

Embedded System Tools Reference Manual

EDK

UG111 (v14.1) April 24, 2012



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

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

© Copyright 2012 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

| Date | Version | Revision |
|------------|---------|---|
| 03/01/2011 | 13.1 | EDK 13.1 release. Revision numbering format change to match release number. |
| 07/06/2011 | 13.2 | EDK 13.2 release. |
| 10/19/2011 | 13.3 | EDK 13.3 release changes: <ul style="list-style-type: none">• Added a <code>-parallel</code> option to Platgen. See Table 4-1, page 44.• Added an <code>xget/xset</code> command option for Verilog Compiler code Simulation (VCS). See Table 5-1, page 48.• Added AXI BFM information to the BFM Chapter. See Using the AXI BFM Package, page 63 through Running AXI BFM Simulations, page 67.• Chapter 7, Simulation Model Generator (Simgen) edits.• Removed <code>_mh</code> Library name from Table 9-9, page 122.• Added Digilint cable option to XMD: See Table 10-7, page 154.• Removed change history from Chapter 15, Version Management Tools (revup). |
| 01/18/2012 | 13.4 | EDK 13.4 release changes: <ul style="list-style-type: none">• Added information about using a <code>.bsb</code> settings file in Using a .bsb Settings File, page 13.• In Table 4-1, page 44, Added information about the Netlist Hierarchy option.• In Creating a New Empty Project, page 47, added link to an answer record for creating a new empty project using 6 series architecture or later. |
| 04/24/2012 | 14.1 | Updated Chapter 10, Xilinx Microprocessor Debugger (XMD) with information about the Dual ARM Cortex-A9 MPCore. |

Table of Contents

| | |
|---|----|
| Revision History | 2 |
| Chapter 1: Embedded System and Tools Architecture Overview | |
| Design Process Overview | 8 |
| EDK Overview | 10 |
| EDK Tools and Utilities | 11 |
| Xilinx Platform Studio | 13 |
| Software Development Kit | 17 |
| Chapter 2: Platform Specification Utility (PsfUtility) | |
| Tool Options | 21 |
| MPD Creation Process Overview | 22 |
| Use Models for Automatic MPD Creation | 23 |
| DRC Checks in PsfUtility | 25 |
| Conventions for Defining HDL Peripherals | 25 |
| Chapter 3: Psf2Edward Program | |
| Program Usage | 41 |
| Program Options | 41 |
| Chapter 4: Platform Generator (Platgen) | |
| Features | 43 |
| Tool Requirements | 43 |
| Tool Usage | 44 |
| Supported Platgen Syntax Options | 44 |
| Load Path | 45 |
| Output Files | 45 |
| Synthesis Netlist Cache | 46 |
| Chapter 5: Command Line Mode | |
| Invoking XPS Command Line Mode | 47 |
| Creating a New Empty Project | 47 |
| Creating a New Project With an Existing MHS | 48 |
| Opening an Existing Project | 48 |
| Saving Your Project Files | 48 |
| Setting Project Options | 48 |
| Executing Flow Commands | 50 |
| Reloading an MHS File | 50 |

| | |
|--------------------------------------|----|
| Adding or Updating an ELF File | 51 |
| Deleting an ELF File | 51 |
| Archiving Your Project Files | 51 |
| Restrictions | 51 |

Chapter 6: Bus Functional Model Simulation

| | |
|---|----|
| Bus Functional Model Use Cases | 54 |
| Bus Functional Simulation Methods | 55 |
| PLB BFM Package | 57 |
| Using the AXI BFM Package | 63 |

Chapter 7: Simulation Model Generator (Simgen)

| | |
|----------------------------------|----|
| Simgen Overview | 69 |
| Simulation Libraries | 69 |
| Compplib Utility | 71 |
| Simulation Models | 71 |
| Simgen Syntax | 74 |
| Output Files | 77 |
| Memory Initialization | 78 |
| External Memory Simulation | 81 |
| Simulating Your Design | 83 |

Chapter 8: Library Generator (Libgen)

| | |
|--|----|
| Overview | 85 |
| Tool Usage | 85 |
| Tool Options | 85 |
| Load Paths | 86 |
| Output Files | 87 |
| Generating Libraries and Drivers | 89 |
| MSS Parameters | 90 |
| Drivers | 90 |
| Libraries | 91 |
| OS Block | 91 |

Chapter 9: GNU Compiler Tools

| | |
|---|-----|
| Overview | 93 |
| Compiler Framework | 94 |
| Common Compiler Usage and Options | 95 |
| MicroBlaze Compiler Usage and Options | 109 |
| PowerPC Compiler Usage and Options | 124 |
| Other Notes | 131 |

Chapter 10: Xilinx Microprocessor Debugger (XMD)

| | |
|-----------------------------------|-----|
| XMD Usage | 134 |
| XMD Console | 135 |
| XMD Command Reference | 136 |
| Connect Command Options | 153 |
| PowerPC Processor Targets | 154 |
| MicroBlaze Processor Target | 167 |
| Cortex A9 Processor Target | 178 |
| XMD Internal Tcl Commands | 181 |

Chapter 11: GNU Debugger

| | |
|------------------------------|-----|
| Overview | 187 |
| Tool Overview | 187 |
| MicroBlaze GDB Targets | 188 |
| PowerPC 405 Targets | 189 |
| PowerPC 440 Targets | 190 |
| Console Mode | 190 |
| GDB Command Reference | 191 |
| Additional Resources | 191 |

Chapter 12: Bitstream Initializer (BitInit)

| | |
|--------------------|-----|
| Overview | 193 |
| Tool Usage | 193 |
| Tool Options | 193 |

Chapter 13: System ACE File Generator (GenACE)

| | |
|-----------------------------|-----|
| Assumptions | 195 |
| Tool Requirements | 195 |
| GenACE Features | 196 |
| GenACE Model | 196 |
| The Genace.tcl Script | 197 |
| Generating ACE Files | 200 |
| Related Information | 205 |

Chapter 14: Flash Memory Programming

| | |
|-------------------------------------|-----|
| Overview | 207 |
| Supported Flash Hardware | 208 |
| Flash Programmer Performance | 209 |
| Customizing Flash Programming | 209 |

Chapter 15: Version Management Tools (revup)

| | |
|--|-----|
| Command Line Option for the Format Revision Tool | 215 |
|--|-----|

| | |
|-------------------------------------|-----|
| The Version Management Wizard | 215 |
|-------------------------------------|-----|

Chapter 16: Microprocessor Peripheral Definition Translation tool (MPDX)

| | |
|--------------------------|-----|
| XBD2 | 217 |
| Define Constraints | 223 |

Appendix A: GNU Utilities

| | |
|--|-----|
| General Purpose Utility for MicroBlaze and PowerPC | 227 |
| Utilities Specific to MicroBlaze and PowerPC..... | 227 |
| Other Programs and Files..... | 229 |

Appendix B: Interrupt Management

| | |
|---|-----|
| Hardware Setup..... | 231 |
| Software Setup and Interrupt Flow | 232 |
| Software APIs..... | 237 |

Appendix C: EDK Tcl Interface

| | |
|---|-----|
| Introduction | 247 |
| Understanding Handles | 247 |
| Data Structure Creation | 248 |
| Tcl Command Usage | 249 |
| EDK Hardware Tcl Commands..... | 250 |
| Tcl Example Procedures | 258 |
| Tcl Flow During Hardware Platform Generation..... | 266 |
| Additional Keywords in the Merged Hardware Datastructure..... | 271 |

Appendix D: Interconnect Settings and Parameter Automations for AXI Designs

| | |
|--|-----|
| Allowed Parameters in Master and Slave Interfaces..... | 273 |
| Building Vectors | 275 |
| Parameter Automations | 275 |

Appendix E: Additional Resources

| | |
|--------------------------------|-----|
| Xilinx Resources | 279 |
| EDK Documentation | 279 |
| EDK Additional Resources | 280 |

Embedded System and Tools Architecture Overview

This chapter describes the architecture of the embedded system tools and flows provided in the Xilinx® Embedded Development Kit (EDK) for developing systems based on the MicroBlaze™ embedded processors and the PowerPC® (405 and 440) processors.

The Xilinx Embedded Development Kit (EDK) system tools enable you to design a complete embedded processor system for implementation in a Xilinx FPGA device.

EDK is a component of the Integrated Software Environment (ISE®) Design Suite Embedded and System Editions. ISE is a Xilinx development system product that is required to implement designs into Xilinx programmable logic devices. EDK includes:

- The Xilinx Platform Studio (XPS) system tools suite with which you can develop your embedded processor hardware.
- The Software Development Kit (SDK), based on the Eclipse open-source framework, which you can use to develop your embedded software application. SDK is also available as a standalone program.
- Embedded processing Intellectual Property (IP) cores including processors and peripherals.

While the EDK environment supports creating and implementing designs, the recommended flow is to begin with an ISE project, then add an embedded processor source to the ISE project. EDK depends on ISE components to synthesize the microprocessor hardware design, to map that design to an FPGA target, and to generate and download the bitstream.

For information about ISE, refer to the ISE software documentation. For links to ISE documentation and other useful information see [Appendix E, Additional Resources](#).

Design Process Overview

The tools provided with EDK are designed to assist in all phases of the embedded design process, as illustrated in Figure 1-1.

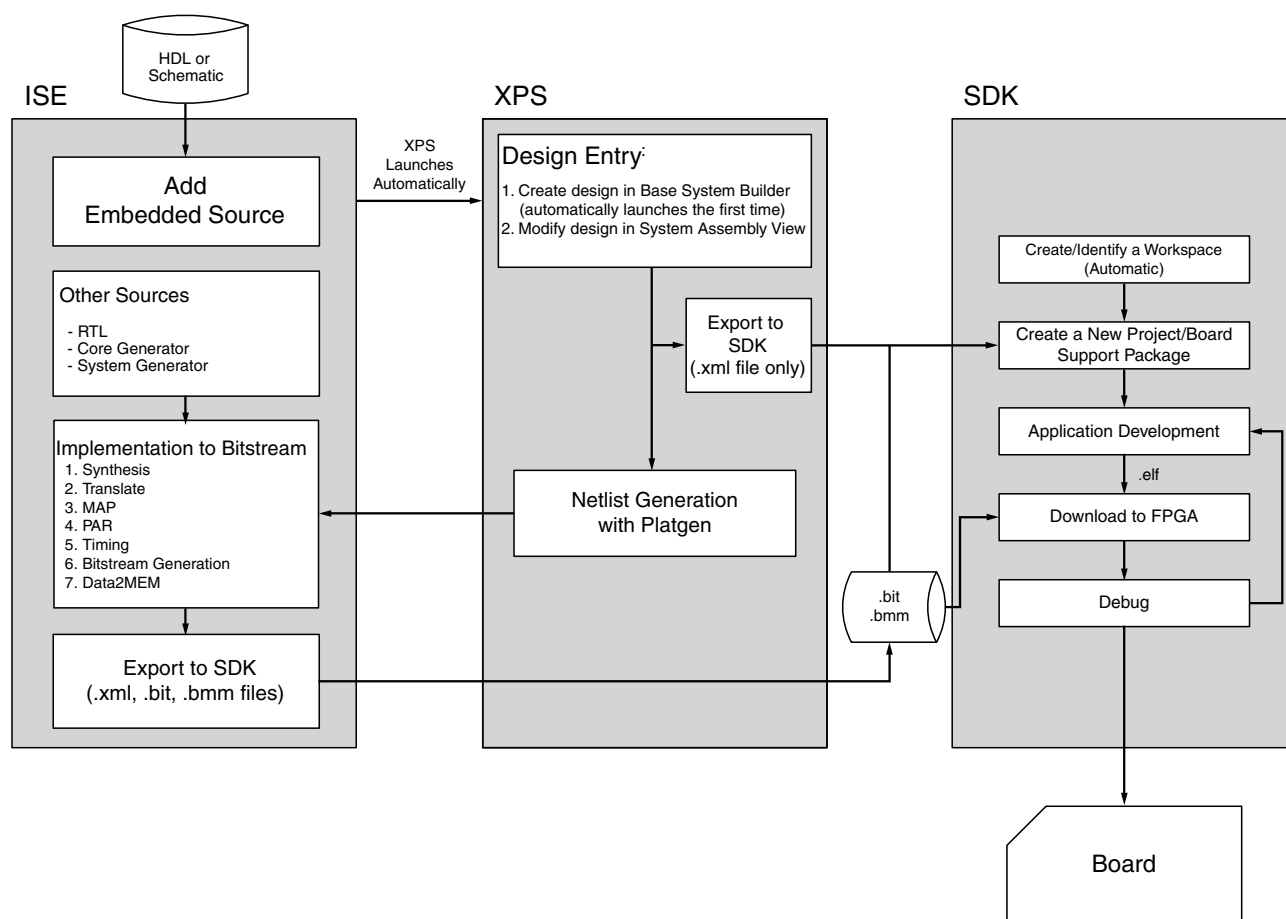


Figure 1-1: Embedded Design Process Flow

Hardware Development

Xilinx FPGA technology allows you to customize the hardware logic in your processor subsystem. Such customization is not possible using standard off-the-shelf microprocessor or controller chips.

The term “Hardware platform” describes the flexible, embedded processing subsystem you are creating with Xilinx technology for your application needs.

The hardware platform consists of one or more processors and peripherals connected to the processor buses. XPS captures the hardware platform description in the Microprocessor Hardware Specification (MHS) file.

The MHS file is the principal source file that maintains the hardware platform description and represents in ASCII text the hardware components of your embedded system.

When the hardware platform description is complete, the hardware platform can be exported for use by SDK.

Software Development

A board support package (BSP) is a collection of software drivers and, optionally, the operating system on which to build your application. The created software image contains only the portions of the Xilinx library you use in your embedded design. You can create multiple applications to run on the BSP.

The hardware platform must be imported into SDK prior to creation of software applications and BSP.

Verification

EDK provides both hardware and software verification tools. The following subsections describe the verification tools available for hardware and software.

Hardware Verification Using Simulation

To verify the correct functionality of your hardware platform, you can create a simulation model and run it on an Hardware Design Language (HDL) simulator. When simulating your system, the processor(s) execute your software programs. You can choose to create a behavioral, structural, or timing-accurate simulation model.

ISim (the ISE simulator) now supports simulation of embedded designs. When you create a project in ISE and add an embedded project source, you can launch ISim from within ISE. When no ISE project is used, you can launch the ISim software directly from within Platform Studio.

Software Verification Using Debugging

The following options are available for software verification:

- You can load your design on a supported development board and use a debugging tool to control the target processor.
- You can gauge the performance of your system by profiling the execution of your code.

Device Configuration

When your hardware and software platforms are complete, you then create a configuration bitstream for the target FPGA device.

- For prototyping, download the bitstream along with any software you require to run on your embedded platform while connected to your host computer.
- For production, store your configuration bitstream and software in a non-volatile memory connected to the FPGA.

EDK Overview

An embedded hardware platform typically consists of one or more processors, peripherals and memory blocks, interconnected via processor buses. It also has port connections to the outside world. Each of the processor cores (also referred to as *pcores* or *processor IPs*) has a number of parameters that you can adjust to customize its behavior. These parameters also define the address map of your peripherals and memories. XPS lets you select from various optional features; consequently, the FPGA needs only implement the subset of functionality required by your application.

Figure 1-2 provides an overview of the EDK architecture structure of how the tools operate together to create an embedded system.

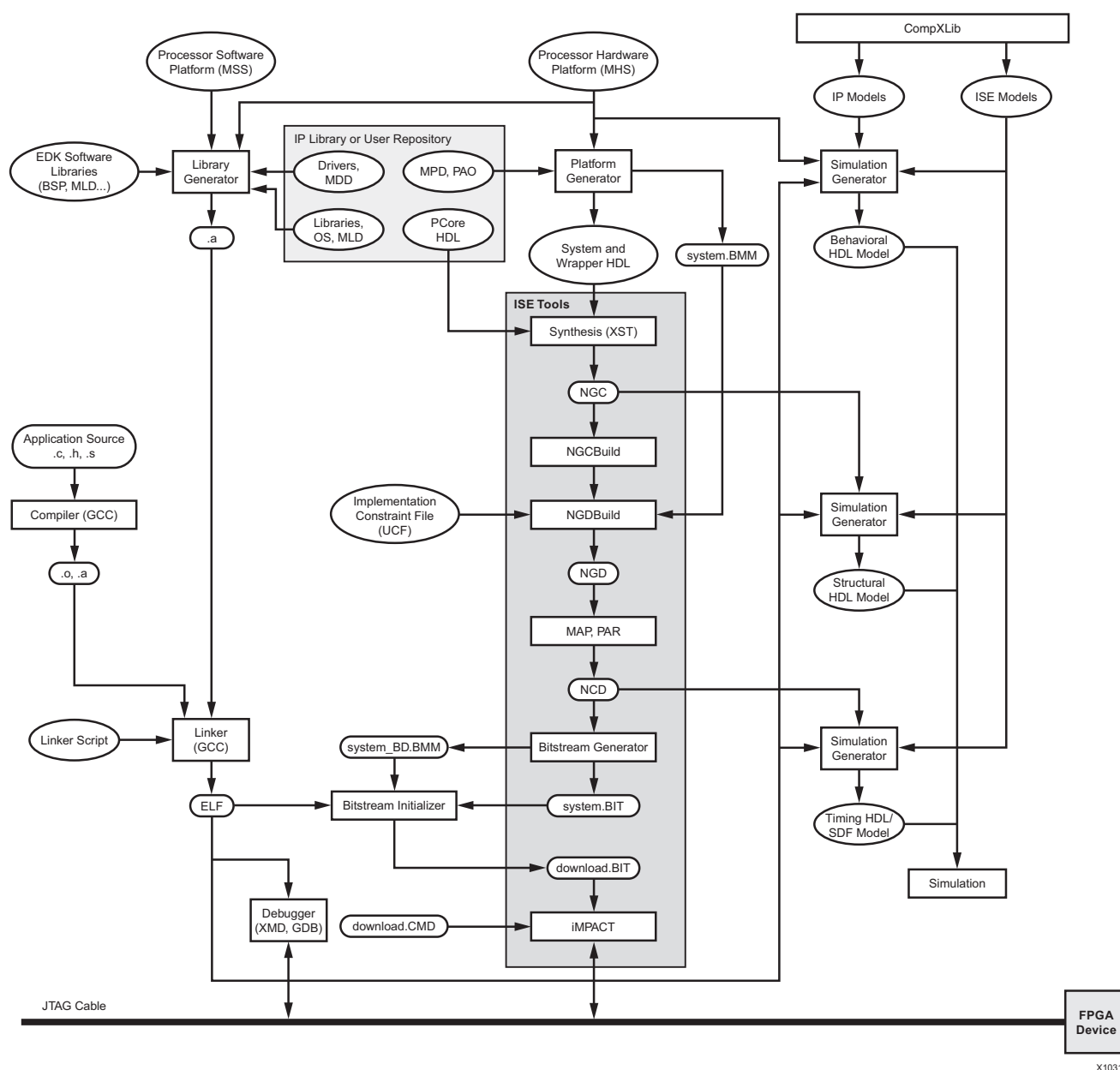


Figure 1-2: Embedded Development Kit (EDK) Tools Architecture

EDK Tools and Utilities

The following table describes the tools and utilities supported in EDK. The subsections that follow provide an overview of each tool, with references to the chapters that contain additional information.

