405JTGTDOEN
 PORT DBGC405DEBUGHALT = DBGC405DEBUGHALT
END
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

Example C Program

```
/*****************************************************************
*   Copyright (c) 2001 Xilinx, Inc.  All rights reserved.
*   Xilinx, Inc.
*
*
* This program uses the timer and gpio to demonstrate interrupt
handling.
* The timer is set to interrupt regularly. The frequency is set in the
code.
* Every time there is an interrupt from the timer,
* a rotating display of leds on the board is updated.
*
* The LEDs and switches are in these bit positions:
* LSB 0: gpio_io<3>
* LSB 1: gpio_io<2>
* LSB 2: gpio_io<1>
* LSB 3: gpio_io<0>
*****************************************************************
/
```

```
/* This is the list of files that must be included to access the
peripherals:
 * xtmrctr.h - to access the timer
 * xgpio_l.h  - to access the general purpose I/O
* xparameters.h  - General purpose definitions.  Must always be included
 *          when any drivers/print routines are accessed.  This defines
 *         addresses of all peripherals, declares the interrupt service
 *           routines, etc.
 */
#include <xtmrctr_l.h>
#include <xgpio_l.h>
#include <xparameters.h>
#include <xexception_l.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/*
 * Interrupt service routine for the timer.  It has been declared as an
ISR in
 * the mss file using the attribute INT_HANDLER. The ISR can be written
as a normal C routine.
 * The peripheral can be accessed using XPAR_<peripheral name in the mhs
file>_BASEADDR
 * as the base address.
 */
void timer_int_handler(void * baseaddr_p) {
   int baseaddr = (int)baseaddr_p;
   unsigned int csr;
   unsigned int gpio_data;
  int baseaddr = (int) baseaddr_p;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Shift the count */

    if ((count <<= 1) > 16) {
      count = 1;
    }

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, ~count);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {
```

```
  int i, j;

  /* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)timer_int_handler, (void *)XPAR_MYTIMER_BASEADDR);

/* Set the gpio as output on high 4 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 8000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );


/* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

  /* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

  /* Wait for interrupts to occur */
  while (1)
      ;
}
```

Example MHS File Snippet (For External Interrupt Signal)

```
PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = LOW, SIGIS =
INTERRUPT

BEGIN ppc405
 PARAMETER INSTANCE = PPC405_i
 PARAMETER HW_VER = 1.00.a
BUS_INTERFACE DPLB = myplb
 BUS_INTERFACE IPLB = myplb
 PORT CPMC405CLOCK = sys_clk
 PORT PLBCLK = sys_clk
 PORT CPMC405CORECLKINACTIVE = net_gnd
 PORT CPMC405CPUCLKEN = net_vcc
 PORT CPMC405JTAGCLKEN = net_vcc
 PORT CPMC405TIMERTICK = net_vcc
 PORT CPMC405TIMERCLKEN = net_vcc
 PORT MCPPCRST = net_vcc
 PORT TIEC405DISOPERANDFWD = net_vcc
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
```

```
 PORT TIEC405MMUEN = net_gnd
 PORT EICC405EXTINPUTIRQ = interrupt_in1
 PORT EICC405CRITINPUTIRQ = net_gnd
 PORT JTGC405TCK = JTGC405TCK
 PORT JTGC405TDI = JTGC405TDI
 PORT JTGC405TMS = JTGC405TMS
 PORT JTGC405TRSTNEG = JTGC405TRSTNEG
 PORT C405JTGTDO = C405JTGTDO
 PORT C405JTGTDOEN = C405JTGTDOEN
 PORT DBGC405DEBUGHALT = DBGC405DEBUGHALT
END
```

Example MSS File Snippet

```
PARAMETER int_handler = global_int_handler, int_port = interrupt_in1
```

Example C Program

```
#include <xparameters.h>

/* global interrupt service routine */
void global_int_handler(void * arg) {
/* Handle the global interrupts here */

}

void
main() {

/* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)global_int_handler, (void *)0);

/* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

/* Wait for interrupts to occur */
  while (1)
      ;

}
```

System *With* an Interrupt Controller (One or More Interrupt Signals)

An Interrupt Controller peripheral (INTC) should be present if more than one interrupt can be raised. When an interrupt is raised, the interrupt handler for the Interrupt Controller, `XIntc_DeviceInterruptHandler`, is called. This function then accesses the interrupt controller to find the highest priority device that raised an interrupt. This is done via the vector table created automatically by Libgen. On return from the peripheral interrupt handler, `intc interrupt handler` acknowledges the interrupt. It then handles any lower priority interrupts, if they exist.

Procedure

To set up a system with one or more interrupting devices and an interrupt controller, you must do the following:

1. Set up the MHS and MSS files as follows:
   - Assign the interrupt signals of all the peripherals to the Intr port of the interrupt controller in the MHS file. The interrupt signal output of INTC is then connected to one of the interrupt inputs of PowerPC. The interrupt inputs can be either critical or non-critical.
   - Give the peripherals instance names using the `INSTANCE` keyword in the MHS file. Libgen creates a definition in `xparameters.h` for `XPAR_<INTC_INSTANCE_NAME>_BASEADDR` mapped to the base address of each peripheral for use in the your program. Libgen also creates an interrupt mask and interrupt ID for each interrupt signal using the priorities `XPAR_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_MASK` and `XPAR_<INTC_INSTANCE_NAME>_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>_INTR`. This can be used to enable or disable interrupts.
2. Write the interrupt handler functions for each interruptible peripheral.
3. Designate each handler function to be the handler for an interrupt signal using the `INT_HANDLER` keyword in the MSS file. Alternately, you can register the routines with the INTC interrupt vector table in your code. For this example, we display both of these use cases by setting the routine for the timer in the MSS file and setting up the `uart` interrupt port handler in your code. The `INTC` interrupt signal must not be given an `INT_HANDLER` keyword. If the `INT_HANDLER` keyword is not present for a particular peripheral, the interrupt signal uses a default dummy interrupt handler.
4. Run Libgen and mb-gcc to do the following:
   - Automatically generate and compile an interrupt vector table in Libgen. Each of the peripherals connected to INTC can also register its interrupt handlers with the INTC interrupt handler.
   - Have `XIntc_DevicenterruptHandler` call the peripheral interrupt handler using the updated interrupt vector table to identify the handler in order of priority.

   Note: PowerPC jumps to `XIntc_DeviceInterruptHandler` using the exception table when an interrupt occurs. The `XIntc_DeviceInterruptHandler` is registered with the exception table in your code.

Example MHS File Snippet

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = timer1
port CaptureTrig0 = net_gnd
END
```

```
BEGIN opb_uartlite
parameter INSTANCE = myuart
parameter HW_VER = 1.00.b
parameter C_BASEADDR  = 0xFFFF8000
parameter C_HIGHADDR = 0xFFFF80FF
parameter C_DATA_BITS  = 8
parameter C_CLK_FREQ   = 30000000
parameter C_BAUDRATE   = 19200
parameter C_USE_PARITY = 0
bus_interface SOPB = opb_bus
port RX = rx
port TX = tx
port Interrupt = uart1
END

BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = timer1 & uart1
END

BEGIN ppc405
 PARAMETER INSTANCE = PPC405_i
 PARAMETER HW_VER = 1.00.a
BUS_INTERFACE DPLB = myplb
 BUS_INTERFACE IPLB = myplb
 PORT CPMC405CLOCK = sys_clk
 PORT PLBCLK = sys_clk
 PORT CPMC405CORECLKINACTIVE = net_gnd
 PORT CPMC405CPUCLKEN = net_vcc
 PORT CPMC405JTAGCLKEN = net_vcc
 PORT CPMC405TIMERTICK = net_vcc
 PORT CPMC405TIMERCLKEN = net_vcc
 PORT MCPPCRST = net_vcc
 PORT TIEC405DISOPERANDFWD = net_vcc
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT TIEC405MMUEN = net_gnd
 PORT EICC405EXTINPUTIRQ = interrupt
 PORT EICC405CRITINPUTIRQ = net_gnd
 PORT JTGC405TCK = JTGC405TCK
 PORT JTGC405TDI = JTGC405TDI
 PORT JTGC405TMS = JTGC405TMS
 PORT JTGC405TRSTNEG = JTGC405TRSTNEG
 PORT C405JTGTDO = C405JTGTDO
 PORT C405JTGTDOEN = C405JTGTDOEN
 PORT DBGC405DEBUGHALT = DBGC405DEBUGHALT
END
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END

BEGIN DRIVER
parameter HW_INSTANCE = myuart
parameter DRIVER_NAME = uartlite
parameter DRIVER_VER = 1.00.b
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xuartlite_l.h>
#include <xintc_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.*/

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* uartlite interrupt service routine */
void uart_int_handler(void *baseaddr_p) {
  char c;
  /* till uart FIFOs are empty */
  while (!XUartLite_mIsReceiveEmpty(XPAR_MYUART_BASEADDR)) {
    /* read a character */
    c = XUartLite_RecvByte(XPAR_MYUART_BASEADDR);
    /* if the character is between "0" and "9" */
    if ((c>47) && (c<58)) {
      timer_count = c-48;
      /* print character on hyperterminal (STDOUT) */
      putnum(timer_count);
      /* Set timer with new value of timer_count */
      XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0, (timer_count*tim
er_count+1) * 1000000);
    }
  }
}


/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;
int baseaddr = (int) baseaddr_p;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
```

```
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }

    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {

  unsigned int gpio_data;

/* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)XIntc_DeviceInterruptHandler, (void
*)XPAR_MYINTC_DEVICE_ID);

/* Connect uart interrupt handler that will be called when an interrupt
* for the uart occurs
*/
  XIntc_RegisterHandler(XPAR_MYINTC_BASEADDR,
XPAR_MYINTC_MYUART_INTERRUPT_INTR,
(XInterruptHandler)uart_int_handler,
(void *)XPAR_MYUART_BASEADDR);

  /* Start the interrupt controller */
  XIntc_mMasterEnable(XPAR_MYINTC_BASEADDR);

  /* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

/* Enable timer and uart interrupts in the interrupt controller */
  XIntc_mEnableIntr(XPAR_MYINTC_BASEADDR, XPAR_MYTIMER_INTERRUPT_MASK
| XPAR_MYUART_INTERRUPT_MASK);

/* Enable Uartlite interrupt */
XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
```

```
/* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

/* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

  /* Wait for interrupts to occur */
  while (1)
      ;

}
```

Chapter 6

Using Xilkernel

Xilkernel is a small, robust embedded kernel that provides for scheduling multiple execution contexts and a set of higher level services. It also provides for synchronization and communication primitives to help design co-operatively executing tasks for solving a problem. The first section of this chapter explains the fundamental operating system concepts and their application to Xilkernel. The next section walks you through the series of steps required within the Xilinx® Platform Studio™ (XPS) to include, configure and build Xilkernel and applications. The final section provides a demonstration and step-by-step analysis of actual applications executing on top of Xilkernel. These design examples are targeted for the Insight 2VP7 FG456 demo board and are available for both MicroBlaze™ and PPC.

This chapter contains the following sections.

- "Xilkernel Concepts"
- "Getting Started with Xilkernel"
- "Xilkernel Design Examples"

Xilkernel Concepts

The following general concepts are key to understanding a kernel: Processes, Threads, Context switching, Scheduling, Synchronization, and Interprocess communication. These general concepts are not explained in their entirety and detail in this chapter. However, a basic introduction to each concept is provided. These concepts should be familiar to anyone comfortable with operating system technology. Therefore, advanced users can skip introductory portions and go directly to Xilkernel relevant parts. Some concepts are completely specific to Xilkernel. These include Xilkernel configuration, application linkage, block memory allocation, system initialization, interrupt handling and accessing standard libraries.

Processes, Threads, Context Switching, and Scheduling

Operating systems typically run multiple user applications as independent *processes*. Each process usually has its own memory space independent of the other processes. A *hardware timer* periodically interrupts the processor to invoke the scheduler. The time interval between these interrupts is called a *time slice*. The *scheduler* is a function within the operating system (or Xilkernel) that determines which process should run in the current time slice. The following sequence is an example of running processes:

1. $Timeslice_1$: The kernel starts running and the scheduler function determines that Process A should start running. The kernel begins Process A.
2. $Timeslice_2$: The kernel stops Process A, saves its state, and begins Process B.

3. Timeslice$_3$: The kernel stops Process B, saves its state, restores the saved state of Process A, and then restarts Process A.
4. Timeslice$_4$: Process A stops. The kernel restores the saved state of Process B and then continues running Process B.

As a result of this sequence of operations, Process A and Process B appear to run concurrently. For example, an embedded system might have two applications that must run concurrently – the first application continuously processes input packets and outputs filtered data, while the other application monitors push-button inputs and displays status on an LCD panel. Without a kernel, the two applications would need to be combined into one giant application. With a kernel, each application becomes an independent process, and can be developed and tested independently. A process can perform several operations on itself or other processes, such as creating a new process, yielding its time slice to another process, killing another process, passing data or control to another process, waiting for another process to finish, changing its own priority, and exiting.

In the context of Xilkernel, a *thread* is the unit of execution and is analogous to a process. The programming interface for these threads is based on the widely used POSIX standard, now supported by the IEEE, called `pthreads`. While other operating systems define a thread as a light weight process and is a sub-unit of a 'process,' a thread in Xilkernel is the primary unit of execution and does not correlate to the concept of thread groups forming a process. Refer to the "Xilkernel" chapter in the *EDK OS and Libraries Reference Manual* for more info about the Xilkernel process model. In further sections of this chapter, the term *process* is used to encompass both threads and processes.

Xilkernel provides different *scheduling algorithms*, one of which can be selected to for a particular combination of applications. The simplest scheduling algorithm provided by Xilkernel is the *round-robin scheduler*. This scheduler places all processes in a queue and assigns each time slice to the next process in the queue. Xilkernel provides another scheduling algorithm called the *priority scheduler*. The priority scheduler places processes in multiple queues, one for each priority level. On each timer interrupt, the priority scheduler picks the first item in the highest priority queue. Each process is assigned a fixed priority when it is created. Unlike other operating systems, Xilkernel only picks a process to run only if it is ready to run and there are no higher priority processes that are ready. Details of the scheduling algorithms are described in the "Xilkernel" chapter in the *EDK OS and Libraries Reference Manual*.

Synchronization Constructs

When two processes try to access a shared resource such as a device, it might be necessary to provide exclusive access to the shared resource for a single process. For example, if Process A is writing to a device, Process B must wait until Process A is done, before it reads the same block. Similarly if Process B is reading a shared memory block, Process A must wait until Process B is done, before it writes a new value to the same block. The constructs that allow one to ensure this kind of constraints are known as *synchronization constructs*. *Semaphores* and *mutexes* are the two constructs provided by Xilkernel. In the previous example, Process A and Process B could use semaphores to ensure correct operation.

Semaphores

Semaphores are much more advanced constructs than mutex locks and provide a counting facility that can be used to co-ordinate tasks in much more complex fashions. A mutex lock is essentially just a binary semaphore. In Xilkernel, mutex locks have a similar but independent implementation from semaphores.

The following example scenario involves semaphores for coordinating access to a piece of shared memory between two processes A and B. Process A uses semaphore X to signal Process B, and Process B uses semaphore Y to signal A.

1. Process A creates semaphore X with an initial value of 0.
2. Process B starts by creating a semaphore Y with an initial value of 0 and waits for semaphore X. The *wait* operation blocks a process until a corresponding *signal* operation releases it.
3. Process A writes to the shared memory and invokes the `sem_post()` function to signal semaphore X. Process A then waits on semaphore Y before writing again.

   Meanwhile, Process B, which was waiting for semaphore X, gets it and does the read operation.
4. Process B signals semaphore Y, and waits again on semaphore X.

Mutexes

Mutexes are similar to semaphores, also allowing mutually exclusive access to a shared resource. However, mutexes are defined only for threads in Xilkernel. The shared resource might be a shared memory block as in the previous example, or it might be a peripheral device such as a display device. A mutex is associated with the shared resource at software design time. Then each thread that wants to access this resource first invokes a `pthread_mutex_lock()` function on this mutex to obtain exclusive access. If the resource is not available, the thread is blocked until the resource becomes available to it. Once the thread obtains the lock, it performs the necessary functions with this resource and then unlocks the resource using `pthread_mutex_unlock()`.

Inter-Process Communication

Independently executing processes or threads can share information among themselves by using standard mechanisms such as *shared memory* or *message passing*. By definition, the data memory used by each process is distinct from the data memory used by other processes.

Shared Memory

Consider a scenario in which you want to build a system with two processes that use shared memory as described in the following:

1. Process A reads data from a serial input device and stores it in memory
2. Process B reads the data from memory and computes some value

In this scenario, you should store the data in a shared memory segment. When you configure Xilkernel, as explained later, you can specify the number and size of shared memory segments. Process A attaches to a named shared memory segment and writes data to it. Process B attaches to the same shared memory segment using the same name and reads data from it.

Message Passing

Another technique to allow Process A and Process B to share data is message passing. The following example describes a system with two processes that use message passing:

1. Process A reads data from a serial device and then creates a message containing this data.
2. Process A sends the message to a message queue, specifying a queue identifier.]
3. Process B then gets the message from a specified message queue and retrieves the data in the message.

The message passing routines are higher level functions that use semaphores internally to provide a simple communication mechanism between processes.

The programming interface for shared memory and messages is based on the POSIX standard.

For more information about these functions, refer to the "Xilkernel" chapter in the *EDK OS and Libraries Reference Manual*.

Concepts Specific to Xilkernel

Code and Runtime Structure

Xilkernel is structured as a library. The user application source files must link with Xilkernel to access Xilkernel functionality. The final image linking the kernel to the application becomes an Executable and Linking Format (ELF) file, which you can download, bootload, debug, transfer around and process as with any other ELF file for stand-alone programs.

Xilkernel on a PPC405 requires a special memory layout for two different sections, as dictated by the hardware requirements. The basic requirement for the PPC is that memory is allocated to the *.vectors* section starting at a 64KB address boundary and a *.boot* section at 0xFFFFFFFC. The rest of the code and data memory can be placed anywhere desired.

Your code must include `xmk.h` as the first include file in your sources. This enables Xilkernel to extract certain definitions from the C standard library header files. Your application must also include the `main()` routine. This routine is the entry point for the kernel + application bundle. When you use `main()`, the kernel is still inactive. You can perform pre-processing before the kernel starts. The entry point for the kernel is `xilkernel_main()`. Use this function to start the kernel, scheduler, timers, and the interrupt system. Control does not return from `xilkernel_main()`. The scheduler is now responsible for scheduling the configured tasks and continuing the execution in that fashion. The following code snippet displays a simple application source file:

```
#include "xmk.h"
#include <stdio.h>
/* various declarations and definitions go here */

int main()
{
  /* Application initialization code goes here */
  /* Application not allowed to invoke kernel interfaces yet */
  xilkernel_main (); /* Start the kernel */

  /* Control does not reach here */
}
```

```
/* Statically created first thread */
int first_thread ()
{
  /* Create more threads and do processing as required */
}
```

The kernel bundled executable mode kernel can still service separately compiled executables through a system call layer. This is similar to a mixed executable mode.

System Calls and Libraries

All of the Xilkernel functions previously described are available to your applications as libraries. Many of these functions require exclusive access to kernel resources, which the code guarantees by disabling interrupts while these kernel functions run. The following flow of control is for invoking a system call, `proc_do_something()`:

1. The `proc_do_something()` function transfers control to the main kernel system call handler.
2. The handler disables interrupts, preforms some pre-processing, and then invokes `sys_proc_do_something()`, the internal version of the system call.
3. Once the system call is complete, the handler performs post-processing. It checks to see if a rescheduling is required.
4. If a rescheduling is required, then the handler invokes the scheduler and does a context switch. If not, it re-enables interrupts before returning to the original caller.

User Interrupts and Xilkernel

In any processor-based hardware system, various peripherals can generate interrupt signals. If multiple interrupting peripherals are present, an interrupt controller collects these signals and feeds a single interrupt signal to the processor. Both PPC and MicroBlaze kernels require a timer device to generate periodic interrupts and this is the only hardware requirement that the kernel places. MicroBlaze requires an external timer interrupt to run Xilkernel, while PowerPC™ has an internal timer that generates interrupts for Xilkernel. Also, if an interrupt controller is present in the system, Xilkernel exports an interrupt handler registering mechanism and invokes the handlers after it pre-processes each hardware interrupt.

On MicroBlaze with interrupts enabled, when the processor receives an interrupt, it jumps to address `0x10`, which contains an instruction to jump to the Xilkernel interrupt handler. By default, the Xilkernel interrupt handler handles only the timer interrupts. On such an interrupt, Xilkernel performs its usual scheduling and time accounting steps. Functions to handle interrupts from other peripherals must be registered with the Xilkernel interrupt handler using the interrupt handler registration functions described in the "Xilkernel" chapter in the *EDK OS and Libraries Reference Manual*.

On MicroBlaze and PowerPC, in addition to the usual interrupts, there might be other hardware exceptions recognized by the processor. To register handlers for these exceptions, follow the steps documented in the *Standalone BSP reference guide.*

Differences Between MicroBlaze & PowerPC Xilkernel Implementations

Xilkernel uses a single common programming interface on MicroBlaze and PowerPC. However, the underlying hardware models are different, so you might notice a difference in terms of compiled code size and performance. The primary difference is in the linker scripts used for PowerPC, and the compiler flags used for MicroBlaze. Otherwise, the kernel exports the same interfaces to both PPC and MicroBlaze.

Using Device Drivers with Xilkernel

Device drivers are software routines that interact with peripheral devices. A number of device drivers are provided as part of the EDK to interface with Xilinx provided devices such as UARTs and various types of memory and input/output device controllers. Xilkernel uses certain device drivers automatically - the UART device driver if a UART is present in the hardware system, and the Interrupt Controller device driver if an interrupt controller is present in the hardware system. Any application can use device drivers by directly invoking the device driver interface. For interrupt handling of devices, your applications must use the interrupt registration mechanism of Xilkernel.

Using Other Libraries with Xilkernel

The libc and libm libraries can be used by any application running as a process or thread managed by Xilkernel. The *XilFile*, *XilMFS*, *XilNet*, and *XilProfile* libraries can also be used by any application thread. However, the XilFile, XilMFS, XilNet and XilProfile libraries access hardware devices either directly or indirectly. These libraries are not designed to be completely re-entrant and thread-safe. As a result they are subject to restrictions on shared access. The libc library is also not completely re-entrant (routines like *malloc*, *free*). If more than one application thread wants to use these libraries at the same time, then the threads need to use some sort of protection mechanism to coordinate access to the device.

Getting Started with Xilkernel

Xilkernel with MicroBlaze

If you want to use Xilkernel with MicroBlaze, your hardware platform must contain the following minimum requirements:

- MicroBlaze processor.
- 7-24 KB of BRAM or other memory connected to the processor over Local Memory Bus (LMB) or On-Chip Peripheral Bus (OPB), with an appropriate memory controller. You might need more or less memory depending on how Xilkernel is configured.
- OPB timer or Fixed Interval Timer (FIT) peripherals connected to the interrupt port of the processor either directly or through an interrupt controller.
- UART or UARTlite peripheral for input/output. This is optional and is only used for demonstration/debug purposes.

Xilkernel with PowerPC

If you want to use Xilkernel with PowerPC, your hardware platform must contain the following minimum requirements:

- PowerPC processor.
- 16-32KB of BRAM or other memory connected to the processor over PLB or OPB, with an appropriate memory controller. You might need more or less memory depending on how Xilkernel is configured. PowerPC systems need more memory than MicroBlaze systems because of a large `.vectors` section.
- UART or UARTlite peripheral for input/output. This is optional and is only used for demonstration/debug purposes.

Note: A separate timer is not needed as in the MicroBlaze system because the PowerPC contains built-in timers for Xilkernel and other operating systems.

Building and Executing Xilkernel and Applications

There are three steps to building and executing Xilkernel and your applications:

1. Configure and generate Xilkernel
2. Create your application(s)
3. Download/Debug Xilkernel

Each of these steps is explained in more detail in the following sections.

Configuring and Generating Xilkernel

You can configure a software platform to include Xilkernel by using the Software Platform Settings dialog box as described in the following steps:

1. Click the System tab on the navigator on the left side of the XPS main window to view the hardware tree view.
2. Right-click the processor name in the processor tree view and select **S/W Settings**.

   The Software Settings dialog box opens. The top half of this dialog box displays the devices in the hardware platform and the drivers assigned to them. The bottom right side of the dialog box allows you to select an OS. The available OS's are VxWorks, Standalone, and Xilkernel.
3. Select **Xilkernel**.

   If multiple versions of Xilkernel are available, you can select an appropriate version.
4. To further configure Xilkernel, click on the Library/OS Parameters tab at the top of the dialog box.

   The Library/OS Parameters tab opens.

   This tab displays all of the configurable parameters for Xilkernel. The parameters are organized into categories and subcategories. For each parameter, the parameter type, default value, and a short explanation are displayed. The default values are pre-populated, and you can change the values as needed. You can collapse or expand the tree view by clicking on the **+** or **-** symbols on the left side.

   Note: You have not yet created any applications. You are only configuring aspects of your software platform.
5. Click **OK** to save your selections.

   Your Xilkernel configuration is complete.

Note: All the GUI settings that you just set are translated to lines in the MSS in the OS section. If you prefer text editing to GUIs, you can create this MSS file and run Libgen in the batch mode flow.

The following MSS fragment reflects the settings as set in the previous example:

```
BEGIN OS
 PARAMETER OS_NAME = xilkernel
 PARAMETER OS_VER = 3.00.a
 PARAMETER STDIN = RS232
 PARAMETER STDOUT = RS232
 PARAMETER proc_instance = microblaze_0
 PARAMETER config_debug_support = true
 PARAMETER verbose = true
 PARAMETER systmr_spec = true
 PARAMETER systmr_dev = system_timer
 PARAMETER systmr_freq = 66000000
 PARAMETER systmr_interval = 100
 PARAMETER sysintc_spec = system_intc
 PARAMETER config_sched = true
 PARAMETER sched_type = SCHED_PRIO
 PARAMETER n_prio = 6
 PARAMETER max_readyq = 10
 PARAMETER config_pthread_support = true
 PARAMETER max_pthreads = 10
 PARAMETER config_sema = true
 PARAMETER max_sem = 4
 PARAMETER max_sem_waitq = 10
 PARAMETER config_msgq = true
 PARAMETER num_msgqs = 1
 PARAMETER msgq_capacity = 10
 PARAMETER config_bufmalloc = true
 PARAMETER config_pthread_mutex = true
 PARAMETER config_time = true
 PARAMETER max_tmrs = 10
 PARAMETER mem_table = ((4,30),(8,20))
 PARAMETER enhanced_features = true
 PARAMETER config_kill = true
 PARAMETER static_pthread_table = ((shell_main,1))
END
```

For more information on Libgen and batch flows, see the "Library Generator" chapter and the "Xilinx Platform Studio" chapter, both in the *Embedded System Tools Guide*.

6. Click **Tools** → **Generate Libraries and BSPs** to generate the Xilkernel library.

   At the end of the Libgen flow, Xilkernel generation is complete and the file `libxilkernel.a` is generated in the corresponding processor's `lib/` folder.

Creating Your Application

You can create and build an application as described in the following steps:

1. Click the Applications tab on the navigator on the left side of the XPS main window.

   This opens the tree view with **Software Projects** on top and all software applications for this processor below it. You can collapse or expand the tree view by clicking on the **+** or **-** symbols on the left side.

   Right-click **Software Projects** and select **Add SW Application Project**.

   The Add SW Application Project dialog box opens.

2. Type a name for your project in the **Project Name** field, and select a processor from the **Processor** drop-down list.
3. The tree view updates to show the new project (*xilkernel_demo* in this example), as shown in the previous image.
4. Now you must set the compiler options for your application. Choose the compiler optimization level, including paths as needed. The only compiler setting that you must specify for your Xilkernel application is “xilkernel” in the list of libraries to link with
5. Create and add your source files, as described here:
   a. Create your application source files. You can use your favorite text editor or IDE to create your application.
   b. In the Applications tab, right click the **Sources** item in the project tree view and select **Add File**.

      The Add Source and Header Files to the project dialog box appears.
   c. Navigate to your application source file and open it.

      Your file is added to the project tree.
6. Right-click the project name in the tree view and select **Build Project** to create the executable file that contains your kernel bundled application.

Downloading/Debugging Your Xilkernel Application

You can download and run the executable file of your application project as with any other standalone application. If you want to debug your application, click the **Run Debugger** toolbar button and select your Xilkernel application project as the project to debug. GDB opens, allowing you to add breakpoints and debug your entire kernel and applications.

Note: If you want to debug your application with GDB, you must compile your project with *-g*.

Xilkernel Design Examples

This section demonstrates an example Xilkernel-based system, illustrating all the features of the kernel. Xilinx recommends that you begin with this design example to see the actual interfaces of the kernel being exercised. This design example is targeted for the Insight V2P7 FG456 Rev 4 board. With minor effort, however, this example can be ported to any board that can accommodate the design.

Hardware and Software Requirements

Your system must meet the following hardware and software requirements for this design example.

Hardware

- Memec Design Insight V2P4/7 FG456 Rev 4 board
- Power supply
- JTAG and serial connection cables.

Software

- Xilinx® EDK 7.1i
- Xilinx® ISE™ 7.1i (G.35)
- User Guide Example Zip Files: `Insight_PPC_XMK.zip` and `Insight_MB_XMK.zip`.
- Hyperterminal or some other terminal client.

Design Example Files

You can obtain the design examples online from the Xilinx website at the following URL:

http://www.xilinx.com/ise/embedded/edk_examples.htm

Download the Xilkernel User Guide design example ZIP files, and extract them to a location on your system. You can access these ZIP files by clicking the links for *PPC XMK Example* and *MB XMK Example*.

In the design example folder, The MicroBlaze design example is located in the `MB` subfolder, and the PowerPC example is located in the `PPC` subfolder.

Description of Example Sets

The hardware systems were built using the Base System Builder Wizard.

Overview of the MicroBlaze System

The MicroBlaze system consists of:

- MicroBlaze at 66 MHz.
- Hardware Debug module with fast download enabled.
- 32KB of LMB BRAM.
- OPB Uartlite at 19200 baud.
- 4-bit LEDs.
- 3 Push Buttons.
- 32 MB of SDRAM.
- DCM module to generate the system clock.
- OPB interrupt controller connected to MicroBlaze interrupt port.
- Two OPB timers connected through the interrupt controller. The second timer is used to illustrate user-level interrupts.

Overview of the PowerPC System

The PowerPC system consists of:

- PowerPC at 100 MHz.
- 64 KB of PLB BRAM.
- OPB UARTlite at 19200 baud.
- 4-bit LEDs.
- 3 Push Buttons.
- 8-bit DIP switches.

- 32 MB of SDRAM.
- DCM module clocking the PLB bus at 50 MHz.
- OPB interrupt controller connected to the PPC external interrupt port.
- An OPB timer connected through the interrupt controller. This timer is used to illustrate user-level interrupt handling.

Overview of the Example Application Sets

The example application sets were designed to illustrate all the features of Xilkernel. All of the features of the kernel are turned on and there are application threads that exercise most of the interfaces, so the code size of the generated kernel is larger than available BRAM. Therefore, the final kernel image is designed to run out of external memory in MicroBlaze and a mix of BRAM and external memory in PPC405. Further description of the examples focus on the MicroBlaze system. A separate section documents the differences in the PowerPC examples.

Since the example applications include those that run from external memory, we must use Xilinx® Microprocessor Debug (XMD) to download the programs and execute them. Each of these EDK design examples contains a simple startup program to print a welcome message and exit. This program initializes BRAM and executes on bitstream download.

Defining the Communications Settings

The steps described below can be run completely from XPS.

You must have hyperterminal or some other terminal client connected to the serial port through which the RS232 interface on the target board is connected.

You should configure your session with the following settings:

- Bits per second: **19200**
- Data bits: **8**
- Parity: **None**
- Stop bits: **1**
- Flow control: **None**

Xilinx recommends that you save these connection settings. To connect using hyperterminal, start up hyperterminal and open up the saved connection settings file. Hyperterminal automatically waits for data from the other end.

Software Platform Specification

The example system has a pre-configured software platform, with chosen values for each parameter. Review the following MSS snippet from the MicroBlaze example, and examine some parameters and what they mean. This snippet is generated for this software platform.

Click a parameter to view its description.