Table 1-1: EDK Tools and Utilities

| Hardware Development and Verification | |
|---|--|
| Xilinx Platform Studio | An integrated design environment (GUI) in which you can create your embedded hardware design. |
| The Base System Builder Wizard | Allows you to quickly create a working embedded design using any features of a supported development board or using basic functionality common to most embedded systems. For initial project creation it is recommended to use the BSB wizard. |
| The Create and Import Peripheral Wizard | Assists you in adding your own custom peripheral(s) to a design. The CIP creates associated directories and data files required by XPS. the Platform Specification Utility (PsfUtility) tool enables automatic generation of Microprocessor Peripheral Definition (MPD) files, which are required to create IP peripherals that are compliant with the Embedded Development Kit (EDK). The CIP wizard in XPS supports features provided by the PsfUtility for MPD file creation (recommended.) |
| Coprocessor Wizard | Helps you add a coprocessor to a CPU. (This applies to MicroBlaze-based designs only.) |
| Platform Generator (Platgen) | Constructs the programmable system on a chip in the form of HDL and synthesized netlist files. |
| XPS Command Line or “No Window” Mode | Allows you to run embedded design flows or change tool options from a command line. |
| Bus Functional Model | Helps simplify the verification of custom peripherals by creating a model of the bus environment to use in place of the actual embedded system. |
| Simulation Model Generator (Simgen) | Generates the hardware simulation model and the compilation script files for simulating the complete system. |
| Simulation Library Compiler (Compplib) | Compiles the EDK simulation libraries for the target simulator, as required, before starting behavioral simulation of the design. |
| Software Development and Verification | |
| Software Development Kit | An integrated design environment, the Software Development Kit (SDK) helps with the development of software application projects. |
| Library Generator (Libgen) | Constructs a BSP comprising a customized collection of software libraries, drivers, and OS. |

Table 1-1: EDK Tools and Utilities (Cont'd)

| | |
|--|--|
| GNU Compiler Tools | Builds a software application based on the platforms created by the Libgen. |
| Xilinx Microprocessor Debugger | Used for software download and debugging. Also provides a channel through which the GNU debugger accesses the device. |
| GNU Debugger | GUI for debugging software on either a simulation model or target device. |
| Bitstream Initializer (Bitinit) | Updates an FPGA configuration bitstream to initialize the on-chip instruction memory with the software executable. |
| Debug Configuration Wizard | Automates hardware and software platform debug configuration tasks common to most designs. |
| System ACE File Generator (GenACE) | Generates a Xilinx System ACE™ configuration file based on the FPGA configuration bitstream and software executable to be stored in a compact flash device in a production system. |
| Flash Memory Programmer | Allows you to use your target processor to program on-board Common Flash Interface (CFI)-compliant parallel flash devices with software and data. |
| Format Revision Tool and Version Management Wizard | Updates the project files to the latest format. The Version Management wizard helps migrate IPs and drivers created with an earlier EDK release to the latest version. |
| Platform Specification Utility (PsfUtility) and PSF2EDWARD Program | The PsfUtility enables automatic generation of Microprocessor Peripheral Definition (MPD) files required to create an IP core compliant with EDK. The psf2Edward is a command line program that converts a Xilinx® Embedded Development Kit (EDK) project into Edward, an internal XML format, for use in programs such as the Software Development Kit (SDK). |
| Microprocessor Peripheral Definition Translation tool (MPDX) | The MPDX is a translation tool that generates the IP-XACT files on disk for the BSB repository. |

Xilinx Platform Studio

Xilinx Platform Studio (XPS) offers the following features:

- Ability to add processor and peripheral cores, edit core parameters, and make bus and signal connections to generate an MHS file.
- Support for tools described in [Table 1-1, page 11](#).
- Ability to generate and view a system block diagram and/or design report.
- Project management support.
- Process and tool flow dependency management.
- Ability to export hardware specification files for import into SDK.

For more information on files and their formats see the *Platform Specification Format Reference Manual*, which is linked in [Additional Resources, page 279](#).

Refer to the *Xilinx Platform Studio Help* for details on using the XPS GUI. The following subsections describe the tool and utility components of XPS.

The Base System Builder Wizard

The Base System Builder (BSB) wizard helps you quickly build a working system. Some embedded design projects can be completed using the BSB wizard alone. For more complex projects, the BSB wizard provides a baseline system that you can then customize to complete your embedded design. The BSB wizard can generate a single-processor design for the supported processor types, and dual processor designs for MicroBlaze. For efficiency in project creation, Xilinx recommends using the BSB wizard in every scenario.

Based on the board you choose, the BSB wizard allows you to select and configure basic system elements such as processor type, debug interface, cache configuration, memory type and size, and peripheral selection. The BSB provides functional default values pre-selected in the wizard that you can modify as needed.

If your target development board is not available or not currently supported by the BSB wizard, you can select the Custom Board option instead of selecting a target board. Using this option, you can specify the individual hardware devices that you expect to have on your custom board. To run the generated system on a custom board, you enter the FPGA pin location constraints into the User Constraints File (UCF). If a supported target board is selected, the BSB wizard inserts these constraints into the UCF automatically.

For detailed information on using the features provided in the BSB wizard, see the *Xilinx Platform Studio Help*.

Using a .bsb Settings File

When you use the Base System Builder, it automatically creates a .bsb settings file upon exit that stores all of the selections that you made in that wizard session. When you load this file in a subsequent session, the wizard pre-loads all the interface selections that were stored in the file rather than the usual defaults. This option is useful if you have an existing design generated by the BSB and want to create another identical or similar to it.

Note: This feature is intended to help you create a new design that is similar to one in another project, not to modify an existing project. The BSB can only be invoked on a new project. After you have created a design, either with the wizard or manually, you cannot re-run the BSB to modify that design.

The .bsb settings file stores only BSB wizard selections and does not reflect any changes made to the system outside of the BSB, such as adding or editing a peripheral in XPS or manually editing the Microprocessor Hardware Specification (MHS) file.

To use a .bsb settings file that you created in an earlier session of the Base System Builder:

1. Start the Base System Builder by selecting **Create New Project Using Base System Builder** from the Welcome page, or by selecting **File > New BSB Project**.
2. In the Create New XPS Project Using BSB Wizard screen, name your new project.
3. In the **Select Existing .bsb Settings File** field, browse to the saved .bsb settings file. When you select the file, details about the project display in the information window.

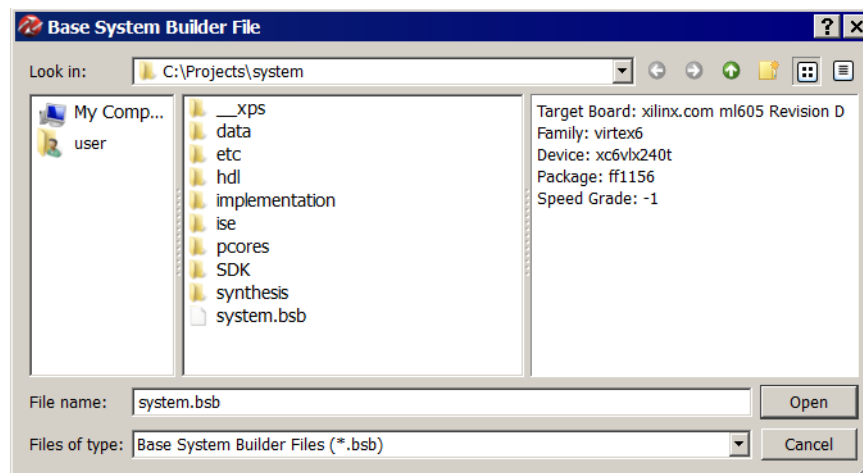


Figure 1-3: Viewing .bsb Settings File Information

The Create and Import Peripheral Wizard

The Create and Import Peripheral (CIP) wizard helps you create your own peripherals and import them into XPS-compliant repositories or projects.

In the *Create* mode, the CIP wizard creates templates that help you implement your peripheral without requiring detailed understanding of the bus protocols, naming conventions, or the formats of special interface files required by XPS. By referring to the examples in the template file and using various auxiliary design support files that are output by the wizard, you can start quickly on designing your custom logic.

In the *Import* mode, this tool creates the interface files and directory structures that are necessary to make your peripheral visible to the various tools in XPS. For the Import operation mode, it is assumed that you have followed the required XPS naming conventions. Once imported, your peripheral is available in the XPS peripherals library.

When you create or import a peripheral, XPS generates the Microprocessor Peripheral Definition (MPD) and Peripheral Analyze Order (PAO) files automatically:

- The MPD file defines the interface for the peripheral.
- The PAO file specifies to Platgen and Simgen what HDL files are required for compilation (synthesis or simulation) for the peripheral and in the order of those files.

For more information about MPD and PAO files, see the *Platform Specification Format Reference Manual*. A link to the document is available in [Additional Resources, page 279](#). For detailed information on using the features provided in the CIP wizard, see the *Xilinx Platform Studio Help*.

Platform Specification Utility (PsfUtility) and PSF2EDWARD Program

The PsfUtility enables automatic generation of Microprocessor Peripheral Definition (MPD) files required to create an IP core compliant with EDK. Features provided by this tool can be used with the help of the CIP wizard.

See [Chapter 2, “Platform Specification Utility \(PsfUtility\).”](#)

The psf2Edward is a command line program that converts a Xilinx® Embedded Development Kit (EDK) project into Edward, an internal XML format, for use in external programs such as the Software Development Kit (SDK). See [Chapter 3, “Psf2Edward Program.”](#)

Coprocessor Wizard

The Configure Coprocessor wizard helps add and connect a coprocessor to a CPU. A coprocessor is a hardware module that implements a user-defined function and connects to the processor through an auxiliary bus. The coprocessor has a Fast Simplex Link (FSL) interface. For MicroBlaze™ processor systems, the coprocessor connects to the FSL interface. For PowerPC® processor systems, the coprocessor connects to the Auxiliary Processor Unit (APU) interface of the PowerPC processor through the fcb2fsl bridge.

For details on the Fast Simplex Link, refer to its data sheet and the *MicroBlaze Processor Reference Guide* (UG). For information about the APU bus, refer to the PowerPC reference guides. For information on the fcb2fsl bridge, refer to its data sheet. Links to document locations are available in the [Additional Resources, page 279](#).

For instructions on using the Coprocessor wizard, refer to the *Xilinx Platform Studio Help*.

Platform Generator (Platgen)

Platgen compiles the high-level description of your embedded processor system into HDL netlists that can be implemented in a target FPGA device.

Platgen:

- Reads the MHS file as its primary design input.
- Reads various processor core (pcore) hardware description files (MPD, PAO) from the XPS project and any user IP repository.
- Produces the top-level HDL design file for the embedded system that stitches together all the instances of parameterized pcores contained in the system. In the process, it resolves the high-level bus connections in the MHS into the actual signals required to interconnect the processors, peripherals and on-chip memories. (The system-level HDL netlist produced by Platgen is used as part of the FPGA implementation process.)
- Invokes the XST (Xilinx Synthesis Technology) compiler to synthesize each of the instantiated pcores.
- Generates the block RAM Memory Map (BMM) file which contains addresses and configuration of on-chip block RAM memories. This file is used later for initializing the block RAMs with software.

[Chapter 4, “Platform Generator \(Platgen\),”](#) provides a detailed description of the Platgen tool.

XPS Command Line or “No Window” Mode

XPS includes a “no window” mode that lets you run from an operating system command line. [Chapter 5, “Command Line Mode,”](#) provides information on the command line feature in XPS.

Bus Functional Model

Bus Functional Model (BFM) simulation simplifies the verification of hardware components that attach to a bus. [Chapter 6, “Bus Functional Model Simulation,”](#) provides information about BFM simulation.

Debug Configuration Wizard

The Debug Configuration wizard automates hardware and software platform debug configuration tasks common to most designs.

You can instantiate a ChipScope™ analyzer core to monitor the AMBA AXI4 interface, Processor Local Bus (PLB), or any other system-level signals. In addition, you can configure the parameters of an existing ChipScope core for hardware debugging. You can also provide JTAG-based virtual input and output.

To configure the software for debugging you can set the processor debug parameters. When co-debugging is enabled for a ChipScope core, you can set up mutual triggering between the software debugger and the hardware signals. The JTAG interface can be configured to transport UART signals to the Xilinx Microprocessor Debugger (XMD).

For detailed information on using the features provided in the Debug Configuration wizard, see the *Xilinx Platform Studio Help*.

Simulation Model Generator (Simgen)

The Simulation Platform Generation tool (Simgen) generates and configures various simulation models for the hardware. To generate a behavioral model, Simgen takes an MHS file as its primary design input. For generating structural or timing models, Simgen takes its primary design input from the post-synthesis or post-place-and-route design database, respectively. Simgen also reads the embedded application executable (ELF) file for each processor to initialize on-chip memory, thus allowing the modeled processor(s) to execute their software code during simulation.

Simgen also provides simulation models for external memory and has automated support to instantiate memory models in the simulation testbench and perform connection with the design under test. To compile memory model into the user library, Simgen also generates simulator-specific compilation/elaboration commands into respective helper/setup scripts.

Refer to [Chapter 7, Simulation Model Generator \(Simgen\)](#) for more information.

Software Development Kit

The Software Development Kit (SDK) provides a development environment for software application projects. SDK is based on the Eclipse open-source standard. SDK has the following features:

- Can be installed independent of ISE and XPS with a small disk footprint.
- Supports development of software applications on single- or multi-processor systems.
- Imports the XPS-generated hardware platform definition.
- Supports development of software applications in a team environment.
- Ability to create and configure board support packages (BSPs) for third-party OS.
- Provides off-the-shelf sample software projects to test the hardware and software functionality.
- Has an easy GUI interface to generate linker scripts for software applications, program FPGA devices, and program parallel flash memory.
- Has feature-rich C/C++ code editor and compilation environment.
- Provides project management.
- Configures application builds and automates the make file generation.
- Supplies error navigation.
- Provides a well-integrated environment for seamless debugging and profiling of embedded targets.

For more information about SDK, see the *Software Development ToolKit (SDK) Help*.

Library Generator (Libgen)

Libgen configures libraries, device drivers, file systems, and interrupt handlers for the embedded processor system, creating a board support package (BSP). The BSP defines, for each processor, the drivers associated with the peripherals you include in your hardware platform, selected libraries, standard input and output devices, interrupt handler routines, and other related software features. Your SDK projects further define software applications to run on each processor, which are based on the BSP.

Taking libraries and drivers from the installation, along with any custom libraries and drivers for custom peripherals you provide, SDK is able to compile your applications, including libraries and drivers, into Executable Linked Format (ELF) files that are ready to run on your processor hardware platform.

Libgen reads selected libraries and processor core (pcore) software description files (Microprocessor Driver Definition (MDD) and driver code) from the EDK library and any user IP repository.

Refer to [Chapter 8, Library Generator \(Libgen\)](#) and the *Xilinx Platform Studio Help* for more information. For more information on libraries and device drivers, refer to the Xilinx software components documented in the *OS and Libraries Document Collection*. Links to the documentation are supplied in the [Additional Resources, page 279](#).

GNU Compiler Tools

GNU compiler tools (GCC) are called for compiling and linking application executables for each processor in the system. Processor-specific compilers are:

- The `mb-gcc` compiler for the MicroBlaze processor.
- The `powerpc-eabi-gcc` compiler for the PowerPC processor.

As shown in the embedded tools architectural overview ([Figure 1-2, page 10](#)):

- The compiler reads a set of C-code source and header files or assembler source files for the targeted processor.
- The linker combines the compiled applications with selected libraries and produces the executable file in ELF format. The linker also reads a linker script, which is either the default linker script generated by the tools or one that you have provided.

Refer to [Chapter 9, “GNU Compiler Tools,”](#) [Chapter 11, “GNU Debugger,”](#) and [Appendix A, GNU Utilities](#) for more information about GNU compiler tools and utilities.

Xilinx Microprocessor Debugger

You can debug your program in software using an Instruction Set Simulator (ISS), or on a board that has a Xilinx FPGA loaded with your hardware bitstream. As shown in [Figure 1-2, page 10](#), the Xilinx Microprocessor Debugger (XMD) utility reads the application executable ELF file. For debugging on a physical FPGA, XMD communicates over the same download cable as used to configure the FPGA with a bitstream. Refer to [Chapter 10, “Xilinx Microprocessor Debugger \(XMD\),”](#) for more information.

GNU Debugger

The GNU Debugger (GDB) is a powerful yet flexible tool that provides a unified interface for debugging and verifying MicroBlaze and PowerPC processor systems during various development phases.

GDB uses Xilinx Microprocessor Debugger (XMD) as the underlying engine to communicate to processor targets.

Refer to [Chapter 11, “GNU Debugger,”](#) for more information.

Simulation Library Compiler (Compplib)

The Compplib utility compiles the EDK HDL-based simulation libraries using the tools provided by various simulator vendors. The Compplib operates in both the GUI and batch modes. In the GUI mode, it allows you to compile the Xilinx libraries (in your ISE installation) using the libraries available in EDK.

For more information about Compplib, see [Simulation Models in Chapter 7](#) and the ISE *Command Line Tools User Guide*. For instructions on compiling simulation libraries, refer to the *Xilinx Platform Studio Help*.

Bitstream Initializer (Bitinit)

The Bitinit tool initializes the on-chip block RAM memory connected to a processor with its software information. This utility reads hardware-only bitstream produced by the ISE tools (`system.bit`), and outputs a new bitstream (`download.bit`) which includes the embedded application executable (ELF) for each processor. The utility uses the BMM file, originally generated by Platgen and updated by the ISE tools with physical placement information on each block RAM in the FPGA. Internally, the Bitstream Initializer tool uses the Data2MEM utility to update the bitstream file.

See [Figure 1-2, page 10](#), to see how the Bitinit tool fits into the overall system architecture. Refer to [Chapter 12, “Bitstream Initializer \(BitInit\),”](#) for more information.

System ACE File Generator (GenACE)

XPS generates Xilinx System ACE configuration files from an FPGA bitstream, ELF, and data files. The generated ACE file can be used to configure the FPGA, initialize block RAM, 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`, that uses XMD commands to generate ACE files. ACE files can be generated for PowerPC processors and MicroBlaze processors with Microprocessor Debug Module (MDM) systems. For more information see [Chapter 13, “System ACE File Generator \(GenACE\).”](#)

Flash Memory Programmer

The Flash Memory Programming solution is designed to be generic and targets a wide variety of flash hardware and layouts. See [Chapter 14, “Flash Memory Programming.”](#)

Format Revision Tool and Version Management Wizard

The Format Revision Tool (`revup`) updates an existing EDK project to the current version. The `revup` tool performs format changes only; it does not update your design.

Backups of existing files such as the project file (XMP), the MHS and MSS files, are performed before the format changes are applied.

The Version Management wizard appears automatically when an older project is opened in a newer version of EDK (for example, when a project created in EDK 10.1 is opened in version 11.3).

The Version Management wizard is invoked after format revision has been performed. The wizard provides information about any changes in Xilinx Processor IPs used in the design. If a new compatible version of an IP is available, then the wizard also prompts you to update to the new version. For instructions on using the Version Management wizard, see [Chapter 15, “Version Management Tools \(revup\),”](#) and the *Xilinx Platform Studio Help*.

Microprocessor Peripheral Definition Translation tool (MPDX)

For board designers not familiar with the IP-XACT tool, a board description can be captured in an ASCII text file similar to the Microprocessor Peripheral Definition (MPD) format that captures a pcore description. This MPD file is known as the Board-MPD. It includes a translation tool, MPDX, which generates the IP-XACT files on disk for the BSB repository. [Chapter 16, Microprocessor Peripheral Definition Translation tool \(MPDX\)](#) describes how to use this tool.

Platform Specification Utility (PsfUtility)

This chapter describes the features and the usage of the Platform Specification Utility (PsfUtility) tool that enables automatic generation of Microprocessor Peripheral Definition (MPD) files. MPD files are required to create IP peripherals that are compliant with the Embedded Development Kit (EDK). The Create and Import Peripheral (CIP) wizard in the Xilinx® Platform Studio (XPS) interface supports features provided by the PsfUtility for MPD file creation (recommended).

Tool Options

Table 2-1 lists the PsfUtility Syntax options and their descriptions.

Table 2-1: PsfUtility Syntax Options

| Option | Command | Description |
|------------------------|--|---|
| Single IP MHS template | -deploy_core <corename> <coreversion> | Generate MHS Template that instantiates a single peripheral. Suboptions are: -lp <Library_Path>— Add one or more additional IP library search paths -o <outfile>— Specify output filename; default is stdout |
| Help | -h, -help | Displays the usage menu and then exits. |
| HDL file to MPD | -hdl2mpd <hdlfile> | Generate MPD from the VHDL/Ver/src/prj file. Suboptions are: -lang {ver vhd1} — Specify language -top <design> — Specify top-level entity or module name -bus {plbv46 axi4 axi4lite dcr lmb fsl m s ms mb ⁽¹⁾ [<busif_name>]}— Specify one or more bus interfaces for the peripheral -p2pbus <busif_name> <bus_std> {target initiator} — Specify one or more point-to-point connections for the peripheral -o <outfile> — Specify output filename; default is stdout |

1. Bus type mb (master that generates burst transactions) is valid for bus standard PLBv4.6 only.

Table 2-1: PsfUtility Syntax Options (Cont'd)

| Option | Command | Description |
|-----------------------------|---------------------------|--|
| PAO file to MPD | -pao2mpd <paofile> | Generate MPD from Peripheral Analyze Order (PAO) file. Suboptions are: -lang {ver vhdl} — Specify language -top <design> — Specify top-level entity or module name -bus {plbv46 axi4 axi4lite dcr lmb fslm s ms mb ⁽¹⁾ [<i><busif_name></i>]} — Specify one or more peripherals and optional interface name(s) -p2pbus <busif_name> <bus_std> {target initiator} — Specify one or more point-to-point connections of the peripheral -o <outfile> — Specify output filename; default is stdout |
| Display version information | -v | Displays the version number |

1. Bus type **mb** (master that generates burst transactions) is valid for bus standard PLBv4.6 only.

MPD Creation Process Overview

You can use the PsfUtility to create MPD specifications from the HDL specification of the core automatically. To create a peripheral and deliver it through EDK:

1. Code the IP in VHDL or Verilog using the required naming conventions for Bus, Clock, Reset, and Interrupt signals. These naming conventions are described in detail in [“Conventions for Defining HDL Peripherals” on page 25](#).