```
BEGIN OS
 PARAMETER OS_NAME = xilkernel
 PARAMETER OS_VER = 3.00.a
 PARAMETER STDIN = RS232
 PARAMETER STDOUT = RS232
 PARAMETER proc_instance = microblaze_0
```

```
# Enable diagnostic/debug messages
PARAMETER config_debug_support = true
PARAMETER verbose = true
# MicroBlaze system timer device specification
PARAMETER systmr_spec = true
PARAMETER systmr_dev = system_timer
PARAMETER systmr_freq = 66000000
PARAMETER systmr_interval = 100
# Specification of the intc device
PARAMETER sysintc_spec = system_intc
# Scheduling type
PARAMETER config_sched = true
PARAMETER sched_type = SCHED_PRIO
PARAMETER n_prio = 6
PARAMETER max_readyq = 10
# Configure pthreads
PARAMETER config_pthread_support = true
PARAMETER max_pthreads = 10
# Semaphore specification
PARAMETER config_sema = true
PARAMETER max_sem = 4
PARAMETER max_sem_waitq = 10
# MSGQ specification
PARAMETER config_msgq = true
PARAMETER num_msgqs = 1
PARAMETER msgq_capacity = 10
# MSGQ's require config_bufmalloc to be true.
PARAMETER config_bufmalloc = true
# Configure pthread mutex
PARAMETER config_pthread_mutex = true
# Configure time related features
PARAMETER config_time = true
PARAMETER max_tmrs = 10
PARAMETER mem_table = ((4,30),(8,20))
# Enhanced features
PARAMETER enhanced_features = true
PARAMETER config_kill = true
PARAMETER static_pthread_table = ((shell_main,1))
END
```

The Xilkernel MSS specification is enclosed within an OS block. The parameters in bold are the required parameters, which you must provide values for.

The following table describes each parameter:

Table 6-1: **Software Platform Parameters**

Parameter	Description
OS_NAME	These parameters combine to tell Libgen what the OS is.
OS_VER	
STDIN	These parameters provide input and output for the example applications. In this example, you must tell Xilkernel the instance names of the peripherals that are to be used for standard inputs and outputs. In this case, we use the OPB Uartlite peripheral in the system, named *RS232*, as the input-output device.
STDOUT	

Table 6-1: **Software Platform Parameters *(Continued)***

Parameter	Description
proc_instance	This parameter ties the OS to a particular processor in the hardware system. In this case, it is tied to the processor whose instance name is microblaze_0.
config_debug_support	This parameter controls various aspects of debugging the kernel.
verbose	This sub-parameter of config_debug_support, if set to true, will enable verbose messages from the kernel, such as on error conditions. We set this to true in our example.
systmr_spec	This parameter is required and specifies whether or not the timer specifications are defined. ***Note:*** There is a different requirement for PPC405. Refer to the section discussing the PPC405 example, Configuring PowerPC.
systmr_dev	Xilkernel requires a timer to tick the kernel. This parameter works with the systmr_freq and systmr_interval parameters.
systmr_freq	This parameter specifies the frequency at which the timer is clocked at, measured in Hz. In this case, the timer is clocked at 66 MHz.
systmr_interval	This parameter allows you to control the time interval at which the kernel ticks are to arrive. This is auto-determined if a *fit_timer* is used, since the interval cannot be programmed. Otherwise, a value in milliseconds is provided. In this case, we provide a large granularity tick of 100ms. This ensures that the output from the application threads are not interleaved and appear mangled on the screen. This setting directly controls the CPU budget of each thread.
sysintc_spec	The example hardware system has multiple interrupting devices, so an interrupt controller is tied to the external interrupt pin of the processor. In this case, the sysintc_spec parameter is set to the instance name of the corresponding peripheral.
config_sched	You configure the scheduling scheme of the kernel with the config_sched parameter. In this example, the scheduling type is chosen to be SCHED_PRIO with 6 priority levels, and the length of each ready queue is 10.
sched_type	
n_prio	
max_readyq	
config_pthread_support	This parameter specifies whether to support threads. Since this example works with threads, this parameter is set to **true**.
max_pthreads	This parameter controls the maximum number of pthreads that can run on Xilkernel at any instant in time.
config_sema	This parameter determines whether to include semaphore support in the kernel. In this example, config_sema parameter is enabled.
max_sem	This parameter specifies the maximum number of semaphores required at run-time.
max_sem_waitq	This parameter specifies the length of each semaphore's wait queue.

Table 6-1: **Software Platform Parameters *(Continued)***

Parameter	Description
config_msgq	This parameter enables the message queue category. The message queue module depends on both the semaphore and buffer memory allocation modules. Each message queue uses two semaphores internally and uses block memory allocation to allocate memory for messages in the queue. This dependency is enforced both in the GUI and in the library generation process with Libgen. Libgen Design Rule Checkers (DRCs) will catch any errors due to missing dependencies.
num_msgqs	This parameter specifies the number of message queues. This example requires a single message queue for the producer consumer demo thread.
msgq_capacity	This parameter specifies the maximum number of messages that the queue accommodates. In this example, the message queue can contain a maximum of 10 messages.
config_bufmalloc	This parameter defines whether or not to use buffer memory allocation. In this example, buffer memory allocation is needed by both the message queue module and the linked list demo thread, so the parameter is set to **true**.
config_pthread_mutex	This parameter defines whether to enable mutex lock support. This example has a mutex lock demo thread, so it sets this parameter to **true**.
config_time	This parameter defines whether to enable software timer support. This example requires software timer support, so it sets this parameter to **true**.
max_tmrs	This parameter specifies the maximum number of timers. This example uses a maximum of 10 timers for the kernel to support.
mem_table	This parameter statically specifies a list of block sizes that the kernel will support when it starts. It consists of a tuple *(m,n)* where *m* is the block size, and *n* is the number of such blocks to be allocated. We have two entries in this array - (**4,30**) and (**8,20**). This means that the kernel must support up to 30 requests for memory of size 4 bytes and 20 requests for memory of size 8 bytes.

Table 6-1: **Software Platform Parameters *(Continued)***

Parameter	Description
enhanced_features config_kill	The enhanced_features parameter defines whether to use enhanced features. One such feature is the ability to kill a process, config_kill, as defined in this example. In order to use enhanced features, the enhanced_features parameter must be defined as **true**, and each feature must also be separately included and defined as **true**.
static_pthread_table	This parameter specifies a list of threads to create at kernel startup. It is made up of an array of tuples (*start_func*, *prio*) which specifies, for each thread, the starting point of execution (*start_func*) and the priority at which it starts (*prio*). For this example, the static_pthread_table parameter is specified as (shell_main, 1). The function shell_main( ) is the start of the shell, and the priority is 1.

Parameters whose default values have not been changed do not appear in the MSS by design in XPS. All of the parameters in the platform specification are translated into configuration directives and definitions in header files. Specifically, for Xilkernel, os_config.h and config_init.h are the generated header files that contain C-language equivalents of the specifications in the MSS file. For the system timer device and system interrupt controller device specifications, the header files contain definitions of base addresses of these devices, which are in turn used by the Xilkernel code. These header files are generated under the main processor include directory. In this example, the include directory is microblaze_0/include.

Building the Hardware and Software System

Open the design example EDK project in XPS. To build the hardware and software system from scratch, select **Tools** → **Update Bitstream**. This builds the hardware bitstream, software libraries including Xilkernel, and software applications including xilkernel_demo, and initializes the bitstream. When you are ready to download, select **Tools** → **Download** to initialize your hardware connections and download the bitstream to the target. Now you can continue to the Xilkernel demo.

The Xilkernel Demo

The Xilkernel demo is organized as a software application project. You can browse through the sources as you go through this demo. The sources consist of the following files:

- shell.c – Main controlling thread. Presents a shell with a few simple commands from which you can launch the other demo threads.
- prodcon.c – One or more producer consumer example threads using message queues.
- llist.c – Linked list demo using the buffer memory allocation interfaces.
- sem.c – Semaphore example displaying multiple competing threads using a semaphore to co-ordinate.
- tictac.c – Simple tic-tac-toe game that illustrates how to dynamically assign stack memory to a thread when creating it.
- timertest.c – Simple time management demo.

- `prio.c` – Thread illustrating dynamically changing priorities and priority queues in the kernel structures.
- `mutexdemo.c` – Mutex demo, illustrating pthread mutex locks
- `clock.c` – Simple thread using the second timer device and handling interrupts from it to keep track of wall-clock time. This illustrates user-level interrupt handling.
- `standby.c` – Simple illustration of how priority affects execution of threads.

Connecting to the Processor

1. Connect to the processor in the system using XMD. You can open up XMD by selecting **Tools → XMD** in the XPS main window, or by typing **`xmd`** in the console. To connect to the processor in the system, type **`connect mb mdm`**.
2. Download the kernel image as shown in the following code snippet:

```
XMD%
Loading XMP File..
Loading MHS File..
Processor(s) in System ::

MicroBlaze(1) : microblaze_0
Address Map for Processor microblaze_0
  (0x00000000-0x00007fff) dlmb_cntlr    dlmb
  (0x00000000-0x00007fff) ilmb_cntlr    ilmb
  (0x0c000000-0x0c0000ff) debug_module  mb_opb
  (0x0c000100-0x0c0001ff) RS232 mb_opb
  (0x0c000200-0x0c0002ff) LEDs_4Bit     mb_opb
  (0x0c000300-0x0c0003ff) Push_Buttons_3Bit     mb_opb
  (0x0c000400-0x0c0004ff) extra_timer   mb_opb
  (0x0c000500-0x0c0005ff) system_timer  mb_opb
  (0x0c000600-0x0c0006ff) system_intc   mb_opb
  (0x0e000000-0x0fffffff) SDRAM_8Mx32   mb_opb

Loading MSS File..
XMD% connect mb mdm
Connecting to cable (Parallel Port - LPT1).
Checking cable driver.
 Driver windrvr6.sys version = 6.0.3.0. LPT base address = 0378h.
 ECP base address = FFFFFFFFh.
Cable connection established.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       05026093           8        XC18V04
 3       0124a093          10        XC2VP7
Assuming, Device No: 3 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
-------------------------------------
Version.............................2.00.a
No of PC Breakpoints................4
No of Read Addr/Data Watchpoints...2
No of Write Addr/Data Watchpoints..2
Instruction Cache Support.........off
```

```
Data Cache Support.................off
JTAG MDM Connected to MicroBlaze 1
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD% dow xilkernel_demo/executable.elf
        section, .text: 0x0e000000-0x0e009bf4
Processor (microblaze_0) I-Side Address Map:
        0x00000000 - 0x00007fff
        0x0e000000 - 0x0fffffff
Checking if Program I-Side Memory within Address Range..
Memory Test... PASSED

        section, .rodata: 0x0e009bf4-0x0e00c881
        section, .sdata2: 0x0e00c888-0x0e00c9d8
        section, .data: 0x0e00c9d8-0x0e00caf0
        section, .bss: 0x0e00caf0-0x0e0113c4
Processor (microblaze_0) D-Side Address Map:
        0x00000000 - 0x00007fff
        0x0e000000 - 0x0fffffff
Checking if Program D-Side Memory within Address Range..
Memory Test... PASSED

Downloaded Program xilkernel_demo/executable.elf
Setting PC with program start addr = 0x0e000000
XMD%
```

Begin Execution of the Processor

1. Once the download step completes, the program counter of the processor points to the start address of execution of the Xilkernel image. Make sure that hyperterminal is connected as described in "Defining the Communications Settings" on page 83.
2. To begin the processor's execution, type **con**. You should see the kernel startup and print messages, as displayed below. The shell starts to run, clears the screen, and then presents the prompt for you to type your commands.

```
XMK: Start
XMK: Initializing Hardware...
XMK: Initializing interrupt controller
XMK: Connecting timer interrupt
XMK: Starting the interrupt controller
XMK: Initializing PIT device.
XMK: System initialization...
XMK: Enabling interrupts and starting system...
Idle Task
SHELL: Starting clock...
CLOCK: Successfully registered a handler for extra timer interrupts.
CLOCK: Configuring extra timer to generate one interrupt per second..
CLOCK: Enabling the interval timer interrupt...
```

This displays the output of the first two programs from the execution of Xilkernel. The lines of output that start with `XMK:` are debug messages from the kernel. The kernel begins by initializing the hardware. This step includes initializing the interrupt controller and the Periodic Interval Timer (PIT) device so that the kernel is interrupted at the configured time interval. Then Xilkernel performs a system initialization. This includes initializing data structures inside the kernel, kernel flags and creating the statically specified processes and threads. The threads and processes are created in the same order that they were specified in the MSS file. There is also an idle task created by the kernel. Thus the shell starts.

When the shell starts, it creates a `clock` thread. The `clock` thread illustrates user-level interrupt handling in the following manner:

1. Initializes the extra timer peripheral to generate interrupts every one second.
2. Registers a handler for this interrupt with Xilkernel.
3. Initializes a semaphore with the value 0.
4. Enables the interrupt.
5. Performs a `sem_wait()` operation on the semaphore, causing the clock thread to be blocked.

Upon each interrupt, the `extra_timer_int_handler()` handler resets the timer counter device and invokes a semaphore system call to post to the semaphore. This causes the clock thread to be unblocked once every second. Whenever the clock thread is unblocked, it increments its concept of time, and returns to a wait operation on the semaphore, blocking again. This simple thread illustrates how you can register handlers for other interrupts in the system and perform communication between the interrupt handler and your application threads, such as a semaphore post operation as in this example. Your user-level interrupt handlers cannot invoke blocking system calls.

Using the Application Threads

At this time, there are three active threads: the idle task, the shell, and the clock. The *idle* task never runs while there are other higher priority threads in the system, so you do not see it run. The *shell* runs, providing a prompt on the user computer and responding to user commands. The shell can be used to launch other example threads.The *clock* thread executes every one second, when the extra timer provides an interrupt.

Consider the POSIX threads API before examining the different application threads. The pthread's API implemented in Xilkernel is very close to the POSIX standards; therefore, many applications in the demo can be directly compiled using the compiler for any POSIX operating system and execute without any changes.

One of the first commands used by an application creating threads is:

```
retval = pthread_attr_init(&attr);
```

This system call initializes an attributes structure `pthread_attr_t` that can be used to configure the creation parameters for a new thread. It includes fields to specify the scheduling priority, the detach state of the created thread, and the stack that is to be used. The `pthread_attr_init` system call inserts default attributes in the attribute structure specified. The default attributes are that the scheduling priority is the lowest priority of the system, with no custom stack space and with the threads created in the default detach state, `PTHREAD_CREATE_DETACHED`. This detach state specifies that the thread's resources will be automatically reclaimed and completely flushed from the system upon the thread's termination. Specifying the detach state as `PTHREAD_CREATE_JOINABLE` causes the thread's storage to be reclaimed only when another thread *joins*, as described in "Making a Thread Joinable" on page 91. Use `pthread_attr_init()` only if you want to modify some of these attributes later. With other system calls, you can change just the other creation attributes, and untouched parameters will have default values. If only default attributes are required, then do not change attributes at all; instead, use a **`NULL`** pointer in the `pthread_create()` system call.

The system call `pthread_create()` creates a new thread dynamically, as described here:

```
retval = pthread_create(&tid1,&attr,thread_func,&arg1);
```

At the end of the create system call, a new thread is created, starting from the function specified in the call. This thread starts in the ready state, waiting to be scheduled. The

system call returns the ID of the newly created thread in the location of the first parameter. The thread identifier is of type `pthread_t`. This identifier must be used in identifying the target of many thread operations. The created thread executes based on the scheduling policy. For example, if the thread was created with a higher priority, then it executes immediately, while if it was created with a lower priority, it executes when the scheduling allows it to. A thread exits by invoking the following code:

```
pthread_exit(&ret);
```

Alternatively, a thread that does not use the `pthread_exit()` call in its body invokes it implicitly. Therefore, use `pthread_exit()` only if you want to return a value to the joining thread.

Making a Thread Joinable

If a thread is configured to be *joinable*, then the call to `pthread_exit()` suspends the calling thread without reclaiming resources and context switches to the next schedulable process or thread. The exit routine takes a pointer argument that points to a return value data structure. This pointer argument can be reclaimed by any thread that joins with this thread. Joins are performed with the `pthread_join()` system call, as shown in the following code:

```
print ("-- shell going into wait-mode to join with launched
program.\r\n");
ret = pthread_join (tid, NULL);
```

This code snippet is run by the shell whenever it launches the application threads. It performs a join, which causes the shell to suspend while waiting for the thread. The thread's identifier is the value contained in `tid` to terminate. When the target thread terminates, the shell is unblocked and it goes about reclaiming the resources of the target thread. The second parameter to `pthread_join()`, if provided, is used for reclaiming the return value of the terminated thread.

Xilkernel Demo, continued

1. Type **`help`** in the shell to see possible commands:

```
shell>help

List of commands
run <program_num>: Run a program. For e.g. "run 0" loads the first
program.
time ?HHMM?      : Set/Display the current time.
standby         : Suspend all tasks for 10 seconds. Idle task executes.
clear            : Clear the screen
list             : List the programs loaded for this example system
help             : This help screen
exit             : Exit this shell

shell>
```

2. Set the current time using the **`time`** command:

```
shell>time 0637
shell>time
Time is: 06:37:01.
```

Note: The **`clear`** command clears a screenful of the hyperterminal window.

3. Use the **`shell>standby`** command to have the shell create the standby thread at a higher priority than itself:

```
shell>standby
Idle Task
Idle Task
Idle Task
Idle Task
Idle Task
Idle Task
shell>
```

The `standby` command illustrates strict priority scheduling. The shell is the highest priority process when it is waiting for input. The shell also always performs a `pthread_join` with the thread that it launches to wait for it to complete. Our standby thread, however, performs a `sleep(1000)` call, causing it to suspend for 10 seconds. Therefore, all threads are suspended and only the idle task executes.

4. Type **`list`** in the shell to view a list of programs that the shell can load. When the standby thread terminates, then the shell gets control again and it returns the prompt.

```
shell>list
List of programs loaded in this example system
0: MEMORY : Linked list example using buffer memory allocation
1: SEM    : Semaphores example
2: PRODCON: Producer consumer example using message queues
3: TIMER  : Timer example, illustrating software timers
4: TICTAC : TicTacToe thread with dynamically assigned large stack
5: MUTEX  : Mutex lock demo
6: PRIO   : Priority queue demo
shell>
```

5. Type **`run 0`** in the hyperterminal session to run the memory allocation example. The shell creates a new thread using the `pthread_create` system call with default attributes. The output from the execution of `llist.elf` is shown below:

```
shell>run 0
-- shell going into wait-mode to join with launched program.

LLIST: Sorted Linked List Implementation.
LLIST: Demonstrates memory allocation interfaces.
LLIST: Creating block memory pool....
LLIST: Adding to list 10 statically defined elements....
( 0 1 2 3 4 5 6 7 8 9 )
LLIST: Deleting the list elements.. 0,5,9
LLIST: The list right now is,
( 1 2 3 4 6 7 8 )
LLIST: Adding to list 1535, 661, 2862 and 8.
LLIST: The list right now is,
( 1 2 3 4 6 7 8 8 661 1535 2862 )
LLIST: Deleting block memory pool...
LLIST: Done. Good Bye..
shell>
```

6. This is a linked list program which uses the dynamic buffer memory allocation interface of Xilkernel. It starts off by creating a buffer memory pool of 20 buffers, each of size 8 bytes with the `bufcreate()` system call.

```
i = bufcreate (&mbuft, membuf, 20, 8);
```

If the call returns successfully, it returns the identifier of the created memory buffer in `mbuft`. Every time the application thread needs to add an element to the linked list, it invokes `bufmalloc()`.

```
temp = (list_t *)bufmalloc (MEMBUF_ANY, sizeof(list_t));
```

The first parameter should be the identifier of the memory buffer that was created; however, programs can obtain a buffer from *any* memory buffer available. This program illustrates such a usage of the `bufmalloc()` interface. When you invoke `bufmalloc()` with `MEMBUF_ANY` as the memory buffer identifier, it requests that the kernel finds the requested sized block from any memory buffer available. The linked list thread, `t`, deletes an element and releases storage for that element with the `buffree()` method. The application starts by adding some elements, then removing some, then again adding some and terminating.

Semaphore Example

Among the system calls used in this example thread, of interest are the POSIX compliant semaphore calls, as described below:

- Creating a semaphore and getting a handle to it

  ```
  sem_init (&protect, 1, 1)
  ```

  The `sem_init()` system call creates a new semaphore inside the kernel and returns the identifier of the created semaphore in the location passed as the first parameter. This identifier is of type `sem_t`. The second argument is ignored. The third argument provides the initial value of the semaphore.

- Performing a wait operation on the semaphore

  ```
  sem_wait (&rzvous_1)
  ```

  The wait operation blocks execution of the calling process until the semaphore is successfully acquired. The semaphore's value indicates the current state of the semaphore. If the semaphore value is greater than zero, then the process decrements this value and successfully acquires the semaphore. If it is less than or equal to zero, then the process blocks.

- Performing a post operation on the semaphore

  ```
  sem_post (&rzvous_1)
  ```

  The post operation increments the value of the referenced semaphore. If the semaphore value indicates that there are processes waiting to acquire the semaphore, then it unblocks exactly one waiting process from the waiting queue. This queue is a priority queue when scheduling is priority driven.

- Destroying the semaphore

  ```
  sem_destroy (&protect)
  ```

  This call deallocates the semaphore resources and removes it from the system. This call fails if there are processes blocked on the semaphore.

The semaphore main thread initializes a total of three semaphores. It uses two of these as flags to converge with two dynamically created threads. That is, it creates these semaphores with an initial value of 0. The created threads, as one of their first steps, perform `sem_wait()` on these converged semaphores. The main thread performs all the thread creation and initialization operations, and flags the threads off by running a post on both threads. This ensures that both threads start their critical sections as closely as possible.

The threads then contend for the console to perform some message output operations. To prevent the interleaving of the output on the console, they do this inside a critical section. The protect semaphore is used to ensure mutual exclusion while executing in the critical section. The protect semaphore has an initial value of 1. The threads use `sem_wait()` to acquire the semaphore and then use `sem_post()` to release it when they are completed with the critical section.

The two threads contend a couple of times and then terminate. The main thread, which was waiting to join with these two threads, now reclaims their resources, destroys the semaphores, and terminates. Type **`run 1`** in the hyperterminal shell prompt to run this example:

```
shell>run 1
-- shell going into wait-mode to join with launched program.
SEM: Starting...
SEM: Spawning 1...
SEM: Returned TID: 00000003
SEM: Spawning 2..
SEM: Returned TID: 00000004
SEM: Rendezvousing with 1.
SEM: Thread 1: Doing sem_wait.
SEM: Thread 1: 00000000
SEM: Thread 1: 00000001
SEM: Thread 1: 00000002
SEM: Thread 1: 00000003
SEM: Thread 1: 00000004
SEM: Thread 1: 00000005
SEM: ThreadSEM: Rendezvousing with 2.
 1: 00000006
SEM: Thread 1: 00000007
SEM: Thread 1: 00000008
SEM: Thread 1: 00000009
SEM: Thread 1: Doing sem_post.
SEM: Thread 1: Doing sem_wait.
SEM: Thread 1SEM: Thread 2: Doing sem_wait.
: 00000000
SEM: Thread 1: 00000001
SEM: Thread 1: 00000002
SEM: Thread 1: 00000003
SEM: Thread 1: 00000004
SEM: Thread 1: 00000005
SEM: Thread 1: 00000006
SEM: Thread 1: 00000007
SEM: Thread 1: 00000008
SEM: Thread 1: 00000009
SEM: Thread 1: Doing sem_post.
SEM: Thread 2: 00000000
SEM: Thread 2: 00000001
SEM: Thread 2: 00000002
SEM: Thread 2: 00000003
SEM: Thread 2: 00000004
SEM: Thread 2: 00000005
SEM: Thread 2: 00000006
SEM: Thread 2: 00000007
SEM: Thread 2: 00000008
SEM: Thread 2: 00000009
SEM: Thread 2: Doing sem_post.
SEM: Thread 2: Doing sem_wait.
SEM: Thread 2: 00000000
```

```
SEM: Thread 2: 00000001
SEM: ThreadSEM: Successfully joined with thread 1. Return value of
terminated thread: 00000064
SEM: Thread 2: 00000002
SEM: Thread 2: 00000003
SEM: Thread 2: 00000004
SEM: Thread 2: 00000005
SEM: Thread 2: 00000006
SEM: Thread 2: 00000007
SEM: Thread 2: 00000008
SEM: Thread 2: 00000009
SEM: Thread 2: Doing sem_post.
SEM: Successfully joined with thread 2. Return value of terminated
thread: 00000
0c8
SEM: Releasing misc resources..
SEM: Good bye !
shell>
```

Next, you execute the producer consumer example. This application solves the producer consumer problem using message queues. An application that uses message queues must include the `sys/ipc.h` and `sys/msg.h` header files to make available standard declarations. Consider the POSIX compliant message queue API that is used in this example:

- Creating a message queue and getting a handle to it

  ```
  msgid = msgget (key, IPC_CREAT);
  ```

  The `msg_get()` system call creates a new message queue inside the kernel and returns an identifier to it. When obtaining a message queue, a unique key is used to identify the message queue. Thus two threads, by agreeing upon a command key, can operate on the same message queue and co-ordinate.

- Performing a blocking message send

  ```
  msgsnd (msgid, &msg_p, 4, 0)
  ```

  The message send operation blocks execution of the calling process until the message in the buffer `msg_p` is successfully stored in the message queue. The size of the message to be sent is passed in as an argument. You can make the message send operation non-blocking by using **`IPC_NOWAIT`** in the flags.

- Performing a blocking message receive

  ```
  msgrcv (msgid, &msg_c, 4, 0,0)
  ```

  The receive operation blocks the calling thread until a message is placed on the message queue. The target buffer to store the message and the size of the target buffer are also passed in as parameters. You can make the receive non-blocking by using the **`IPC_NOWAIT`** flag. When this call returns successfully, a message is stored in the requested buffer.

- Retrieving the message queue statistics

  ```
  msgctl(msgid, IPC_STAT, &stats)
  ```

  Statistics about the message queue can be retrieved using the `msgctl()` API. The statistics are placed in the stats structure, which is of type **`msgid_ds`**. The statistics that can be retrieved are the number of messages in the queue, the maximum size of the message queue, the identifier of the last process that performed a send operation on the queue, and the identifier of the last process that performed a receive on the queue.

- Removing the message queue

  ```
  msgctl(msgid, IPC_RMID, NULL)
  ```

  The message queue is removed, again, with the `msgctl()` API. The operation requested should be **`IPC_RMID`**. This forcefully clears the message queue and flushes out messages that are in the queue and processes that are waiting on the message queue, either on a send or a receive.

The message queue application begins by creating two threads: a *producer thread* and a *consumer thread*. The producer thread continues producing all of the english alphabets from *a* to *t* while the consumer consumes the same. One way to synchronize both the producer and the consumer is to use message queues to store whatever the producer produces, and have the consumer consume from the message queue. The message queue acts as the synchronizing agent in this case. Both the producer and the consumer use blocking sends and receives. Therefore, the producer blocks when all the production buffers (message queue) are full and is unblocked whenever the consumer consumes a message. Similarly, the consumer blocks on empty buffers and gets unblocked whenever the producer produces an item. The main thread performs some additional operations, such as requesting the statistics of the message queue, verifying that it cannot acquire an existing message queue in exclusive mode, removing the message queue from the system, and ensuring that processes that are blocked on the queue are flushed out of the queue.

Type **`run 2`** in the hyperterminal shell prompt. The following code displays the output from this example:

```
shell>run 2
-- shell going into wait-mode to join with launched program.
PRODCON: Starting..
PRODCON: Spawning Producer..
PRODCON: Producer -- Start !
PRODCON: Producer -- a
PRODCON: Producer -- b
PRODCON: Producer -- c
PRODCON: Producer -- d
PRODCON: Producer -- e
PRODCON: Producer -- f
PRODCON: Producer -
PRODCON: Returned TID: 00000003
PRODCON: Spawning consumer...
- g
PRODCON: Producer -- h
PRODCON: Producer -- i
PRODCON: Producer -- j
PRODCON: Consumer -- Start !

PRODCON: Returned TID: 00000004
PRODCON: Waiting for these guys to finish.
PRODCON: Producer -- k
PRODCON: Consumer -- a
PRODCON: Producer -- l
PRODCON: Consumer -- b
PRODCON: Producer -- m
PRODCON: Consumer -- c
PRODCON: Producer -- n
PRODCON: Consumer -- d
PRODCON: Producer -- o
PRODCON: Consumer -- e
PRODCON: Producer -- p
```

```
PRODCON: Consumer -- f
PRODCON: Producer -- q
PRODCON: Consumer -- g
PRODCON: Producer -- r
PRODCON: Consumer -- h
PRODCON: Producer -- s
PRODCON: Consumer -- i
PRODCON: Producer -- t
PRODCON: Producer done !
PRODCON: Consumer -- j
PRODCON: Consumer -- k
PRODCON: Consumer -- l
PRODCON: Consumer -- m
PRODCON: Consumer -- n
PRODCON: Consumer -- o
PRODCON: Consumer -- p
PRODCON: Consumer -- q
PPRODCON: Successfully joined with producer. Return value of terminated
thread:
00000000
RODCON: Consumer -- r
PRODCON: Consumer -- s
PRODCON: Consumer -- t
PRODCON: Consumer -- Done. ERRORS (1 indicates error in corresponding
message):
00000000
PRODCON: Consumer -- Signalling main.
PRODCON: Starting other tests..
PRODCON: Trying to create a message queue with the same key.
PRODCON: EXCL mode...
PRODCON: SuccePRODCON: Consumer -- Doing other tests...Blocking on
message queue
ssfully failed :). Errno: 00000011
PRODCON: Retrieving msgid for already created msgQ.
PRODCON: Retrieving statistics from message queue.
PRODCON: MsgQ stats:
        msg_qnum    : 00000000
        msg_qbytes  : 00000000
        msg_lspid   : 00000004
        msg_lrpid   : 00000005
End Stats
PRODCON: Attempting to destroy message Q while a process is occupying
it.
PRODCON: Consumer -- Great! Got Kicked out of msgrcv appropriately.
PRODCON: Consumer -- Terminating.
PRODCON: Successfully removed message queue.
PRODCON: Successfully joined with consumer. Return value of terminated
thread: 0
0000000
PRODCON: Releasing misc resources..
PRODCON: Done !
shell>
```

You can run this example unlimited times. Since the semaphore and producer consumer examples are based on the POSIX API, they should be completely portable onto a POSIX OS without any change.

Timer Example

The following are some of the basic interfaces used in this example:

- Getting number of clock ticks elapsed since kernel start

  ```
  ticks = xget_clock_ticks ();
  ```

  This routine returns the number of times the kernel received a timer interrupt, such as a kernel tick, since the kernel was started. This is a useful measure as a kind of timestamp, or in other time-related calculations.

- Suspending the current task for a certain number of milliseconds

  ```
  sleep(1000);
  ```

  This routine causes the kernel to suspend the invoking thread for the specified number of milliseconds. The thread regains control after the time elapses.

The timer test thread begins by creating multiple threads, each of which reports the current timestamp. Each thread then attempts to sleep for a different time amount. This can be seen by the idle task executing in between the suspension of the threads. The time actually slept by each thread can be compared against wall clock time and verified.

Type **`run 3`** in the shell prompt. The following code displays output from this demo thread:

```
shell>run 3
-- shell going into wait-mode to join with launched program.
TIMER_TEST: Starting...
TIMER_TEST: Creating 3 threads...
Thread 0 starting..
Thread 0 clock ticks currently: 00021e6a. Sleeping for 6 seconds
Thread 1 starting..
Thread 1 clock ticks currently: 00021e6a. Sleeping for 1 second
Thread 2 starting..
Thread 2 clock ticks currently: 00021e6a. Sleeping for 2 seconds
TIMER_TEST: Clock ticks before: 138858.
Thread 1 done.
TIMER_TEST: Clock ticks after: 138868.
TIMER_TEST: Clock ticks before: 138868.
TIMER_TEST: Creating 1 threads...
Thread 1 starting..
Thread 1 clock ticks currently: 00021e74. Sleeping for 1 second
Idle Task
Thread 2 done.
Thread 1 done.
Idle Task
Idle Task
Thread 0 done.
TIMER_TEST: Clock ticks after: 138918.
TIMER_TEST: End demo...
shell>
```

Tic-Tac-Toe Game

This application thread does not illustrate any kernel interfaces; however, it is slightly unique and brings to light a common run-time requirement, the run-time stack of a thread. Since this thread has nine levels of recursion in its MIN-MAX algorithm, it makes sense to give a bigger stack to it. The static pthread stack size specification, however, applies globally to all threads. You control the stack size selectively for a few special threads by using the shell when creating the tictac thread. This example is displayed in the following code snippet:

```
static char tictac_stack[TICTAC_STACK_SIZE]
   __attribute__ ((aligned(4)));
.
.
.
pthread_attr_init (&attr);
if (opt == 4) {
  /* Special attention to tictac thread */
  pthread_attr_setstack (&attr, tictac_stack, TICTAC_STACK_SIZE);
}
ret = pthread_create (&tid, &attr, (void*)proginfo[opt].start_addr,
NULL);
```

You use the `pthread_attr_setstack()` interface to modify the default thread creation stack attributes. You do this by assigning a memory buffer, `tictac_stack` in this example, to be used as a stack and by also telling the implementation the size of the stack. Therefore, you can selectively choose to assign a different stack space to threads, preferring over the default fixed stack size allocated by the kernel.

Type **`run 4`** in the shell prompt. Here is a part of the output from the tictac thread:

```
shell>run 4
-- shell going into wait-mode to join with launched program.
TICTAC: Game Starting
TICTAC: Current board -->
   |   |
------------
   |   |
------------
   |   |
TICTAC: Make a move (1-9):
TICTAC: Current board -->
 A |   |
------------
   |   |
------------
   |   |
TICTAC: I am thinking..
TICTAC: Current board -->
 A |   |
------------
   | X |
------------
   |   |
TICTAC: Make a move (1-9):
```

Mutex Demo Application

The following are some of the basic mutex operations that are performed by this application.

- Initializing a mutex lock

  ```
  pthread_mutex_init (&mutex, NULL)
  ```

  The `pthread_mutex_init()` system call creates a new mutex lock within the kernel and returns the identifier of the mutex in the location passed as the first parameter. The type of this mutex identifier is `pthread_mutex_t`. The initialization call requires a second parameter, which gives a pointer to a mutex initialization attributes structure. Because only the basic mutex types are supported, this parameter is unused and NULL or an attribute initialized with `pthread_mutexattr_init()` should be passed in.

  There is an alternative way to initialize the mutex lock statically, by assigning the value `PTHREAD_MUTEX_INITIALIZER` to the `pthread_mutex_t` structure. This allows the kernel to initialize the mutex lock, in a lazy fashion, whenever it is operated on for the first time.

- Performing a lock operation on the mutex

  ```
  pthread_mutex_lock (&mutex)
  ```

  This call locks the mutex for the calling process or thread and returns. If the mutex is already locked, then the calling process or thread blocks until it is unblocked by some mutex unlock operation.

- Performing a mutex unlock operation

  ```
  pthread_mutex_unlock (&mutex)
  ```

  This call unlocks the mutex, which must be currently locked by the calling process or thread, and returns. If there are processes blocked on the mutex, this call unlocks exactly one of them. If scheduling is priority-driven, then it unlocks the highest-priority process in the wait queue. If scheduling is round-robin, then it unlocks the first process in the wait queue.

- Destroying the mutex lock

  ```
  pthread_mutex_destroy (&mutex)
  ```

  This call destroys the mutex lock. No consideration is given for blocked processes and mutex lock and unlock state.

  The mutex demo application creates some configured number of threads (3 in this case). Each thread contends for a critical section in which it increments a global variable. The main thread looks at this global variable to reach a particular value and then proceeds to join with the threads. The threads use the lock and unlock primitives to access the critical section. The threads also delay inside the critical section to demonstrate contention by other threads. There is also a “bad thread” that tries to do an illegal operation and therefore runs into an error. There are also interfaces for testing the recursive type pthread mutex locks.

  Type **`run5`** to start the mutex demo application. The following code sample displays the output for this example:

  ```
  shell>run 5
  -- shell going into wait-mode to join with launched program.
  MUTEXDEMO: Starting..
  MUTEXDEMO: Launching 3 contending threads..
  MUTEXDEMO: Thread(0) starting...
  ```

```
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication frMUTEXDEMO: Thread(1)
starting...
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting for indication from main.
MUTEXDEMO: Thread(0) waiting fm main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: Thread(1) waiting for indication from main.
MUTEXDEMO: ThreaMUTEXDEMO: Thread(2) starting...
MUTEXDEMO: Thread(2) waiting for indication from main.
MUTEXDEMO: Thread(2) waiting for indication from main.
MUTEXDEMO: Thread(2) waiting for indication fro
MUTEXDEMO: Providing indication to waiting threads to start
contending...
MUTEXDEMO: Waiting for threads to get past critical section...
or indication from main.
MUTEXDEMO: Thread(0) contending...
MUTEXDEMO: Thread(0) in critical section..Will spend some time here.
d(1) waiting for indication from main.
MUTEXDEMO: Thread(1) contending...
m main.
MUTEXDEMO: Thread(2) contending...
MUTEXDEMO: Waiting for threads to get past critical section...
MUTEXDEMO: Thread(1) in critical section..Will spend some time here.
MUTEXDEMO: Thread(0) mutex done...
MUTEXDEMO: Waiting for threads to get past critical section...
MUTEXDEMO: Thread(2) in critical section..Will spend some time here.
MUTEXDEMO: Thread(1) mutex done...
MUTEXDEMO: Thread(2) mutex done...
MUTEXDEMO: BAD_THREAD: Starting..
MUTEXDEMO: I am going to try to unlock a mutex I don't own and force an
error.
MUTEXDEMO: Good! I got the right error!
MUTEXDEMO: BAD_THREAD Done.
MUTEXDEMO: Destroying mutex locks...
MUTEXDEMO: Done. Good Bye !
shell>
```

You can run this demo an unlimited number of times.

Next, run the last application thread. This thread illustrates priority scheduling and dynamic priority. It also illustrates how priority is ingrained even in the wait queues of primitives like semaphores, mutex locks, and message queues.

Create several threads with different priority ordering. All the threads block on a single semaphore which is initialized to 0. The main thread unblocks one single thread using `sem_post`. Subsequently, threads must be unblocked in priority order, instead of the original blocking order. This is confirmed by the outputs issued by the threads. One single

LOW priority thread also tests the `sem_timedwait()` API by repeatedly attempting to acquire the semaphore with a time out specified. This thread is the last thread that acquires the semaphore, but should have timed out several times in between. The main thread joins with the remaining threads and terminates this portion of the test.

In the second stage of the test, the main thread creates another thread at highest priority. It then tests this thread by changing its priority at regular intervals and sleeping in the intervals. This is confirmed by the puppet thread not producing any output in the intervals that the main thread sleeps. The main thread then kills this puppet thread using the *kill* system call.

The following are the relevant interfaces illustrated with this example:

- Specifying thread priority during creation

  ```
  spar.sched_priority = prio[i];
  pthread_attr_setschedparam(&attr,&spar);
  ```

  This snippet illustrates how the priority of a thread is controlled while creating a thread. By setting the priority attribute of the thread creation attributes and passing the same attributes structure to the `pthread_create()` call, the priority of a thread is controlled.

- Dynamically changing the priority of a thread

  ```
  retval = pthread_create(&lowpriotid, &attr, low_prio_thread_func,
  NULL);
  spar.sched_priority = (NUM_PUPPET_THREADS - 1) - i;
  if ((retval = pthread_setschedparam (puppet_tid[i], 0, &spar)) != 0)
  ```

  The `pthread_setschedparam()` call changes the priority of a thread. This snippet shows how the priority of the puppet threads are flipped at runtime by the main thread.

- Enhanced features; killing a thread

  ```
  if (kill (main_thread_pid) != 0) {
  ```

  This snippet shows how the low priority thread kills the main thread. Notice that the identifier passed to the kill interface is a different identifier, not of type `pthread_t`. This is because POSIX threads does not define a `kill()` interface. This is a custom interface exported by Xilkernel. Therefore, use the underlying process context identifier to kill the thread. The process context identifier is retrieved by using the following call:

  ```
  main_thread_pid = get_currentPID ();
  ```

Type **`run 6`** in the shell prompt to run this application thread. The following code snippet displays the output for this example:

```
shell>run 6
-- shell going into wait-mode to join with launched program.
PRIOTEST: Starting...
PRIOTEST: Spawning: 0.
Thread(0): Starting...
PRIOTEST: Returned TID: 3.
PRIOTEST: Spawning: 1.
Thread(1): Starting...
PRIOTEST: Returned TID: 4.
Thread(LOWPRIO): Starting. I should be the lowest priority of them
all...
PRIOTEST: Returned TID: 5.
PRIOTEST: Yawn..sleeping for a while (400 ms)
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
```

```
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
PRIOTEST: Time to wake up the sleeping threads...
Thread(1): Acquired sem. Doing some processing...
Thread(1): Done...
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(0): Acquired sem. Doing some processing...
Thread(0): Done...
PRIOTEST: Joining with threads...
PRIOTEST: Allowing the LOWPRIO thread to finish...
PRIOTEST: Joining with LOWPRIO thread...
Thread(LOWPRIO): TIMEDOUT while trying to acquire sem.
Thread(LOWPRIO): Acquired sem. Doing some processing...
Thread(LOWPRIO): Done...
PRIOTEST: Dynamic priority test phase starting !
PRIOTEST: Initializing barrier semaphore to value 0...
PRIOTEST: Creating puppet threads...
PUPPET(0): Starting...
PUPPET(0): Blocking on semaphore...
PUPPET(1): Starting...
PUPPET(1): Blocking on semaphore...
PRIOTEST: Now I am flipping the priorities of all the blocked puppet
threads.
PRIOTEST: Now I am posting to the semaphores and releasing all the
puppets.
PUPPET(1): Got semaphore...
PUPPET(1): Releasing semaphore...
PUPPET(1): DONE...
PUPPET(0): Got semaphore...
PUPPET(0): Releasing semaphore...
PUPPET(0): GRR Taking revenge for being demoted in priority..
PUPPET(0): Killing main thread before I die...
shell>PUPPET(0): SUCCESS.
PUPPET(0): DONE...

shell>
```

When you are done with the examples, type **`exit`** in the shell prompt. This ends all the application threads and leaves only the idle task to continue.

PowerPC Example Sets

The same example sets that are available for MicroBlaze are available for PowerPC. The examples follow the same configuration and organization in each set, as they do in the MicroBlaze system. There is no difference in the way that the example application threads are written and the way that they execute on the shell. However, you will see that run-time behavior is different, which is basically due to the hardware differences. Refer to the instructions in "Xilkernel with MicroBlaze" on page 78 to run the demo Xilkernel application.

Configuring PowerPC

The software specification for the Xilkernel demo system in PPC405 differs very slightly from that of the MicroBlaze. The following portions of the software specification differ from that of MicroBlaze:

```
PARAMETER proc_instance = ppc405_0

PARAMETER systmr_freq = 100000000

PARAMETER systmr_interval = 80
```

The processor instance for this platform specification must be changed to reflect the PPC405 processor. Since the PPC405 in this design is clocked at 100MHz, we change the `systmr_freq` parameter to reflect this. The interval has also been changed slightly to allow for the same kind of output; even the hardware is different.

Building the Hardware and Software System

Open the design example EDK project in XPS. To build the hardware and software system from scratch, select **Tools → Update Bitstream**. This action builds the hardware bitstream, software libraries including Xilkernel, and software applications including xilkernel_demo, and initializes the bitstream. When you are ready to download, initialize your hardware connections and download the bitstream to the target by selecting **Tools → Download**. Next, run the Xilkernel demo.

Running the Xilkernel Demo

Connect to the processor in the system using XMD. You can open XMD either through the XPS main window by selecting **Tools → XMD**, or by typing **`xmd`** in the console. Connect to the processor in the system with the **`connect ppc hw`** command. Once you have connected to the processor, download the kernel image. The code snippet for this example is displayed below:

```
XMD% connect ppc hw
Connecting to cable (Parallel Port - LPT1).
Checking cable driver.
 Driver windrvr6.sys version = 6.0.3.0. LPT base address = 0378h.
 ECP base address = FFFFFFFFh.
Cable connection established.
INFO:EDK - Assumption: Selected Device 3 for debugging.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       05026093           8        XC18V04
 3       0124a093          10        XC2VP7

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
    ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
```

```
Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
XMD% dow xilkernel_demo/executable.elf
```

Once the download step is complete, the program counter of the processor points to the start address of Xilkernel, since Xilkernel is the last of the programs to be downloaded. Make sure you have hyperterminal connected as described in "Defining the Communications Settings" on page 83. Type **con** to begin the demo. Follow the steps described in "Xilkernel with MicroBlaze" on page 78 to walk through the design examples.

Chapter 7

Using XilMFS

XilMFS is a memory-based file system library that can be used in standalone mode or along with the Xilkernel library. You can use XilMFS functions to read and write files and to create and manage directories and files in RAM. You can also use XilMFS to access read-only files from read-only memory (ROM or Flash).

This chapter contains the following sections.

XilMFS Concepts

XilMFS allows you to treat the RAM, ROM, Flash, or other memory in your system as a collection of files, organized into directories and subdirectories as needed. You can create and delete files from your program, allowing you to log information in a convenient format. You can also read data from files, allowing your software to be more data-driven.

XilMFS requires the use of BRAM or other external RAM for a read-write file system.It works on MicroBlaze™, PowerPC™, and on the host platforms Solaris, Linux, and PC / Cygwin.

When XilMFS is used *in conjunction with* a Flash or ROM for a read-only file system, a separate tool called `mfsgen` is used to create the read-only file system image. This tool is bundled along with the XilMFS libraries in source code form. It is run on the host machine (Solaris, Linux or PC / Cygwin) to generate a file system image that can then be loaded into the target memory.

For example, if you want your MicroBlaze or PowerPC target system to have a read-only file system that contains a directory `d1` which has two files `a.txt` and `b.txt`. You first create the directory `d1` and the files `a.txt` and `b.txt` somewhere on your host machine. Then you run `mfsgen` to create a memory image file containing `d1`, `a.txt`, and `b.txt`. Finally, you download this memory image file to the target memory, and the XilMFS file system is available for use.

Getting Started with XilMFS

If you want to use XilMFS with MicroBlaze or PowerPC, you will need a hardware platform that contains at least the following:

- MicroBlaze or PowerPC processor
- 8KB or larger BRAM or other memory connected to the processor over LMB, PLB, or OPB, with an appropriate memory controller (more or less memory might be needed depending on how XilMFS is configured)

Using XilMFS

There are three steps to using XilMFS with your applications:

1. Configure XilMFS
2. Create your application
3. Create a file system image on your host and download to target (Optional)

The following sections explain each of these steps in detail.

Configuring XilMFS

You can configure Xilkernel using the XPS main window or by editing the MSS file.

Configuring XilMFS in the XPS Main Window

1. In the XPS main window, click the System tab in the tree view panel to open the hardware tree view.
2. Right-click the processor name in the tree view and select **S/W Settings**.
3. The Software Platform Settings dialog box opens.

   The top half of this dialog box displays various devices in the hardware platform and the drivers assigned to them. The bottom right side of the dialog box allows you to select an OS: **Standalone** or others.
4. Select **Standalone**.
5. Click the Library / OS Parameters tab at the top of the dialog box to configure XilMFS.

   The Library / OS Parameters tab shows all the configurable parameters for XilMFS. For each parameter, the parameter type, default value, and a short explanation are shown. These parameters are pre-populated with the default values, and you can change them as needed. Collapse or expand the tree view by clicking the **+** or **-** symbols on the left side.
6. After setting all the parameters, click **OK** to save.

   You can always go back and change the saved values if you want to.

   The base address and numbytes parameters allow you to indicate that a certain region of memory starting at that base address and extending to numbytes bytes is reserved for XilMFS.

   Caution! The current version of the library generator and related tools does not verify that the memory reserved for XilMFS is actually present in the hardware or whether this memory is used by some other code or peripheral device.

7. When you are done with this panel, click **OK**.
8. Select **Tools** → **Generate Libraries and BSPs**.

   This updates the standard Xilinx C libraries to be created, and the XilMFS library functions are included in this library. You do not have to specify any additional libraries in the compiler settings for your application. A file called `mfs_config.h` is generated in the standard include area. You must include this file in your application code to use XilMFS.

Configuring XilMFS Parameters in MSS

All the main window settings corresponding to XilMFS are stored in an MSS file in the libraries section. If you prefer text editing, you can directly create this MSS file and run Libgen in the batch mode flow.

The following code snippet displays a sample MSS fragment for XilMFS:

```
BEGIN LIBRARY
 PARAMETER LIBRARY_NAME = xilmfs
 PARAMETER LIBRARY_VER = 1.00.a
 PARAMETER base_address = 0x10000000
END
```

For more information on Libgen and batch flows, see the "Library Generator" chapter and the "Xilinx Platform Studio" chapter, both in the *Embedded System Tools Guide*.

Creating Your Application

You can use XPS to set up a software project as described in Chapter 3, "Writing Applications for a Platform Studio Design." After configuring your libraries to include XilMFS, you are ready to create your application source files, as described in the following steps:

1. Use your favorite text editor or Integrated Development Environment (IDE) to create your application code.
2. Right-click **Sources** in the project tree view, and select **Add File** to add your source files.
3. Right-click on the project name in the tree view and select **Build Project** to create the executable that contains your application.

To use XilMFS in your application, you must include `mfs_config.h` in your source. This file defines three parameters that determine the size, location, and type of your Memory File System (MFS). These values have been set in XPS or in the MSS file, depending on your choice of tools. The following code displays a sample `mfs_config.h` file:

```
/*****************************************************************
*
* CAUTION: This file is automatically generated by Libgen.
* Version: Xilinx® EDK 6.2 EDK_Gm.10
* DO NOT EDIT.
*
* Copyright (c) 2003 Xilinx, Inc.  All rights reserved.
*
* Description: MFS Parameters
*
*****************************************************************/

#ifndef _MFS_CONFIG_H
#define _MFS_CONFIG_H
#include <xilmfs.h>
#define MFS_NUMBYTES  100000
#define MFS_BASE_ADDRESS 0x10000000
#define MFS_INIT_TYPE  MFSINIT_NEW
#endif
```

To use XilMFS in your application code, call the `mfs_init_fs()` function as shown in the following code sample, using the parameters defined in `mfs_config.h` Create a directory, open a file and write to it, and then open a file and read from it, as shown. See the "LibXil Memory File System" chapter in the *EDK OS and Libraries Reference Manual* for more information.