Note: Following these naming conventions enables the PsfUtility to create a correct and complete MPD file.

2. Create an XST (Xilinx Synthesis Technology) project file or a PAO file that lists the HDL sources required to implement the IP.
3. Invoke the PsfUtility by providing the XST project file or the PAO file with additional options.

For more information on invoking the PsfUtility with different options, see the following section, [Use Models for Automatic MPD Creation, page 23](#).

Use Models for Automatic MPD Creation

You can run the PsfUtility in a variety of ways, depending on the bus standard and bus interface types used with the peripheral and the number of bus interfaces a peripheral contains. Bus standards and types can be one of the following:

- AXI4 MASTER
- AXI4 SLAVE
- AXI4LITE MASTER
- AXI4LITE SLAVE
- AXI STREAMING (same as POINT TO POINT)
- DCR (design control register) SLAVE
- FSL (fast simplex link) SLAVE
- FSL MASTER
- LMB (local memory bus) SLAVE
- PLBV46 (processor local bus version 4.6) SLAVE
- PLBV46 MASTER
- POINT TO POINT BUS (special case)

Peripherals with a Single Bus Interface

Most processor peripherals have a single bus interface. This is the simplest model for the PsfUtility. For most such peripherals, complete MPD specifications can be obtained without any additional attributes added to the source code.

Signal Naming Conventions

The signal names must follow the conventions specified in [“Conventions for Defining HDL Peripherals” on page 25](#). When there is only one bus interface, no bus identifier need be specified for the bus signals.

Invoking the PsfUtility

The command line for invoking PsfUtility is as follows:

```
psfutil -hdl2mpd <hdlfile> -lang {vhdl|ver} -top <top_entity>  
-bus <busstd> <bustype> -o <mpdfile>
```

For example, to create an MPD specification for an PLB slave peripheral such as UART, the command is:

```
psfutil -hdl2mpd uart.vhd -lang vhdl -top uart -bus plb s -o uart.mpd
```

Alternatively, you can use a .prj file as input for invoking PsfUtility, as follows:

```
psfutil -hdl2mpd uart.prj -lang vhdl -top uart -bus plb s -o uart.mpd
```

Peripherals with Multiple Bus Interfaces

Some peripherals might have multiple associated bus interfaces. These interfaces can be exclusive bus interfaces, non-exclusive bus interfaces, or a combination of both. All bus interfaces on the peripheral that can be connected to the peripheral simultaneously are exclusive interfaces. For example, an OPB Slave bus interface and a DCR Slave bus interface are exclusive because they can be connected simultaneously.

On a peripheral containing exclusive bus interfaces: a port can be connected to only one of the exclusive bus interfaces.

Non-exclusive bus interfaces cannot be connected simultaneously.

Peripherals with non-exclusive bus interfaces have ports that can be connected to multiple non-exclusive interfaces. Non-exclusive interfaces have the same bus interface standard.

Non-Exclusive and Exclusive Bus Interfaces

Signal Naming Conventions

Signal names must adhere to the conventions specified in [“Conventions for Defining HDL Peripherals” on page 25](#).

- For non-exclusive bus interfaces, bus identifiers need not be specified.
- For exclusive bus interfaces, identifiers must be specified only when the peripheral has more than one bus interface of the same bus standard and type.

Invoking the PsfUtility With Buses Specified in the Command Line

You can specify buses on the command line when the bus signals do not have bus identifier prefixes. The command line for invoking the PsfUtility is as follows:

```
psfutil -hdl2mpd <hdlfile> -lang {vhdl|ver} -top <top_entity>
[-bus <busstd> <bustype>] -o <mpdfile>
```

Exclusive and Non-exclusive Bus Interface Command Line Examples

For an example of a non-exclusive bus interface, to create an MPD specification for a peripheral with a PLB slave interface and a PLB Master/Slave interface such as gemac, the command is:

```
psfutil -hdl2mpd gemac.prj -lang vhdl -top gemac -bus plb s -bus plb ms
-o gemac.mpd
```

For an example of an exclusive bus identifier, to create an MPD specification for a peripheral with a PLB slave interface and a DCR Slave interface, the command is:

```
psfutil -hdl2mpd mem.prj -lang vhdl -top mem -bus plb s -bus dcr s -o
mem.prj
```

Peripherals with Point-to-Point Connections

Some peripherals, such as multi-channel memory controllers, might have point-to-point connections (BUS_STD = XIL_MEMORY_CHANNEL, BUS_TYPE = TARGET).

Signal Naming Conventions

The signal names must follow conventions such that all signals belonging to the point-to-point connection start with the same bus interface name prefix, such as MCH0_*.

Invoking the PsfUtility with Point-to-Point Connections from Command Line

You can specify point-to-point connections in the command line using the bus interface name as a prefix to the bus signals.

The command line for invoking PsfUtil is:

```
psfutil -hdl2mpd <hdlfile> -lang {vhdl|ver} -top <top_entity>  
-p2pbus <busif_name> <bus_std> {target|initiator} -o <mpdfile>
```

For example, to create an MPD specification for a peripheral with an MCH0 connection, the command is:

```
psfutil -hdl2mpd mch_mem.prj -lang vhdl -top mch_mem -p2pbus MCH0  
XIL_MEMORY_CHANNEL TARGET -o mch_mem.mpd
```

DRC Checks in PsfUtility

To enable generation of correct and complete MPD files from HDL sources, the PsfUtility reports DRC errors. The DRC checks are listed in the following subsections in the order they are performed.

HDL Source Errors

The PsfUtility returns a failure status if errors are found in the HDL source files.

Bus Interface Checks

For every specified bus interface, the PsfUtility checks and reports any missing or repeated bus signals. It generates an MPD file when all bus interface checks are completed.

Conventions for Defining HDL Peripherals

The top-level HDL source file for an IP peripheral defines the interface for the design and has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters (generics) and default values
- Parameters defined in the MHS overwrite corresponding HDL source parameters

Individual peripheral documentation contains information on source file options.

For components that have more than one bus interface of the same type, naming conventions must be followed so the automation tools can group the bus interfaces.

Naming Conventions for Bus Interfaces

A bus interface is a grouping of related interface signals. For the automation tools to function properly, you must adhere to the signal naming conventions and parameters associated with a bus interface.

When the signal naming conventions are correctly specified, the following interface types are recognized automatically, and the MPD file contains the bus interface label shown in

Table 2-2.

Table 2-2: Recognized Bus Interfaces

| Description | Bus Label in MPD |
|--------------------------|------------------|
| Slave AXI interface | S_AXI |
| Master AXI interface | M_AXI |
| Slave DCR interface | SDCR |
| Master FSL interface | MFSL |
| Slave FSL interface | SFSL |
| Slave LMB interface | SLMB |
| Master PLBV4.6 interface | MPLB |
| Slave PLBV4.6 interface | SPLB |

Naming Conventions for VHDL Generics

For peripherals that contain more than one of the same bus interface, a *bus identifier* must be used. The bus identifier must be attached to all associated signals and generics.

Generic names must be VHDL-compliant. Additional conventions for IP peripherals are:

- The generic must start with C_.
- If more than one instance of a particular bus interface type is used on a peripheral, a bus identifier *<BI>* must be used in the signal.
- If a bus identifier is used for the signals associated with a port, the generics associated with that port can optionally use *<BI>*.
- If no *<BI>* string is used in the name, the generics associated with bus parameters are assumed to be global. For example, C_DOPB_DWIDTH has a bus identifier of D and is associated with the bus signals that also have a bus identifier of D. If only C_OPB_DWIDTH is present, it is associated with all OPB buses regardless of the bus identifier on the port signals.
Note: For the PLBV4.6 bus interface, the bus identifier *<BI>* is treated as the bus tag (bus interface name). For example, C_SPLB0_DWIDTH has a bus identifier (tag) SPLB0 and is associated with the bus signals that also have a bus identifier of SPLB0 as the prefix.
- For peripherals that have only a single bus interface (which is the case for most peripherals), the use of the bus identifier string in the signal and generic names is optional, and the bus identifier is typically not included.
- All generics that specify a base address must end with _BASEADDR, and all generics that specify a high address must end with _HIGHADDR. Further, to tie these addresses with buses, they must also follow the conventions for parameters, as listed above.
- For peripherals with more than one bus interface type, the parameters must have the bus standard type specified in the name. For example, parameters for an address on the PLB bus must be specified as C_PLB_BASEADDR and C_PLB_HIGHADDR.

The Platform Generator (Platgen) expands and populates certain reserved generics automatically. For correct operation, a bus tag must be associated with these parameters.

To have the PsfUtility infer this information automatically, all specified conventions must be followed for reserved generics as well. This can help prevent errors when your peripheral requires information on the platform that is generated.

Reserved Generic Names

Table 2-3 lists the reserved generic names.

Table 2-3: Automatically Expanded Reserved Generics

| Parameter | Description |
|-----------------------|-----------------------------|
| C_<BI>AXI_ADDR_WIDTH | AXI address width. |
| C_<BI>AXI_DATA_WIDTH | AXI data width. |
| C_<BI>AXI_ID_WIDTH | AXI master ID width. |
| C_<BI>AXI_NUM_MASTERS | Number of AXI masters. |
| C_<BI>AXI_NUM_SLAVES | Number of AXI slaves. |
| C_FAMILY | FPGA device family. |
| C_INSTANCE | Instance name of component. |
| C_<BI>DCR_AWIDTH | DCR address width. |
| C_<BI>DCR_DWIDTH | DCR data width. |
| C_<BI>DCR_NUM_SLAVES | Number of DCR slaves. |
| C_<BI>FSL_DWIDTH | FSL data width. |
| C_<BI>LMB_AWIDTH | LMB address width. |
| C_<BI>LMB_DWIDTH | LMB data width. |
| C_<BI>LMB_NUM_SLAVES | Number of LMB slaves. |

Reserved Parameters

Table 2-4 lists the parameters that Platgen populates automatically.

Table 2-4: Reserved Parameters

| Parameter | Description |
|------------------|--|
| C_BUS_CONFIG | Defines the bus configuration of the MicroBlaze processor. |
| C_FAMILY | Defines the FPGA device family. |
| C_INSTANCE | Defines the instance name of the component. |
| C_DCR_AWIDTH | Defines the DCR address width. |
| C_DCR_DWIDTH | Defines the DCR data width. |
| C_DCR_NUM_SLAVES | Defines the number of DCR slaves on the bus. |
| C_LMB_AWIDTH | Defines the LMB address width. |
| C_LMB_DWIDTH | Defines the LMB data width. |
| C_LMB_NUM_SLAVES | Defines the number of LMB slaves on the bus. |

Naming Conventions for Bus Interface Signals

This section provides naming conventions for bus interface signal names. The conventions are flexible to accommodate embedded processor systems that have more than one bus interface and more than one bus interface port per component. When peripherals with more than one bus interface port are included in a design, it is important to understand how to use a bus identifier. (As explained previously, a bus identifier must be used for peripherals that contain more than one of the same bus interface. The bus identifier must be attached to all associated signals and generics.)

The names must be HDL compliant. Additional conventions for IP peripherals are:

- The first character in the name must be alphabetic and uppercase.
- The fixed part of the identifier for each signal must appear exactly as shown in the applicable section below. Each section describes the required signal set for one bus interface type.
- If more than one instance of a particular bus interface type is used on a peripheral, the bus identifier *<BI>* must be included in the signal identifier. The bus identifier can be as simple as a single letter or as complex as a descriptive string with a trailing underscore (*_*) peripheral. *<BI>* must be included in the port signal identifiers in the following cases:
 - The peripheral has more than one slave AXI port
 - The peripheral has more than one master AXI port
 - The peripheral has more than one slave LMB port
 - The peripheral has more than one slave DCR port
 - The peripheral has more than one master DCR port
 - The peripheral has more than one slave FSL port
 - The peripheral has more than one master FSL port
 - The peripheral has more than one slave PLBV4.6 port
 - The peripheral has more than one master PLBV4.6 port
 - The peripheral has more than one port of any type and the choice of *<Mn>* or *<Sl n>* causes ambiguity in the signal names.

For peripherals that have only a single bus interface (which is the case for most peripherals), the use of the bus identifier string in the signal names is optional, and the bus identifier is typically not included.

Global Ports

The names for the global ports of a peripheral, such as clock and reset signals, are standardized. You can use any name for other global ports, such as the interrupt signal.

AXI Master - Clock and Reset ⁽¹⁾

```
M_AXI_ACLK  
M_AXI_ARESETN
```

AXI Slave - Clock and Reset ⁽¹⁾

```
S_AXI_ACLK  
S_AXI_ARESETN
```

LMB - Clock and Reset

```
LMB_Clk  
LMB_Rst
```

PLBV46 Master - Clock and Reset

```
MPLB_Clk  
MPLB_Rst
```

PLBV46 Slave - Clock and Reset

```
SPLB_Clk  
SPLB_Rst
```

1. ACLK and/or ARESETN can be bus interface specific or can be global across bus interfaces. Global ports, must be named ACLK and ARESETN.

Master AXI4 Ports

Master AXI4 ports must use the naming conventions shown in [Table 2-5](#):

Table 2-5: Master AXI4 Port Naming Conventions

| | |
|-------------------|---|
| <BI> | <p>A bus identifier.</p> <p>For peripherals with multiple AXI4 ports, the <BI> strings must be unique for each bus interface. Trailing underline characters such as '_' in the <BI> string are ignored.</p> |
|-------------------|---|

AXI4 Master Outputs

```

<BI>_awaddr      : out std_logic_vector(C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_awlen       : out std_logic_vector(7 downto 0);
<BI>_awsize      : out std_logic_vector(2 downto 0);
<BI>_awburst     : out std_logic_vector(1 downto 0);
<BI>_awprot      : out std_logic_vector(2 downto 0);
<BI>_awcache     : out std_logic_vector(3 downto 0);
<BI>_awvalid     : out std_logic;
<BI>_wdata       : out std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_wstrb       : out std_logic_vector((C_<BI>_DATA_WIDTH/9)-1downto
0);
<BI>_wlast       : out std_logic;
<BI>_wvalid      : out std_logic;
<BI>_bready      : out std_logic;
<BI>_araddr      : out std_logic_vector (C_<BI>_ADDR_WIDTH-1 downto 0)
;
<BI>_arlen       : out std_logic_vector(7 downto 0);
<BI>_arsize      : out std_logic_vector(2 downto 0);
<BI>_arburst     : out std_logic_vector(1 downto 0);
<BI>_arprot      : out std_logic_vector(2 downto 0);
<BI>_arcache     : out std_logic_vector(3 downto 0);
<BI>_arvalid     : out std_logic;
<BI>_rready      : out std_logic;

```

Examples:

```

m_axi_sg_awlen   : out std_logic_vector(7 downto 0);
m_axi_sg_awsz    : out std_logic_vector(2 downto 0);
m_axi_sg_awburst : out std_logic_vector(1 downto 0);

```

AXI4 Master Inputs

```

<BI>_awready     : in std_logic;
<BI>_wready      : in std_logic;
<BI>_bresp       : in std_logic_vector(1 downto 0);
<BI>_bvalid      : in std_logic;
<BI>_arready     : in std_logic;
<BI>_rdata       : in std_logic_vector (C_<BI>_DATA_WIDTH-1 downto 0) ;
<BI>_rresp       : in std_logic_vector(1 downto 0);
<BI>_rlast       : in std_logic;
<BI>_rvalid      : in std_logic;

```

Examples:

```

m_axi_sg_awready : in std_logic;
m_axi_sg_bresp   : in std_logic_vector(1 downto 0);
m_axi_sg_bvalid  : in std_logic;

```

Slave AXI4 Ports

Slave AXI4 ports must use the naming conventions shown in [Table 2-6](#):

Table 2-6: Slave AXI4 Port Naming Conventions

| | |
|------|---|
| <BI> | <p>A bus identifier.</p> <p>For peripherals with multiple AXI4 ports, the <BI> strings must be unique for each bus interface. Trailing underline characters such as '_' in the <BI> string are ignored.</p> |
|------|---|

AXI4 Slave Outputs

```

<BI>_awready      : out std_logic;
<BI>_wready       : out std_logic;
<BI>_bid          : out std_logic_vector(C_<BI>_ID_WIDTH-1 downto 0);
<BI>_bresp        : out std_logic_vector(1 downto 0);
<BI>_bvalid       : out std_logic;
<BI>_arready      : out std_logic;
<BI>_rid          : out std_logic_vector(C_<BI>_ID_WIDTH-1 downto 0);
<BI>_rdata        : out std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_rresp        : out std_logic_vector(1 downto 0);
<BI>_rlast        : out std_logic;
<BI>_rvalid       : out std_logic;

```

Examples:

```

s_axi_bid         : out std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
s_axi_bresp       : out std_logic_vector(1 downto 0);
s_axi_bvalid      : out std_logic;

```

AXI4 Slave Inputs

```

<BI>_awid         : in std_logic_vector(C_<BI>_ID_WIDTH-1 downto 0);
<BI>_awaddr       : in std_logic_vector(C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_awlen        : in std_logic_vector(7 downto 0);
<BI>_awsize       : in std_logic_vector(2 downto 0);
<BI>_awburst      : in std_logic_vector(1 downto 0);
<BI>_awlock       : in std_logic;
<BI>_awcache      : in std_logic_vector(3 downto 0);
<BI>_awprot       : in std_logic_vector(2 downto 0);
<BI>_awqos        : in std_logic_vector(3 downto 0);
<BI>_awvalid      : in std_logic;
<BI>_wdata        : in std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_wstrb        : in std_logic_vector(C_<BI>_DATA_WIDTH/8-1 downto 0);
<BI>_wlast        : in std_logic;
<BI>_wvalid       : in std_logic;
<BI>_bready       : in std_logic;
<BI>_arid         : in std_logic_vector(C_<BI>_ID_WIDTH-1 downto 0);
<BI>_araddr       : in std_logic_vector(C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_arlen        : in std_logic_vector(7 downto 0);
<BI>_arsize       : in std_logic_vector(2 downto 0);
<BI>_arburst      : in std_logic_vector(1 downto 0);
<BI>_arlock       : in std_logic;
<BI>_arcache      : in std_logic_vector(3 downto 0);
<BI>_arprot       : in std_logic_vector(2 downto 0);
<BI>_arqos        : in std_logic_vector(3 downto 0);
<BI>_arvalid      : in std_logic;
<BI>_rready       : in std_logic;

```

Examples:

```
s_axi_arburst      : in std_logic;
s_axi_arlock       : in std_logic;
s_axi_arcache      : in std_logic;
```

Master AXI4LITE Ports

Master AXI4LITE ports must use the naming conventions shown in [Table 2-7](#):

Table 2-7: Master AXI4LITE Port Naming Conventions

| | |
|------|---|
| <BI> | <p>A bus identifier.</p> <p>For peripherals with multiple AXI4 ports, the <BI> strings must be unique for each bus interface. Trailing underline characters such as '_' in the <BI> string are ignored.</p> |
|------|---|

AXI4LITE Master Outputs

```
<BI>_arvalid       : out std_logic;
<BI>_araddr        : out std_logic_vector(C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_arprot        : out std_logic_vector(2 downto 0);
<BI>_rready        : out std_logic;
<BI>_awvalid       : out std_logic;
<BI>_awaddr        : out std_logic_vector(C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_awprot        : out std_logic_vector(2 downto 0);
<BI>_wvalid        : out std_logic;
<BI>_wdata         : out std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_wstrb         : out std_logic_vector((C_<BI>_DATA_WIDTH/8)-1 downto
0);
<BI>_bready        : out std_logic;
```