```
#include <stdio.h>
#include "mfs_config.h"

int main(int argc, char *argv[]) {
  char buf[512];
  char buf2[512];
  int buflen;
  int fdr;
  int fdw;
  int tmp;
  int num_iter;
  mfs_init_fs(MFS_NUMBYTES, MFS_BASE_ADDRESS, MFSINIT_NEW);

  tmp = mfs_create_dir("testdir1");

  fdw = mfs_file_open("testfile1", MFS_MODE_CREATE);
  strcpy(buf,"this is a test string");
  for (num_iter = 0; num_iter < 100; num_iter++)
    tmp = mfs_file_write(fdw, buf, strlen(buf));
  fdr = mfs_file_open("testfile1", MFS_MODE_READ);
  while((tmp= mfs_file_read(fdr, buf2, 512))== 512){
    buf2[511]='\0';
    strcpy(buf, buf2);
  }
  tmp = mfs_file_close(fdr);
}
```

Using a Pre-Built XilMFS Image

1. Create a directory on your host file system containing the files you want in your image. In the following example, the directory is called `testmfs`, and the files are called `a.txt` and `b.txt`:

   ```
   mfs> ls -R testmfs
   testmfs:
   a.txt  b.txt
   ```

2. Run `mfsgen` to create the image and write the image to a file called `image.mfs`.

   ```
   mfs> mfsgen -cvbf image.mfs 10 testmfs
   testmfs:
   a.txt  15
   b.txt  18
   MFS block usage (used / free / total) = 4 / 6 / 10
   Size of memory is 5320 bytes
   Block size is 532
   mfs>
   ```

   In this example, `mfsgen` is called with a block size of 10, resulting in a memory image size of 5320 bytes, or 10 blocks, of which four are used to store the directory `testmfs` and the files `a.txt` and `b.txt`.

3. Load this image file into memory at a suitable address, such as 0x10000000. You can use XMD to download data, or you can use another tool to copy this data to memory.

   For more information about using XMD, refer to the "Xilinx Microprocessor Debugger (XMD)" chapter in the *Embedded System Tools Guide*.

4. In your application, initialize the file system as follows:

   ```
   mfs_init_genimage(5324, 0x10000000, MFSINIT_ROM_IMAGE);
   ```

Now you are ready to use the file system in read-only mode. You can traverse directories, open and close files, and read files using the XilMFS functions in your application.

Note: Linux and Cygwin hosts are typically little-endian machines, while Solaris hosts and MicroBlaze and PowerPC targets are big-endian. When using mfsgen on little-endian machines, use the **`-s`** option to generate an image in big-endian format so it is readily usable on both MicroBlaze and PowerPC targets.

Chapter 8

Simulation in EDK

This chapter describes the basic Hardware Description Language (HDL) simulation flow using the Xilinx® EDK with third-party software. It includes the following sections:

Introduction

Increasing design size and complexity and recent improvements in design synthesis and simulation tools have made HDL the preferred design language of most integrated circuit designers. The two leading HDL synthesis and simulation languages today are Verilog and VHDL. Both of these languages have been adopted as IEEE standards.

The two most common design methods used in both VHDL and Verilog logic designs are structural and behavioral.

Structural design is a method by which a designer instantiates and utilizes predefined components (or structures) and describes how they are to be connected together. For example, a four-bit adder can be created by instantiating four one-bit adders and connecting them together. The one-bit adder itself can be created by instantiating and connecting the appropriate lower-level gates. A design that applies the structural design method can be easily represented as a netlist that describes the components used and their connections.

Behavioral design is a method by which a designer uses a much higher level of abstraction than structural design. The design might contain high-level operations, such as a four-bit addition operator (this is not an adder as in structural design), without having a knowledge of how the design will be implemented. Synthesis tools then take these behavioral designs and infer the actual gate structures and connections to be used, generating a netlist description. As synthesis tools evolve, behavioral designs will become more and more common.

EDK IP components are designed with a mix of behavioral and structural descriptions. For simplicity, we refer to these source descriptions only as behavioral, even though they have some structural instantiations.

EDK Simulation Basics

This section introduces the basic facts and terminology of HDL simulation in EDK. There are three stages in the FPGA design process in which you conduct verification through simulation. Figure 8-1 shows these stages.

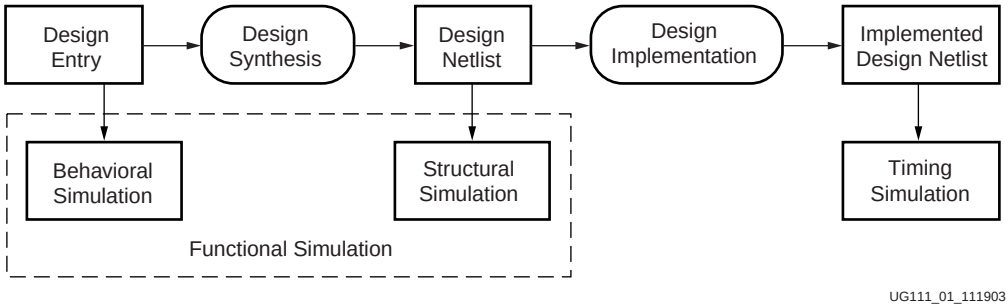

Figure 8-1: **FPGA Design Simulation Stages**

Behavioral Simulation

EDK refers to pre-synthesis simulation as *Behavioral Simulation*. Behavioral simulation is done from an HDL description of the design before it is synthesized to gates. This is the quickest and least detailed HDL simulation method.

Behavioral simulation is used to verify the syntax and functionality without timing information. The majority of the design development is done through behavioral simulation until the required functionality is obtained. Errors identified early in the design cycle are inexpensive to fix compared to functional errors identified during silicon debug.

Structural Simulation

EDK refers to post-synthesis simulation as *Structural Simulation*. After the behavioral simulation is error free, the HDL design is synthesized to gates and a purely structural description of the design is generated. At this point, you can verify what the synthesis tool did with the behavioral design. The post-synthesized structural simulation is a functional simulation that can be used to identify initialization issues and to analyze “don’t care” conditions. Timing information is not used at this simulation level. This method is slower and has more details than behavioral simulation.

Xilinx tools have the ability to write out purely structural HDL netlists for a post-synthesized design. These VHDL or Verilog netlists are written using UNISIM library components, which describe all the low-level hardware primitives available in Xilinx® FPGAs.

Timing Simulation

EDK refers to post-implementation simulation as *Timing Simulation*. This is the same as structural simulation, but with back-annotated timing information. Timing simulation is important in verifying the operation of your circuit after the worst-case place and route (PAR) delays are calculated for your design. The back annotation process produces a netlist of library components annotated in an SDF file with the appropriate block and net delays from the place and route process. The simulation identifies any race conditions and setup-and-hold violations based on the operating conditions for the specified functionality. This simulation method is the slowest and is more detailed.

EDK and ISE Simulation Points

Xilinx® ISE™ supports the following simulation points:

- Behavioral (RTL)
- Post-Synthesis Structural
- Post-NGDBuild (Pre-Map) Structural with no Timing
- Post-Map with Partial Timing
- Post-PAR with Full Timing

Refer to the "Verifying Your Design" chapter in the *Synthesis and Verification Design Guide* in your ISE distribution to learn more about the ISE simulation points.

EDK does not support the third and fourth points above. The other three have the equivalencies described in Table 8-1.

Table 8-1: **Simulation Terminology Equivalency**

Xilinx Simulation Points	EDK Simulation Model	Project Navigator Model
Register Transfer Level (RTL)	Behavioral	Behavioral
Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation	Structural	Not Available
Post-NGDBuild (Pre-Map) Gate-Level Simulation	Not Available	Post-Translate
Post-Map Partial Timing (CLB and IOB Block Delays)	Not Available	Post-Map
Timing Simulation Post-PAR Full Timing (Block and Net Delays)	Timing	Post-PAR

Simulation Libraries

EDK simulation netlists use low-level hardware primitives available in Xilinx® FPGAs. Xilinx provides simulation models for these primitives in the libraries listed in this section.

Xilinx Simulation Libraries

Xilinx® ISE provides the following libraries for simulation:

UNISIM Library

This is a library of functional models used for behavioral and structural simulation. It includes all of the Xilinx Unified Library components that are inferred by most popular synthesis tools. The UNISIM library also includes components that are commonly instantiated such as I/Os and memory cells.

You can instantiate the UNISIM library components in your design (VHDL or Verilog) and simulate them during behavioral simulation. Structural simulation models generated by Simgen instantiate UNISIM library components.

All asynchronous components in the UNISIM library have zero delay. All synchronous components have a unit delay to avoid race conditions. The clock-to-out delay for these is 100 ps.

SIMPRIM Library

This is a library used for timing simulation. This library includes all of the Xilinx Primitives Library components that are used by Xilinx implementation tools.

Timing simulation models generated by Simgen instantiate SIMPRIM library components.

XilinxCoreLib Library

The Xilinx® CORE Generator™ is a graphical intellectual property design tool for creating high-level modules like FIR Filters, FIFOs, CAMs, and other advanced IP. You can customize and pre-optimize modules to take advantage of the inherent architectural features of Xilinx FPGA devices, such as block multipliers, SRLs, fast carry logic, and on-chip, single-port, or dual-port RAM.

The CORE Generator HDL library models are used for behavioral simulation. You can select the appropriate HDL model to integrate into your HDL design. The models do not use library components for global signals.

EDK Library

All EDK IP components are written in VHDL. Some of the IP components distributed with EDK have an encrypted source. EDK provides precompiled libraries of these components that can be used for behavioral simulation. Unencrypted EDK IP components can be compiled using the CompEDKLib utility provided by Xilinx. CompEDKLib deploys the encrypted and unecrypted VHDL only compiled models into a common location.

Compiling Simulation Libraries

Most modern simulators require you to compile the HDL libraries before you can use them for design simulations. The advantages of compiling HDL libraries are speed of execution and economy of memory.

A GUI utility is available for compiling these libraries. To run this utility, do the following:

1. Launch XPS

   You can choose to create a XPS project, but a XPS project is *not* required for this step.

2. If you have an XPS project open, then do the following:
   a. Select **Options** → **Project Options**.
   b. Click the HDL and Simulation tab.
   c. In the Simulation Libraries Path area, select the EDK and Xilinx libraries to compile.
   d. Click **Compile**.
3. If you do not have an XPS project open, then select **Options** → **Compile Simulation Libraries**.

   This opens the Simulation Library Compilation Wizard, which guides you through the compilation process.

Command line executables for compiling the simulation libraries are also available and are described in the subsequent sections. However, Xilinx strongly recommends that you use this GUI tool. Using the GUI ensures that the correct set of tool options are used, and generally enhances your experience through better interaction with XPS.

Compiling Xilinx Simulation Libraries

Xilinx provides *CompXLib* to compile the HDL libraries for all Xilinx-supported simulators. This utility compiles the UNISIM, SIMPRIM and XilinxCoreLib libraries for all supported device architectures using the tools provided by the simulator vendor.

Refer to the "Verifying Your Design" chapter in the *Synthesis and Verification Design Guide* in your ISE distribution to learn more about compiling and using Xilinx simulation libraries.

Library Compilation

To compile your HDL libraries using *CompXLib*, follow these steps. You must have an installation of the Xilinx implementation tools:

1. Run *CompXLib* with the **`-help`** option if you need to display a brief description for the available options:

   ```
   compxlib -help
   ```

2. The *CompXLib* tool uses the following syntax:

   ```
   compxlib -s <simulator> -f <family[:lib],<family[:lib],...|all>
            [-l <language>]
            [-o <compxlib_output_directory>]
            [-w]
            [-p <simulator_path>]
   ```

   Note: Each simulator uses certain environment variables which you must set before invoking CompXLib. Consult your simulator documentation to ensure that the environment is properly set up to run your simulator.

Note: Make sure you use the **`-p <simulator_path>`** option to point to the directory where the modelsim executable is, if it is not in your path.

CompXLib Command Line Example

The following is an example of a command for compiling Xilinx libraries for MTI_SE:

```
> compxlib -s mti_se -f all -l vhdl -w -o .
```

This command compiles all the necessary XILINX libraries into the current working directory.

Compiling EDK Behavioral Simulation Libraries

Before starting behavioral simulation of your design, you must compile the EDK Simulation Libraries for the target simulator. For this purpose, Xilinx provides CompEDKLib, a tool for compiling the EDK HDL-based simulation libraries using the tools provided by the simulator vendor. It compiles all of the HDL sources for all IP that is provided unencrypted and copies the precompiled libraries for encrypted IP to the same location.

Usage

```
compedklib [ -h ] [ -o output-dir-name ] [ -lp repository-dir-name ]
     [ -E compedklib-output-dir-name ] [ -lib core-name ]
     [ -compile_sublibs ] [ -exclude deprecated ]
     -s mti_se|mti_pe|ncsim -X compxlib-output-dir-name
```

This tool compiles the HDL in EDK *pcore* libraries for simulation using the simulators supported by EDK. Currently, the only supported simulators are MTI PE/SE and NCSIM.

CompEDKLib Command Line Examples

Use Case I: Launching the GUI Tool for Compiling Both the Xilinx and EDK Simulation Libraries.

```
compedklib
```

No options are required. This launches the same GUI as when selecting **Options → Compile Simulation Libraries** in XPS.

Note: This is the only mode of `CompEDKLib` that also compiles the Xilinx Libraries. All the other modes only compile the EDK libraries.

Use Case II: Compiling HDL Sources in the Built-In Repositories in the EDK

The most common use case is as follows:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name> -exclude deprecated
```

In this case, the pcores available in the EDK install are compiled and then stored in `<compedklib-output-dir-name>`. The value to the **`-X`** option indicates the directory containing the models outputted by CompXLib such as the `unisim`, `simprim` and `XilinxCoreLib` compiled libraries.

Pcores can be in development, active, deprecated, or obsolete state. Adding **`-exclude deprecated`** has the effect of not compiling deprecated cores. However, if you have deprecated cores in your design, do not use the **`-exclude deprecated`** option.

Use Case III: Compiling HDL Sources in Your Own Repository

If you have your own repository of EDK style pcores, you may to compile them into `<compedklib-output-dir-name>` as follows:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name>
     -E <compedklib-output-dir-name>
     -lp <Your-Repository-Dir>
```

In this form, the **-E** value accounts for the possibility that some of the pcores in your repository may need to access the compiled models generated by Use Case I. This is very likely because the pcores in your repository are likely to refer to HDL sources in the EDK built-in repositories.

You can limit the compilation to named cores in the repository:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name>
     -E <compedklib-output-dir-name>
     -lp <Your-Repository-Dir>
     -lib core1
     -lib core2
```

In this case, the entire repository is read but only the pcores indicated by the **-c** options are compiled.

You can add **-compile_sublibs** to the above to compile the pcores that the indicated pcore depends on.

Other Details

- If the simulator is not indicated, then MTI is assumed.
- You can supply multiple **-X** and **-E** arguments. The order is important. If you have the same pcore in two places, the first one is used.
- Some pcores are secure in that their source code is not available. In such cases, the repository contains the compiled models. These are copied out into `<compedklib-output-dir-name>`.
- If your pcores are in your XPS project, you do not need to do Use Case II. XPS/SIMGEN will create the scripts to compile them.
- Only VHDL is supported.
- The execution log is available in `compedklib.log`.
- Starting in EDK 7.1, the file indicated by your MODELSIM environment variable is not modified by CompEDKLib. However, the simulation scripts generated by SIMGEN will modify the file pointed to by the MODELSIM variable.

Setting Up SmartModels

SmartModels represent integrated circuits and system buses as black boxes that accept input stimulus and respond with appropriate output behavior. Such behavioral models provide improved performance over gate-level models, while at the same time protecting the proprietary designs created by semiconductor vendors. SmartModels connect to hardware simulators through the SWIFT interface, which is integrated with over 30 commercial simulators, including Synopsys VCS, Cadence Verilog-XL, Cadence NcSim, and Model Technology ModelSim SE/PE(1).

The Xilinx® Virtex™-II Pro simulation flow uses Synopsys LMC models to simulate the IBM PowerPC™ microprocessor and Rocket I/O multi-gigabit transceiver. LMC models are simulator-independent models that are derived from the actual design and are therefore accurate evaluation models. To simulate these models, a simulator that supports the SWIFT interface must be used. The SmartModels are included in the ISE Implementation Tools.

The following steps outline how to set up the models. After setting up the SmartModels, refer to "Third-Party Simulators" on page 120 to setup your simulator to use SmartModels.

Windows

On Windows, go to **Start → Control Panel → System**. The System Properties dialog box opens. Select the Advanced tab and click **Environment Variables**. The Environment Variables dialog box opens.

The most commonly used value of LMC_HOME is suggested below. The `compedklib` utility indicates what the correct setting is for your installation.

Set the variables to the following values (if not already set):

```
LMC_HOME %XILINX%\smartmodel\nt\installed_nt
PATH %LMC_HOME%\bin;%LMC_HOME%\lib\pcnt.lib;%PATH%
```

Note: %PATH% represents what your PATH variable had before doing the changes. Make sure you keep this.

Solaris

Set the following variables, if not already set:

```
setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv PATH $LMC_HOME/bin:${PATH}
```

Linux

Set the following variables, if not already set:

```
setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
setenv PATH $LMC_HOME/bin:${PATH}
```

Third-Party Simulators

EDK requires that some third-party simulators are obtained and set up. This section provides some information on these tools.

ModelSim Setup for Using SmartModels

Although ModelSim PE/SE supports the SWIFT interface, certain modifications must be made to the default ModelSim setup to enable this feature. The ModelSim installation directory contains an initialization file called `modelsim.ini`. In this initialization file, you can edit GUI and simulator settings so that they default to your preferences. You must edit

1. ModelSim XE does not support SmartModels.

parts of this `modelsim.ini` file in order for it to work properly with the Virtex-II Pro device simulation models.

Make the following changes to the `modelsim.ini` file located in the `MODEL_TECH` directory. Alternately, you can change the `MODELSIM` environment variable setting in the MTI setup script so that it points to the `modelsim.ini` file located in the project design directory.

1. Edit the statement **`Resolution = ns`** and change it to **`Resolution = ps`**.
2. Comment the statement called `PathSeparator = /` by adding `;` at the beginning of the line.
3. For Verilog designs, enable smartmodels by searching for the variable `Veriuser` and change it to:
   - On Windows: `Veriuser = $MODEL_TECH/libswiftpli.dll`
   - On Solaris: `Veriuser = $MODEL_TECH/libswiftpli.sl`
   - On Linux: `Veriuser = $MODEL_TECH/libswiftpli.sl`
4. Search for the `[lmc]` section and uncomment the `libsm` and `libswift` definitions according to your operating system.

   For Example:

   - On Windows, uncomment these lines:

     ```
     libsm = $MODEL_TECH/libsm.dll
     libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
     ```

   - On Solaris, uncomment these lines:

     ```
     libsm = $MODEL_TECH/libsm.sl
     libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
     ```

   - On Linux, uncomment these lines:

     ```
     libsm = $MODEL_TECH/libsm.sl
     libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
     ```

   Note: It is important to make the changes in the order in which the commands appear in the `modelsim.ini file`. The simulation might not work if the order recommended above is not followed.

After you edit the modelsim.ini file, add some environment variables, if they are not already defined.

- In Windows, go to **Start** → **Control Panel** → **System**. The System Properties dialog box opens. Select the Advanced tab and click **Environment Variables**. The Environment Variables dialog box opens.

  Set the variables to the following values:

  ```
  MODELSIM <path_to_modelsim.ini_script>\modelsim.ini
  PATH <MTI_path>\win32;%PATH%
  ```

  Note: %PATH% represents what your PATH variable had before doing the changes. Make sure you keep this.

- On Solaris and Linux, add the following environment variables to the MTI ModelSim setup script:

  ```
  setenv MODELSIM <path_to_modelsim.ini_script>/modelsim.ini
  setenv PATH <MTI_path>/bin:${PATH}
  ```

Note: Change the parameters included within the brackets <> to match the system configuration.

Note: If the MODELSIM environment variable is not set to point to the file you edited, MTI might not use this INI file, and the simulator will not read the initialization settings required for simulation.

NcSim Setup for Using SmartModels

Add the following environment variables, if they are not already defined.

- In Windows, go to **Start** → **Control Panel** → **System**. The System Properties dialog box opens. Select the Advanced tab and click **Environment Variables**. The Environment Variables dialog box opens.

  Set the variables to the following values:

  ```
  CDS_HOME <Cadence path>
  PATH %LMC_HOME%\bin;%LMC_HOME%\lib\pcnt.lib;
       %CDS_HOME%\tools\bin;%CDS_HOME%\tools\lib;%PATH%
  ```

  Note: %PATH% represents what your PATH variable had before doing the changes. Make sure you keep this.

- On Solaris and Linux, add the following environment variables to the Cadence NcSim setup script:

  ```
  setenv CDS_HOME <Cadence path>
  setenv LD_LIBRARY_PATH ${CDS_HOME}/tools/lib:\
                         $LMC_HOME/lib/sun4Solaris.lib:${LD_LIBRARY_PATH}
  setenv PATH ${LMC_HOME}/bin:${CDS_HOME}/bin:${PATH}
  ```

Note: Change the parameters included within the brackets <> to match the system configuration.

Creating Simulation Models

You can create simulation models using the XPS Graphical User Interface, XPS batch mode, and Simgen. Refer to the "Simulation Model Generator" chapter of the *Embedded System Tools Reference Manual* for details on how to use Simgen.

Creating Simulation Models Using XPS

Do the following to generate simulation models using XPS:

1. Open your project.
2. Create or modify your MHS file in XPS.
3. Select **Options** → **Project Options**.
4. Click the HDL and Simulation tab.
   a. Select a simulator from the **Simulator Compile Script** option list. The options are "ModelSim", "NCSim", or "None."
   b. Specify the path to the Xilinx and EDK precompiled libraries in the **Simulation Libraries Path** box.
   c. Select the Simulation Model: Behavioral, Structural, or Timing.
5. Click **OK** to save your settings .
6. To generate the simulation model, select **Tools** → **Sim Model Generation**.
7. When the process finishes, HDL models are saved in the simulation directory.
8. To open the simulator application, select **Tools** → **Hardware Simulation**.

Creating Simulation Models Using XPS Batch

Do the following to generate simulation models using the XPS batch mode:

1. Open your project by loading your XMP file:

   ```
   XPS% load xmp <filename>.xmp
   ```

2. Set the following simulation values at the XPS prompt.

   a. Select the simulator of your choice using the following command:

      ```
      XPS% xset simulator [ mti | ncs | none ]
      ```

   b. Specify the path to the Xilinx and EDK precompiled libraries using the following commands:

      ```
      XPS% xset sim_x_lib <path>
      XPS% xset sim_edk_lib <path>
      ```

   c. Select the Simulation Model using the following command:

      ```
      XPS% xset sim_model [ behavioral | structural | timing ]
      ```

3. To generate simulation model, type the following:

   ```
   XPS% run simmodel
   ```

4. When the process finishes, HDL models are saved in the simulation directory
5. To open the simulator, type the following:

   ```
   XPS% run sim
   ```

Memory Initialization

After a simulation model is created for a system, the software data must be included in the memory simulation models for the system to run the program.

A C program that has been compiled to generate an executable file can be put inside the simulation models. When running and debugging the software program in the simulator, there will be several iterations of changes and compilation only of the C program and not the hardware design. Creating this data inside the simulation models would be very inefficient. If there are no hardware changes, there is no need to run any hardware creation or implementation tools again.

EDK supports initialization of data in BRAM blocks. BRAM blocks are formed of several BRAMs arranged and configured to create a memory of the size specified in the design. The simulation models that Xilinx provides for these BRAMs use generics in the case of VHDL and parameters in the case of Verilog as the means to initialize them with data.

If there is any program with which to initialize memory, EDK creates separate memory initialization HDL files that include the data for the design.

VHDL Models

For VHDL simulation models, EDK generates a VHDL file that contains a configuration for the system with all initialization values. For a design described in a file called `system.mhs`, there is a VHDL system configuration in the file `system_init.vhd`. The configuration in this file maps the required generics for each of the BRAM blocks connected to a processor in the system.

Verilog Models

For Verilog simulation models, EDK generates a Verilog file that contains an extra module to be used along with the system with all initialization values. For a design described in a file called `system.mhs`, there is a VHDL system configuration in the file `system_init.v`. This module does not have any ports and does not instantiate any other module. It only contains a set of `defparam` statements to define the required parameters for each of the BRAM blocks connected to a processor in the system.

Simulating a Basic System

You have created a hardware system and the software to run on it. You also created simulation models and compile scripts for your hardware simulator. Next you will use a hardware simulator to simulate your design as described in the following sections.

XPS creates all simulation files in the simulation directory in the project directory and inside a subdirectory for each of the simulation models.

`<project_directory>/simulation/<sim_model>`

For example, if the three supported simulation models (behavioral, structural, and timing) are created and your project directory is `proj_dir`, you have the following directory structure:

```
proj_dir/
  simulation/
    behavioral/
    structural/
    timing/
```

Note: There will be other directories in your project directory, but these are not relevant for now.

Older hardware simulators use an interpreted technique. These kinds of simulators have reasonable performance when simulating small designs but become inefficient as hardware designs become larger and larger.

Modern hardware simulators use a compiled technique. They can either translate an HDL representation and generate a C representation that is later translated to native machine code by a C compiler, or they can directly generate native machine code. In either of these cases, simulations runs much faster.

EDK 7.1 generates compile scripts for two compiled simulators: Model Technology's ModelSim and Cadence's NcSim.

Simulation Model Files

Behavioral Model

After a successfully generating behavioral simulation models, the `simulation` directory contains the following files in the `behavioral` subdirectory:

Table 8-2: **Files in the simulation\behavioral Directory**

File	Description
*peripheral*_wrapper.[vhd|v]	Modular simulation files for each component.
elaborate/	Subdirectory with HDL files for components that require elaboration.
system_name.[vhd|v]	The top-level HDL file of the design.
*system_name*_init.[vhd|v]	Memory initialization file, if needed.
system_name.[do|sh]	Script to compile the HDL files and load the compiled simulation models in the simulator.

Structural Model

After a successfully generating structural simulation models, the `simulation` directory contains the following files in the `structural` subdirectory:

Table 8-3: **Files in the simulation\structural Directory**

File	Description
*peripheral*_wrapper.[vhd|v]	Modular simulation files for each component.
system_name.[vhd|v]	The top-level HDL file of the design.
*system_name*_init.[vhd|v]	Memory initialization file, if needed.
system_name.[do|sh]	Script to compile the HDL files and load the compiled simulation models in the simulator.

Timing Model

After successfully generating timing simulation models, the `simulation` directory contains the following files in the `timing` subdirectory:

Table 8-4: **Files in the simulation\timing Directory**

File	Description
system_name.[vhd|v]	The top-level HDL file of the design.
system_name.sdf	The Standard Delay Format (SDF) file with the appropriate block and net delays from the place and route process used only for timing simulation.
*system_name*_init.[vhd|v]	Memory initialization file, if needed.
system_name.[do|sh]	Script to compile the HDL files and load the compiled simulation models in the simulator.

ModelSim

To simulate using ModelSim, do the following steps, which are described in detail in the following subsections.

1. Compile the design
2. Load the design
3. Provide stimulus
4. Run the simulation

Compiling the Simulation Models

EDK generates a `.do` file to compile the generated simulation models. This file contains a set of library mapping and compile commands.

When this file is used, the library mappings are recorded by ModelSim in your `modelsim.ini` file. If you have a `$MODELSIM` environment variable pointing to a `modelsim.ini` file, this file is used by ModelSim. However, if the environment variable is not set, a new `modelsim.ini` file is created by ModelSim in the current directory.

To use the compile script generated for an MHS file named `system.mhs`, type the following at the ModelSim command prompt:

```
ModelSim> do system.do
```

Loading Your Design

Before simulating your design, you must load it. There are differences in how you load the design depending on the language that was used for the top level and whether or not you have memory blocks to initialize.

Loading VHDL Designs

If your top level created by EDK was in VHDL, you should load the design using

```
ModelSim> vsim system_conf
```

This command loads the configuration. The simulator uses the design and all of the parameters specified in the configuration. As a result, you will have all of the memory blocks initialized with the appropriate data.

If you do not have any data to put into memory blocks, you can use the following command to load the design only:

```
ModelSim> vsim system
```

Loading Verilog Designs

If your top level created by EDK was in Verilog, you should load the design using

```
ModelSim> vsim system_conf system glbl
```

This loads the module containing the parameter definitions that initialize the memory blocks in your system, loads the system module itself, and loads the `glbl` module.

If you do not have any data to put into memory blocks, use the following command to load only the system and the `glbl` modules:

```
ModelSim> vsim system glbl
```

Note: The Verilog files written by EDK employ the `uselib` directive to load simulation libraries. Still, you only need to use **`-Lf`** for user-defined libraries.

Providing Stimulus to Your Design

After loading the design and before running the simulation, you must stimulate the clock and reset signals. You can do this with the **force** command:

```
force -freeze sim:/system/sys_clk 1 0, 0 {10 ns} -r 20 ns
force -freeze sim:/system/sys_reset 1
force -freeze sim:/system/sys_reset 0 100 ns, 1 {200 ns}
```

The first command sets the clock signal to a 20 ns period. The second and third commands set the reset signal (active Low) to be active after 100 ns and go back to inactive after 200 ns. You should change these values as appropriate for your simulation.

Note: You can also use ModelSim's GUI to set the values for these signals. ModelSim's user documentation describes how to do it.

Simulating Your Design

To run the simulation, type **run** and the number of time units for which you want the simulation to run. For example,

```
VSIM> run 100
```

Using ModelSim's Script Files

You can put all the commands to compile the HDL files, load the design, give stimulus, and simulate your design in a single `.do` file. For example, you can create a script file called `run.do` with the following:

```
# Compile Design
do system.do

# Load Design
vsim system

# Set Stimulus
force -freeze sim:/system/sys_clk 1 0, 0 {10 ns} -r 20 ns
force -freeze sim:/system/sys_reset 1
force -freeze sim:/system/sys_reset 0 100 ns, 1 {200 ns}

# Run simulation
run 2 us
```

To run this script, on ModelSim's command prompt type:

```
ModelSim> do run.do
```

NcSim

To simulate using ModelSim, you must do the following:

1. Compile the design
2. Elaborate the design
3. Load the design
4. Run simulation

Compiling the Simulation Models

EDK generates a `.sh` file to compile the generated simulation models. The library mappings are created in the `cds.lib` file in the same directory.

To use the compile script generated for an MHS file named `system.mhs`, type the following at the Command line prompt:

```
> sh system.sh
```

Elaborating Your Design

After the design compiles, it requires elaboration. Elaboration is different for VHDL and Verilog, or if you are using swift models.

VHDL

To elaborate the design, type:

```
> ncelab -relax -noxilinxaccl -access +rwc system:structure
```

If the design has any memories to be initialized, type:

```
> ncelab -relax -noxilinxaccl -access +rwc system_conf
```

Verilog

To elaborate the design, type:

```
> ncelab -relax -noxilinxaccl -access +rwc -timescale 1ns/1ps \
   system glbl
```

If the design has any memories to be initialized, type:

```
> ncelab -relax -noxilinxaccl -access +rwc -timescale 1ns/1ps \
   system_conf system glbl
```

Swift Models

If you are using swift models, add the following switch to the `ncelab` command line:

```
-loadpli1 swiftpli:swift_boot
```

Special Cases

For VHDL timing simulation, compile the SDF file using the SDF compiler:

```
> ncsdfc system.sdf
```

and point to the compiled SDF file using a command file and the following switch on the ncelab command line:

```
-sdf_cmd_file sdf.cmd
```

where the `sdf.cmf` file contains the following

```
COMPILED_SDF_FILE = “system.sdf.X”,
   SCOPE = :SYSTEM;
```

Loading Your Design

Before simulating your design, you must load the design. There are differences in how you load the design depending on the language that was used for the top level and whether or not you have memory blocks to initialize.

Loading VHDL Designs

If your top level created by EDK was in VHDL, you should load the design using

```
nclaunch> ncsim system_conf
```

This command loads the configuration. The simulator uses the design and all of the parameters specified in the configuration. As a result, you have all the memory blocks initialized with the appropriate data.

If you do not have any data to put into memory blocks, you can use the following command to load only the design:

```
nclaunch> ncsim system:structure
```

Loading Verilog Designs

If your top level created by EDK was in Verilog, you should load the design using

```
nclaunch> ncsim system_conf:v
```

This loads the module containing the parameter definitions that initialize the memory blocks in your system, loads the system module itself, and loads the `glbl` module.

If you do not have any data to put into memory blocks, use the following command to load only the system and the `glbl` modules:

```
nclaunch> ncsim system:v
```

Simulating Your Design

To run the simulation, type **`run`** and the number of time units for which you want the simulation to run. For example,

```
ncsim> run 100
```

Submodule or Testbench Simulation

In certain cases, the EDK design will not be the top level of your design. This is the case when you instantiate it in a higher level design for synthesis or when you instantiate the EDK design in a testbench for simulation. In either case, the design hierarchy is modified and the EDK design is pushed down the hierarchy one level. This section describes instantiating simulation models in testbenches or higher level systems.

VHDL

You can instantiate an EDK design in a testbench or higher level design using the following syntax:

```
entity <testbench name> is
end <testbench name>;

architecture <testbench architecture> of <testbench name>is
  <design instance name>: <design name>
    generic map (
      <generics>
    )
    port map (
      <ports>
    );
end <testbench architecture>;

configuration <configuration name> of <testbench name> is
  for <testbench architecture>
    for <design instance name> : <design name>
      use configuration work.<design configuration name>;
    end for;
  end for;
end <configuration name>;
```

For example, if the design name is `system`, you write the following code in a VHDL file such as `tb.vhd`:

```
entity tb is
end tb;

architecture STRUCTURE of tb is

  mysystem: system
    port map (
      sys_clk => my_clk,
      sys_rst => my_rst
    );

end STRUCTURE;

configuration tb_conf of tb is
  for STRUCTURE
    for mysystem : system
      use configuration work.system_conf;
    end for;
  end for;
end tb_conf;
```

You should also include all of the other components, signals, and stimulus generation code in the same file.

The EDK system configuration, `system_conf` in this case, defines all of the memory initialization definitions for your design. If this configuration is not used, there will be no data in any memory blocks. If the design has no memory blocks to be initialized, you can omit using the configuration.

Verilog

You can instantiate an EDK design in a testbench or higher level design using the following syntax:

```
module <testbench name> ();

  <design configuration name>
    <design configuration name>();

  <design name>
    <design name>
    (
      .<port1> (),
      .<port2> (),
    );

endmodule
```

For example, if the design name is `system`, you write the following code in a Verilog file, such as `tb.v`:

```
module tb();

  system_conf
    system_conf();

  system
    system
    (
      .sys_clk (my_clk),
      .sys_rst (my_rst)
    );

endmodule
```

You should also include all of the other components, signals, or stimulus generation code in the same file.

The configuration module, `system_conf` in this case, is a module that has all of the memory initialization parameter definitions for your design. If this module is not instantiated, there will be no data in any memory blocks. If the design has no memory blocks to be initialized, you can omit the configuration module instantiation.

ModelSim

This section provides partial instructions on how to compile, load, and simulate using ModelSim. For a more extensive explanation, refer to "Simulating a Basic System" on page 124 or the ModelSim user documentation.

VHDL

- To compile the `tb.vhd` file, type
  `ModelSim> vcom -93 -work work tb.vhd`
- To load the testbench, type
  `ModelSim> vsim tb_conf`

  Or if there are no memory blocks to initialize, type
  `ModelSim> vsim tb`
- To simulate, type
  `VSIM> run 100`

Verilog

- To compile the `tb.v` file, type
  `ModelSim> vlog -incr -work work tb.v`
- To load the testbench, type
  `ModelSim> vsim tb glbl`
- To simulate, type
  `VSIM> run 100`

NcSim

This section provides partial instructions on how to compile, load, and simulate using NcSim. For a more extensive explanation, refer to "Simulating a Basic System" on page 124 or the NcSim documentation.

VHDL

- To compile the `tb.vhd` file, type
  `nclaunch> ncvhdl -v93 -work work tb.vhd`
- To elaborate it, type
  `nclaunch> ncelab tb_conf`

  Or if there are no memory blocks to initialize, type
  `nclaunch> ncelab tb:structure`
- To load the testbench, type
  `nclaunch> ncsim tb_conf`

  Or if there are no memory blocks to initialize, type
  `nclaunch> ncsim tb`
- To simulate, type
  `ncsim> run 100`

Verilog

- To compile the `tb.v` file, type
  `nclaunch> ncvlog -update -work work tb.v`
- To elaborate it, type
  `nclaunch> ncelab tb glbl`
- To load the testbench, type
  `nclaunch> ncsim tb glbl`
- To simulate, type
  `ncsim> run 100`

Using SmartModels

Accessing SmartModel's Internal Signals

The lmcwin Command

The **`lmcwin`** command has the following options:

`lmcwin read <window_instance> [-<radix>]`

The lmcwin read command displays the current value of a window. The optional radix argument is **`-binary`**, **`-decimal`**, or **`-hexadecimal`**, which can be abbreviated.

`lmcwin write <window_instance> <value>`

The **`lmcwin write`** command writes a value into a window. The format of the value argument is the same as that used in other simulator commands which take value arguments.

`lmcwin enable <window_instance>`

The **`lmcwin enable`** command enables continuous monitoring of a window. The specified window is added to the model instance as a signal (with the same name as the window) of type **`std_logic`** or **`std_logic_vector`**. You can then reference this signal in other simulator commands just as you would any other signal.

`lmcwin disable <window_instance>`

The **`lmcwin disable`** command disables continuous monitoring of a window. The window signal is not deleted, but it no longer updates when the model's window register changes value.

`lmcwin release <window_instance>`

Some windows are actually nets, and the lmcwin **`write`** command behaves more like a continuous force on the net. The lmcwin **`release`** command disables the effect of a previous lmcwin **`write`** command on a window net.

Viewing PowerPC Registers in ModelSim

You can enable and view the contents of PowePC registers during simulation. The **`lmc`** command provides access to the PowerPC SmartModel. This section provides some examples of the command usage.

Note: The InstanceName path below is an example only. You should replace <*InstanceName*> with the path that matches your design.

- To see the SmartModel Status Report, type

  ```
  VSIM> lmc -all reportstatus
  ```

  This lists all the available registers and shows the full instance name path to the SmartModel. For example,

  ```
  # InstanceName:
  /system/ppc405_i/ippc405_swift/ppc405_swift_inst
  ```

- To enable a register to be read, use:

  ```
  VSIM> lmcwin enable <InstanceName>/GPR0
  ```

  This enables GPR0 to be read. You must enable each register to which you want access.

- To read the value of a register, type

  ```
  VSIM> examine <InstanceName>/GPR0
  ```

- To add a register to the wave window, use

  ```
  VSIM> add wave <InstanceName>/GPR0
  ```

Chapter 9

Debugging in EDK

This chapter describes the basics in debugging a system designed using EDK. This includes software debugging using Xilinx® Microprocessor Debugger (XMD) and the GNU Debugger (GDB) as well as hardware debugging using the ChipScope Pro™ cores and ChipScope Analyzer tool. This chapter is organized into the following sections.

Introduction

A significant percentage of the design cycle of complex processor systems is spent in debugging and verification. This includes debugging of software running on processors and verification of on-chip bus transactions and hardware logic. Since time-to-market is a key factor in most designs, appropriate tools are needed for both these time-consuming tasks.

EDK provides software tools, XMD, GDB, and Platform Studio™ SDK to debug software running on processors. The Debugging Software sections describe the debugging processes using XMD and GDB. For more information on debugging using Platform Studio SDK, refer to the "Platform Studio SDK Online Documentation."

EDK provides IP cores to access processors, buses, and random logic in your design inside the FPGA. These are described in "Hardware Debugging Using ChipScope Pro."

Debugging PowerPC Software

Hardware Setup Using a JTAG Cable

PowerPC™ 405 has a built-in JTAG port for debugging software. The JTAG ports of the PowerPC™ processors can be chained with the JTAG port present in Xilinx® FPGAs using the FPGA primitive called JTAGPPC. This way, the same JTAG cable used by the Xilinx® ISE™ iMPACT tool for configuring the FPGA with a bitstream can also be used for debugging PowerPC programs.

Connecting JTAGPPC and PowerPC

EDK provides wrappers for connecting the PowerPC and JTAGPPC FPGA primitives to use the previously mentioned JTAG debug setup. If you use the Base System Builder (BSB) to create the design, this connection is done automatically when you select "Use FPGA JTAG pins" for the JTAG Debug Interface option. The following Microprocessor Hardware Specification (MHS) file snippet of an EDK design demonstrates how the two wrappers must be connected for the simple case of FPGAs with a single PowerPC processor.

```
.......
BEGIN ppc405
 PARAMETER INSTANCE = ppc405_0
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE DPLB = plb
 BUS_INTERFACE IPLB = plb
 BUS_INTERFACE JTAGPPC = jtagppc_0_0
 PORT PLBCLK = sys_clk_s
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT CPMC405CLOCK = proc_clk_s
END

BEGIN jtagppc_cntlr
 PARAMETER INSTANCE = jtagppc_0
 PARAMETER HW_VER = 2.00.a
 BUS_INTERFACE JTAGPPC0 = jtagppc_0_0
END
```

For more information about the JTAGPPC connection, refer the PPC405 JTAG port section of the *PowerPC 405 Processor Block Reference Guide*. For more information about MHS files and making port connections in Platform Studio, refer the *Embedded Systems Tools Reference Manual*.

Software Setup

Once the FPGA is configured with a bitstream containing the above JTAG connection, you can use XMD to connect to the PowerPC processor and debug programs using GNU Debugger (GDB).

Example Debug Session

The example design contains a PowerPC 405 processor, on-chip BRAM, UART peripheral, OPB_MDM peripheral, and GPIO peripheral. The OPB_MDM peripheral has the UART enabled and is used as STDIN / STDOUT for the processor. The peripheral communicates with XMD over the JTAG interface. The PowerPC PIT timer is used to generate interrupts at equal intervals.

1. Specify XMD Debug Options
   a. In XPS, select **Options** → **XMD Debug Options**.
   b. Select **Hardware** as the connection type.

Figure 9-1: **XMD Debug Options, PowerPC Hardware Target**

2. Start XMD and connect to the PowerPC processor.

   In XPS, select **Tools → XMD**. This launches XMD with the **`xmd -xmp <system.xmp> -opt etc/xmd_ppc405_0.opt`** option.

```
XMD%
Loading XMP File..
Processor(s) in System ::
PowerPC405(1) : ppc405_0
Address Map for Processor ppc405_0
  (0x40000000-0x4000ffff) LEDs_4Bit     plb->plb2opb->opb
  (0x40400000-0x4040ffff) debug_module  plb->plb2opb->opb
  (0x40600000-0x4060ffff) RS232 plb->plb2opb->opb
  (0xffffc000-0xffffffff) plb_bram_if_cntlr_1   plb

Executing Connect Cmd: connect ppc hw -cable type xilinx_parallel port
LPT1 -debugdevice cpunr 1
Connecting to cable (Parallel Port - LPT1).
Checking cable driver.
 Driver windrvr6.sys version = 6.2.2.2. LPT base address = 03BCh.
 ECP base address = 07BCh.
 ECP hardware is detected.
Cable connection established.
Connecting to cable (Parallel Port - LPT1) in ECP mode.
Checking cable driver.
 File C:/Xilinx7.1/bin/nt/xpc4drvr.sys not found.
 Driver xpc4drvr.sys version = 1.0.4.0. LPT base address = 03BCh.
 Cable Type = 1, Revision = 3.
Cable connection established.
INFO:EDK - Assumption: Selected Device 3 for debugging.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05046093           8        XCF04S
 2       05046093           8        XCF04S
 3       0124a093          10        XC2VP7
```

```
XMD: Connected to PowerPC target. Processor Version No : 0x20010820
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
   ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD%
```

Now XMD is connected to PowerPC and has started a GDB server port 1234 for source level debugging using the GDB console. For more information on options for the `connect` command, refer to the *"XMD"* chapter of the *Embedded System Tools Reference Manual*.

3. Start the STDIN / OUT terminal in XMD.
   a. In the XMD shell, type **`connect mdm -uart`**. XMD connects to the UART interface of OPB_MDM.

      ```
      Connected to the JTAG MicroProcessor Debug Module (MDM)
      No of processors = 0
      Connected to MDM UART target
      ```

   b. In the XMD shell, type **terminal**. This opens a terminal shell.

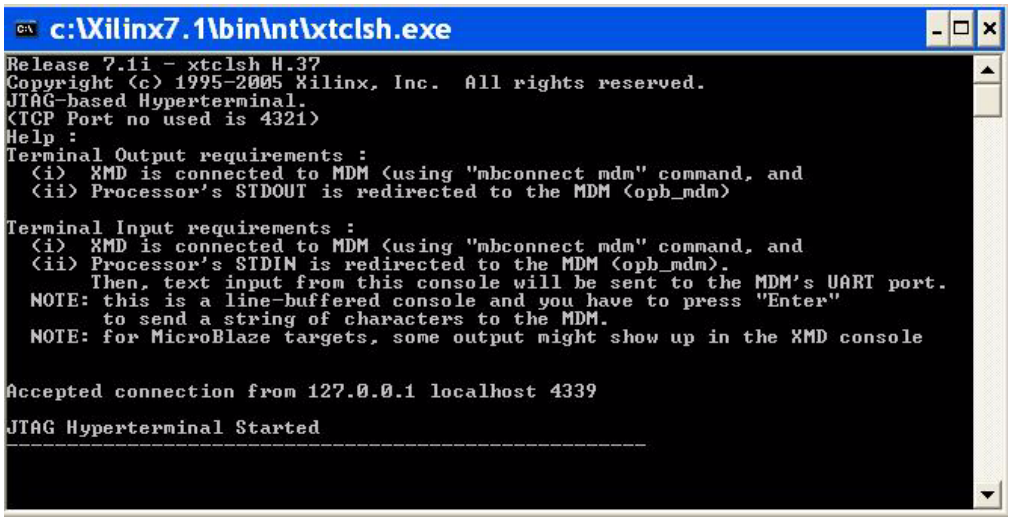

Figure 9-2: **JTAG-Based Hyperterminal**

4. Start the GDB (powerpc-eabi-gdb) console.

   In XPS, select **Tools** → **Software Debugger**.

5. Connect GDB to XMD's GDB server TCP port 1234 and debug the program locally (hostname=localhost) or remotely (hostname = IP addr).
6. For help in using GDB, select **Help** → **Help Topics** in GDB.

Advanced PowerPC Debugging Tips

PowerPC Simulator

You can use the PowerPC 405 Instruction Set Simulator to debug and verify your software before you have a Virtex™-II Pro hardware board with a working PowerPC system. The ISS supports only the processor and memory models by default. Therefore, the default setup cannot be used to debug firmware that accesses peripheral cores.

For more information about using the PowerPC ISS with XMD / GDB and its restrictions, refer to the *"XMD"* chapter in the *Embedded System Tools Reference Manual*. For more information about the PowerPC ISS and how to extend it to add bus models for peripherals, refer to the IBM PowerPC 405 documentation.

Configuring Instruction Step

XMD supports two Instruction Step modes. You can use the **debugconfig** command to select between the modes. The two modes are:

- Instruction step with Interrupts disabled

  This is the default mode. In this mode the interrupts are disabled.

- Instruction step with Interrupts enabled

  In this mode the interrupts are enabled during step operation. XMD sets a hardware breakpoint at the next instruction and executes the processor. If an interrupt occurs, it is handled by the registered interrupt handler. The program stops at the next instruction.

  Note: The Instruction Memory of the program should be connected to the processor d-side interface.

```
.XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Disabled

XMD% debugconfig -step_mode enable_interrupt
XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Enabled
Memory Data Width Matching... Disabled
```

Note: This configuration also applies to MicroBlaze™ Hardware Targets.

Configuring Memory Access

XMD supports handling different memory data width accesses. The supported data widths are Word (32 bits), Half Word (16 bits) and Byte (8 bits). By default, XMD uses Word accesses when performing memory read / write operations. You can use the **debugconfig** command to configure XMD to match the data width of memory operation. This is usually necessary for accessing Flash devices of different data widths.

```
XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode................... Interrupt Disabled
Memory Data Width Matching... Disabled

XMD% debugconfig -memory_datawidth_matching enable
XMD% debugconfig
Debug Configuration for Target 0
--------------------------------
Step Mode................... Interrupt Disabled
Memory Data Width Matching... Enabled
```

Note: This configuration also applies to MicroBlaze Hardware Targets

Support for Running Programs from ISOCM and ICACHE

There are restrictions on debugging programs from PowerPC ISOCM memory and Instruction CACHEs, particularly that you cannot use software breakpoints. In such cases, XMD can automatically set hardware breakpoints, if the address ranges for the ISOCM or ICACHES are provided as options to the connect command in XMD. In this case of ICACHE, this is only necessary if you try to run programs completely from the ICACHE by locking its contents in ICACHE.

For more information about the specific XMD connect options, refer to the "XMD" chapter in the *Embedded System Tools Reference Manual*. Refer also to the "Setting Debug Options and Invoking XMD" section of the *Platform Studio Online Help*.

Accessing DCR Registers, TLB, ISOCM, Instruction and Data Caches

The previously mentioned special features of the PowerPC can be accessed from XMD by specifying the appropriate options to the `connect` command in the XMD console. Refer to the "XMD" chapter in the *Embedded System Tools Reference Manual* for more information and example debug sessions to demonstrate the usage.

Debugging Setup for Third-Party Debug Tools

In order to use third-party debug tools such as Wind River SingleStep and Green Hills Multi, Xilinx recommends that you bring the JTAG signals of the PowerPC out of the FPGA as User IO to appropriate debug connectors on the hardware board. Apart from the JTAG signals, TCK, TMS, TDI, and TDO, you must also bring the DBGC405DEBUGHALT and C405JTGTDOEN signals out of the FPGA as User IO. In the case of multiple PowerPCs, Xilinx recommends that you chain the PowerPC JTAG signals inside the FPGA. For more information about connecting the PowerPC JTAG port to FPGA User IO, refer to the PPC405 JTAG port section of the *PowerPC 405 Processor Block Reference Guide*.

Note: You must NOT use the JTAGPPC module while bringing the PowerPC JTAG signals out as User IO.

Debugging MicroBlaze Software

MicroBlaze is a soft processor implemented using FPGA logic and is configurable according to your requirements. One of the configurable options is the debug mode. You can use the hardware debug logic in MicroBlaze for better control over the processor resources and advanced debug features. Alternatively, you can debug programs using XMDStub (a ROM monitor) over a UART (JTAG-based or RS232-based) and trade-off control and features for debug logic area.

Hardware Setup for MDM-Based Debugging Using JTAG (HW-Based)

Connecting MDM and MicroBlaze

Microprocessor Debug Module (MDM) is the core that facilitates the use of JTAG for debugging one or more MicroBlaze processors. The MDM core instantiates the FPGA primitive named BSCAN_VIRTEX[2] to connect the FPGA's JTAG port to the MicroBlaze debug ports. Only a single MDM core can be instantiated in a system and that core can in turn connect to multiple MicroBlaze processors. When you use the BSB to create a MicroBlaze system, the MicroBlaze-MDM connection is automatically setup when you select **Use On-chip Debug Logic** as the debug option. The following MHS snippet of an EDK design demonstrates how to connect the MDM and MicroBlaze for a simple case of a single MicroBlaze system with hardware debug logic enabled.

```
.....
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 2
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
 PORT DBG_CAPTURE = DBG_CAPTURE_s
 PORT DBG_CLK = DBG_CLK_s
 PORT DBG_REG_EN = DBG_REG_EN_s
 PORT DBG_TDI = DBG_TDI_s
 PORT DBG_TDO = DBG_TDO_s
 PORT DBG_UPDATE = DBG_UPDATE_s
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0x80002000
 PARAMETER C_HIGHADDR = 0x800020ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT DBG_CAPTURE_0 = DBG_CAPTURE_s
 PORT DBG_CLK_0 = DBG_CLK_s
 PORT DBG_REG_EN_0 = DBG_REG_EN_s
```

```
 PORT DBG_TDI_0 = DBG_TDI_s
 PORT DBG_TDO_0 = DBG_TDO_s
 PORT DBG_UPDATE_0 = DBG_UPDATE_s
END
.....
```

Software Setup for MDM-Based Debugging

Once the FPGA is configured with a bitstream containing the previous debug setup, you can use XMD to connect to the MicroBlaze processor and debug programs using GDB.

Example Debug Session

The example design contains a MicroBlaze processor, on-chip BRAM, UART peripheral, OPB_MDM peripheral, and GPIO peripheral. The OPB_MDM peripheral has the UART enabled and is used as STDIN / STDOUT for the processor. The peripheral communicates with XMD over the JTAG interface.

1. Specify XMD Debug Options
   a. In XPS, select **Options** → **XMD Debug Options**.
   b. Select **Hardware** as the connection type.

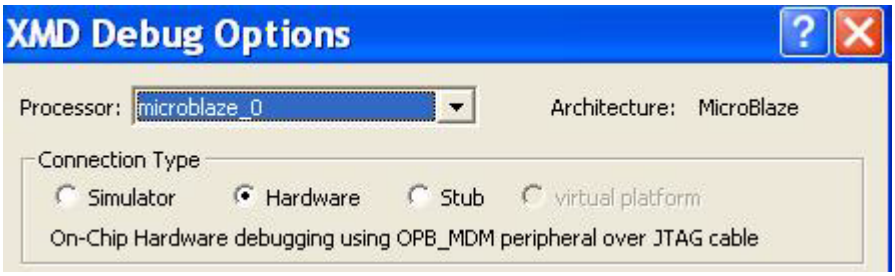

Figure 9-3: **XMD Debug Options, MicroBlaze Hardware Target**

2. Start XMD and connect to the MicroBlaze processor (via MDM).

   In XPS, select **Tools** → **XMD**. This launches XMD with the **`xmd -xmp <system.xmp> -opt etc/xmd_microblaze_0.opt`** option.

```
XMD%
Loading XMP File..
Processor(s) in System ::

MicroBlaze(1) : microblaze_0
Address Map for Processor microblaze_0
  (0x00000000-0x00003fff) dlmb_cntlr    dlmb
  (0x00000000-0x00003fff) ilmb_cntlr    ilmb
  (0x84000000-0x8400ffff) opb_timer_1   mb_opb
  (0x84010000-0x8401ffff) opb_intc_0    mb_opb
  (0x84020000-0x8402ffff) debug_module  mb_opb
  (0x84030000-0x8403ffff) RS232 mb_opb
  (0x84040000-0x8404ffff) LEDs_4Bit     mb_opb

Executing Connect Cmd: connect mb mdm -cable type xilinx_parallel port
LPT1
-debugdevice cpunr 1
Connecting to cable (Parallel Port - LPT1).
```

```
Checking cable driver.
 Driver windrvr6.sys version = 6.2.2.2. LPT base address = 03BCh.
 ECP base address = 07BCh.
 ECP hardware is detected.
Cable connection established.
Connecting to cable (Parallel Port - LPT1) in ECP mode.
Checking cable driver.
 File C:/Xilinx7.1/bin/nt/xpc4drvr.sys not found.
 Driver xpc4drvr.sys version = 1.0.4.0. LPT base address = 03BCh.
 Cable Type = 1, Revision = 3.
Cable connection established.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05046093           8        XCF04S
 2       05046093           8        XCF04S
 3       0124a093          10        XC2VP7
Assuming, Device No: 3 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
--------------------------------------
Version.............................4.00.a
No of PC Breakpoints................2
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.........off
Data Cache Support.................off
Exceptions  Support................off
FPU  Support.......................off
FSL DCache Support.................off
FSL ICache Support.................off
Hard Divider Support...............off
Hard Multiplier Support............on
Barrel Shifter Support.............off
MSR clr/set Instruction Support....off
Compare Instruction Support........off
JTAG MDM Connected to MicroBlaze 1
Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD%
```

Now XMD is connected to the MicroBlaze processor and has started a GDB server port 1234 for source-level debugging using the GDB console.

3. Start the STDIN / OUT terminal in XMD.

   In the XMD shell, type **`terminal`**. This opens a terminal shell.

   Note: Explicit connection to MDM is not necessary, since MicroBlaze is connected via the MDM interface.

4. Start the GDB (mb-gdb) console.

   In XPS, select **Tools → Software Debugger** or type **`mb-gdb`**.

5. Connect GDB to XMD's GDB server TCP port 1234 and debug the program locally (hostname=localhost) or remotely (hostname = IP addr).

6. For help using GDB, select **Help** → **Help Topics** in GDB.

   For more information, refer to the following reference guides:

   - *MicroBlaze Processor Reference Guide*
   - *MDM* datasheet
   - "XMD" chapter in the *Embedded System Tools Reference Manual*

Hardware Setup for XMDStub-Based Debugging Using JTAG (SW-Based)

Connecting MDM (as JTAG-Based UART) and MicroBlaze

As previously mentioned, MicroBlaze programs can also be debugged using a ROM monitor program called XMDStub. In this case, the hardware debug logic in MicroBlaze need not be enabled - parameter C_DEBUG_ENABLED can be set to "0" or false. Also, the C_MB_DBG_PORTS parameter on the MDM core can be set to "0" to reduce logic usage. But the UART interface on the MDM must be enabled as follows:

```
.....
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_DEBUG_ENABLED = 0
BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
PARAMETER C_MB_DBG_PORTS = 0
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0x80002000
 PARAMETER C_HIGHADDR = 0x800020ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
END
.....
```

Configuring XMDStub Software Settings in an MSS File

In the Software Platform Settings dialog box in XPS, set up the `xmdstub_peripheral` parameter to enable XMDStub-based debugging.

```
BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = microblaze_0
 PARAMETER COMPILER = mb-gcc
 PARAMETER ARCHIVER = mb-ar
 PARAMETER XMDSTUB_PERIPHERAL = debug_module
END
```

Software Setup for XMDStub-Based Debugging

For XMDStub-based debugging, you must connect to the MicroBlaze processor from XMD using the **`connect mb stub -comm jtag`** command.

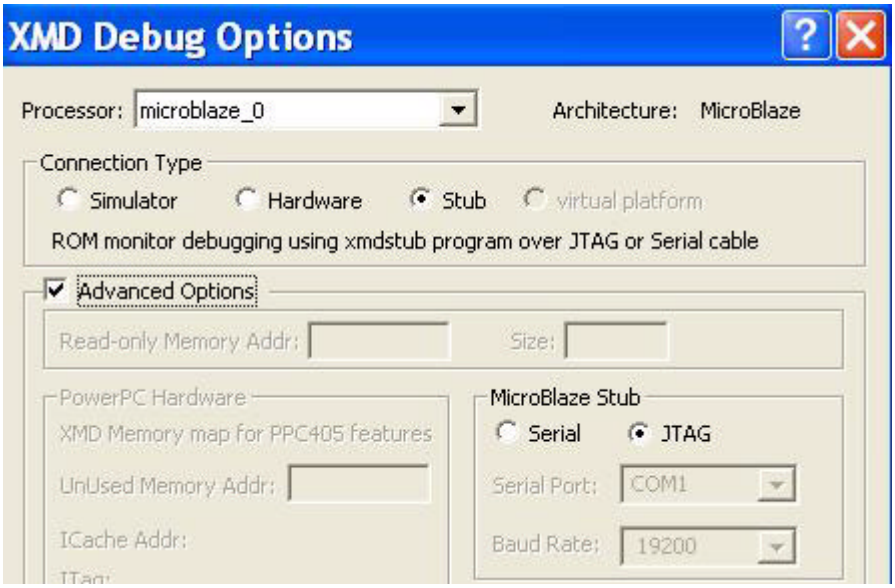

Figure 9-4: **XMD Debug Options, MicroBlaze Stub**

Connecting GDB to XMD is similar to the hardware-based debugging case described earlier.