Examples:

```
m_axi_lite_wdata   : out std_logic_vector(C_M_AXI_LITE_DATA_WIDTH-1
downto 0);
m_axi_lite_wstrb    : out std_logic_vector((C_M_AXI_LITE_DATA_WIDTH/8)-1
downto 0);
m_axi_lite_bready   : out std_logic;
```

AXI4LITE Master Inputs

```
<BI>_arready       : in std_logic;
<BI>_rvalid        : in std_logic;
<BI>_rdata         : in std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_rresp         : in std_logic_vector(1 downto 0);
<BI>_awready       : in std_logic;
<BI>_wready        : in std_logic;
<BI>_bvalid        : in std_logic;
<BI>_bresp         : in std_logic_vector(1 downto 0);
```

Examples:

```
m_axi_lite_rdata    : in std_logic_vector(C_M_AXI_LITE_DATA_WIDTH-1
downto 0);
m_axi_lite_rresp    : in std_logic_vector(1 downto 0);
m_axi_lite_awready   : in std_logic;
```

Slave AXI4LITE ports

Slave AXI4LITE ports must use the naming conventions shown in [Table 2-8](#):

Table 2-8: Slave AXI4LITE Port Naming Conventions

| | |
|------|---|
| <BI> | <p>A bus identifier.</p> <p>For peripherals with multiple AXI4 ports, the <BI> strings must be unique for each bus interface. Trailing underline characters such as '_' in the <BI> string are ignored.</p> |
|------|---|

AXI4LITE Slave Outputs

```

<BI>_AWREADY      : out std_logic;
<BI>_WREADY       : out std_logic;
<BI>_BRESP        : out std_logic_vector(1 downto 0);
<BI>_BVALID       : out std_logic;
<BI>_ARREADY      : out std_logic;
<BI>_RDATA        : out std_logic_vector(C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_RRESP        : out std_logic_vector(1 downto 0);
<BI>_RVALID       : out std_logic;

```

Examples:

```

<BI>_RDATA        : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
<BI>_RRESP        : out std_logic_vector(1 downto 0);
<BI>_RVALID       : out std_logic;

```

AXI4LITE Slave Inputs

```

<BI>_AWADDR       : in std_logic_vector (C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_AWVALID      : in std_logic;
<BI>_WDATA        : in std_logic_vector (C_<BI>_DATA_WIDTH-1 downto 0);
<BI>_WSTRB        : in std_logic_vector ((C_<BI>_DATA_WIDTH/8)-1 downto
0);
<BI>_WVALID       : in std_logic;
<BI>_BREADY       : in std_logic;
<BI>_ARADDR       : in std_logic_vector (C_<BI>_ADDR_WIDTH-1 downto 0);
<BI>_ARVALID      : in std_logic;
<BI>_RREADY       : in std_logic;

```

Examples:

```

S_AXI_ARADDR      : in std_logic_vector (C_S_AXI_ADDR_WIDTH-1 downto 0);
S_AXI_ARVALID     : in std_logic;
S_AXI_RREADY      : in std_logic;

```

Slave DCR Ports

Slave DCR ports must follow the naming conventions shown in [Table 2-9](#).

Note: If *<BI>* is present, *<Sln>* is optional.

Table 2-9: Slave DCR Port Naming Conventions

| | |
|---------------------|--|
| <i><Sln></i> | A meaningful name or acronym for the slave output. <i><Sln></i> must <i>not</i> contain the string DCR (upper, lower, or mixed case), so that slave outputs are not confused with bus outputs. |
| <i><nDCR></i> | A meaningful name or acronym for the slave input. The last three characters of <i><nDCR></i> must contain the string DCR (upper, lower, or mixed case). |
| <i><BI></i> | A bus identifier. Optional for peripherals with a single slave DCR port, and required for peripherals with multiple slave DCR ports. <i><BI></i> must <i>not</i> contain the string DCR (upper, lower, or mixed case). For peripherals with multiple slave DCR ports, the <i><BI></i> strings must be unique for each bus interface. |

DCR Slave Outputs

For interconnection to the DCR, all slaves must provide the following outputs:

```

<BI><Sln>_dcrDBus      : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
<BI><Sln>_dcrAck       : out std_logic;

```

Examples:

```

Uart_dcrAck           : out std_logic;
Intc_dcrAck           : out std_logic;
Memcon_dcrAck         : out std_logic;
Bus1_timer_dcrAck     : out std_logic;
Bus1_timer_dcrDBus    : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
Bus2_timer_dcrAck     : out std_logic;
Bus2_timer_dcrDBus    : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);

```

DCR Slave Inputs

For interconnection to the DCR, all slaves must provide the following inputs:

```

<BI><nDCR>_ABus        : in std_logic_vector(0 to C_<BI>DCR_AWIDTH-1);
<BI><nDCR>_DBus        : in std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
<BI><nDCR>_Read        : in std_logic;
<BI><nDCR>_Write       : in std_logic;

```

Examples:

```

DCR_DBus              : in std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
Bus1_DCR_DBus         : in std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);

```

Slave FSL Ports

Table 2-10 contains the required Slave FSL port naming conventions:

Table 2-10: Slave FSL Port Naming Conventions

| | |
|---|--|
| <code><nFSL></code> or <code><nFSL_S></code> | A meaningful name or acronym for the slave I/O. The last five characters of <code><nFSL_S></code> must contain the string <code>FSL_S</code> (upper, lower, or mixed case). |
| <code><BI></code> | A bus identifier. Optional for peripherals with a single slave FSL port and required for peripherals with multiple slave FSL ports. <code><BI></code> must <i>not</i> contain the string <code>FSL_S</code> (upper, lower, or mixed case). For peripherals with multiple slave FSL ports, the <code><BI></code> strings must be unique for each bus interface. |

FSL Slave Outputs

For interconnection to the FSL, slaves must provide the following outputs:

```

<BI><nFSL_S>_Data      : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
<BI><nFSL_S>_Control   : out std_logic;
<BI><nFSL_S>_Exists    : out std_logic;

```

Examples:

```

FSL_S_Control          : out std_logic;
Memcon_FSL_S_Control   : out std_logic;
Bus1_timer_FSL_S_Control : out std_logic;
Bus1_timer_FSL_S_Data   : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
Bus2_timer_FSL_S_Control : out std_logic;
Bus2_timer_FSL_S_Data   : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);

```

FSL Slave Inputs

For interconnection to the FSL, slaves must provide the following inputs:

```

<BI><nFSL>_Clk         : in std_logic;
<BI><nFSL>_Rst         : in std_logic;
<BI><nFSL_S>_Clk       : in std_logic;
<BI><nFSL_S>_Read      : in std_logic;

```

Examples:

```

FSL_S_Read             : in std_logic;
Bus1_FSL_S_Read        : in std_logic;

```

Master FSL Ports

Table 2-11 lists the required Master FSL ports naming conventions:

Table 2-11: Master FSL Port Naming Conventions

| | |
|---|--|
| <code><nFSL></code> or <code><nFSL_M></code> | A meaningful name or acronym for the master I/O. The last five characters of <code><nFSL_M></code> must contain the string <code>FSL_M</code> (upper, lower, or mixed case). |
| <code><BI></code> | A bus identifier. Optional for peripherals with a single master FSL port, and required for peripherals with multiple master FSL ports. <code><BI></code> must <i>not</i> contain the string <code>FSL_M</code> (upper, lower, or mixed case). For peripherals with multiple master FSL ports, the <code><BI></code> strings must be unique for each bus interface. |

FSL Master Outputs

For interconnection to the FSL, masters must provide the following outputs:

```
<BI><nFSL_M>_Full : out std_logic;
```

Examples:

```
FSL_M_Full          : out std_logic;
Memcon_FSL_M_Full  : out std_logic;
```

FSL Master Inputs

For interconnection to the FSL, masters must provide the following inputs:

```
<BI><nFSL>_Clk       : in std_logic;
<BI><nFSL>_Rst       : in std_logic;
<BI><nFSL_M>_Clk     : in std_logic;
<BI><nFSL_M>_Data    : in std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
<BI><nFSL_M>_Control : in std_logic;
<BI><nFSL_M>_Write   : in std_logic;
```

Examples:

```
FSL_M_Write          : in std_logic;
Bus1_FSL_M_Write     : in std_logic;
Bus1_timer_FSL_M_Control : out std_logic;
Bus1_timer_FSL_M_Data  : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
Bus2_timer_FSL_M_Control : out std_logic;
Bus2_timer_FSL_M_Data  : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
```

Slave LMB Ports

Slave LMB ports must follow the naming conventions shown in [Table 2-12](#):

Table 2-12: Slave LMB Port Naming Conventions

| | |
|---------------------------|--|
| <code><Sln></code> | A meaningful name or acronym for the slave output. <code><Sln></code> must <i>not</i> contain the string <code>LMB</code> (upper, lower, or mixed case), so that slave outputs will not be confused with bus outputs. |
| <code><nLMB></code> | A meaningful name or acronym for the slave input. The last three characters of <code><nLMB></code> must contain the string <code>LMB</code> (upper, lower, or mixed case). |
| <code><BI></code> | Optional for peripherals with a single slave LMB port and required for peripherals with multiple slave LMB ports. <code><BI></code> must <i>not</i> contain the string <code>LMB</code> (upper, lower, or mixed case). For peripherals with multiple slave LMB ports, the <code><BI></code> strings must be unique for each bus interface. |

Note: If `<BI>` is present, `<Sln>` is optional.

LMB Slave Outputs

For interconnection to the LMB, slaves must provide the following outputs:

```
<BI><Sln>_DBus      : out std_logic_vector(0 to C_<BI>LMB_DWIDTH-1);
<BI><Sln>_Ready     : out std_logic;
```

Examples:

```
D_Ready           : out std_logic;
I_Ready           : out std_logic;
```

LMB Slave Inputs

For interconnection to the LMB, slaves must provide the following inputs:

```
<BI><nLMB>_ABus      : in std_logic_vector(0 to C_<BI>LMB_AWIDTH-1);
<BI><nLMB>_AddrStrobe : in std_logic;
<BI><nLMB>_BE        : in std_logic_vector(0 to C_<BI>LMB_DWIDTH/
8-1);
<BI><nLMB>_Clk       : in std_logic;
<BI><nLMB>_ReadStrobe : in std_logic;
<BI><nLMB>_Rst       : in std_logic;
<BI><nLMB>_WriteDBus  : in std_logic_vector(0 to C_<BI>LMB_DWIDTH-1);
<BI><nLMB>_WriteStrobe : in std_logic;
```

Examples:

```
LMB_ABus          : in std_logic_vector(0 to C_LMB_AWIDTH-1);
DLMB_ABus         : in std_logic_vector(0 to C_DLMB_AWIDTH-1);
```

Master PLBV4.6 Ports

Master PLBV4.6 ports must use the naming conventions shown in [Table 2-13](#).

Table 2-13: Master PLBV4.6 Port Naming Conventions

| | |
|----------------------------|---|
| <code><M></code> | Prefix for the master output. |
| <code><PLB_M></code> | Prefix for the master input. |
| <code><BI></code> | <p>A bus identifier. Optional for peripherals with a single master PLBV46 port and required for peripherals with multiple master PLBV46 ports.</p> <p>For peripherals with multiple master PLBV46 ports, the <code><BI></code> strings must be unique for each bus interface. Trailing underline character '_' in the <code><BI></code> string are ignored.</p> |

PLB v4.6 Master Outputs

For interconnection to the PLB v4.6, masters must provide the following outputs:

```

<BI>M_abort          : out std_logic;
<BI>M_ABus           : out std_logic_vector(0 to C_<BI>/MPLB>_AWIDTH-1);
<BI>M_UABus          : out std_logic_vector(0 to C_<BI>/MPLB>_AWIDTH-1);
<BI>M_BE             : out std_logic_vector(0 to C_<BI>/MPLB>_DWIDTH/8-1);
<BI>M_busLock        : out std_logic;
<BI>M_lockErr        : out std_logic;
<BI>M_MSize          : out std_logic;
<BI>M_priority       : out std_logic_vector(0 to 1);
<BI>M_rdBurst        : out std_logic;
<BI>M_request        : out std_logic;
<BI>M_RNW            : out std_logic;
<BI>M_size           : out std_logic_vector(0 to 3);
<BI>M_TAttribute     : out std_logic_vector(0 to 15);
<BI>M_type           : out std_logic_vector(0 to 2);
<BI>M_wrBurst        : out std_logic;
<BI>M_wrDBus        : out std_logic_vector(0 to C_<BI>/MPLB>_DWIDTH-1);

```

Examples:

```

IPLBM_request        : out std_logic;
Bridge_M_request     : out std_logic;
O20b_M_request       : out std_logic;

```

PLB v4.6 Master Inputs

For interconnection to the PLBV4.6, masters must provide the following inputs:

```

<BI>MPLB_Clk         : in std_logic;
<BI>MPLB_Rst         : in std_logic;
<BI>PLB_MBusy        : in std_logic;
<BI>PLB_MRdErr       : in std_logic;
<BI>PLB_MWrErr       : in std_logic;
<BI>PLB_MIRQ         : in std_logic;
<BI>PLB_MWrBTerm     : in std_logic;
<BI>PLB_MWrDAck      : in std_logic;
<BI>PLB_MAddrAck     : in std_logic;
<BI>PLB_MRdBTerm     : in std_logic;
<BI>PLB_MRdDAck      : in std_logic;

```

```

<BI>PLB_MRdDBus      : in std_logic_vector(0 to C_<BI>/MPLB>_DWIDTH-1);
<BI>PLB_MRdWdAddr    : in std_logic_vector(0 to 3);
<BI>PLB_MRearbitrate : in std_logic;
<BI>PLB_MSSize       : in std_logic_vector(0 to 1);
<BI>PLB_MTimeout     : in std_logic;

```

Examples:

```

IPLB0_PLB_MBusy      : in std_logic;
Bus1_PLB_MBusy       : in std_logic;

```

Slave PLBV46 Ports

Table 2-14 shows the required naming conventions for Slave PLBV4.6 ports.

Table 2-14: Slave PLBV46 Port Naming Conventions

| | |
|-------|--|
| <Sl> | Prefix for the slave output |
| <PLB> | Prefix for the slave input |
| <BI> | <p>A bus identifier. Optional for peripherals with a single slave PLBV46 port and required for peripherals with multiple slave PLBV46 ports.</p> <p>For peripherals with multiple PLBV46 ports, the <BI> strings must be unique for each bus interface. Trailing underline character '_' in the <BI> string are ignored.</p> |

PLBV46 Slave Outputs

For interconnection to the PLBV4.6, slaves must provide the following outputs:

```

<BI>Sl_addrAck       : out std_logic;
<BI>Sl_MBusy         : out std_logic_vector(0 to C_<BI>/SPLB>_NUM_MASTERS-1);
<BI>Sl_MRdErr        : out std_logic_vector(0 to C_<BI>/SPLB>_NUM_MASTERS-1);
<BI>Sl_MWrErr        : out std_logic_vector(0 to C_<BI>/SPLB>_NUM_MASTERS-1);
<BI>Sl_MIRQ          : out std_logic;
<BI>Sl_rdBTerm       : out std_logic;
<BI>Sl_rdComp        : out std_logic;
<BI>Sl_rdDAck        : out std_logic;
<BI>Sl_rdDBus        : out std_logic_vector(0 to C_<BI>/SPLB>_DWIDTH-1);
<BI>Sl_rdWdAddr      : out std_logic_vector(0 to 3);
<BI>Sl_rearbitrate    : out std_logic;
<BI>Sl_SSize         : out std_logic(0 to 1);
<BI>Sl_wait          : out std_logic;
<BI>Sl_wrBTerm       : out std_logic;
<BI>Sl_wrComp        : out std_logic;
<BI>Sl_wrDAck        : out std_logic;

```

Examples:

```

Tmr_Sl_addrAck       : out std_logic;
Uart_Sl_addrAck      : out std_logic;
IntcSl_addrAck       : out std_logic;

```

PLBV4.6 Slave Inputs

For interconnection to the PLBV4.6, slaves must provide the following inputs:

```

<BI>SPLB_Clk           : in std_logic;
<BI>SPLB_Rst           : in std_logic;
<BI>PLB_ABus           : in std_logic_vector(0 to C_<BI>SPLB>_AWIDTH-1);
<BI>PLB_UABus          : in std_logic_vector(0 to C_<BI>SPLB>_AWIDTH-1);
<BI>PLB_BE             : in std_logic_vector(0 to C_<BI>PLB>_DWIDTH/8-1);
<BI>PLB_busLock        : in std_logic;
<BI>PLB_lockErr        : in std_logic;
<BI>PLB_masterID       : in std_logic_vector(0 to C_<BI>SPLB>_MID_WIDTH-1);
<BI>PLB_PValid         : in std_logic;
<BI>PLB_rdPendPri      : in std_logic_vector(0 to 1);
<BI>PLB_wrPendPri      : in std_logic_vector(0 to 1);
<BI>PLB_rdPendReq      : in std_logic;
<BI>PLB_wrPendReq      : in std_logic;
<BI>PLB_rdBurst        : in std_logic;
<BI>PLB_rdPrim         : in std_logic;
<BI>PLB_reqPri         : in std_logic_vector(0 to 1);
<BI>PLB_RNW            : in std_logic;
<BI>PLB_SValid         : in std_logic;
<BI>PLB_MSize          : in std_logic_vector(0 to 1);
<BI>PLB_size           : in std_logic_vector(0 to 3);
<BI>PLB_TAttribute     : in std_logic_vector(0 to 15);
<BI>PLB_type           : in std_logic_vector(0 to 2);
<BI>PLB_wrBurst        : in std_logic;
<BI>PLB_wrDBus         : in std_logic_vector(0 to C_<BI>SPLB>_DWIDTH-1);
<BI>PLB_wrPrim         : in std_logic;

```

Examples:

```

PLB_size               : in std_logic_vector(0 to 3);
IPLB_size              : in std_logic_vector(0 to 3);
DPORT0_PLB_size       : in std_logic_vector(0 to 3);

```

Psf2Edward Program

The `psf2Edward` is a command line program that converts a Xilinx® Embedded Development Kit (EDK) project into Edward, an internal XML format, for use in external programs such as the Software Development Kit (SDK).

The DTD for the Edward Format can be found in
`<EDK installation directory>/data/xml/DTD/Xilinx/Edward`.

Program Usage

You can use `Psf2Edward` to:

- Convert Platform Specification Format (PSF) project to XML format. To do this, use the following command:
`psf2Edward -inp <psf input source> -xml <xml output file> <options>`
- Synchronize an existing XML file with a PSF project, as follows:
`psf2Edward -inp <psf input source> -sync <XML file to sync> <options>`

Program Options

`Psf2Edward` has the following options:

| Option | Description |
|------------------------------------|--|
| <code>- dont_run_checkhwsys</code> | Do not run full set of system DRC checks. |
| <code>- edwver</code> | Set schema version of Edward to write. For example, 1.1 and 1.2. |
| <code>- exit_on_error</code> | Exit on first DRC error. By default, non-fatal errors are ignored. |
| <code>- inp</code> | Input PSF source. This can be either a Microprocessor Hardware Specification (MHS) file or a Xilinx Microprocessor Project (XMP) file. |
| <code>- p</code> | Part Name. This must be used if the PSF source is an MHS file. |
| <code>- sync</code> | Input sync XML file. This outputs to the same file. |
| <code>- xml</code> | Output XML file. |

Platform Generator (Platgen)

The Hardware Platform Generation tool (Platgen) customizes and generates the embedded processor system, in the form of hardware netlists files.

By default, Platgen synthesizes each processor IP core instance found in your embedded hardware design using the XST compiler. Platgen also generates the system-level HDL file that interconnects all the IP cores, to be synthesized later as part of the overall Xilinx® Integrated Software Environment (ISE®) implementation flow.

For more information, refer to the *Platform Specification Format Reference Manual*. A link to this document is provided in [Appendix E, Additional Resources](#).

Features

The features of Platgen includes the creation of:

- The programmable system on a chip in the form of hardware netlists (HDL and implementation netlist files.)
- A hardware platform using the Microprocessor Hardware Specification (MHS) file as input.
- Netlist files in various formats such as NGC and EDIF.
- Support files for downstream tools and top-level HDL wrappers to allow you to add other components to the automatically generated hardware platform.

After running Platgen, XPS spawns the Project Navigator interface for the FPGA implementation tools to complete the hardware implementation, allowing you full control over the implementation. At the end of the ISE flow, a bitstream is generated to configure the FPGA. This bitstream includes initialization information for block RAM memories on the FPGA chip. If your code or data must be placed on these memories at startup, the Data2MEM tool in the ISE toolset updates the bitstream with code and data information obtained from your executable files, which are generated at the end of the software application creation and verification flow.

Tool Requirements

Set up your system to use the Xilinx Integrated Development System. Verify that your system is properly configured. Consult the release notes and installation notes for more information.

Tool Usage

Run Platgen as follows:

```
platgen -p <partname> system.mhs
```

where:

platgen is the executable name.

-p is the option to specify a part.

<partname> is the partname.

system.mhs is the output file.

Supported Platgen Syntax Options

Table 4-1: Platgen Syntax Options

| Option | Command | Description |
|---|--|---|
| Help | -h, -help | Displays the usage menu and then exits without running the Platgen flow. |
| Filename | -f <filename> | Reads command line arguments and options from file. |
| Integration Style | -intstyle {ise default} | Indicates contextual information when invoking Xilinx applications within a flow or project environment. |
| Language | -lang {verilog vhdl} | Specifies the HDL language output. Default: vhdl |
| Log output | -log <logfile[.log]> | Specifies the log file. Default: platgen.log |
| Library path for user peripherals and driver repositories | -lp <Library_Path> | Adds <Library_Path> to the list of IP search directories. A library is a collection of repository areas. |
| Netlist Hierarchy | -netlist_hierarchy {as_optimized rebuilt} | as_optimized (default) – XST takes into account the Keep Hierarchy (KEEP_HIERARCHY) constraint, and generates the NGC netlist in the form in which it was optimized. In this mode, some hierarchical blocks can be flattened, and some can maintain hierarchy boundaries. rebuilt – XST writes a hierarchical NGC netlist, regardless of the Keep Hierarchy (KEEP_HIERARCHY) constraint. |
| Output directory | -od <output_dir> | Specifies the output directory path. Default: The current directory. |
| Part Name | -p <partname> | Uses the specified part type to implement the design. |
| Parallel Synthesis | -parallel {yes no} | Specifies the use of parallel synthesis. |
| Top-level module | -tm <top_module> | Specifies the top-level module name. |

Table 4-1: Platgen Syntax Options (Cont'd)

| Option | Command | Description |
|-----------|---------------------------|--|
| Top level | -toplevel {yes no} | Specifies if the input design represents a whole design or a level of hierarchy. Default: yes |
| Version | -v | Displays the version number of Platgen and then exits without running the Platgen flow. |

Load Path

Figure 4-1 shows the peripheral directory structure.

To specify additional directories, use one of the following options:

- Use the current directory (from which Platgen was launched.)
- Set the EDK tool **-lp** option.

Platgen uses a search priority mechanism to locate peripherals in the following order:

1. The `pcores` directory in the project directory.
2. The `<Library_Path>/<Library_Name>/pcores` as specified by the **-lp** option.
3. The `$XILINX_EDK/hw/<Library_Name>/pcores`.

Note: Directory path names are case-sensitive in Linux. Ensure that you use **pcore** and *not* Pcore.

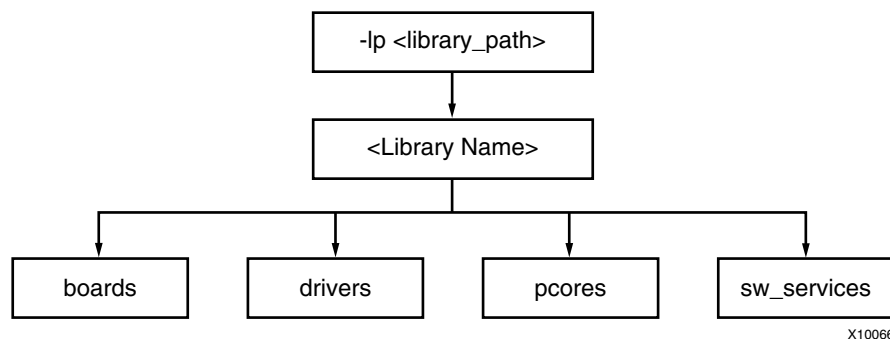


Figure 4-1: Peripheral Directory Structure

From the `pcores` directory, the root directory is the `<peripheral_name>`.

From the root directory, the underlying directory structure is:

```

data/
hdl/
netlist/
  
```

Output Files

Platgen produces directories and files from the project directory in the following underlying directory structure:

```

/hdl
/implementation
/synthesis
  
```

HDL Directory

The `/hdl` directory contains the following files:

- `system.{vhd|v}` is the HDL file of the embedded processor system as defined in the MHS, and the toplevel file for your project.
- `system_stub.{vhd|v}` is the toplevel template HDL file of the instantiation of the system. Use this file as a starting point for your own toplevel HDL file.
- `<inst>_wrapper.{vhd|v}` is the HDL wrapper file for the of individual IP components defined in the MHS.

Implementation Directory

The implementation directory contains implementation netlist files with the naming convention `<instance_name>_wrapper.ngc`.

Synthesis Directory

The synthesis directory contains the `system.[prj|scr]` synthesis project file.

BMM Flow

Platgen generates the `<system>.bmm` and the `<system>_stub.bmm` in the `<Project_Name>/implementation` directory.

- The `<system>.bmm` is used by the implementation tools when EDK is the top-level system.
- The `<system>_stub` is used by the implementation when EDK is a sub-module of the top-level system.

The EDK tools implementation tools flow using Data2MEM is as follows:

```
ngdbuild -bm <system>.bmm <system>.ngc
map
par
bitgen -bd <system>.elf
```

Bitgen outputs `<system>_bd.bmm`, which contains the physical location of block RAMs.

A block RAM Memory Map (BMM) file contains a syntactic description of how individual block RAMs constitute a contiguous logical data space.

The `<system>_bd.bmm` and `<system>.bit` files are input to the Data2MEM program. Data2MEM translates contiguous fragments of data into the proper initialization records for the Virtex[®] series block RAMs.

Synthesis Netlist Cache

An IP rebuild is triggered when one of the following changes occur:

- Instance name change
- Parameter value change
- Core version change
- Core is specified with the MPD `CORE_STATE=DEVELOPMENT` option
- Core license change

Command Line Mode

This chapter describes the XPS command line (no window) mode.

Invoking XPS Command Line Mode

To invoke the XPS command line or “no window” mode, type the command **xps -nw** at the LINUX Shell or Windows command prompt. XPS performs the specified operation, then presents a command prompt.

From the command line, you can:

- Generate the make files
- Run the complete project flow in batch mode
- Create an XMP project file
- Load a Xilinx Microprocessor Project (XMP) file created by the XPS GUI
- Read and reload project files
- Execute flow commands
- Archive your project

XPS batch provides the ability to query the EDK design database; Tcl commands are available for this purpose. In batch mode for XPS, you can specify a Tcl script by using the **-scr** option. You can also provide an existing XMP file as input to XPS.

Creating a New Empty Project

To create a new project with no components, use the command:

```
xload new <basename> .xmp
```

XPS creates a project with an empty Microprocessor Hardware Specification (MHS) file. All of the files have same base name as the XMP file. If XPS finds an existing project in the directory with same base name, then the XMP file is overwritten.

If you are using 6 series architecture or later, refer to answer record [44371](#), which provides a workaround.

Creating a New Project With an Existing MHS

To create a new project, use the command:

```
xload mhs <basename>.mhs
```

XPS reads in the MHS file and creates the new project. The project name is the same as the MHS base name. All of the files generated have the same name as MHS. After reading in the MHS file, XPS also assigns various default drivers to each of the peripheral instances, if a driver is known and available to XPS.

Opening an Existing Project

If you already have an XMP project file, you can load that file using the command:

```
xload xmp <basename>.xmp
```

XPS reads in the XMP file.

Saving Your Project Files

To save XMP and make files for your project, use the command:

```
save [xmp|make|proj]
```

Command **save proj** saves the XMP, MHS and make files. To save the make file, use the **save make** command explicitly.

Setting Project Options

Using the **xset** commands, you can set project options and other fields in XPS.

Using the **xget** commands, you can display the current value of those field; it also returns the result as a Tcl string result, which you can save into a Tcl variable. [Table 5-1](#) shows the options you can use with the **xget** and **xset** commands:

```
xset option <value>
xget option
```

Table 5-1: **xset** and **xget** Command Options

| Option Name | Description |
|--|--|
| arch | Set the target device architecture. |
| dev | Set the target part name. |
| enable_par_timing_error [0 1] | When set to 1, enables PAR timing error. |
| external_mem_sim [0 1] | When set to 1, enables external memory simulation. Default: 0. |
| gen_sim_tb [true false] | Generate test bench for simulation models. |
| hdl [vhdl verilog] | Set the HDL language to be used. |
| hier [top sub] | Set the design hierarchy. |

Table 5-1: **xset and xget Command Options (Cont'd)**

| Option Name | Description |
|--|--|
| intstyle [ise sysgen default] | Set the instantiation style. <ul style="list-style-type: none"> intstyle = ise: the project is instantiated in Project Navigator. intstyle = sysgen: the project is instantiated in System Generator. Default: default. |
| is_external_mem_present | xget command only. Returns 1 if AXI DDRx memory controller (Virtex [®] -6 and 7 series) is present; otherwise returns 0. |
| mix_lang_sim [true false] | Specify if the available simulator tool can support both VHDL and Verilog. |
| package | Set the package of the target device. |
| parallel_synthesis [yes no] | Set the parallel synthesis option. Default: no. |
| sdk_export_bmm_bit [0 1] | When set to 1, export BMM and BIT files for SDK. |
| sdk_export_dir <directory path> | Directory to which to export SDK files. Default: project_directory/sdk. |
| searchpath <directories> | Set the search path as a semicolon-separated list of directories. |
| speedgrade | Set the speedgrade of the target device. |
| sim_model [structural behavioral timing] | Set the current simulation mode. |
| simulator [mgm ies isim questa vcs none] | Set the simulator for which you want simulation scripts generated. mgm = Mentor Graphics ModelSim ies = Cadence Incisive Enterprise Simulator isim = ISE [®] Design Suite Simulator (ISim) questa = Mentor Graphics QuestaSim vcs = Verilog Compiler code Simulation none = No simulator specified. |
| sim_x_lib | Set the simulation library. For details, refer to Chapter 7, “Simulation Model Generator (Simgen).” |
| sim_elf imp_elf | Read Simulation/Implementation ELF files associated with the processors. Instead of the xset command, use add_elf , 'help_elf' . |
| ucf_file | Specify a path to the User Constraints File (UCF) to be used for implementation tools. |
| usercmd1 | Set the user command 1. |
| usercmd2 | Set the user command 2. |
| user_make_file <directory path> | Specify a path to the make file. This file should not be same as the make file generated by XPS. |

Executing Flow Commands

You can run various flow tools using the **run** command with appropriate options. XPS creates a make file for the project and runs that make file with the appropriate target. XPS generates the make file every time the **run** command is executed. Table 5-2 lists the valid options for the **run** command:

```
run <option>
```

Table 5-2: **run** Command Options

| Option Name | Description |
|---------------------|--|
| ace | Generate the System ACE™ technology file after the BIT file is updated with block RAM information. |
| bits | Run the Xilinx implementation tools flow and generate the bitstream. |
| bitsclean | Delete the BIT, NCD, and BMM files in the implementation directory. |
| clean | Delete all tool-generated files and directories. |
| download | Download the bitstream onto the FPGA. |
| hwclean | Delete the implementation directory. |
| init_bram | Update the bitstream with block RAM initialization information. |
| makeiplocal | Make an IP (and all its dependent libraries) local to the project. |
| netlist | Generate the netlist. |
| netlistclean | Delete the NGC or EDN netlist. |
| resync | Update any MHS file changes into the memory, and rewrites the XMP and makefile if required. |
| sim | Generate the simulation models and run the simulator. |
| simmodel | Generate the simulation models without running the simulator. |
| simclean | Delete the <i>simulation</i> directory. |

Reloading an MHS File

All EDK design files refer to MHS files. Any changes in MHS files have impact on other design files. If there are any changes in the MHS file after you loaded the design, use the following command to re-read MHS and XMP files:

```
run resync
```

Adding or Updating an ELF File

You can add or update the ELF files associated with a processor instance using this command:

```
xadd_elf <procinst> <elf type - sim|imp|both> <elf file>
```

| Option Name | Description |
|-----------------|--|
| procinst | The processor instance |
| elf type | The type of ELF file(s) to add or update. sim = simulation ELF file imp = implementation ELF file both = both simulation and implementation ELF files |
| <elf file> | The file name to add/update |

Deleting an ELF File

You can delete the ELF file associated with a processor instance using this command:

```
xdel_elf <procinst> <elf type - sim|imp|both>
```

| Option Name | Description |
|-----------------|---|
| procinst | The processor instance |
| elf type | The type of ELF file(s) to delete. sim = simulation ELF file imp = implementation ELF file both = both simulation and implementation ELF files |

Archiving Your Project Files

To archive a project, use the command:

```
xps_archiver
```

The `xps_archiver` tool compacts the files into a zip file. Refer to the *XPS Online Help* for the list of files that are archived.

Restrictions

XMP Changes

Xilinx® recommends that you *do not* edit the XMP file manually. The XPS **-batch** mode supports changing project options through commands. Any other changes must be done from XPS.

Bus Functional Model Simulation

This chapter describes Bus Functional Model (BFM) simulation within Xilinx® Embedded System Kit (EDK). You can run BFM simulation with ModelSim, QuestaSim, and ISim.

Bus Functional Simulation provides the ability to generate bus stimulus and thereby simplifies the verification of hardware components that attach to a bus. Bus Functional Simulation circumvents the drawbacks to the two typical validation methods, which are:

- Creating a test bench: This is time-consuming because it involves describing the connections and test vectors for all combinations of bus transactions.
- Creating a larger system with other known-good components that create or respond to bus transactions: This is time-consuming because it requires that you describe the established connections to the device under test, program the added components to generate the bus transactions to which the device will respond, and potentially respond to bus transactions that the device is generating. Such a system usually involves creating and compiling code, storing that code in memory for the components to read, and generating the correct bus transactions.

Bus Functional Model Use Cases

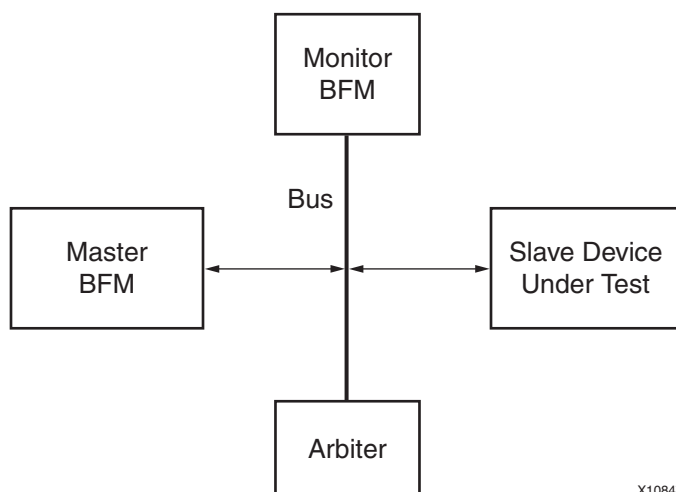
There are two main BFM use cases:

- IP verification
- Speed Up simulation

IP Verification

When verifying a single piece of IP that includes a bus interface you concern yourself with the internal details of the IP design and the bus interactions. It is inefficient to attach the IP to a large system only to verify that it is functioning properly.

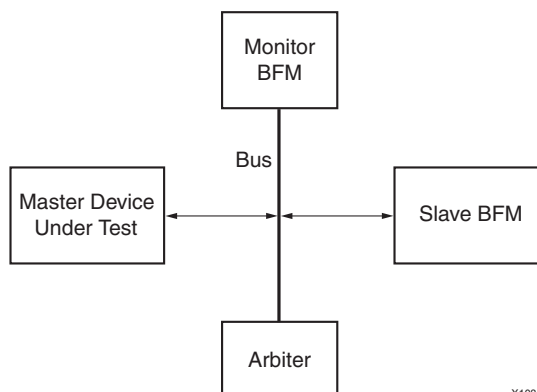
Figure 6-1 shows an example in which a master BFM generates bus transactions to which the device under test responds. The monitor BFM reports any errors regarding the bus compliance of the device under test.



X10847

Figure 6-1: Slave IP Verification Use Case

Figure 6-2 shows an example in which a slave BFM responds to bus transactions that the device under test generates. The monitor BFM reports any errors regarding the bus compliance of the device under test.



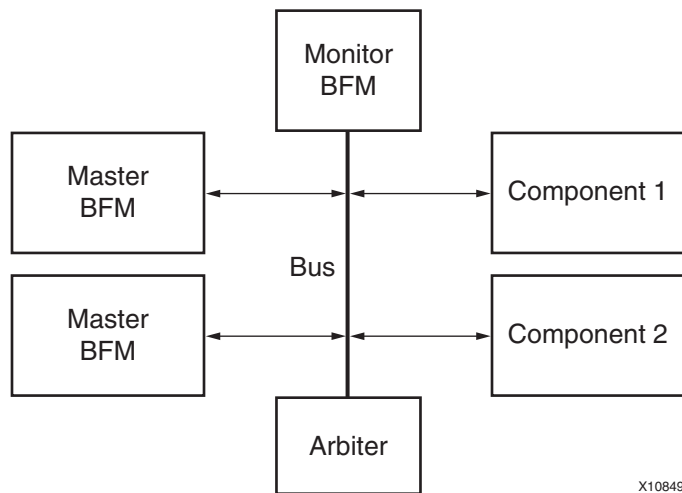
X10848

Figure 6-2: Master IP Verification Use Case

Speed-Up Simulation

When verifying a large system design, it can be time consuming to simulate the internal details of each IP component that attaches to a bus. There are certain complex pieces of IP that take a long time to simulate and could be replaced by a Bus Functional Model, especially when the internal details of the IP are not of interest. Additionally, some IP components are not easy to program to generate the desired bus transactions.

Figure 6-3 shows how two different IP components that are bus masters have been replaced by BFM master modules. These modules are simple to program and can provide a shorter simulation time because no internal details are modeled.



X10849

Figure 6-3: Speed-Up Simulation Use Case

Bus Functional Simulation Methods

There are two software packages that let you perform Bus Functional Simulation, and each applies its own methodology:

- PLBv4.6 BFM and the Xilinx EDK BFM package, which is based upon the IBM CoreConnect™ Toolkit.
- AXI BFM, which was provided by Cadence Design Systems, and is available on the Xilinx website.

IBM CoreConnect and AXI BFM are not included with EDK, but are required if you intend to perform bus functional simulation.

EDK includes a BFM package that provides a set of CoreConnect BFMs, the Bus Functional Compiler, and CoreConnect documents tailored for use within EDK. You can use this package after licensing the IBM CoreConnect Toolkit. The EDK BFM package lets you specify bus connections from a high-level description, such as an MHS file. By allowing the EDK tools to write the HDL files that describe the connections, the time and effort required to set up the test environment are reduced.

IBM CoreConnect Toolkit-Based PLB BFM

The IBM CoreConnect Toolkit is a collection of toolkits. Each toolkit includes a collection of HDL files that represents predefined systems, including a bus, bus masters, bus slaves, and bus monitors.

You can modify the predefined systems included in the toolkits manually to connect the hardware components you want to test. This is a labor-intensive process because you must describe all the connections to the bus and ensure there are no errors in setting up the test environment.

Refer to the CoreConnect Toolkit documentation for more information on how to verify your hardware module. You can download IBM CoreConnect™ Toolkit free of charge after you obtain a license for the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to documentation, Bus Functional Models, and the Compiler.

Xilinx provides a Web-based licensing mechanism that lets you obtain CoreConnect from the Xilinx website. To license CoreConnect, use an internet browser to access: http://www.xilinx.com/products/ipcenter/dr_pcentral_coreconnect.htm. After the request is approved (typically within 24 hours), you receive an E-mail granting you access to the protected web site from which to download the toolkit.