For more information on the connect command options, refer to the MicroBlaze Stub Target section in the "XMD" chapter of the *Embedded System Tools Reference Manual*.

Using Serial Cable for XMDStub Debugging

In the absence of a JTAG connection to the FPGA containing the MicroBlaze system, you can also use a standard serial cable to debug programs on MicroBlaze. In this case, you must instantiate the UARTlite peripheral (opb_uartlite) in the EDK design and connect it to the OPB bus. As described above in the Software settings MSS file, the UARTlite peripheral must be chosen as the xmdstub_peripheral. In XMD, use the **`connect stub -comm serial`** command.

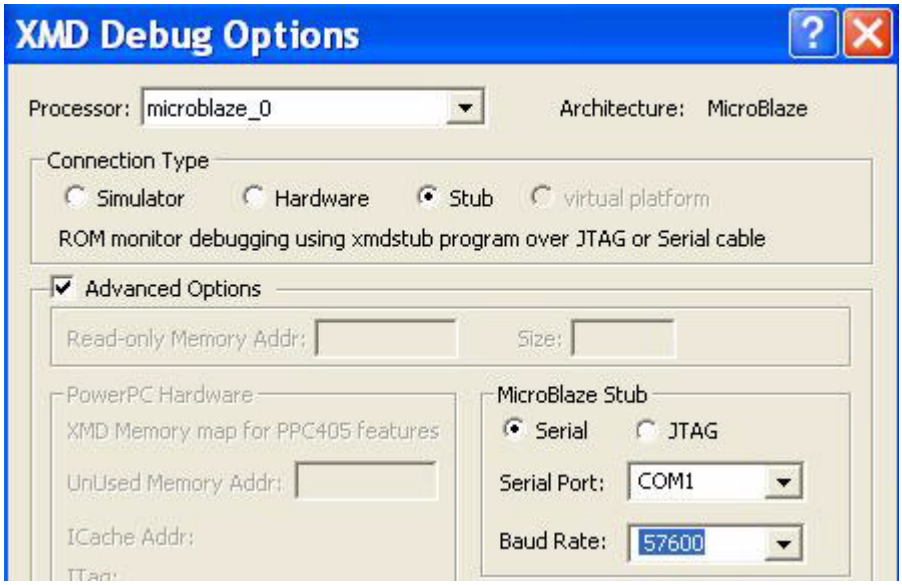

Figure 9-5: **XMD Debug Options, MicroBlaze Serial**

Connecting GDB to XMD is similar to the hardware-based debugging case described in "Hardware Setup for MDM-Based Debugging Using JTAG (HW-Based)" on page 141.

For more information on the connect command options, refer to the MicroBlaze Stub Target section in the "XMD" chapter of the *Embedded System Tools Reference Manual*.

Debugging Software on a Multi-Processor System

XMD enables debugging multiple heterogeneous processors in a system. You can use an XMD terminal to connect to multiple processors or use a separate XMD terminal for each processor. Use GDB for debugging program on processor.

Hardware Setup

Multiple PowerPC Debug

In order to debug multiple PowerPCs using a single JTAG cable, the JTAG signals of all the PowerPCs, specifically TDI and TDO, must be chained together and connected to the FPGA's JTAG port using the JTAGPPC module.

For more information about chaining the JTAG ports of multiple PowerPCs, refer to the PPC405 JTAG port section of the *PowerPC 405 Processor Block Reference Guide* and the JTAGPPC Controller Data Sheet, available online at *http://www.xilinx.com/bvdocs/ipcenter/data_sheet/jtagppc_cntlr.pdf*.

Multiple MicroBlaze Debug

In order to debug multiple MicroBlaze processors using a single JTAG cable, use MDM. All of the MicroBlaze processors should be connected to MDM. The following MHS snippet of an EDK design demonstrates how to connect the MDM and two MicroBlaze processors.

```
.....
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 2
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
 PORT DBG_CAPTURE = DBG_CAPTURE_s_0
 PORT DBG_CLK = DBG_CLK_s_0
 PORT DBG_REG_EN = DBG_REG_EN_s_0
 PORT DBG_TDI = DBG_TDI_s_0
 PORT DBG_TDO = DBG_TDO_s_0
 PORT DBG_UPDATE = DBG_UPDATE_s_0
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_MB_DBG_PORTS = 2
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0x80002000
 PARAMETER C_HIGHADDR = 0x800020ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT DBG_CAPTURE_0 = DBG_CAPTURE_s_0
 PORT DBG_CLK_0 = DBG_CLK_s_0
 PORT DBG_REG_EN_0 = DBG_REG_EN_s_0
 PORT DBG_TDI_0 = DBG_TDI_s_0
 PORT DBG_TDO_0 = DBG_TDO_s_0
 PORT DBG_UPDATE_0 = DBG_UPDATE_s_0
PORT DBG_CAPTURE_1 = DBG_CAPTURE_s_1
 PORT DBG_CLK_1 = DBG_CLK_s_1
 PORT DBG_REG_EN_1 = DBG_REG_EN_s_1
 PORT DBG_TDI_1 = DBG_TDI_s_1
 PORT DBG_TDO_1 = DBG_TDO_s_1
 PORT DBG_UPDATE_1 = DBG_UPDATE_s_1
END

BEGIN microblaze
 PARAMETER INSTANCE = microblaze_1
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 2
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
 PORT DBG_CAPTURE = DBG_CAPTURE_s_1
 PORT DBG_CLK = DBG_CLK_s_1
```

```
PORT DBG_REG_EN = DBG_REG_EN_s_1
PORT DBG_TDI = DBG_TDI_s_1
PORT DBG_TDO = DBG_TDO_s_1
PORT DBG_UPDATE = DBG_UPDATE_s_1
.....
```

Software Setup

While debugging a multi-processor system, you can connect to a specific processor from XMD / GDB by providing options to the `connect` command in the XMD console. The option to specify a particular processor from the XMD console is **`-debugdevice cpunr <processor number>`**. The exact value for the **`processor number`** depends on the order in which the PowerPC processors were chained in the hardware system or the order in which the MicroBlaze processors are connected to MDM. For each processor connection, XMD opens a GDB server at different TCP port numbers such as `1234` and `1235`. From GDB, connect to the appropriate processor using the different TCP ports.

Debugging Software on Virtual Platform

XMD enables debugging of Virtual Platform systems. You can connect to Virtual Platform using the **`vpconnect`** command in an XMD shell. Use GDB or Platform Studio SDK for debugging the program.

For more information on generating Virtual Platform, refer to the "Virtual Platform Generator" chapter of the *Embedded System Tools Reference Manual*.

Hardware Debugging Using ChipScope Pro

EDK provides five ChipScope Pro cores for hardware debugging:

1. chipscope_icon — Provides communication to other ChipScope cores
2. chipscope_opb_iba — Facilitates monitoring of OPB Bus transactions
3. chipscope_plb_iba — Facilitates monitoring of PLB Bus transactions
4. chipscope_vio — Facilities Virtual IO to probe FPGA signals via JTAG
5. chipscope_ila — Facilities monitoring individual non-bus signals in the processor design

For more information on each of these cores, refer to the Debug and Verification category of the *Processor IP Reference Guide* or the *ChipScope Pro Software and Cores User Manual* in the ChipScope installation.

Instantiating ChipScope Pro Cores in an EDK Design

Connecting ChipScope ICON and ChipScope OPB or PLB IBA

ChipScope ICON core provides the JTAG connectivity for all other ChipScope cores and therefore is necessary to use any of the other ChipScope cores. A single ICON can connect to one or more ChipScope cores via one or more 36-bit *control* connections. The OPB or PLB IBA must be connected to the appropriate bus using a monitor (Bus Analyzer) connection. The following example MHS file snippet demonstrates an example setup for a MicroBlaze design containing the MDM debug peripheral and the ChipScope ICON and IBA cores.

```
BEGIN chipscope_icon
 PARAMETER INSTANCE = chipscope_icon_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_SYSTEM_CONTAINS_MDM = 1 # Needed if opb_mdm is also present
 PARAMETER C_NUM_CONTROL_PORTS = 1
 PORT control0 = opb_iba_control
END

BEGIN chipscope_opb_iba
 PARAMETER INSTANCE = chipscope_opb_iba_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_DATA_UNITS = 1
 PARAMETER C_CONTROL_UNITS = 1
 PARAMETER C_WRDATA_UNITS = 1
 PARAMETER C_RDDATA_UNITS = 1
 PARAMETER C_ADDR_UNITS = 1
 BUS_INTERFACE MON_OPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT chipscope_icon_control = opb_iba_control
END
```

The following figures demonstrate how to set this up in Platform Studio.

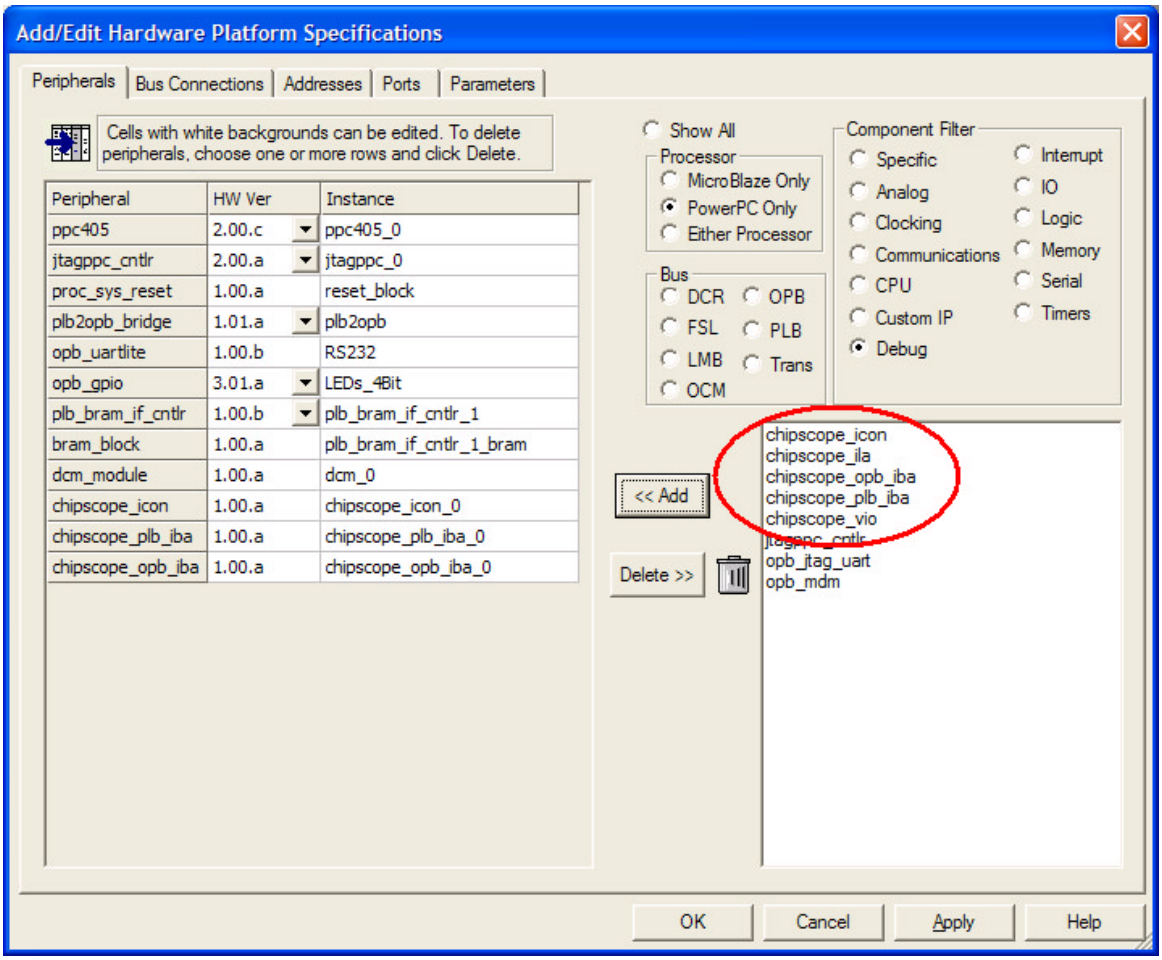

Figure 9-6: **Adding ChipScope ICON and OPB/PLB IBA Cores**

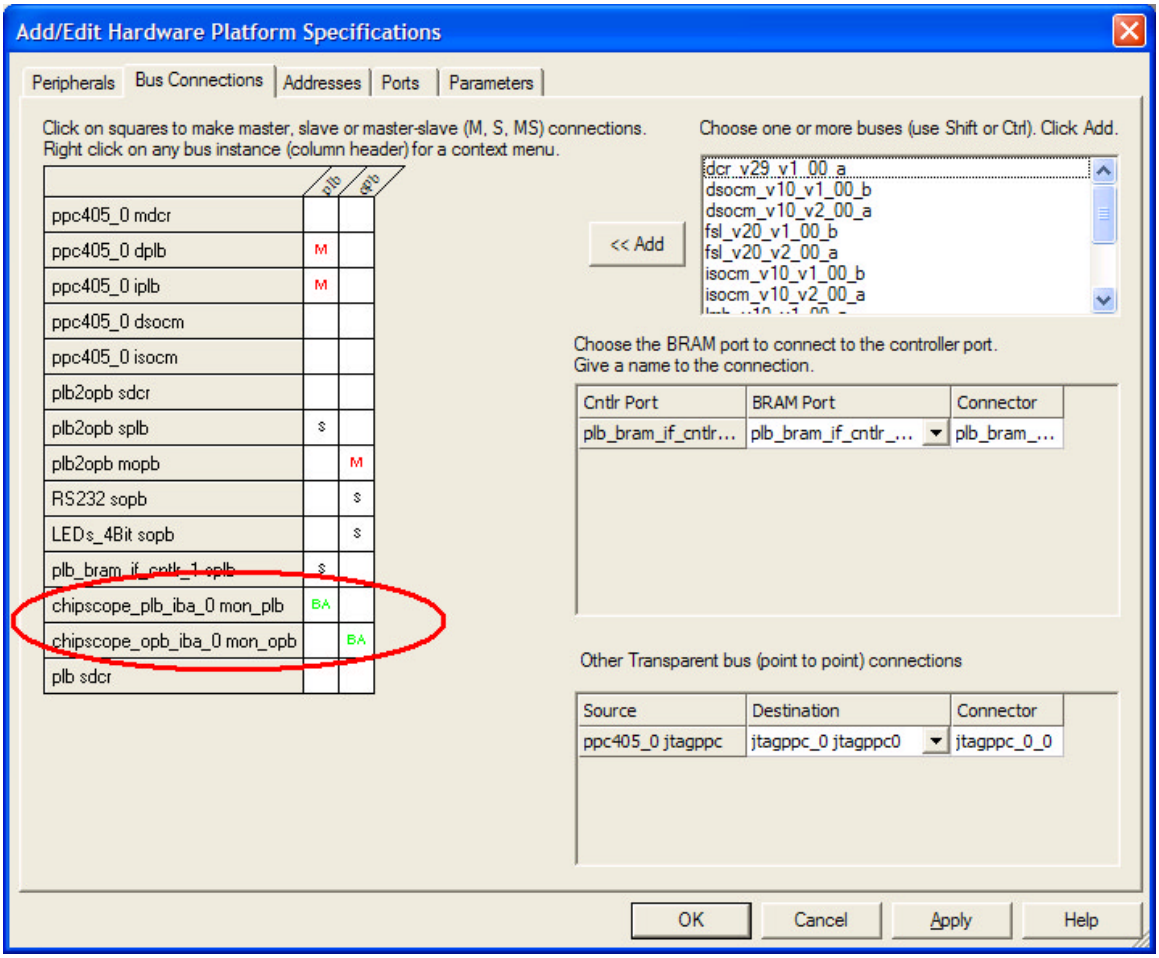

Figure 9-7: **Connecting the OPB/PLB IBA to the OPB/PLB as a Bus Analyzer**

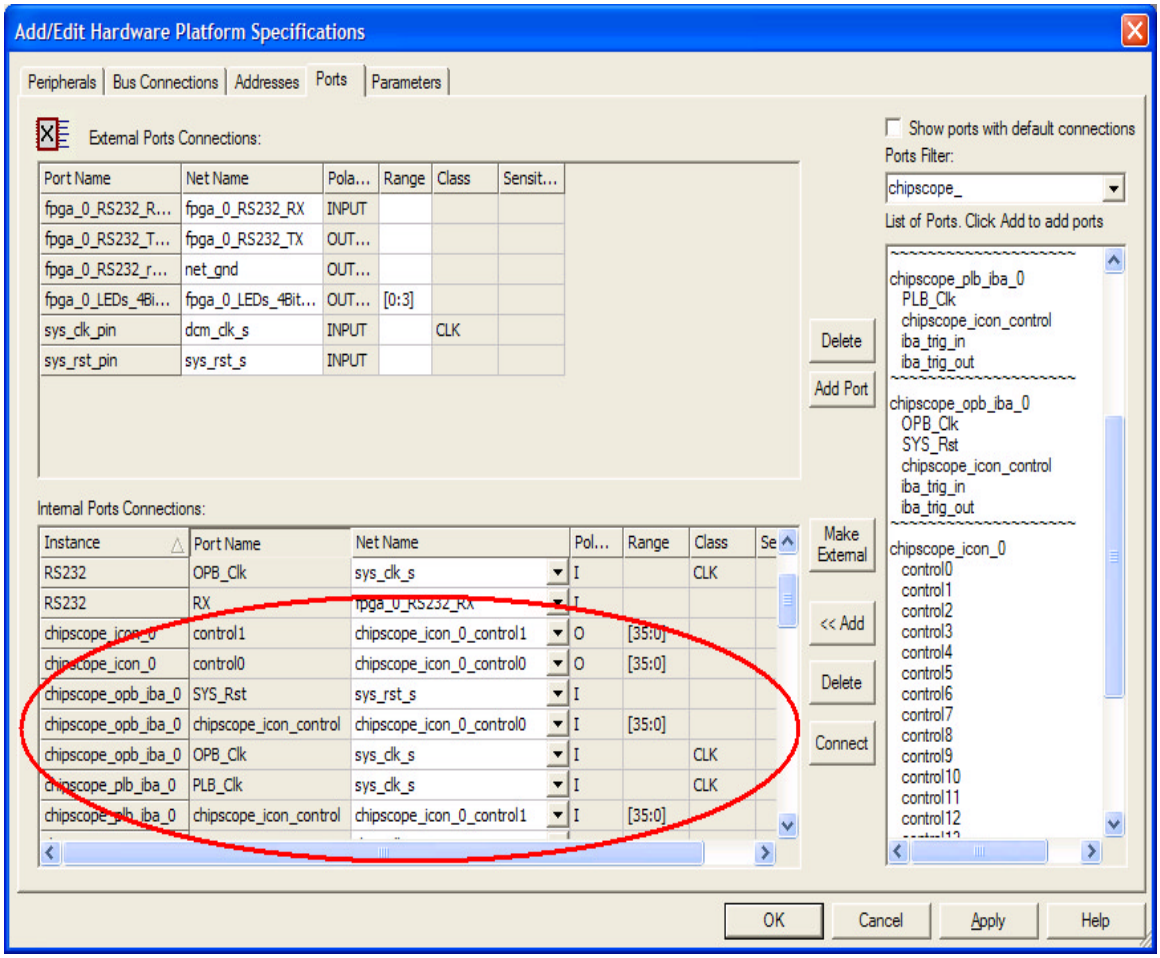

Figure 9-8: **Connecting the ICON "control" Signals to the OPB/PLB IBA**

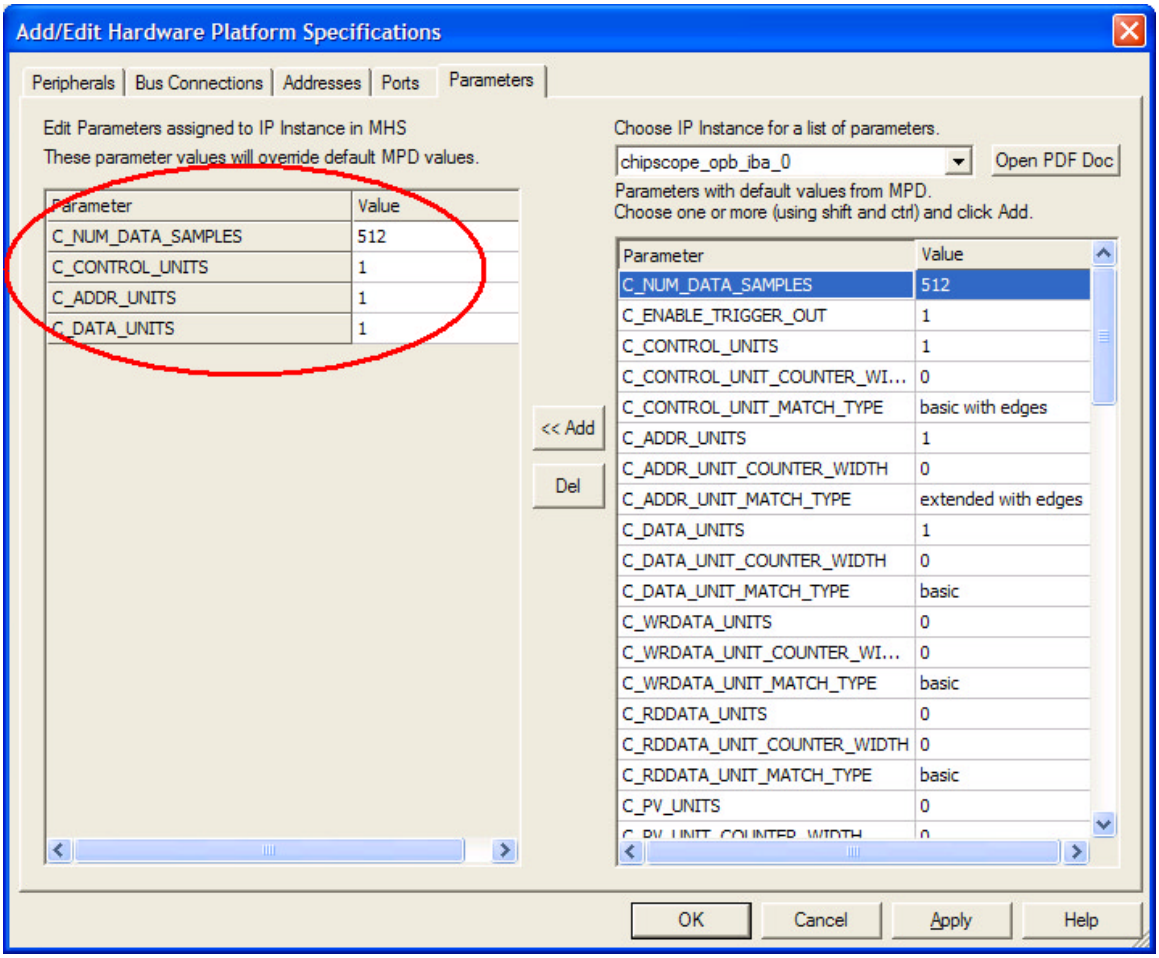

Figure 9-9: **Enabling Individual OPB IBA Monitors**

Steps Involved in Using ChipScope Pro Analyzer with an EDK Design

1. Generate a netlist with the ChipScope cores.

   Once a valid MHS design is created with the ChipScope ICON and IBA setup, you can implement this design from XPS and generate a netlist. For the ChipScope cores, Platgen automatically calls some Tcl scripts in the background that will run the ChipScope Core Generator to create netlists for these cores with the appropriate parameters.

2. Implement the design to create a bitstream.

   When `ngdbuild` is run, the netlists are all merged into a single design and a bitstream file is then generated that contains the embedded ChipScope Logic or Bus Analyzer.

3. Download the bitstream to the board.

   When this bitstream is downloaded to the FPGA, you can use the ChipScope Analyzer to view bus transactions in the design.

4. Open ChipScope Analyzer.

   From the Analyzer, connect to the JTAG cable. The Analyzer automatically connects to the FPGA and detects the IBA present in the design, but the signal names do not reflect that of the OPB IBA.

5. Import the ChipScope OPB IBA signals `.cdc` file into the Analyzer.

   Platgen and the Tcl wrappers for the ChipScope cores create the signals `.cdc` file based on the design parameters in the following directory:

   ```
   <EDK Project Directory>/implementation/<OPB IBA instance
   name>_wrapper/<OPB IBA instance name>.cdc
   ```

   For example, if the instance name of the OPB IBA in your design is `chipscope_opb_iba_0` then `<Project>/implementation/chipscope_opb_iba_0_wrapper/chipscope_opb_iba_0.cdc` is the ChipScope signals `.cdc` file that must be imported into the Analyzer.

6. Now, you can use the ChipScope Analyzer to monitor the bus transactions.

   For more information on using the ChipScope Analyzer, refer to the *ChipScope Pro Software and Cores User Manual*.

Using ChipScope ILA Core

You can use ChipScope ILA to monitor individual signals in a processor design. When the ChipScope ILA core is used in Platform Studio, only signals at the top level of the processor design (at the MHS level) can be monitored using the ILA. By using the ILA connection feature in the FPGA Editor, you can monitory signals at level of hierarchy using this ILA.

Using ChipScope Virtual IO (VIO) Core

The ChipScope VIO core provides peek-and-poke access into FPGA internal or external signals at the MHS level. Unlike the other ChipScope cores, this core is not used like a logic analyzer to collect a signal trace, but to directly read and write logic signals at runtime from ChipScope Analyzer. The ChipScope VIO core supports asynchronous or synchronous input or output probes into the hardware. They can also be visualized as LEDs, numerical display for virtual outputs and pushbuttons, DIP switches, or pulse trains for virtual inputs. Refer to the *ChipScope Pro Software and Cores User Manual* for more information.

Advanced ChipScope Debugging Tips

- If more than one ChipScope core (excluding ICON) is used in a design, then set the `C_NUM_CONTROL_PORTS` parameter appropriately in the ICON and make one control port connection for each ChipScope core.
- If the opb_mdm core is used in a design along with the ChipScope cores, then set the `C_SYSTEM_CONTAINS_MDM` to **`1`**. Otherwise, set it to **`0`**, the default value.
- ChipScope VIO does not have any bus interfaces. The synchronous or asynchronous input or output ports should be connected directly to other ports in the MHS.
- Generic Trigger Units in the OPB and PLB IBA with variable width can be used to monitor individual non-bus signals in the design. This is equivalent to using the ChipScope ILA core.

- For the ChipScope OPB and PLB IBA cores, for each match unit, the `C_<UNIT>_MATCH_TYPE` parameter can be used to have different kinds of trigger matching. The possible string values for this `C_<UNIT>_MATCH_TYPE` parameter are `basic`, `basic with edges`, `extended`, `extended with edges`, `range`, and `range with edges`.
- For detailed explanations of various ChipScope core parameters, refer to the *ChipScope Pro Software and Cores User Manual* in the ChipScope Installation.

XILINX®

Chapter 10

Profiling Embedded Designs

This chapter describes profiling standalone PowerPC™ and MicroBlaze™ programs on Hardware, Simulator, and virtual platform (system simulator) targets. Function Call Graph and Histogram information of the program can be generated using different targets. Profiling can be performed from Xilinx® Platform Studio™ IDE (XPS) or Platform Studio SDK IDE. Profiling in XPS is command-line-based using Xilinx® MicroProcessor Debugger (XMD) and GNU gprof tools. Profiling in SDK is interactive and provide graphical profile views. This chapter describes profiling in XPS and SDK.

Topics covered in this chapter include:

Assumptions

This chapter assumes that you:

- Are familiar with EDK and have built systems with XPS.
- Are familiar with debugging applications using XMD.
- Are familiar with C programming and have used GNU tools.