For further documentation on the CoreConnect Bus Architecture, refer to the IBM CoreConnect web site: http://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture

Note: There are differences between IBM CoreConnect and the Xilinx implementation of CoreConnect. These are described in the *Processor IP Reference Guide*, available in your \$XILINX_EDK/doc/usenglish directory. Refer to “Device Control Register Bus (DCR) V2.9” for differences in the DCR bus.

AXI BFM Package

AXI BFM lets you verify and simulate communication with AXI-based, in-development IP. Complete verification of these interfaces and protocol compliance is outside the scope of the AXI BFM solution; for compliance testing and complete system-level verification of AXI interfaces, use the Cadence AXI Universal Verification Component (UVC).

The AXI BFM solution is an optional product that is purchased separate from the ISE® Design Suite software. Licensing is handled through the standard Xilinx licensing scheme.

A license feature, XILINX_AXI_BFM, is required in addition to the standard ISE license features. A license is checked out at simulation run time. While the Xilinx ISE software does not need to be running while the AXI BFM solution is in use, the AXI BFM only operates on a computer that has the Xilinx software installed and licensed.

The BFM solution is encrypted using either the Verilog P1735 IEEE standard or a vendor-specific encryption scheme. To use the AXI BFM with Cadence Incisive Unified Simulator (IUS) and Incisive Enterprise Simulator (IES) products, an export control regulation license feature is required. Contact your Cadence sales office for more information.

See the *AXI BFM User Guide (UG783)* and the *AXI Bus Functional Model Data Sheet (DS824)* for more information.

PLB BFM Package

The use of the IBM CoreConnect PLB BFM components requires the acceptance of a license agreement. For this reason, the BFM components are not installed along with EDK. Xilinx provides a separate installer for these called the “Xilinx EDK BFM Package.”

To use the Xilinx EDK BFM Package, you must register and obtain a license to use the IBM CoreConnect Toolkit at:

http://www.xilinx.com/products/ipcenter/dr_pcentral_coreconnect.htm

After you register, you receive instructions and a link to download the CoreConnect Toolkit files. You can then install the files using the registration key provided.

After running the installer, you can verify that the files were installed by typing the following command:

```
xilbfc -check
```

A **Success!** message indicates you are ready to continue; otherwise, you will receive instructions on the error.

IBM Bus Functional Simulation Basics

Bus Functional Simulation usually involves the following components:

- A Bus Functional Model
- A Bus Functional Language
- A Bus Functional Compiler

Bus Functional Models (BFMs)

BFMs are hardware components that include and model a bus interface. There are different BFMs for different buses. For example, PLB BFM components are used to connect to their respective bus.

For each bus, there are different model types. For example the PLB bus has PLB Master, PLB Slave, and PLB Monitor BFM components. The same set of components and more could exist for other busses, or the functionality of BFM components could be combined into a single model.

Bus Functional Language (BFL)

The BFL describes the behavior of the BFM components. You can specify how to initiate or respond to bus transactions using commands in a BFL file.

Bus Functional Compiler (BFC)

The BFC translates a BFL file into the commands that actually program the selected Bus Functional Model.

Using the PLB BFM Package

After you download and install the PLB (IBM CoreConnect Toolkit) BFM Package, you can launch EDK. The following components are available:

- **PLB v4.6 Master BFM (plbv46_master_bfm)**
The PLB v4.6 master model contains logic to initiate bus transactions on the PLB v4.6 bus automatically. The model maintains an internal memory that can be initialized through the Bus Functional Language and may be dynamically checked during simulation or when all bus transactions have completed.
- **PLB v4.6 Slave BFM (plbv46_slave_bfm)**
The PLB v4.6 slave contains logic to respond to PLB v4.6 bus transactions based on an address decode operation. The model maintains an internal memory that can be initialized through the Bus Functional Language and can be dynamically checked during simulation or when all bus transactions have completed.
- **PLB v4.6 Monitor (plbv46_monitor_bfm)**
The PLB v4.6 monitor is a model that connects to the PLB v4.6 and continuously samples the bus signals. It checks for bus compliance or violations of the PLB v4.6 architectural specifications and reports warnings and errors.
- **BFM Synchronization Bus (bfm_synch)**
The BFM Synchronization Bus is a simple bus that connects BFM in a design and allows communication between them. The BFM Synchronization Bus is required whenever BFM devices are used.

These components can be instantiated in an MHS design file for the EDK tools to create the simulation HDL files.

Note: Xilinx has written an adaptation layer to connect the IBM CoreConnect Bus Functional Models to the Xilinx implementation of CoreConnect. Some of these BFM devices have different data and instruction bus widths.

PLB v4.6 BFM Component Instantiation

The following is an example MHS file that instantiates PLB v4.6 BFM components and the BFM synchronization bus.

```
# Parameters
PARAMETER VERSION = 2.1.0

# Ports
PORT sys_clk = sys_clk, DIR = I, SIGIS = CLK
PORT sys_reset = sys_reset, DIR = IN

# Components
BEGIN plb_v46
PARAMETER INSTANCE = myplb
PARAMETER HW_VER = 1.01.a
PARAMETER C_DCR_INTFCE = 0
PORT PLB_Clk = sys_clk
PORT SYS_Rst = sys_reset
END

BEGIN plb_bram_if_cntlr
PARAMETER INSTANCE = myplbbam_cntlr
```

```
PARAMETER HW_VER = 1.00.a
PARAMETER C_BASEADDR = 0xFFFF8000
PARAMETER C_HIGHADDR = 0xFFFFFFFF
BUS_INTERFACE PORTA = porta
BUS_INTERFACE SPLB = myplb
END

BEGIN bram_block
PARAMETER INSTANCE = bram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = porta
END

BEGIN plbv46_master_bfm
PARAMETER INSTANCE = my_master
PARAMETER HW_VER = 1.00.a
PARAMETER PLB_MASTER_ADDR_LO_0 = 0xFFFF0000
PARAMETER PLB_MASTER_ADDR_HI_0 = 0xFFFFFFFF
BUS_INTERFACE MPLB = myplb
PORT SYNCH_OUT = synch0
PORT SYNCH_IN = synch
END

BEGIN plbv46_slave_bfm
PARAMETER INSTANCE = my_slave
PARAMETER HW_VER = 1.00.a
PARAMETER PLB_SLAVE_ADDR_LO_0 = 0xFFFF0000
PARAMETER PLB_SLAVE_ADDR_HI_0 = 0xFFFF7FFF
BUS_INTERFACE SPLB = myplb
PORT SYNCH_OUT = synch1
PORT SYNCH_IN = synch
END

BEGIN plbv46_monitor_bfm
PARAMETER INSTANCE = my_monitor
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE MON_PLB = myplb
PORT SYNCH_OUT = synch2
PORT SYNCH_IN = synch
END

BEGIN bfm_synch
PARAMETER INSTANCE = my_synch
PARAMETER HW_VER = 1.00.a
PARAMETER C_NUM_SYNCH = 3
PORT FROM_SYNCH_OUT = synch0 & synch1 & synch2
PORT TO_SYNCH_IN = synch
END
```

BFM Synchronization Bus Usage

The BFM synchronization bus collects the `SYNCH_OUT` outputs of each BFM component in the design. The bus output is then connected to the `SYNCH_IN` of each BFM component. Figure 6-4 depicts an example for three BFMs, and the MHS example above shows its instantiation for PLB v4.6 BFMs.

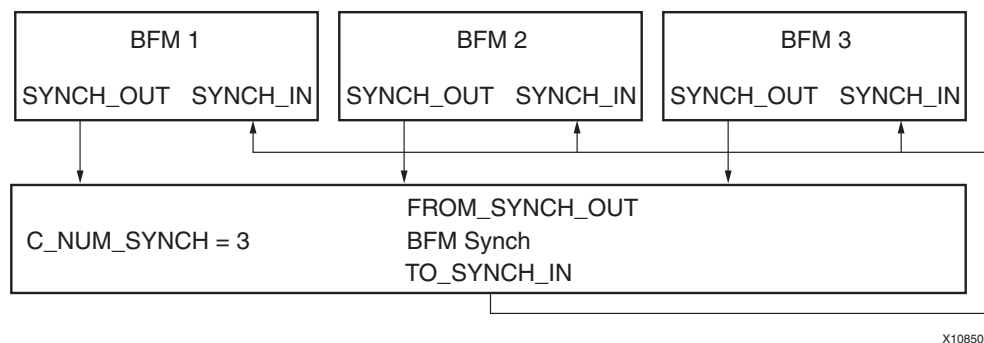


Figure 6-4: BFM Synchronization Bus Usage

PLB Bus Functional Language Usage

The following is a sample BFL file written for the [PLB v4.6 BFM Component Instantiation, page 58](#), which instantiate the PLB v4.6 BFM components.

```
-- FILE: sample.bfl
-- This test case initializes a PLB master

-- Initialize my_master
-- Note: The instance name for plb_master is duplicated in the
-- path due to the wrapper level inserted by the tools
set_device(path=/system/my_master/my_master/
master,device_type=plb_master)

-- Configure as 64-bit master
configure(msize=01)

-- Write and read 64-bit data using byte-enable architecture
mem_update(addr=ffff8000,data=00112233_44556677)
mem_update(addr=ffff8008,data=8899aabb_ccddeeff)
write      (addr=ffff8000,size=0000,be=11111111)
write      (addr=ffff8008,size=0000,be=11111111)
read       (addr=ffff8000,size=0000,be=11111111)
read       (addr=ffff8008,size=0000,be=11111111)

-- Write and read 32-bit data using byte-enable architecture
mem_update(addr=ffff8010,data=11111111_22222222)
write      (addr=ffff8010,size=0000,be=11110000)
write      (addr=ffff8014,size=0000,be=00001111)
read       (addr=ffff8010,size=0000,be=11110000)
read       (addr=ffff8014,size=0000,be=00001111)

-- Write and read 16-bit data using byte-enable architecture
mem_update(addr=ffff8020,data=33334444_55556666)
write      (addr=ffff8020,be=1100_0000)
write      (addr=ffff8022,be=0011_0000)
```

```

write      (addr=ffff8024,be=0000_1100)
write      (addr=ffff8026,be=0000_0011)
read       (addr=ffff8020,be=1100_0000)
read       (addr=ffff8022,be=0011_0000)
read       (addr=ffff8024,be=0000_1100)
read       (addr=ffff8026,be=0000_0011)

-- Write and read 8-bit data using byte-enable architecture
mem_update(addr=ffff8030,data=778899aa_bbccdde)
write      (addr=ffff8030,be=1000_0000)
write      (addr=ffff8031,be=0100_0000)
write      (addr=ffff8032,be=0010_0000)
write      (addr=ffff8033,be=0001_0000)
write      (addr=ffff8034,be=0000_1000)
write      (addr=ffff8035,be=0000_0100)
write      (addr=ffff8036,be=0000_0010)
write      (addr=ffff8037,be=0000_0001)
read       (addr=ffff8030,be=1000_0000)
read       (addr=ffff8031,be=0100_0000)
read       (addr=ffff8032,be=0010_0000)
read       (addr=ffff8033,be=0001_0000)
read       (addr=ffff8034,be=0000_1000)
read       (addr=ffff8035,be=0000_0100)
read       (addr=ffff8036,be=0000_0010)
read       (addr=ffff8037,be=0000_0001)

-- Write and read a 16-word line
mem_update(addr=ffff8080,data=01010101_01010101)
mem_update(addr=ffff8088,data=02020202_02020202)
mem_update(addr=ffff8090,data=03030303_03030303)
mem_update(addr=ffff8098,data=04040404_04040404)
mem_update(addr=ffff80a0,data=05050505_05050505)
mem_update(addr=ffff80a8,data=06060606_06060606)
mem_update(addr=ffff80b0,data=07070707_07070707)
mem_update(addr=ffff80b8,data=08080808_08080808)
write      (addr=ffff8080,size=0011,be=1111_1111)
read       (addr=ffff8080,size=0011,be=1111_1111)

```

More information about the PLB Bus Functional Language is in the `PlbToolkit.pdf` document in the `$XILINX_EDK/third_party/doc` directory.

Bus Functional Compiler Usage

The Bus Functional Compiler provided in the CoreConnect toolkit is a Perl script called `BFC`. The script uses a `bfcrc` configuration file that specifies to the script which simulator is used and the paths to the BFM. EDK includes a helper executable, called `xilbfc`, that enables this configuration.

To compile a BFL file, type the following at a command prompt:

- For ModelSim: **`xilbfc -s mti sample.bfl`**
- For ISim: **`xilbfc -s isim sample.bfl`**

This creates a script targeted for the selected simulator that initializes the BFM devices. In the case of ModelSim, it creates a file called `sample.do`. In the case of ISim, it creates a file called `sample.tcl`.

Simulation Examples

The following subsections provide examples for the supported simulators.

ModelSim/Questasim Example

The following is an example ModelSim or Questasim script called `run.do` that you can write to perform the BFM simulation steps:

```
do system.do
vsim system
do sample.do
do wave.do
force -freeze sim:/system/sys_clk 1 0, 0 {10 ns} -r 20 ns
force -freeze sim:/system/sys_reset 0, 1 {200 ns}
run 2 us
```

Note: If your design has an input reset that is active high, replace the reset line with:
`force -freeze sim:/system/sys_reset 1 , 0 {200 ns}`

At the ModelSim prompt, type:

```
do run.do
```

ISim Example

The following is an example ISim script called `run.tcl` that you can write to perform the BFM simulation steps:

```
isim force add /system/sys_clk 1 -time 0 ns, -value 0 -time 10 ns
-repeat 20 ns
isim force add /system/sys_reset 1 -time 100 ns -value 0 -time 200 ns
do sample.tcl
do wave.do
run 2 us
```

At the ISim prompt, type:

```
source run.tcl
```

Using the AXI BFM Package

After you launch XPS, you can find the following Xilinx AXI BFM components in the IP Catalog list under verification category:

- AXI3 Master BFM(`cdn_axi3_master_bfm_wrap`)
This is a master pcore on an AXI3 bus; it contains logic to initiate bus transactions on an AXI3 bus automatically.
- AXI3 Slave BFM(`cdn_axi3_slave_bfm_wrap`)
This is a slave pcore on an AXI3 bus; it contains logic to respond to AXI3 bus transactions based on an address decode operation.
- AXI4-Lite Master BFM(`cdn_axi4_lite_master_bfm_wrap`)
This is an AXI4-Lite Master pcore on an AXI4 bus; it contains logic to initiate Axi4-Lite bus transactions on an AXI4 bus automatically.
Note: An AXI4-Lite Master can initiate only a single read and a single write transaction, AXI4-Lite protocol does not allow burst transaction.
- AXI4-Lite Slave BFM(`cdn_axi4_lite_slave_bfm_wrap`)
This is AXI4-Lite Slave pcore on AXI4 bus and it contains logic to respond to AXI4-Lite bus transaction based on an address decode operation.
Note: An AXI4-Lite Slave can respond only to a single read and write transaction, AXI4 lite protocol does not allow burst transaction.
- AXI4 Master BFM(`cdn_axi4_master_bfm_wrap`)
This is an AXI4 Slave pcore on AXI4 bus and it contains logic to initiate AXI4 bus transactions on an AXI4 bus automatically. This master can initiate burst transaction.
- AXI4 Slave BFM(`cdn_axi4_slave_bfm_wrap`)
This is an AXI4 Slave pcore on AXI4 bus and it contains logic to respond to AXI4 bus transaction based on an address decode operation. This slave can respond to burst transactions.
- AXI4-Stream Master BFM(`cdn_axi4_streaming_master_bfm_wrap`)
This is an AXI4-Stream Master pcore on AXI4- Stream Point-to-Point (P2P) connection to initiate an AXI4-Stream transaction on a streaming connection.
- AXI4-Stream Slave BFM(`cdn_axi4_streaming_slave_bfm_wrap`)
This is AXI4-Stream Slave pcore on AXI4 Streaming Point-to-Point (P2P) connection to respond to an AXI4-Stream transaction on a streaming connection.

These components can be instantiated in an MHS design file for the Platform Studio tools to create the simulation HDL files.

AXI4 BFM Component Instantiation

Example MHS File

The following is an example MHS file that instantiates AXI4 BFM component

```
#
#####
# BFM simulation system
#
#####
PARAMETER VERSION = 2.1.0
PORT sys_reset = sys_reset, DIR = IN, SIGIS = RST
PORT sys_clk = sys_clk, DIR = IN, SIGIS = CLK, CLK_FREQ = 100000000

#AXI4 Lite Master BFM
BEGIN cdn_axi4_lite_master_bfm_wrap
  PARAMETER INSTANCE = bfm_processor
  PARAMETER HW_VER = 2.00.a
  BUS_INTERFACE M_AXI_LITE = axi4lite_bus
  PORT M_AXI_LITE_ACLK = sys_clk
END

#AXI Interconnect
BEGIN axi_interconnect
  PARAMETER INSTANCE = axi4lite_bus
  PARAMETER HW_VER = 1.03.a
  PARAMETER C_INTERCONNECT_CONNECTIVITY_MODE = 0
  PORT INTERCONNECT_ARESETN = sys_reset
  PORT INTERCONNECT_ACLK = sys_clk
END

#DUT
BEGIN test_slave_lite
  PARAMETER INSTANCE = test_slave_lite_inst
  PARAMETER HW_VER = 1.00.a
  PARAMETER C_BASEADDR = 0x30000000
  PARAMETER C_HIGHADDR = 0x3000ffff
  BUS_INTERFACE S_AXI = axi4lite_bus
  PORT S_AXI_ACLK = sys_clk
END
```

Figure 6-5 shows an example for AXI BFM, and the MHS example above shows its instantiations for an AXI4-Lite Master BFM:

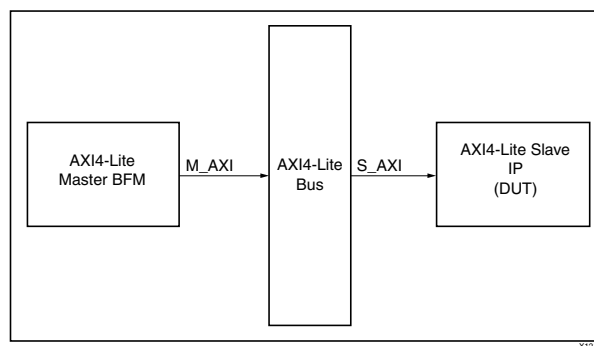


Figure 6-5: Master AXI BFM Bus Usage for DUT Verification

AXI4 BFM Usage

Figure 6-6 shows the usage of a Cadence AXI BFM.

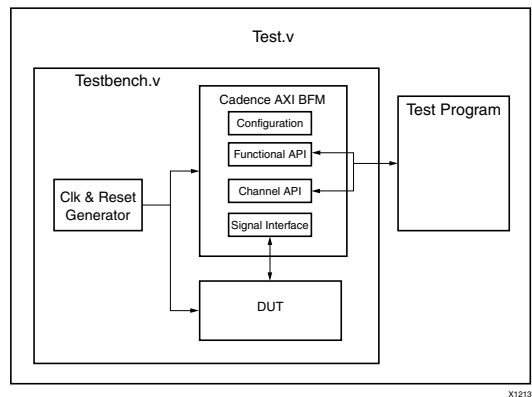


Figure 6-6: Cadence AXI BFM

Cadence AXI BFMs consist of three main layers; the:

- **Signal interface:** A standard Verilog input/output ports and associated signal.
- **Channel API:** A set of defined Verilog tasks that operate at the basic transaction level inherent in the AXI protocol, namely:
 - Read Address Channel
 - Write Address Channel
 - Read Data Channel
 - Write Data Channel
 - Write Response Channel
- **Function-level API:** This level has complete transaction level control, for example a complete AXI read burst process is encapsulated in a single Verilog task.

The configuration is implemented using Verilog parameters and/or BFM internal variables and is used to set, for example, the address bus width and the data bus width.

The testbench can contain multiple instances of Cadence AXI BFMs but for simplicity only one has been shown in Figure 6-6. The basic principle is that the Design Under Test (DUT) and AXI BFMs are instantiated in a testbench that also contains a clock and reset generator. Then you instantiate the testbench into your test module and create a test program using the Cadence BFM API layers. Such a test program does calls to these API tasks either sequentially or concurrently using a fork and join method.

Sample AXI BFM Program

The following is a sample BFM test program for the [Example MHS File, page 64](#), which instantiates the AXI4-Lite BFM component:

```

reg      rst_n;
reg      sys_clk;
integer  number_of_bytes;
integer  i;
integer  j;
reg[31 : 0] test_data;
reg[31 : 0] mtestAddr;
reg[2 : 0] mtestProtection;
reg[31 : 0] rd_data;
reg[1 : 0] response;

//-----
// Instantiate bfm_system
//-----

bfm_system
dut(.sys_reset(rst_n),.sys_clk(sys_clk));

initial begin
    //Wait for end of reset
    wait(rst_n == 0) @(posedge sys_clk);
    wait(rst_n == 1) @(posedge sys_clk);
    $display("-----");
    $display("Full Registers write");
    $display("-----");
    number_of_bytes = 4;

    //writing the all register
    for( i = 0 ; i <4; i = i+1) begin
        for(j=0 ; j < number_of_bytes ; j = j+1)
            test_data[j*8 +: 8] = j+(i*number_of_bytes);
        mtestAddr = `SLAVE_BASE_ADDR + i*number_of_bytes;
        $display("Writing to Slave Register addr=0x%h",mtestAddr, " data=0x%h",test_data);
        fork

dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_WRITE_ADDRESS(mtestAddr,mtestProtection);
dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_WRITE_DATA(4'b1111,test_data);
        dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.RECEIVE_WRITE_RESPONSE(response);
        join
        CHECK_RESPONSE_OKAY(response);
    end

    //reading the all register
    for( i = 0 ; i <4; i = i+1) begin
        for(j=0 ; j < number_of_bytes ; j = j+1)
            test_data[j*8 +: 8] = j+(i*number_of_bytes);
        mtestAddr = `SLAVE_BASE_ADDR + i*number_of_bytes;

dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_READ_ADDRESS(mtestAddr,mtestProtection);
        dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.RECEIVE_READ_DATA(rd_data,response);
        CHECK_RESPONSE_OKAY(response);
        $display("Reading from Slave Register addr=0x%h",mtestAddr, " data=0x%h",rd_data);
    COMPARE_DATA(test_data,rd_data);
        if (rd_data != test_data) begin
            $display("TESTBENCH FAILED! Data expected is not equal to actual.", "\n expected = 0x%h",expected,
                "\n actual    = 0x%h",actual);
            $stop;
        end
    end
    $display("-----");
    $display("Peripheral Verification Completed Successfully");
    $display("-----");
End

```

Running AXI BFM Simulations

The overall steps to run the an AXI BFM simulation are:

1. Compile the simulation HDL files.
2. Load the system into the simulator.
3. Initialize the Bus Functional Models.
4. (Optionally) create a waveform list or load a previously created waveform.
5. Provide the clock and reset stimulus to the system.
6. Run the simulation.

Specifically, in EDK, the steps are:

1. In XPS, open your test design.
2. Go to **Project > Project options > design flow**, and select the following:
 - a. **Verilog** for HDL language
 - b. **Generate test bench template**
 - c. **Behavioral** for simulation models
3. Select **Edit > Preference > Simulation** and select the simulator (ISim, ModelSim, QuestaSim)
4. Generate the simulation file (**Simulation > Generate Simulation HDL Files**). This also generates a `<projectname>_tb.v` file inside the `./simulation/behavioral/` directory.
5. Write you test program in the `./simulation/behavioral/<projectname>_tb.v` file.
6. Launch Simulation: Go to **Simulation > Launch HDL Simulator**
The simulator wizard opens.
7. Take the action according to simulator:
 - Select **ISim > run timelength**. For example: **run 1ms**
 - For ModelSim and Questa Sim:
 - Compile the HDL files
 - Simulate
 - Specify run timelength. For example: **run 1ms**

Simulation Model Generator (Simgen)

This chapter introduces the basics of Hardware Description Language (HDL) simulation and describes the Simulation Model Generator tool, Simgen, and usage of the Compplib utility tool.

Simgen Overview

Simgen creates and configures various VHDL and Verilog simulation models for a specified hardware. Simgen takes, as the input file, the Microprocessor Hardware Specification (MHS) file, which describes the instantiations and connections of hardware components.

Simgen is also capable of creating scripts for a specified vendor simulation tool. The scripts compile the generated simulation models.

The hardware component is defined by the MHS file. Refer to the “Microprocessor Hardware Specification (MHS)” chapter in the *Platform Specification Format Reference Manual* for more information. [Appendix E, Additional Resources](#), contains a link to the document web site. For more information about simulation basics and for discussions of behavioral, structural, and timing simulation methods, refer to the *Platform Studio Online Help*.

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.

The libraries described in the following sections are available for the Xilinx simulation flow. The HDL code must refer to the appropriate compiled library. The HDL simulator must map the logical library to the physical location of the compiled library.

Xilinx ISE Libraries

ISE provides the following libraries for simulation:

- [UNISIM Library](#)
- [SIMPRIM Library](#)
- [XilinxCoreLib Library](#)

UNISIM Library

The UNISIM Library 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.

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

SIMPRIM Library

The SIMPRIM Library is used for timing simulation. It includes all the Xilinx primitives library components used by Xilinx implementation tools. Timing simulation models generated by Simgen instantiate SIMPRIM library components.

XilinxCoreLib Library

The Xilinx CORE Generator™ software is a graphical Intellectual Property (IP) 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- or dual-port RAM.

The CORE Generator software 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.

Xilinx EDK Library

The EDK library is used for behavioral simulation. It contains all the EDK IP components, precompiled for ModelSim SE and PE, or Cadence Incisive Enterprise Simulator (IES). This library eliminates the need to recompile EDK components on a per-project basis, minimizing overall compile time. The EDK IP components library is provided for VHDL only and can be encrypted.

The Xilinx Compplib utility deploys compiled models for EDK IP components into a common location. Unencrypted EDK IP components can be compiled using Compplib. Precompiled libraries are provided for encrypted components.

Compplib Utility

Xilinx provides the `Compplib` utility to compile the HDL libraries for Xilinx-supported simulators. `Compplib` compiles the UNISIM, SIMPRIM, and XilinxCoreLib libraries for supported device architectures using the tools provided by the simulator vendor. You must have an installation of the Xilinx implementation tools to compile your HDL libraries using `Compplib`.

Run `Compplib` with the `-help` option if you need to display a brief description for the available options:

```
compplib -help
```

Each simulator uses certain environment variables that you must set before invoking `Compplib`. Consult your simulator documentation to ensure that the environment is properly set up to run your simulator.

Note: Use the `-p <simulator_path>` option to point to the directory where the ModelSim executable is, if it is not in your path.

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

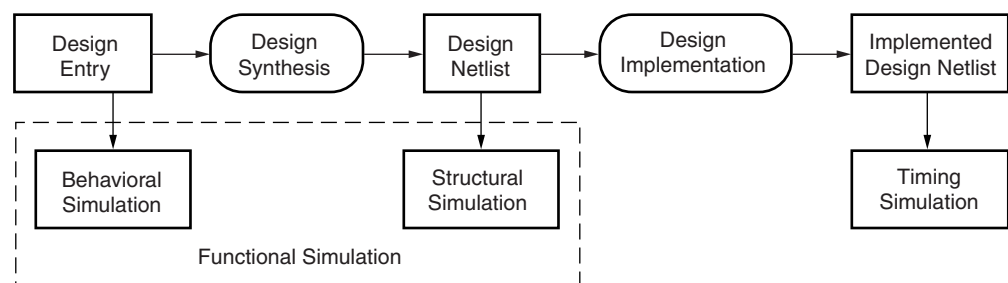
```
Compplib -s mti_se -arch all -l vhdl -w -dir .
```

This command compiles the necessary Xilinx libraries into the current working directory. Refer to the *Command Line Tools User Guide* for information `Compplib`. Refer to the “Simulating Your Design” chapter of the *Synthesis and Simulation Design Guide (UG626)* to learn more about compiling and using Xilinx ISE simulation libraries. A link to the documentation website is provided in [Appendix E, Additional Resources](#).

Simulation Models

This section describes how and when each of three FPGA simulation models are implemented, and provides instructions for creating simulation models using XPS batch mode. At specific points in the design process, Simgen creates an appropriate simulation model, as shown in the following figure.

[Figure 7-1](#) illustrates the FPGA design simulation stages:



UG111_01_111903

Figure 7-1: FPGA Design Simulation Stages

Behavioral Models

To create a behavioral simulation model as displayed in [Figure 7-2](#), Simgen requires an MHS file as input. Simgen creates a set of HDL files that model the functionality of the design. Optionally, Simgen can generate a compile script for a specified vendor simulator.

If specified, Simgen can generate HDL files with data to initialize block RAMs associated with any processor that exists in the design. This data is obtained from an existing Executable Linked Format (ELF) file. [Figure 7-2](#) illustrates the behavioral simulation model generation.

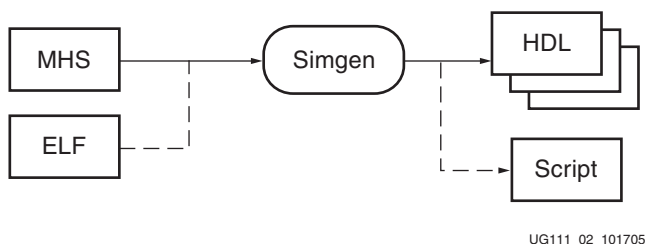


Figure 7-2: Behavioral Simulation Model Generation

Structural Models

To create a structural simulation model as shown in the following figure, Simgen requires an MHS file as input and associated synthesized netlist files. From these netlist files, Simgen creates a set of HDL files that structurally model the functionality of the design.

Optionally, Simgen can generate a compile script for a specified vendor simulator.

If specified, Simgen can generate HDL files with data to initialize block RAMs associated with any processor that exists in the design. This data is obtained from an existing ELF file. [Figure 7-3](#) illustrates the structural simulation model simulation generation.

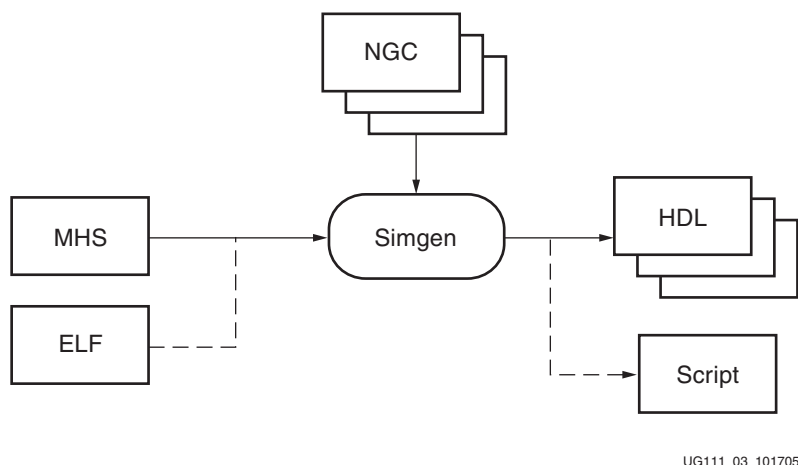
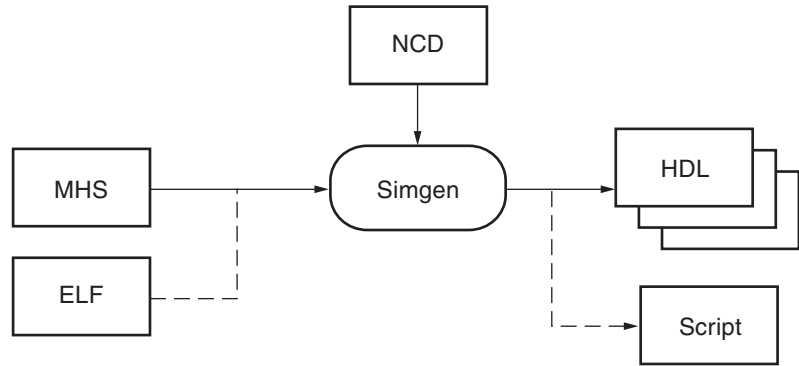


Figure 7-3: Structural Simulation Model Generation

Note: The EDK design flow is modular. Platgen generates a set of netlist files that Simgen uses to generate structural simulation models.

Timing Models

To create a timing simulation model, as shown in Figure 7-4, Simgen requires an MHS file as input and an associated implemented netlist file. From this netlist file, Simgen creates an HDL file that models the design and a Standard Data Format (SDF) file with the appropriate timing information. Optionally, Simgen can generate a compile script for a specified vendor simulator. If specified, Simgen can generate HDL files with data to initialize block RAMs associated with any processor that exists in the design. This data is obtained from an existing ELF file.



UG111_04_101705

Figure 7-4: Timing Simulation Model Generation

Single and Mixed Language Models

Simgen allows the use of mixed language components in behavioral files for simulation. By default, Simgen takes the native language in which each component is written. Individual components cannot be mixed language. To use this feature, a mixed language simulator is required.

Xilinx IP components are written in VHDL. If a mixed language simulator is not available, Simgen can generate single language models by translating the HDL files that are not in the HDL language. The resulting translated HDL files are structural files.

Structural and Timing simulation models are always single language.

Creating Simulation Models Using 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 [ mgm | questa | ies | isim | vcs | none ]
```

 Where:

```
mgm = Mentor Graphics ModelSim
questa = Mentor Graphics QuestaSim
ies = Cadence Incisive Enterprise Simulator (IES)
isim = ISE Simulator (ISIM)
vcs = Verilog Compiler code Simulator
```
 - 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 ]
```
 - d. Enable or disable external memory simulation using following command:

```
xset external_mem_sim [ 0 | 1 ]
```

 Optionally, before setting external memory simulation flag you might need to check if a DDRx memory controller (for Virtex-6) is present in the system. Use the following command:

```
xget is_external_mem_present
```

 Check for more detail in [External Memory Simulation](#), page 81.
3. To generate the simulation model, type:

```
XPS% run simmodel
```

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

```
XPS% run sim
```

Simgen Syntax

At the prompt, run Simgen with the MHS file and appropriate options as inputs.

```
simgen <system_name>.mhs [options]
```

Requirements

Verify that your system is properly configured to run the Xilinx ISE tools. Consult the release notes and installation notes that came with your software package for more information.

Options

Table 7-1 list the supported Simgen options:

Table 7-1: Simgen Syntax Options

| Option | Command | Description |
|--|--|---|
| EDK Library Directory | -E <i><edklib_dir></i> | Path to EDK simulation libraries directory. This switch is not required if the -X switch is used. The default location of the EDK libraries is inferred from the -X switch. |
| External Memory Simulation | -external_mem_sim [yes no] | yes - Instantiate external memory model into testbench. no - Generate testbench without external memory model instances. Default: no |
| External Memory Model Entity/ Module name | -external_mem_module <i><mem_module></i> | Simgen searches for an external memory model file with name <i><mem_module></i> .v/vhd in the /XPS project directory. Inside the model file, a module declaration must exist with the name <i><mem_module></i> . The default value of the <i><mem_module></i> is: <ul style="list-style-type: none"> • ddr3 if DDR3 is present in the system MHS. • ddr2 if DDR2 is present in the system MHS. By default, if the -external_mem_module flag is not present, Simgen searches the XPS project directory for ddr[3 2].v files. If Simgen finds a file, it uses that file during simulation; otherwise, it uses MIG-generated model files that reside in the <i><XPS project directory>/__xps/<DDRx_INST>/ddr[3 2]_module.v</i> |
| Help | -h, -help | Displays the usage menu and then quits. |
| Options File | -f <i><filename></i> | Reads command line arguments and options from file. |
| HDL Language | -lang [vhd1 verilog] | Specifies the HDL language: VHDL or Verilog. Default: vhd1 |
| Log Output | -log <i><logfile.log></i> | Specifies the log file. Default: simgen.log |
| Library Directories | -lp <i><Library_Path></i> | Allows you to specify library directory paths. This option can be specified more than once for multiple library directories. |
| Simulation Model Type | -m [beh str tim] | Allows you to select the type of simulation models to be used. The supported simulation model types are: <ul style="list-style-type: none"> • behavioral (beh) • structural (str) • timing (tim). Default: beh |
| Mixed Language | N/A | This option is obsolete. The tool assumes -mixed=yes . |
| Output Directory | -od <i><output_dir></i> | Specifies the project directory path. The default is the current directory. |

Table 7-1: Simgen Syntax Options (Cont'd)

| Option | Command | Description |
|--------------------------|--|--|
| Target Part or Family | -p <i><partname></i> | Lets you target a specific part or family. This option must be specified. The <i><partname></i> is available in the XPS Project Option tab. |
| Processor ELF Files | -pe <i><proc_instance></i> <i><elf_file></i> <i><elf_file></i> | Specifies a list of ELF files to be associated with the processor with instance name as defined in the MHS. |
| Simulator | -s [mgm questa ies isim vcs] | Generates compile script and helper scripts for vendor simulators. The options are: mgm = Mentor Graphics ModelSim questa = Mentor Graphics QuestaSim ies = Cadence Incisive Enterprise Simulator (IES) isim = ISE Simulator (ISIM) vcs = Verilog Compiler code Simulator |
| Source Directory | -sd <i><source_dir></i> | Specifies the source directory to search for netlist files. |
| Testbench Template | -tb | Creates a testbench template file. Use -ti and -tm to define the design under test name and the testbench name, respectively. |
| Top-Level Instance | -ti <i><top_instance></i> | When a testbench template is requested, use <i><top_instance></i> to define the instance name of the design under test. When design represents a sub-module, use <i><top_instance></i> for the top-level instance name. |
| Top-Level Module | -tm <i><top_module></i> | When a testbench template is requested, use <i>top_module</i> to define the name of the testbench. When the design represents a sub-module, use <i><top_module></i> for the top-level entity/module name. |
| Top-Level | -toplevel [yes no] | yes - Design represents a whole design. no - Design represents a level of hierarchy (sub-module). Default: yes |
| Version | -v | Displays the version then quits. |
| Xilinx Library Directory | -x <i><xlib_directory></i> | Path to the Xilinx simulation libraries (unisim, simprim, XilinxCoreLib) directory. This is the output directory of the Compxlib tool. Note: This option is not required for ISim because it does not use a pre-compiled simulation library. |

Output Files

Simgen produces all simulation files in the `/simulation` directory, which is located inside the `/output_directory`. In the `/simulation` directory, there is a subdirectory for each simulation model such as:

```
output_directory/simulation/<sim_model>
```

Where `<sim_model>` is one of: behavioral, structural, or timing

After a successful Simgen execution, the simulation directory contains the files listed in [Table 7-2](#). The generated file extension reflects the simulator used.

Table 7-2: Simgen Output Files

| Filename | Simulator File Extension | Description |
|--|---|---|
| <code>peripheral_wrapper.[vhd v]</code> | File extension common to all simulators | Modular simulation files for each component. Not applicable for timing models. |
| <code>system_name.[vhd v]</code> | File extension common to all simulators | Top-level HDL design file. |
| <code>system_name.sdf</code> | File extension common to all simulators | File with the appropriate block and net delays from the place and route process. (Used only for timing simulation.) |
| <code>xilinxsim.ini</code> | ISim Only | ISIM initialization file. |
| <code>system.[do prj tcl]</code> | .do - ModelSim/ QuestaSim .prj - ISim .tcl - IES | Project file specifying HDL source files and libraries to compile. |
| <code><system_name>_fuse.sh</code> | ISim Only | Helper script to create a simulation executable. |
| <code><system_name>_setup.[do sh tcl]</code> | .do - ModelSim/ QuestaSim .sh - ISim or IES .tcl - IES | Script to compile the HDL files and load the compiled simulation models in the simulator. |
| <code><system_name>_wave.[do sh tcl]</code> | .do - ModelSim/ QuestaSim .sh - ISim or IES .tcl - IES | Helper script to set up simulation waveform display. |
| <code><system_name>_list.[do sv tcl]</code> | .do - ModelSim/ QuestaSim .sv - ISim or IES .tcl - IES | Helper script to set up simulation tabular list display. |

Table 7-2: Simgen Output Files (Cont'd)

| Filename | Simulator File Extension | Description |
|--|---|---|
| <code><instance>_wave.[do sv tcl]</code> | .do - ModelSim/ QuestaSim .sv - ISim or IES .tcl - IES | Helper script to set up simulation waveform display for the specified instance. |
| <code><instance>_list.[do sv tcl]</code> | .do - ModelSim/ QuestaSim .sv - ISim or IES .tcl - IES | Helper script to set up simulation tabular list display for the specified instance. |

Memory Initialization

If a design contains banks of memory for a system, the corresponding memory simulation models can be initialized with data. You can specify a list of ELF files to associate to a given processor instance using the **-pe** switch.