Tool Requirements

Program profiling requires the following tools in EDK:

- Xilinx Platform Studio (XPS).
- EDK GNU tools.
- Library generator (Libgen).
- Xilinx Microprocessor Debugger (XMD).
- Xilinx® Platform Studio SDK.
- The opb_timer peripheral for MicroBlaze (For Hardware targets).
- virtual platform (system simulator).

Features

Program profiling has the following features:

- Histogram (flat profile) and Call Graph information.
  - *Histogram:* Shows how much time the program spent in each function and how many times the function was called.
  - *Call Graph*: Shows for each function, which functions called it, which other functions it called, and how many times.
- Profiling parameters are `configurable`, `Sampling Frequency`, `Histogram Binsize`, and `Timer to use`.
- Graphical profile views in Platform Studio SDK.

Note: Profiling is not supported on the PowerPC Simulator target.

Note: EDK6.3i LibXil Profile Library has been deprecated. Xilinx recommends that you use the new profiling method that is described in "Profiling the Program on Hardware Target" on page 160.

Profiling the Program on Simulator/virtual platform

MicroBlaze simulator (in-built XMD) and virtual platform support *software non-intrusive profiling*; that is, no profiling software code is instrumented in your program. This provides accurate profiling information.

This section illustrates the steps to profile a MicroBlaze program. The program to be profiled is a dhrystone application.

Using Xilinx Platform Studio IDE

Steps involved in Profiling using XPS are:

1. Building Your Application
2. Collecting and Generating the Profile Data
3. Viewing the Profile Output

Building Your Application

Your application is compiled and built using Software Settings in XPS. Refer to Chapter 3, "Writing Applications for a Platform Studio Design" for more details.

Collecting and Generating the Profile Data

XMD helps in the generation of output files that can be read by GNU gprof tools. You can do the following actions in XMD for profiling:

- Open XMD Tcl shell by selecting **Tools** → **XMD**. Connect to the MicroBlaze simulator or virtual platform target.
- Download the executable file.
- Run the Program.
- After the Program has continued execution for sufficient time and when profiling information is needed, stop the Program or set a breakpoint at the exit location.
- To generate Profile output file, type **`profile`** in XMD. This generates a profile output file, `gmon.out`.

Sample profiling session using XMD:

```
XMD%
XMD% connect mb sim

Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD% dow executable.elf
        section, .text: 0x00000000-0x00000cd0
        section, .rodata: 0x00000cd0-0x00000d4f
        section, .sdata2: 0x00000d50-0x00000d64
        section, .data: 0x00000d68-0x00000d94
        section, .sdata: 0x00000d98-0x00000d9c
        section, .bss: 0x00000da0-0x00003998
Downloaded Program executable.elf
Setting PC with program start addr = 0x00000000
XMD% bps exit
Setting breakpoint at 0x00000090
XMD% con
XMD% profile
        Profile data written to gmon.out
XMD%
```

Viewing the Profile Output

Use GNU gprof tool (mb-gprof) to read the output file. Refer to the UNIX Manual page **gprof** for more information.

```
mb-gprof -b microblaze_0/code/executable.elf gmon.out
```

```
Flat profile:

Each sample counts as 1e-08 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 19.14      0.00     0.00      100  1350.10  1350.10  divsi3_proc
 14.95      0.00     0.00        1 105470.00 602150.00  main
 14.52      0.00     0.00        1 102435.00 102455.00  _crtinit
 13.33      0.00     0.00      100   940.00   940.00  strcmp
 10.35      0.00     0.00      100   730.00  1283.33  Proc_1
  6.95      0.00     0.00      100   490.00  1486.67  Func_2
  5.53      0.00     0.00      100   390.00   390.00  Proc_8
  3.69      0.00     0.00      100   260.00   320.00  Proc_6
  2.41      0.00     0.00      300    56.67    56.67  Func_1
  1.98      0.00     0.00      300    46.67    46.67  Proc_7
  1.98      0.00     0.00      100   140.00   186.67  Proc_3
  1.84      0.00     0.00      100   130.00   130.00  Proc_4
  1.56      0.00     0.00      100   110.00   110.00  Proc_2
  0.85      0.00     0.00      100    60.00    60.00  Func_3
  0.71      0.00     0.00      100    50.00    50.00  Proc_5
  0.09      0.00     0.00        2   335.00   335.00  malloc
  0.07      0.00     0.00                             XNullHandler
  0.04      0.00     0.00                             _start1
  0.00      0.00     0.00        1    20.00    20.00  _program_clean
  0.00      0.00     0.00        1    20.00    20.00  _program_init
  0.00      0.00     0.00                             _start
  0.00      0.00     0.00                             exit


                        Call graph
```

```
granularity: each sample hit covers 8 byte(s) for 0.00% of 0.00 seconds

index % time    self  children    called     name
                                                <spontaneous>
[1]     99.9    0.00    0.00                 _start1 [1]
                0.00    0.00       1/1           main [2]
0.00    0.00       1/1           _crtinit [6]
                0.00    0.00       1/1           _program_clean [19]
-----------------------------------------------
                0.00    0.00       1/1           _start1 [1]
[2]     85.4    0.00    0.00       1         main [2]
                0.00    0.00     100/100         Func_2 [3]
                0.00    0.00     100/100         divsi3_proc [4]
                0.00    0.00     100/100         Proc_1 [5]
                0.00    0.00     100/100         Proc_8 [8]
                0.00    0.00     100/100         Proc_4 [13]
                0.00    0.00     200/300         Func_1 [11]
                0.00    0.00     100/100         Proc_2 [14]
                0.00    0.00     100/100         Proc_5 [16]
                0.00    0.00     100/300         Proc_7 [12]
                0.00    0.00       2/2           malloc [17]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[3]     21.1    0.00    0.00     100         Func_2 [3]
                0.00    0.00     100/100         strcmp [7]
                0.00    0.00     100/300         Func_1 [11]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[4]     19.1    0.00    0.00     100         divsi3_proc [4]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[5]     18.2    0.00    0.00     100         Proc_1 [5]
                0.00    0.00     100/100         Proc_6 [9]
                0.00    0.00     100/100         Proc_3 [10]
                0.00    0.00     100/300         Proc_7 [12]
-----------------------------------------------
                0.00    0.00       1/1           _start1 [1]
[6]     14.5    0.00    0.00       1         _crtinit [6]
                0.00    0.00       1/1           _program_init [20]
-----------------------------------------------
                0.00    0.00     100/100         Func_2 [3]
[7]     13.3    0.00    0.00     100         strcmp [7]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[8]      5.5    0.00    0.00     100         Proc_8 [8]
-----------------------------------------------
                0.00    0.00     100/100         Proc_1 [5]
[9]      4.5    0.00    0.00     100         Proc_6 [9]
                0.00    0.00     100/100         Func_3 [15]
-----------------------------------------------
                0.00    0.00     100/100         Proc_1 [5]
[10]     2.6    0.00    0.00     100         Proc_3 [10]
                0.00    0.00     100/300         Proc_7 [12]
-----------------------------------------------
                0.00    0.00     100/300         Func_2 [3]
                0.00    0.00     200/300         main [2]
[11]     2.4    0.00    0.00     300         Func_1 [11]
```

```
-----------------------------------------------
                0.00    0.00     100/300         main [2]
                0.00    0.00     100/300         Proc_1 [5]
                0.00    0.00     100/300         Proc_3 [10]
[12]     2.0    0.00    0.00     300         Proc_7 [12]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[13]     1.8    0.00    0.00     100         Proc_4 [13]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[14]     1.6    0.00    0.00     100         Proc_2 [14]
-----------------------------------------------
                0.00    0.00     100/100         Proc_6 [9]
[15]     0.9    0.00    0.00     100         Func_3 [15]
-----------------------------------------------
                0.00    0.00     100/100         main [2]
[16]     0.7    0.00    0.00     100         Proc_5 [16]
-----------------------------------------------
                0.00    0.00       2/2           main [2]
[17]     0.1    0.00    0.00       2         malloc [17]
-----------------------------------------------
                                                <spontaneous>
[18]     0.1    0.00    0.00                 XNullHandler [18]
-----------------------------------------------
                0.00    0.00       1/1           _start1 [1]
[19]     0.0    0.00    0.00       1         _program_clean [19]
-----------------------------------------------
                0.00    0.00       1/1           _crtinit [6]
[20]     0.0    0.00    0.00       1         _program_init [20]
-----------------------------------------------
                                                <spontaneous>
[21]     0.0    0.00    0.00                 _start [21]
-----------------------------------------------
                                                <spontaneous>
[22]     0.0    0.00    0.00                 exit [22]
-----------------------------------------------


   Index by function name

[11] Func_1             [9] Proc_6              [1] _start1
[3] Func_2              [12] Proc_7             [4] divsi3_proc
[15] Func_3             [8] Proc_8              [22] exit
[5] Proc_1              [18] XNullHandler       [2] main
[14] Proc_2             [6] _crtinit            [17] malloc
[10] Proc_3             [19] _program_clean     [7] strcmp
[13] Proc_4             [20] _program_init
[16] Proc_5             [21] _start
```

Using Platform Studio SDK IDE

The steps involved in Profiling using SDK are:

1. Create / build the application in Debug build configuration.
2. Run the application.
   a. Create a Run Configuration.
   b. Specify the MicroBlaze simulator or virtual platform target in the Target Connection tab.
   c. Enable Profiling in the Profiler tab.
   d. Click **Run** to profile the program and collect profile information.
3. Open the Profiling perspective to view the profile outputs.

For more detailed information, refer to the Profiling section of the *Platform Studio SDK Online Help*.

Profiling the Program on Hardware Target

A program running on a hardware target is profiled using the *software intrusive* method. In this method, profiling software code is instrumented in your program. On program execution, this profiling code stores information on the hardware. XMD collects the profile information and generates the output file, which can be read by the GNU *gprof* tool. The program's functionality remains unchanged but it slows down the execution.

Software intrusive profiling requires memory for storing profile information and timer for sampling instruction address. *opb_timer* is the supported profile timer. For PowerPC systems, you can also use *Programmable Interrupt Timer (PIT)* as profile timer.

The following steps illustrate profiling on a MicroBlaze program. The system has BRAM and External Memory. The system has an opb_timer, and the Interrupt signal is directly connected to MicroBlaze External Interrupt signal. The opb_timer is used as the profile timer on MicroBlaze.

The program to be profiled is a dhrystone application. It is executed from BRAM address space.

Note: The profiling steps for PowerPC target is similar to MicroBlaze. Any difference in steps are highlighted.

Using Xilinx Platform Studio IDE

Steps involved in Profiling using XPS are:

1. Enabling the Profiling Functions.
2. Building the User Application
3. Collecting and Generating the Profile Data
4. Viewing the Profile Output

Enabling the Profiling Functions

Open the EDK design in XPS.

1. Open the Software Platform Settings window by selecting **Projects** → **Software Platform Settings**.
2. Click the Library / OS Parameters tab. Under MicroBlaze Standalone configuration, enable enable_sw_intrusive_profiling by selecting **true**. Select **opb_timer** as the profile_timer to use for profiling.
3. Click **OK**.

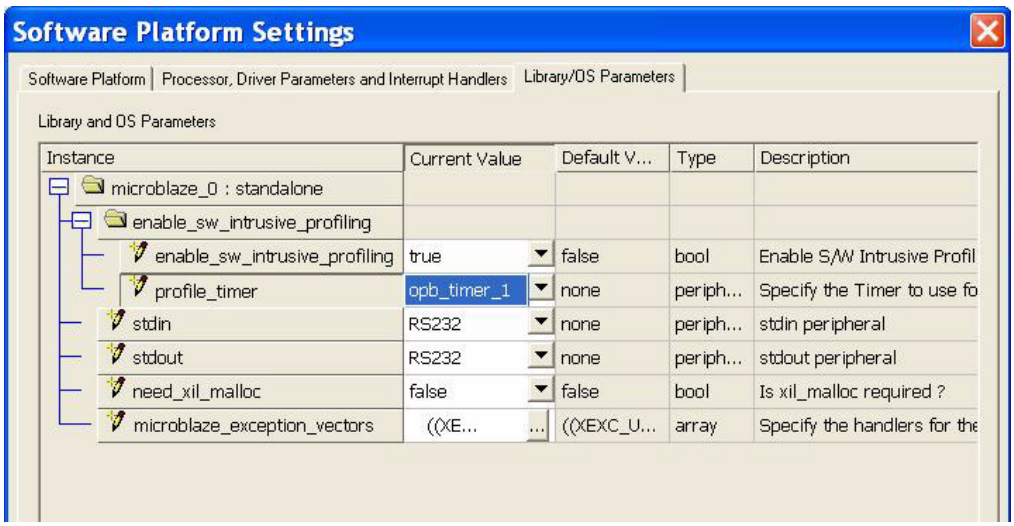

Figure 10-1: **Profile Configuration Parameters**

4. Generate libraries by selecting **Tools** → **Generate Libraries and BSPs**. This generates the profile functions in the `libxil.a` library.

Note: For PowerPC targets, you can use in-built Programmable Interrupt Timer (PIT) as a profile timer. Specify **none** for profile_timer to use PIT for profiling.

Building the User Application

1. Open / Create the dhrystone application. Refer to Chapter 3, "Writing Applications for a Platform Studio Design" for more details.
2. Right-click the project name in the tree view and select **Set Compiler Options**. In the Advanced tab, specify **-pg** as Program Sources Compiler Options.

   Refer to the "Timer / Interrupts Initialization in Your Application" section below for any changes in your application.
3. Build the application.

Timer/Interrupts Initialization in Your Application

Initialize the profile timer by setting the timer interval, registering the timer handler, and enabling interrupts. Enable system interrupts for proper profiling. The instrumented profiling functions perform all the initialization required.

In the following system configurations, no changes are required in your program for initialization:

- The system has an opb_timer. This opb_timer is used for profiling and its Interrupt signal is directly connected to the Processor Interrupt signal.

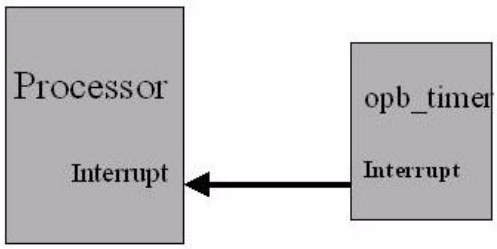

Figure 10-2: **An opb_timer Connected to A Processor Interrupt Port**

- The system has an interrupt controller with an opb_timer as a single interrupt source. The interrupt controller is connected to Processor Interrupt signal and the opb_timer is used for profiling.

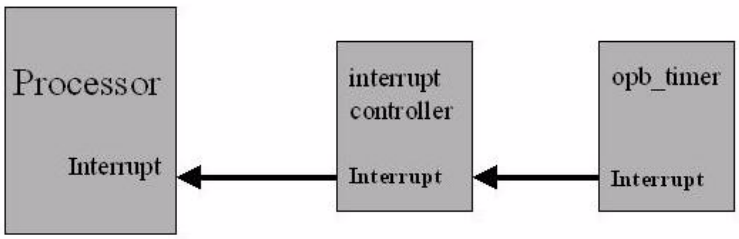

Figure 10-3: **An opb_timer Connected to A Processor Using An Interrupt Controller**

Note: For PowerPC, a Programmable Interrupt Timer is used for profiling.

For system configurations which have multiple interrupt sources, the profiling functions perform the initialization required for profile timers. Perform the following initialization in your application:

1. Perform interrupt controller initialization.
2. Enable the profile timer interrupt in the interrupt controller.
3. Enable processor interrupts and exceptions.

The following code snippet is for a MicroBlaze-based system:

```
XIntc_mMasterEnable( SYSINTC_BASEADDR );
XIntc_SetIntrSvcOption( SYSINTC_BASEADDR, XIN_SVC_ALL_ISRS_OPTION);
XIntc_mEnableIntr( SYSINTC_BASEADDR, PROFILE_TIMER_INTR_MASK );
microblaze_enable_interrupts();
```

For more information on Profiling functions, refer to the "Standalone Board Support Package" chapter of the *OS and Libraries Reference Manual*.

Collecting and Generating the Profile Data

XMD helps in the generation of output files that can be read by GNU gprof tools. You can do the following functions in XMD for profiling.

- Open XMD Tcl shell by selecting **Tools** → **XMD**. Connect to the MicroBlaze simulator or virtual platform target.
- Specify the profiling sampling frequency, histogram bin size, and memory address for collecting profile data. Use the **`profile -config`** command.
- Download the executable file.
- Run the Program
- After the Program has continued execution for sufficient time and when profiling information is needed, stop the Program or set a breakpoint at the exit location.

Sample profiling session using XMD:

```
XMD% connect mb mdm
Connecting to cable (Parallel Port - parport0).
 WinDriver v6.03 Jungo (c) 1997 - 2003 Build Date: Aug 10 2003 X86
15:27:16.
 parport0: baseAddress=0x378, ecpAddress=0x778
 LPT base address = 0378h.
 ECP base address = 0778h.
 ECP hardware is detected.
Cable connection established.
Connecting to cable (Parallel Port - parport0) in ECP mode.
 LPT base address = 0378h.
 Cable Type = 1, Revision = 0.
 Setting cable speed to 5 MHz.
Cable connection established.
JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05046093           8        XCF04S
 2       05046093           8        XCF04S
 3       0124a093          10        XC2VP7
Assuming, Device No: 3 contains the MicroBlaze system
Connected to the JTAG MicroProcessor Debug Module (MDM)
No of processors = 1
MicroBlaze Processor 1 Configuration :
-------------------------------------
Version.............................4.00.a
No of PC Breakpoints................2
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support..........off
Data Cache Support.................off
Exceptions  Support................off
```

```
FPU  Support.....................off
FSL DCache Support...............off
FSL ICache Support...............off
Hard Divider Support.............off
Hard Multiplier Support..........on
Barrel Shifter Support...........off
MSR clr/set Instruction Support....off
Compare Instruction Support........off
JTAG MDM Connected to MicroBlaze 1
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD% profile -config sampling_freq_hz 10000 binsize 4 profile_mem
0x22070000
XMD% dow executable.elf
        section, .text: 0x22000000-0x22001cac
        section, .rodata: 0x22001cac-0x22001d34
        section, .data: 0x22001d38-0x22001db8
        section, .sdata: 0x22001db8-0x22001dbc
        section, .bss: 0x22001dc0-0x220055e0
Downloaded Program dhrystone/executable.elf
Setting PC with program start addr = 0x22000000
Program dhrystone/executable.elf being Profiled on Hardware
Initialized Profile Configurations for the Program :
-------------------------------------------------
Sampling Frequency..............10000 Hz
Histogram Bin Size..............4 Words
Memory for Profiling used from...0x22070000
Memory Used for Profiling Data...2140 Bytes
XMD% bps exit
Setting breakpoint at 0x220000f4
XMD% con
XMD% profile
        Profile data written to gmon.out
XMD%
```

Viewing the Profile Output

Use GNU gprof tool (mb-gprof or powerpc-eabi-gprof) to read the output file. Refer to the UNIX Manual page **gprof** for more information.

Using Platform Studio SDK IDE

Steps involved in Profiling using SDK are:

1. Enable the Profiling function in XPS as described in "Enabling the Profiling Functions" on page 161.
2. Create / build the application. Build the application in Profile build configuration, this compiles the application using the **-pg** option.
3. Refer to "Timer / Interrupts Initialization in Your Application" on page 162 for any changes in your application.

4. Run the application.
   a. Create a Run Configuration.
   b. Specify the Hardware target in the Target Connection tab.
   c. Enable Profiling in the Profiler tab. Specify the sampling frequency, histogram bin size, and address for storing profile information.
   d. Click **Run** to run the program and collect profile information.
5. Open the Profiling Perspective to view the profile outputs.

For more detailed information, refer to the Profiling section of the *Platform Studio SDK Online Help*.

Chapter 11

System Initialization and Download

This chapter describes the basics of system initialization and download using the Xilinx® Platform Studio™ (XPS) tools. It contains the following sections.

Assumptions

This chapter assumes that you have already done the following:

- Created an EDK project
- Created a hardware system and a software application in your EDK project
- Used the Xilinx® ISE™ tools to create a bitstream for the hardware, and the EDK compiler tools to create an Executable and Linking Format (ELF) file for the application

Introduction

System initialization and download consists of updating the hardware bitstream with BRAM initialization data, downloading this modified bitstream to the FPGA, and initializing external memories if necessary. There are several methods available to do this. Select the appropriate method for your project considering the application and the stage of the project.

If the entire software application is contained within Field Programmable Gate Array (FPGA) BRAM blocks and no external memories must be initialized, you can initialize the system by updating the bitstream. Refer to "Initialize Bitstreams with Applications" on page 168 for more details.

If a part of the software application resides in external memory, you can use the Xilinx® Microprocessor Debug (XMD) to download the software application after the FPGA is programmed with the bitstream. Refer to "Downloading an Application Using XMD" on page 169 for more details. You must use a bootloop to ensure that the processor does not enter a bad state between download of the bitstream and download of the application. Refer to "Initialize Bitstreams Using Bootloops" on page 168 for more information on bootloops.

You can also initialize the system using System ACE™. System ACE CF configures devices using Boundary-Scan (JTAG) instructions and a Boundary-Scan Chain. System ACE CF is a two-chip solution that requires the System ACE CF controller and either a CompactFlash card or one-inch Microdrive disk drive technology as the storage medium. Refer to "Generating ACE Files" on page 179 for more details. You can also use a bootloader as the mechanism to load the application from some nonvolatile memory into processor memory and execute it. Refer to "Bootloaders" on page 169 for more details.

Bitstream Initialization

Initialize Bitstreams with Applications

If your entire software application fits on FPGA BRAM blocks, you can initialize the system by updating the hardware bitstream with the BRAM initialization data. You can then download this updated bitstream to the FPGA.

After building the applications, XPS allows you to select the application that must be initialized in the generated bitstream. To initialize an executable in the bitstream, do the following:

1. Right-click on the project name in the tree view and select **Mark to Initialize BRAMs**.
2. From the XPS main window, select **Tools** → **Update Bitstream** to initialize the BRAMs with the selected executable information.

   Once the bitstream is initialized with the executable, the bitstream downloads to the board.
3. To download the application, set up the board and the parallel cable as required.
4. To download and execute the selected application, select **Tools** → **Download** in the XPS main window. This downloads the bitstream onto the board, brings the processor out of reset, and starts execution.

Initialize Bitstreams Using Bootloops

Once the FPGA is configured with a bitstream, the processor comes out of reset and begins executing. If the system is not yet initialized with the software application, the processor might execute code that puts it into a state that it cannot be brought out of with a soft reset. The processor must therefore be kept in a known good state until the system is completely initialized.

A bootloop is a software application that keeps the processor in a defined state until the actual application can be downloaded and run. It consists of a simple branch instruction, and is located at the processor's boot location. XPS contains a predefined bootloop application.

To use a bootloop, create the software application as usual. The linker script used is no different from the case in which no bootloop is used; that is, the software application should contain instructions at the processor's boot location.

The software application should not be used to initialize the system BRAMs. Right-click the project name in the tree view, and ensure that **Mark to Initialize BRAMs** is not selected. If it is, click to deselect it. To initialize BRAMs with the bootloop, right-click on the default bootloop project (*processor name*_bootloop) in the tree view and select **Mark to Initialize BRAMs**.

Update the bitstream with the bootloop by selecting **Tools** → **Update Bitstream** in the XPS main window.

You can then download the bitstream to the FPGA. You can then download the software application using either XMD or System ACE.

Software Program Loading

Downloading an Application Using XMD

Using XMD is one way to initialize application memory that lies external to the FPGA. To download an application using XMD, do the following:

1. Initialize the bitstream with a bootloop, and configure the FPGA with this information.

   Note: For MicroBlaze™ systems, the processor must be connected to an MDM module.
2. Select **Tools** → **XMD** from the main XPS window to start XMD.
3. Connect to the processor.
   - For a PowerPC™ system, use `connect ppc hw`
   - For a MicroBlaze system, use `connect mb mdm`.
4. Download the software application using `dow <path to executable file>`.

Refer to the XMD documentation for more XMD commands and options.

Bootloaders

Bootloaders are small software programs that reside in internal BRAM memory. They run when the processor comes out of reset. A bootloader sets up registers and copies the main application program from external nonvolatile memory, such as flash memory, into internal BRAM memory. Once the application is copied, the bootloader runs it by branching to the entry point of the application. All interrupts are turned off when the bootloader is executing. The application is responsible for all interrupt handling.

System ACE

You can use the System ACE solution to download software through Joint Test Action Group (JTAG) in a similar way to the debugger. This way a single System ACE controller can be augmented to contain both the bitstream and software initialization data. Refer to "Generating a System ACE File" on page 175 for more information.

Fast Download on a MicroBlaze System

This section describes the steps to build a MicroBlaze system for downloading data at high speeds to internal and external memory from XMD during debugging. The system uses a unidirectional Fast Simplex Link (FSL) from the MicroBlaze Debug Module (`opb_mdm`) to the MicroBlaze processor. Faster download speeds are necessary when debugging a large program, such as a uClinux-based application, or when downloading large data files to external memory. The download mechanism described below provides speeds up to 500 Kb per second.

The section contains the following topics:

Assumptions

This chapter assumes that you:

- Have created an EDK project, including hardware and software, using XPS.
- Are familiar with debugging a program using XMD / GDB.

Tool Requirements

Fast download on MicroBlaze is supported on IP and tools of the following or later versions:

- `microblaze 2.10.a`
- `opb_mdm 2.00.a`
- `xmd EDK 7.1`
- `mb-gdb EDK 7.1`

Step 1: Building the Hardware

The MicroBlaze Debug Module `opb_mdm` has Slave (Input) and Master (Output) FSL Interfaces. For downloading purposes, we use only the Master FSL Interface. This Interface is connected to the Slave FSL Interface (SFSL0) of MicroBlaze through an FSL Bus Interface.

To build the hardware, you must instantiate the FSL Bus Interface. You must also connect the MFSL0 Interface of `opb_mdm` and SFSL0 Interface of MicroBlaze. To perform these tasks, select **Project** → **Add/Edit Cores** in XPS. For more information, refer Chapter 2, "Creating a Basic Hardware System in XPS".

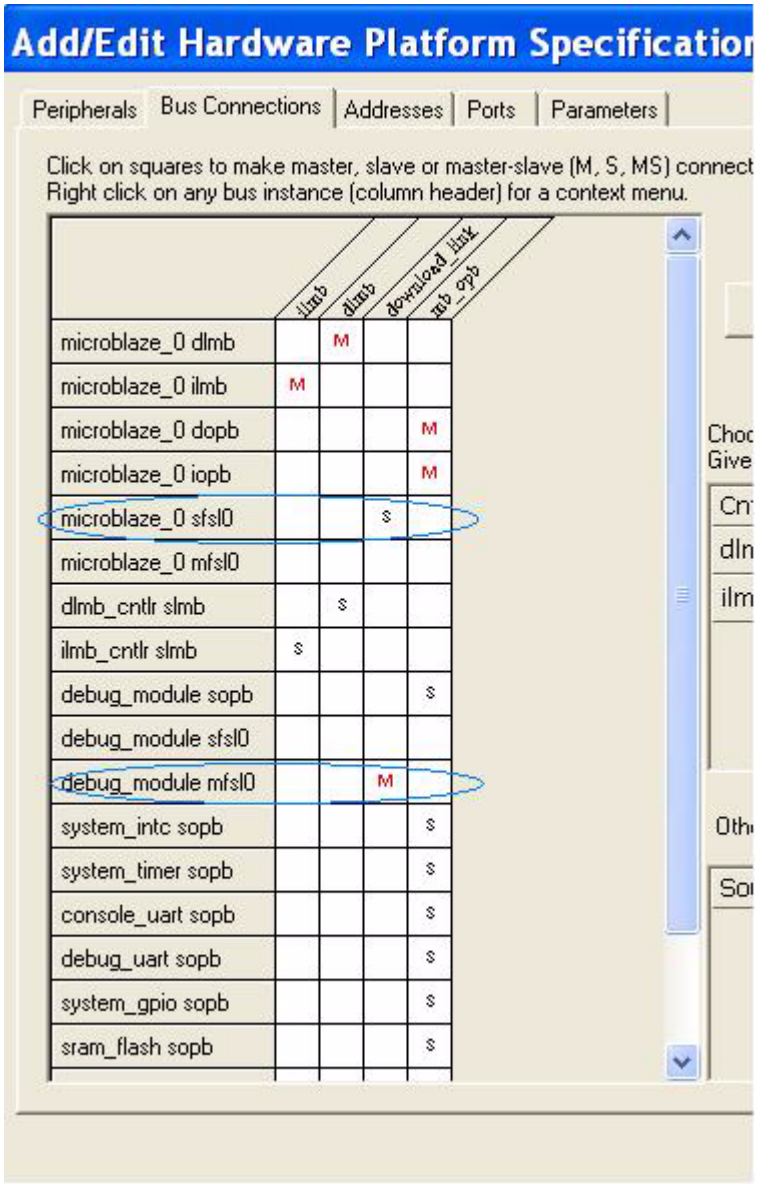

Figure 11-1: **Add/Edit Core Bus Connections**

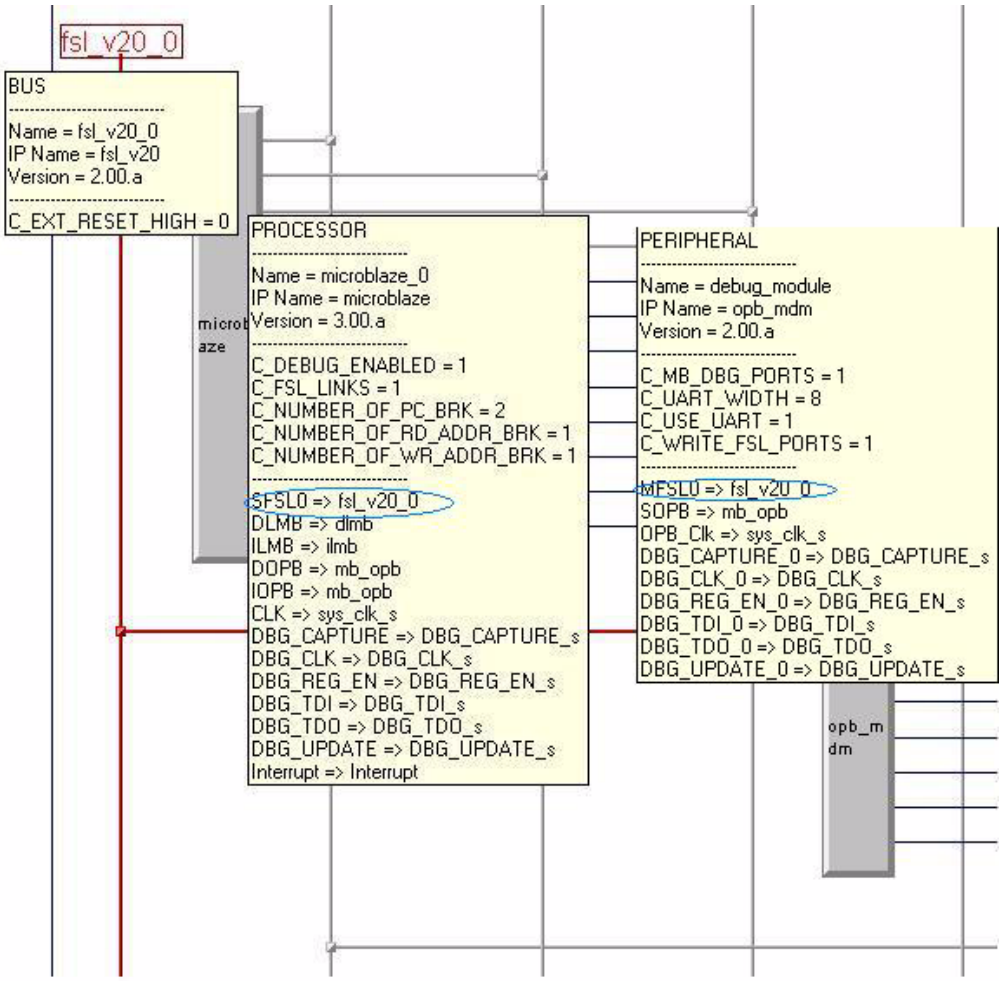

Figure 11-2: **MicroBlaze-FSL-MDM Connection in the PBD Editor**

The following is the MHS code snippet of the connection:

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_i
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_DIV = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 4
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 PARAMETER C_FSL_LINKS = 1
 BUS_INTERFACE SFSL0 = download_link
 BUS_INTERFACE DLMB = d_lmb_v10
 BUS_INTERFACE ILMB = i_lmb_v10
 BUS_INTERFACE DOPB = d_opb_v20
 BUS_INTERFACE IOPB = d_opb_v20
 PORT CLK = sys_clk
 PORT INTERRUPT = interrupt
END
```

```
BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0xFFFFC000
 PARAMETER C_HIGHADDR = 0xFFFFC0FF
 PARAMETER C_WRITE_FSL_PORTS = 1
 BUS_INTERFACE MFSL0 = download_link
 BUS_INTERFACE SOPB = d_opb_v20
 PORT OPB_Clk = sys_clk
END

BEGIN fsl_v20
 PARAMETER INSTANCE = download_link
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst
 PORT FSL_Clk = sys_clk
END
```

Step 2: Downloading Program/Data to Memory

XMD determines if opb_mdm has an MFSL0 Interface and if the MicroBlaze SFSL0 Interface is connected to opb_mdm while connecting to a MicroBlaze target. If both conditions are satisfied, XMD uses the fast download methodology to increase download speeds.

The MFSL0 Interface on opb_mdm is determined by reading the config word of the MDM module on hardware, so XMD should be connected to the board. The SFSL0 Interface connection is determined by loading the MHS file of your EDK design. If MHS file is not loaded, then XMD will not be able to determine the connection and therefore will not use fast download methodology.

The following is an XMD debug session example:

```
bash-2.05$ xmd -xmp system.xmp
Xilinx® Microprocessor Debug (XMD) Engine
Xilinx® EDK 6.3 Build EDK_Gmm.8
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

XMD%
Loading XMP File..
Loading MHS File..
Processor(s) in System ::

MicroBlaze(1) : microblaze_0
Address Map for Processor microblaze_0
  (0x00000000-0x00003fff) dlmb_cntlr    dlmb
  (0x00000000-0x00003fff) ilmb_cntlr    ilmb
  (0x80000000-0x81ffffff) sdram_controller      mb_opb
  (0xc0000000-0xc0003fff) Ethernet_MAC  mb_opb
  (0xff000000-0xff7fffff) sram_flash    mb_opb
  (0xffe00000-0xfffffff) sram_flash     mb_opb
  (0xffff0000-0xffff01ff) sram_flash    mb_opb
  (0xffff1000-0xffff10ff) system_timer  mb_opb
  (0xffff2000-0xffff20ff) console_uart  mb_opb
  (0xffff3000-0xffff30ff) system_intc   mb_opb
```

```
  (0xffff4000-0xffff40ff) debug_uart    mb_opb
  (0xffff5000-0xffff50ff) system_gpio   mb_opb
  (0xffffc000-0xffffc0ff) debug_module  mb_opb

Loading MSS File..
XMD% connect mb mdm
Connecting to cable (Parallel Port - parport0).
 WinDriver v6.03 Jungo (c) 1997 - 2003 Build Date: Aug 10 2003 X86
15:27:16.
 parport0: baseAddress=0x378, ecpAddress=0x778
 LPT base address = 0378h.
 ECP base address = 0778h.
 ECP hardware is detected.
Cable connection established.
Connecting to cable (Parallel Port - parport0) in ECP mode.
 LPT base address = 0378h.
 Cable Type = 1, Revision = 0.
Cable connection established.
openCable xilinx_parallel, returned CSSUCCESS

JTAG chain configuration
-----------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05046093           8        XCF04S
 2       05046093           8        XCF04S
 3       0124a093          10        XC2VP7
Assuming, Device No: 3 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
-------------------------------------
Version.............................2.10.a
No of PC Breakpoints................4
No of Read Addr/Data Watchpoints...2
No of Write Addr/Data Watchpoints..2
Instruction Cache Support..........on
Instruction Cache Base Address.....0x80000000
Instruction Cache High Address.....0x81ffffff
Data Cache Support.................on
Data Cache Base Address............0x80000000
Data Cache High Address............0x81ffffff
MBsfsl-MDMmfsl Connected...........Yes
JTAG MDM Connected to MicroBlaze 1
Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD% dow microblaze_i/code/executable.elf
Downloaded Program microblaze_0/code/executable.elf
Setting PC with program start addr = 0x00000000
       section, .text: 0x00000000-0x00001274
        section, .rodata: 0x00001274-0x000015f5
        section, .sdata2: 0x000015f8-0x00001748
        section, .data: 0x00001748-0x0000177c
        section, .bss: 0x00001780-0x00001ba0
Downloaded Program microblaze_0/code/executable.elf
Setting PC with program start addr = 0x00000000

XMD%
```

Note: When XMD is used for generation of a System ACE file, it does not read the config word from MDM; that is, XMD does not know if the MFSL0 Interface is available on MDM. When the MHS file is not available, XMD cannot verify the FSL connection. In these cases, XMD can be forced to use the fast download methodology by using **-pfsl** option of the **connect** command. For more information, refer to the "Xilinx Microprocessor Debugger (XMD)" chapter in the *Embedded System Tools Guide*.

Note: For the Xilinx Parallel Cable, the default JTAG clock speed is set to 5 MHz. You can change the speed by setting the *XIL_IMPACT_ENV_LPT_SETCLOCK_VALUE* environment variable in the shell. The allowed values are 10000000, 5000000, 2500000, 1250000, and 625000 (Hz).

Generating a System ACE File

This section describes the steps to generate System ACE configuration files from an FPGA bitstream and ELF / data files. The ACE file generated can be used to configure the FPGA, initialize BRAM, initialize external memory with valid program or data, and bootup the processor in a production system. EDK provides a Tool Command Language (Tcl) script, `genace.tcl`, which uses XMD commands to generate ACE files. ACE files can be generated for PowerPC and MicroBlaze with MDM systems.

Assumptions

This chapter assumes that you:

- Are familiar with debugging programs using XMD and with using XMD commands.
- Are familiar with general hardware and software system models in EDK.
- Have a basic understanding of Tcl scripts.

Tool Requirements

Generating an ACE file requires the following tools:

- `genace.tcl`
- `xmd`
- `iMPACT` (from ISE)

GenACE Features

GenACE has the following features:

- Supports PowerPC and MicroBlaze with MDM targets
- Generates ACE files from hardware (Bitstream) and software (ELF / data) files.
- Initializes external memories on PowerPC and MicroBlaze systems.
- Supports Single / Multiple FPGA device systems.

GenACE Model

The System ACE files generated support the System ACE CF family of configuration solutions. System ACE CF configures devices using Boundary-Scan (JTAG) instructions and a Boundary-Scan Chain. System ACE CF is a two-chip solution that requires the System ACE CF controller, and either a CompactFlash card or one-inch Microdrive disk drive technology as the storage medium. The System ACE file is generated from a Serial Vector Format (SVF) file. An SVF file is a text file containing both programming instructions and configuration data to perform JTAG operations.

XMD and iMPACT generate SVF files for software and hardware system files respectively. The set of JTAG instructions and data used to communicate with the JTAG chain on board is an SVF file. It includes the instructions and data to perform operations such as configuring FPGA using iMPACT, connecting to the processor target, downloading the program, and running the program from XMD are captured in an SVF file format. The SVF file is then converted to an ACE file and written to the storage medium. These operations are performed by the System ACE controller to achieve the determined operation.

The following is the sequence of operations using iMPACT and XMD for a simple hardware and software configuration that gets translated into an ACE file.

1. Download the bitstream using iMPACT. The bitstream, `download.bit`, contains system configuration and bootloop code.
2. Bring the device out of reset, causing the Done pin to go high. This starts the Processor system.
3. Connect to the Processor using XMD.
4. Download multiple data files to BRAM or External memory.
5. Download multiple executable files to BRAM or External memory. The PC points to the start location of the last downloaded ELF file.
6. Continue execution from the PC instruction address.

The flow for generating System ACE files is bit → svf, elf → svf, binary data → svf and svf → ace file. The `genace.tcl` script allows the following operations to perform.

The Genace.tcl Script

Syntax

```
xmd -tcl genace.tcl [-opt <genace_options_file>] [-jprog]  [-target
<target_type>] [-hw <bitstream_file>] [-elf <Elf_Files>] [-data
<Data_files>] [-board <board_type>] -ace <ACE_file>
```

Table 11-1: **genace.tcl Script Command Options**

Options	Default	Description
`-opt` <*genace_options_file*>	none	GenACE options are read from the options file.
`-jprog`	false	Clear the existing FPGA configuration. This option should not be specified if performing runtime configuration.

`-target` <*target_type*>	ppc_hw	Target to use in the system for downloading ELF / Data file. Target types are:
		`ppc_hw`
		To connect to a ppc405 system
		`mdm`
		To connect to a MicroBlaze system. This assumes the presence of `opb_mdm` in the system.

`-hw` <*bitstream_file*>	none	The bitstream file for the system. If an SVF file is specified, the SVF file is used.
`-elf` <*list_of_Elf_Files*>	none	List of ELF files to download. If an SVF file is specified, it is used.

Table 11-1: **genace.tcl Script Command Options *(Continued)***

Options	Default	Description
`-data` <*data_file*> <*load_address*>	none	List of Data / Binary file and its load address. The load address can be in decimal or hex format (0x prefix needed). If an SVF file is specified, it is used.

`-board` <*board_type*>	auto	This identifies the JTAG chain on the board (Devices, IR length, Debug device, and so on). The options are given with respect to the System ACE controller. The script contains the options for some pre-defined boards. Board type options are:
		`ml300`
		ml300 board with Virtex2P7 device
		`memec`
		Memec board with Virtex2P4 device and P160
		`mbdemo`
		Xilinx® MicroBlaze Demo Board Virtex21000 device
		`auto`
		Auto Detect Scan Chain and form options for any generic board. The board should be connected for this option. The GenACE options are written out as a genace.opt file. The user can use this file to generate an ACE file for the given system.
		`user`
		The user specifies the `-configdevice` and `-debugdevice` option in the Options file. Refer to the `genace.opt` file for details.

`-configdevice` (only for `-user` board type)	none	Configuration parameters for the device on the JTAG chain: • `devicenr`: Device position on the JTAG chain • `idcode`: ID code • `irlength`: Instruction Register (IR) length • `partname`: Name of the device The device position is relative to the System ACE device and these JTAG devices should be specified in the order in which they are connected in the JTAG chain on the board. This option can be specified only in the options file.
`-debugdevice` (only for `-user` board type)	none	The device containing PowerPC / MicroBlaze to debug or configure in the JTAG chain. Specify the device position on the chain, `devicenr`, and number of processors, `cpunr`. This option can be specified only in the options file.
`-ace` <*ACE_file*>	none	The output ACE file. The file prefix should not match any of input files (bitstream, elf, data files) prefix.

Usage

```
xmd -tcl genace.tcl -jprog -target mdm -hw implementation/download.bit
-elf executable1.elf executable2.svf -data image.bin 0xfe000000 -board
auto -ace system.ace
```

Equivalent `genace.opt` file (for Memec V2P4FF672 board):

```
-jprog
-target mdm
-hw implementation/download.bit
-elf executable1.elf executable2.svf
-data image.bin 0xfe000000
-ace system.ace
-board user
-configdevice devicenr 1 idcode 0x5026093 irlength 8 partname XC18V04
-configdevice devicenr 2 idcode 0x123e093 irlength 10 partname XC2VP4
-debugdevice devicenr 2 cpunr 1
```

Note: The board option in the above options file has been changed from auto to user. The `genace.opt` file should not have a blank line, the parser would error out.

Supported Target Boards

The Tcl script supports the following three boards.

- Memec 2VP4/7 FG456: This board has the following devices in the JTAG chain: XC18V04 → XC18V04 → XC2VP4/7
- ML300: This board has the following device in the JTAG chain: XC2VP7.
- MicroBlaze Demo Board: This board has the following device in the JTAG chain: XC2V1000.

For placing software in OCM memory or for using cache, XMD options in the Tcl script must be changed. These are described in later sections.

GenACE Script Flow and Files Generated

Hardware Bitstream Downloading and FPGA Programming

1. The bitstream file is converted to SVF file format by calling iMPACT.
2. The `bit2svf.scr` command file is passed to iMPACT.
3. The generated SVF file is copied to the final <*ACE_filename*>`.svf` file.

   The bitstream file is assumed to contain a bootloop code or some valid program initialized in BRAM. If an SVF file already exists for the BIT file, the SVF file is used and iMPACT is not called. The script uses the file extension SVF to determine the file type.

Downloading Software ELF/Data Files

1. If software is being downloaded, a delay is introduced to allow the system to come from reset. This is appended to <*ACE_filename*>`.svf` file by routine `write_swprefix`.
2. The data files are converted to an SVF file <*data_filename*>`.svf` by the `xmd_data2svf` routine. This contains instructions and data to connect to the target, download the file, and disconnect from the target.

3. The executable ELF files are converted to an SVF file, <*elf_filename*>.svf, by the xmd_elf2svf routine. This contains instructions and data to connect to the target, download the file, and disconnect from the target.
4. The generated SVF files are appended to the final SVF file.

Running the Software ELF Program

1. A sw_suffix.svf file is generated by the write_swsuffix routine, which contains commands to run the ELF file. This is appended to the final SVF file.
2. The final <*ACE_filename*>.svf file is converted to an ACE file by calling the svf2ace command in iMPACT (iMPACT batch mode command). The GenACE script writes out the svf2ace.scr command file with the appropriate svf2ace command arguments. This svf2ace.scr command file is passed on to iMPACT to generate the final ACE file.

Generating ACE Files

Single FPGA Device

System ACE files can be generated for the following scenarios.

Hardware and Software Configuration

```
xmd -tcl genace.tcl -jprog -target mdm -hw implementation/download.bit
-elf executable1.elf executable2.svf -data image.bin 0xfe000000 -board
mbdemo -ace system.ace
```

Hardware and Software Partial Reconfiguration

```
xmd -tcl genace.tcl -target mdm -hw implementation/download.bit -elf
executable1.elf executable2.svf -data image.bin 0xfe000000 -board
mbdemo -ace system.ace
```

Hardware Only Configuration

```
xmd -tcl genace.tcl -jprog -target mdm -hw implementation/download.bit
-board mbdemo -ace system.ace
```

Hardware Only Partial Reconfiguration

```
xmd -tcl genace.tcl -target mdm -hw implementation/download.bit -board
mbdemo -ace system.ace
```

Software Only Configuration (Downloading and Running)

```
xmd -tcl genace.tcl -target mdm -elf executable1.elf executable2.svf -
data image.bin 0xfe000000 -board mbdemo -ace system.ace
```

Software Only Configuration (Downloading)

This configuration requires changing the genace.tcl script. In the Tcl procedure `genace{},` comment out the following lines, which create and append `sw_suffix.svf`.

```
write_swsuffix $param(target) "sw_suffix.elf" $xmd_options
# Generate code to execute the program downloaded
set suffixsvf [open "sw_suffix.svf" "r"]
puts $final_svf "\n// Issuing Run command to PowerPC to start
execution\n"
fcopy $suffixsvf $final_svf
close $suffixsvf
```

Command line genace options are:

```
xmd -tcl genace.tcl -target mdm -elf executable1.elf executable2.svf -
data image.bin 0xfe000000 -board mbdemo -ace system.ace
```

Software Only Configuration on MicroBlaze; Downloading Using Fast Download

For a MicroBlaze target, software files can be downloaded with greater speeds using Fast Download methodology. This requires the presence of an FSL link between MicroBlaze and MDM, which makes downloading faster and creates a small ACE file.

This configuration requires changing the `genace.tcl` script. In procedure `genace` add the following line:

```
append xmd_options " -pfsl port 0 type s"
# Get the irLength of FPGA.
set options_list [concat $xmd_options]
```

The command line options are the same as the configurations above.

ACE Generation for a Single Processor in Multiple Processors System:

Many of the Virtex™-II Pro and Virtex™-4 devices contain two PowerPC processors or the System might contain multiple MicroBlaze processors. To generate ACE file for a single processor use **-debugdevice** option. Use `cpunr` to specify the Processor instance.

In the example we assume a configuration with two PowerPC processors and ACE file is generated for processor number 2. The options file for this configuration is:

```
-jprog
-target ppc_hw
-hw implementation/download.bit
-elf executable.elf
-ace final_system.ace
-board user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
-debugdevice devicenr 1 cpunr 2 <= Note: The cpunr is 2
```

Multiple PowerPC Processors System Configuration:

Many of the Virtex-II Pro and Virtex-4 devices contain two or more processors. Typically, each of these processors requires a separate ELF file. We assume a configuration with two PowerPC processors, each loaded with a single ELF file. The configuration of the board is specified in the options file.

This configuration requires the following steps to generate the ACE file:

1. Generate an SVF for the BIT file.

   The options file text is displayed below:

   ```
   -jprog
   -target ppc_hw
   -hw implementation/download.bit
   -ace final_system.ace
   -board user
   -configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
   -debugdevice devicenr 1 cpunr 1
   ```

2. Generate an SVF for the First Processor ELF file.

   The options file text is displayed below:

   ```
   -target ppc_hw
   -elf executable1.elf
   -ace elf1.ace
   -board user
   -configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
   -debugdevice devicenr 1 cpunr 1
   ```

   This generates the `executable1.svf and sw_suffix.svf` files. Copy the `sw_suffix.svf` file to the `sw_suffix1.svf` file.

3. Generate an SVF for the Second Processor ELF file.

   The options file text is displayed below:

   ```
   -target ppc_hw
   -elf executable2.elf
   -ace elf2.ace
   -board user
   -configdevice devicenr 1 idcode 0x1266093 irlength 14 partname
   XC2VP20
   -debugdevice devicenr 1 cpunr 2 <= Note: The cpunr is 2.
   ```

   This generates the `executable2.svf` and `sw_suffix.svf` files. Copy the `sw_suffix.svf` file to the `sw_suffix2.svf` file.

4. Concatenate the files in the following order: `final_system.svf`, `executable1.svf`, `executable2.svf`, `sw_suffix1.svf`, and `sw_suffix2.svf` to `final_system.svf`.

5. Generate the ACE file by calling `impact -batch svf2ace.scr`. The following SCR file should be used:

   ```
   svf2ace -wtck -d -i final_system.svf -o final_system.ace
   quit
   ```

Multiple MicroBlaze Processors System Configuration

This example uses a configuration with two MicroBlaze processors connected to MDM, each loaded with a single ELF file. The configuration of the board is specified in the options file.

This configuration requires multiple steps to generate the ACE file and change to `genace.tcl` script.

1. Generate an SVF for the BIT file.

   The options file text is displayed below:

   ```
   -jprog
   -target mdm
   -hw implementation/download.bit
   -ace final_system.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 1 cpunr 1
   ```

2. Generate an SVF for the First Processor ELF file.

   The options file text is displayed below:

   ```
   -target mdm
   -elf executable1.elf
   -ace elf1.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 1 cpunr 1
   ```

   This generates the `executable1.svf` and `sw_suffix.svf` files. Copy `sw_suffix.svf` file to the `sw_suffix1.svf` file.

3. Generate an SVF for the Second Processor ELF file.

   The options file text is displayed below:

   ```
   -target mdm
   -elf executable2.elf
   -ace elf2.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 1 cpunr 2 <= Note: The cpunr is 2.
   ```

   This generates the **e**`xecutable2.svf` and s`w_suffix.svf` files. Copy the `sw_suffix.svf` file to the `sw_suffix2.svf` file.

4. Concatenate the files in the following order: `final_system.svf`, `executable1.svf`, `executable2.svf`, `sw_suffix1.svf`, and `sw_suffix2.svf` to `final_system.svf`.

5. Generate the ACE file by calling `impact -batch svf2ace.scr`. The following SCR file should be used:

   ```
   svf2ace -wtck -d -i final_system.svf -o final_system.ace
   quit
   ```

Multiple FPGA Devices

Generation of System ACE files for boards with multiple FPGA devices follows the same pattern as multi-processor configuration. We assume a configuration with two FPGA devices, each with a single Processor and single ELF file. The configuration of the board is specified in the options file.

This configuration requires multiple steps to generate the ACE file.

1. Generate an SVF for the first FPGA device.

   The options file is given below:

   ```
   -jprog
   -target ppc_hw
   -hw implementation/download.bit
   -elf executable1.elf
   -ace fpga1.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -configdevice devicenr 2 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 1 cpunr 1
   ```

   This generates the file `fpga1.svf`.

2. Generate an SVF for the second FPGA device.

   The options file is given below:

   ```
   -jprog
   -target ppc_hw
   -hw implementation/download.bit
   -elf executable2.elf
   -ace fpga2.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -configdevice devicenr 2 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 2 cpunr 1 <= Note: The change in Devicenr
   ```

   This generates the file `fpga2.svf`.

3. Concatenate the files in the following order: `fpga1.svf` and `fpga2.svf` to `final_system.svf`.
4. Generate the ACE file by calling `impact -batch svf2ace.scr`. The following SCR file should be used:

   ```
   svf2ace -wtck -d -i final_system.svf -o final_system.ace
   quit
   ```

Related Information

Adding a New Device to the JTAG Chain

An XMD supported device list is located at `$XILINX_EDK/data/xmd/deviceid.lst` or `$XILINX_EDK/data/xmd/devicetable.lst`. If XMD detects a device as "UNKNOWN" in the JTAG chain, the IR length of the device has to be specified manually by the user. This can be done in the following ways:

- Edit the `devicetable.list`.

  Add a new entry for the device to the `devicetable.lst`. XMD then supports the device. You must add Device ID Code, IR Length, and Name in the list file. For more details, refer the `devicetable.lst` file.

- Edit the `deviceid.list`.

  Add a new entry for the device to the `deviceid.lst`. XMD then supports the device. You must add entry to all three sections in the list file. For more details, refer the `deviceid.lst` file.

- Provide a GenACE OPT file with options.

  You can specify the `-configdevice` and `-debugdevice` options in the options file.

- Edit the `genace.tcl` file.

  Add a new board type in `genace.tcl`. Specify the `xmd_options`, `jtag_fpga_position`, and `jtag_devices` values for the board. Refer to the `genace.tcl` file in EDK installation at `$XILINX_EDK/data/xmd/genace.tcl` for more details.

CF Device Format

To have the System ACE controller read the CF device, do the following:

1. Format the CF device as FAT 16.
2. Create a `Xilinx.sys` file in the root directory. This file contains the directory structure to use by the ACE controller. Copy the generated ACE file to the appropriate directory. For more information refer to the "iMPACT" section of the ISE documentation.

Glossary

Click on a letter, or scroll down to view the entire glossary.

B C D E F G H I J L M N O P S U V X Z

B

BBD file

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

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.

BMM file

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

BSB

Base System Builder. A wizard for creating a complete EDK design.

BSP

See Standalone BSP.

C

CSV file

Comma Separated Value file.

D

DCR

Device Control Register.

DLMB

Data-side Local Memory Bus. See also: LMB

DMA

Direct Memory Access.

DOPB

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

DRC

Design Rule Check.

E

EDIF file

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

EDK

Embedded Development Kit.

ELF file

Executable Linked Format file.

EMC

Enclosure Management Controller.

EST

Embedded System Tools.

F

FATfs (XilFATfs)

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

FPGA

Field Programmable Gate Array.

FSL

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

G

GDB

GNU Debugger.

GPIO

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

H

HDL

Hardware Description Language.

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.

ISS

Instruction Set Simulator.

J

JTAG

Joint Test Action Group.

L

Libgen

Library Generator sub-component of the Platform Studio(tm) 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.

LibXil File

A module that provides block access to files and devices. The LibXil File module provides standard routines such as open, close, read, and write.

LibXil Net

The network library for embedded processors.

LibXil Profile

A software intrusive profile library that generates call graph and histogram information of any program running on a board.

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

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.

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.

MVS file

Microprocessor Verification Specification file.

N

NCF file

Netlist Constraints file.

O

OCM

On Chip Memory.

OPB

On-chip Peripheral Bus.

P

PACE

Pinout and Area Constraints Editor.

PAO file

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

PBD file

Processor Block Diagram file.

Platgen

Hardware Platform Generator sub-component of the Platform Studio(tm) technology.

PLB

Processor Local Bus.

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.

Simgen

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

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).

U

UCF

User Constraints File.

V

VHDL

VHSIC Hardware Description Language.

VP

Virtual Platform.

VPgen

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

X

XCL

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

Xilkernel

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

XMD

Xilinx® Microprocessor Debugger.

XMK

Xilinx® Microkernel. The entity representing the collective software system comprising the standard C libraries, Xilkernel, Standalone BSP, LibXil Net, LibXil MFS, LibXil File, and LibXil Drivers.

XMP

Xilinx® Microprocessor Project. 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 Tool.

Z

ZBT

Zero Bus Turnaround™.