The compiled executable files are generated with the appropriate GNU Compiler Collection (GCC) compiler or assembler, from corresponding C or assembly source code.

Note: Memory initialization of structural simulation models is only supported when the netlist file has hierarchy preserved.

For VHDL/Verilog simulation models, run Simgen with the **-pe** option to generate .mem files. These files contain a configuration for the system with all initialization values. For example:

```
simgen system.mhs -pe mblaze executable.elf -l vhd1 ...
simgen system.mhs -pe mblaze executable.elf -l verillog ...
```

The .mem files are used along with your system to initialize memory. The BRAM blocks connected to the mblaze processor contain the data in executable.elf.

Test Benches

Simgen can create test bench templates. When you use the **-tb** switch, Simgen creates a test bench that:

- Instantiates the top-level design
- Creates default stimulus for clock and reset signals

Clock stimulus is inferred from any global port which is tagged `SIGIS = CLK` in the MHS file. The frequency of the clock is given by the `CLK_FREQ` tag. The phase of the clock is given by the `CLK_PHASE` tag, which takes values from 0 to 360.

Reset stimulus is inferred for all global ports tagged `SIGIS = RST` in the MHS file.

- The polarity of the reset signal is given by the `RST_POLARITY` tag.
- The length of the reset is given by the `RST_LENGTH` tag.

For more information about the clock and reset tags, refer to the *Platform Studio Online Help*.

VHDL Test Bench Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

entity system_tb is
end system_tb;

architecture STRUCTURE of system_tb is

    constant sys_clk_PERIOD : time := 10 ns;
    constant sys_reset_LENGTH : time := 160 ns;
    constant sys_clk_PHASE : time 2.5 ns;

    component system is
        port (
            sys_clk : in std_logic;
            sys_reset : in std_logic;
            rx : in std_logic;
            tx : out std_logic;
            leds : inout std_logic_vector(0 to 3)
        );
    end component;

    -- Internal signals

    signal leds : std_logic_vector(0 to 3);
    signal rx : std_logic;
    signal sys_clk : std_logic;
    signal sys_reset : std_logic;
    signal tx : std_logic;

begin

    dut : system
        port map (
            sys_clk => sys_clk,
            sys_reset => sys_reset,
            rx => rx,
            tx => tx,
            leds => leds
        );

    -- Clock generator for sys_clk

    process
    begin
        sys_clk <= '0';
        wait for (sys_clk_PHASE);
        loop
            wait for (sys_clk_PERIOD/2);
            sys_clk <= not sys_clk;
        end loop;
    end process;
```

```

-- Reset Generator for sys_reset

process
begin
    sys_reset <= '0';
    wait for (sys_reset_LENGTH);
    sys_reset <= not sys_reset;
    wait;
end process;

-- START USER CODE (Do not remove this line)
-- User: Put your stimulus here. Code in this
--       section will not be overwritten.
-- END USER CODE (Do not remove this line)

end architecture STRUCTURE;

```

You can add your own VHDL code between the lines tagged BEGIN USER CODE and END USER CODE. The code between these lines is maintained if simulation files are created again. Any code outside these lines will be lost if a new test bench is created.

Verilog Test Bench Example

```

`timescale 1 ns/10 ps

`uselib lib=unisims_ver

module system_tb
(
);

    real sys_clk_PERIOD = 10;
    real sys_clk_PHASE = 2.5;
    real sys_reset_LENGTH = 160;

    // Internal signals

    reg [0:3] leds;
    reg rx;
    reg sys_clk;
    reg sys_reset;
    reg tx;

    system
    dut (
        .sys_clk ( sys_clk ),
        .sys_reset ( sys_reset ),
        .rx ( rx ),
        .tx ( tx ),
        .leds ( leds )
    );

```

```
// Clock generator for sys_clk

initial
begin
    sys_clk = 1'b0;
    #(sys_clk_PHASE); forever
    #(sys_clk_PERIOD/2)
    sys_clk = ~sys_clk;
end

// Reset Generator for sys_reset

initial
begin
    sys_reset = 1'b0;
    #sys_clk_LENGTH sys_reset = ~sys_reset;
end

// START USER CODE (Do not remove this line)
// User: Put your stimulus here. Code in this
//      section will be not be overwritten.
// END USER CODE (Do not remove this line)

endmodule
```

You can add your own Verilog code between the lines tagged BEGIN USER CODE and END USER CODE. The code between these lines is maintained if simulation files are created again. Any code outside these lines is lost if you create a new test bench.

External Memory Simulation

Simgen provides simulation models for external memory and has automated support to instantiate memory models in the simulation testbench and performs connection with the design under test.

To compile memory model into the user library, Simgen also generates simulator-specific compilation/elaboration commands into respective helper/setup scripts.

Restrictions

The restrictions on external memory simulation models are:

1. Supported for DDR2 and DDR3 (Virtex[®]-6, Kintex[®]-7 and Virtex-7 DDRx IPs).
2. Supported for behavioral simulation only.
3. When you select external memory simulation, the memory model is instantiated only if an AXI DDRx memory controller is present. The IP nomenclature is `axi_<device_family>_ddrx`. If the AXI memory controller is not present, Simgen continues to generate the testbench without the external memory model.
4. The following are not supported:
 - 72-bit wide memory interface (for example, with ECC)
 - RDIMM memory types

Enabling External Memory Simulation

To enable external memory simulation, pass the following flag to Simgen:

```
-external_mem_sim [yes|no]
```

By default, Simgen uses memory model files generated by MIG, as follows:

- `<XPS project directory>/__xps/<DDRx_Inst>/[ddr3|ddr2]_model.v`
- `<XPS project directory>/__xps/<DDRx_Inst>/[ddr3|ddr2]_model_parameters.vh`

It is not necessary to copy any other memory model files. You can specify other memory model files by placing the file in the `<XPS project directory>` with the name `[ddr2|ddr3].v/vhd`.

Note: Simgen is tested with MIG-generated memory model files; to use any customized simulation model files you must download and specify those files from vendor websites, provided that it is renamed accordingly. You must also have a copy of the `[ddr2|ddr3]_model_parameters.vh`.

Command Line Option

An additional Simgen command line option lets you specify the external simulation file:

```
-external_mem_module <external memory entity/module name>
```

Note: This option is not supported in the XPS GUI or XPS “no window” mode.

Recommended Steps

The following are optional, recommended steps when working in XPS:

1. For better tracking model initialization on the simulator waveforms, expose the `phy_init_done` or the `init_calib_complete` pin on the top-level.

To make the port external:

- a. Go to **XPS > System Assembly View**.
 - b. Select the Port tab and expand the `DDRx_SDRAM` IP instance.
 - c. Select the pin. For:
 - Virtex-6, the pin name is `phy_init_done`
 - Kintex-7, the pin name is `init_calib_complete`
 - d. In the net drop-down, select **Make External**.
2. To speed simulation, manually set the `*_init_call` to **FAST** in the MHS (this feature is not available in the GUI IP configuration).

Under the `axi_ddrx` IP instance Add **PARAMETER** `<parameter name> = FAST` to the MHS, where the parameter name for:

- Virtex-6 is `C_BYPASS_INIT_CAL`
 - Kintex-7 is `C_SIM_BYPASS_INIT_CAL`
3. For a Micron memory model higher than 1.62, the model files must have density defined before compilation. Add the ``define <density>` construct into the model file, where `<density>` is: `den1024Mb`, `den2048Mb`, or `den4096Mb`.

Note: This step is not required for version 1.60 micron memory models.
 4. Configure the **Memory part**, **Memory datawidth**, and other parameters using the **DDRx IP Configuration** MIG window.

Considerations and Use Restrictions

1. The memory model is always instantiated with x8 configuration; consequently, if the `DQ_WIDTH` parameter is 64 then eight instances are generated in the testbench and other parameters are modified accordingly.
2. External memory simulation with multiple instances of DDRx memory controller in XPS is not supported.
3. External memory simulation is supported only for Micron Memory Models used with XPS MIG.
4. During simulation the following initialization errors can be observed in the simulator console, but can be ignored for behavioral simulation:

```
system_axi_tb.inst_ddr_00.dqs_neg_timing_check: at time 5485244.0 ps  
ERROR: tDQSH violation on DQS bit 0 by 1039.0 ps.  
system_axi_tb.inst_ddr_03.cmd_task: at time 3438850.0 ps ERROR: Load  
Mode 0 Illegal value. Reserved address bits must be programmed to zero.
```
5. There is no support from Simgen to allow an application to load an ELF into external memory.
6. Other, non-supported external memory models must be manually instantiated and connected in the simulation testbench and initialized according to the model specifications.

Simulating Your Design

When simulating your design, there are some special considerations to keep in mind, such as the global reset and 3-state nets. Xilinx ISE tools provide detailed information on how to simulate your VHDL or Verilog design. Refer to the “Simulating Your Design” chapter in the *ISE Synthesis and Simulation Design Guide (UG626)*, for more information. [Appendix E, Additional Resources](#), contains a link to the document website.

Helper scripts generated at the test harness (or testbench) level are simulator setup scripts. When run, the setup script performs initialization functions and displays usage instructions for creating waveform and list (ModelSim only) windows using the waveform and list scripts. The top-level scripts invoke instance-specific scripts. You might need to edit hierarchical path names in the helper scripts for test harnesses not created by Simgen.

Commands in the scripts are commented or not commented to define the displayed set of signals. Editing the top-level waveform or list scripts lets you include or exclude signals for an instance; editing the instance-level scripts lets you include or exclude individual port signals. For timing simulations, only top-level ports are displayed.

Library Generator (Libgen)

This chapter describes the Library Generator utility, Libgen, which is required for the generation of libraries and drivers for embedded processors.

Overview

Libgen is the first Embedded Design Kit (EDK) tool that you run to configure libraries and device drivers. Libgen takes an XML hardware specification file and a Microprocessor Software Specification (MSS) file that you create. The hardware specification file defines the hardware system to Libgen and the MSS file describes the content and configuration of the software platform for a particular processor. Components are instantiated as blocks in the MSS file, and configuration is specified using parameters. Libgen reads the MSS file and generates the software components, configuring them as specified in the MSS.

For further description on generating the XML hardware specification file refer to the Software Development Kit (SDK) documentation in the *SDK Online Help*. For further description of the MSS file format, refer to the “Microprocessor Software Specification (MSS)” chapter in the *Platform Specification Format Reference Manual*. A link to the document is supplied in [Appendix E, Additional Resources](#).

Note: EDK includes a Format Revision tool to convert older MSS file formats to a new MSS format. Refer to [Chapter 15, “Version Management Tools \(revup\),”](#) for more information.

Tool Usage

To run Libgen, type the following:

```
libgen [options] <filename>.mss
```

Tool Options

[Table 8-1](#) list the supported Libgen command options.

Table 8-1: Libgen Syntax Options

| Option | Command | Description |
|------------|---------------------------|--|
| Help | -h, -help | Displays the usage menu and quits. |
| Version | -v | Displays the version number of Libgen and quits. |
| Log output | -log <logfile.log> | Specifies the log file. Default: libgen.log |

Table 8-1: Libgen Syntax Options (Cont'd)

| Option | Command | Description |
|---|---|--|
| Output directory | -od <output_dir> | Specifies the output directory output_dir. The default is the current directory. All output files and directories are generated in the output directory. The input file filename.mss is taken from the current working directory. This output directory is also called OUTPUT_DIR, and the directory from which Libgen is invoked is called YOUR_PROJECT for convenience in the documentation. |
| Source directory | -sd <source_dir> | Specifies the source directory <source_dir> for searching the input files. The default is the current working directory. |
| Path to a software component repository | -lp <Repository_Path> | Specifies a library containing repositories of user peripherals, drivers, OSs, and libraries. Libgen looks for the following: Drivers in the directory <Library_Path>/drivers/ Libraries in the directory <Library_Path>/sw_services/ OSs in the directory <Library_Path>/bsp/ |
| Hardware Specification File | -hw <hwspecfile.xml> | Specifies the hardware specification file (XML) to be used for Libgen. The hardware specification file describes the complete hardware system to LibGen. |
| Libraries | -lib | Use this option to copy libraries and drivers but not to compile them. |
| Processor instance-specific Libgen run | -pe <processor_instance_name> | This command runs Libgen for a specific processor instance. |

Load Paths

Figure 8-1 shows the directory structure of peripherals, drivers, libraries, and operating systems.

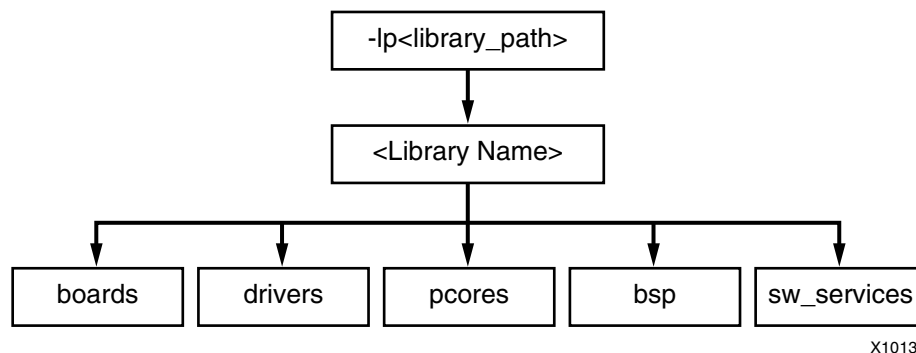


Figure 8-1: Directory Structure of Peripherals, Drivers, Libraries, and OSs

Default Repositories

By default, Libgen scans the following repositories for software components:

- `$XILINX_EDK/sw/lib/XilinxProcessorIPLib`
- `$XILINX_EDK/sw/lib`
- `$XILINX_EDK/sw/ThirdParty`

It also treats the directory from which Libgen is invoked as a repository and therefore scans for cores under sub-directories with standard directory names, such as `drivers`, `bsp`, and `sw_services`. Figure 8-2 shows the repository directory structure.

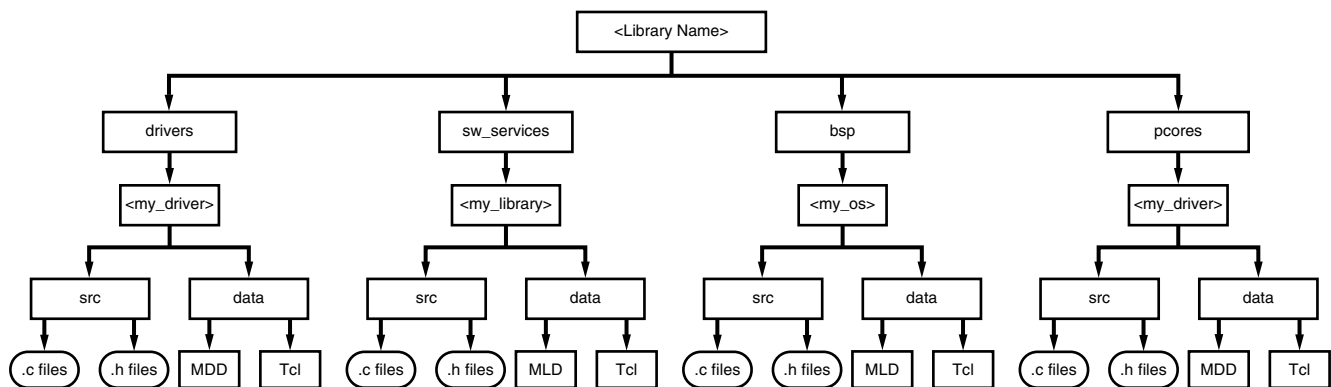


Figure 8-2: Repository Directory Structure

Search Priority Mechanism

Libgen uses a search priority mechanism to locate drivers and libraries, as follows:

1. Search the current working directory:
2. Search the repositories under the library path directory specified using the `-lp` option:
3. Search the default repositories as described in “Default Repositories.”

Output Files

Libgen generates directories and files in the `<YOUR_PROJECT>` directory. For every processor instance in the MSS file, Libgen generates a directory with the name of the processor instance. Within each processor instance directory, Libgen generates the following directories and files, which are described in the following subsections:

- [The include Directory](#)
- [lib Directory](#)
- [libsrc Directory](#)
- [code Directory](#)

The include Directory

The `include` directory contains C header files needed by drivers. The include file `xparameters.h` is also created through Libgen in this directory. This file defines base addresses of the peripherals in the system, `#defines` needed by drivers, OSs, libraries and user programs, as well as function prototypes.

- The Microprocessor Driver Definition (MDD) file for each driver specifies the definitions that must be customized for each peripheral that uses the driver. See the “Microprocessor Driver Definition (MDD)” chapter in the *Platform Specification Format Reference Manual (UG642)* for more information.
- The Microprocessor Library Definition (MLD) file for each OS and library specifies the definitions that you must customize. See the “Microprocessor Library Definition (MLD)” chapter in the *Platform Specification Format Reference Manual (UG642)* for more information.

A link to the *Platform Specification Format Reference Manual, (UG642)*, is in [Appendix E, Additional Resources](#).

lib Directory

The `lib` directory contains `libc.a`, `libm.a`, and `libxil.a` libraries. The `libxil` library contains driver functions that the particular processor can access. For more information about the libraries, refer to the introductory section of the *OS and Libraries Document Collection (UG643)*. A link to the document is in [Appendix E, Additional Resources](#).

libsrc Directory

The `libsrc` directory contains intermediate files and make files needed to compile the OSs, libraries, and drivers. The directory contains peripheral-specific driver files, BSP files for the OS, and library files that are copied from the EDK and your driver, OS, and library directories. Refer to the [Drivers, page 90](#), [OS Block, page 91](#), and [Libraries, page 91](#) sections of this chapter for more information.

code Directory

The `code` directory is a repository for EDK executables. Libgen creates an `xmdstub.elf` file (for MicroBlaze™ on-board debug) in this directory.

Note: Libgen removes these directories every time you run the tool. You must put your sources, executables, and any other files in an area that you create.

Generating Libraries and Drivers

Overview

This section provides an overview of generating libraries and drivers.

The hardware specification file and the MSS files define a system. For each processor in the system, Libgen finds the list of addressable peripherals. For each processor, a unique list of drivers and libraries are built. Libgen does the following for each processor:

- Builds the directory structure as defined in the [Output Files, page 87](#).
- Copies the necessary source files for the drivers, OSs, and libraries into the processor instance specific area: `OUTPUT_DIR/processor_instance_name/libsrc`.
- Calls the Design Rule Check (DRC) procedure, which is defined as an option in the MDD or MLD file, for each of the drivers, OSs, and libraries visible to the processor.
- Calls the `generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor. This generates the necessary configuration files for each of the drivers, OSs, and libraries in the `include` directory of the processor.
- Calls the `post_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.
- Runs `make` (with targets `include` and `libs`) for the OSs, drivers, and libraries specific to the processor. On the Linux platform, the `gmake` utility is used, while on NT platforms, `make` is used for compilation.
- Calls the `execs_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.

MDD, MLD, and Tcl

A driver or library has two associated data files:

- Data Definition File (MDD or MLD file): This file defines the configurable parameters for the driver, OS, or library.
- Data Generation File (Tcl): This file uses the parameters configured in the MSS file for a driver, OS, or library to generate data. Data generated includes but is not limited to generation of header files, C files, running DRCs for the driver, OS, or library, and generating executables.

The Tcl file includes procedures that Libgen calls at various stages of its execution. Various procedures in a Tcl file include:

- `DRC`
The name of DRC given in the MDD or MLD file
- `generate`
A Libgen-defined procedure that is called after files are copied
- `post_generate`
A Libgen-defined procedure that is called after `generate` has been called on all drivers, OSs, and libraries
- `execs_generate`
A Libgen-defined procedure that is called after the BSPs, libraries, and drivers have been generated

Note: The data generation (Tcl) file is not necessary for a driver, OS, or library.

For more information about the Tcl procedures and MDD/MLD related parameters, refer to the “Microprocessor Driver Definition (MDD)” and “Microprocessor Library Definition (MLD)” chapters in the *Platform Specification Format Reference Manual (UG642)*. A link to the document is supplied in [Appendix E, Additional Resources](#).

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, refer to the “Microprocessor Software Specification (MSS)” chapter in the *Platform Specification Format Reference Manual*. A link to the document is supplied in [Appendix E, Additional Resources](#).

Drivers

Most peripherals require software drivers. The EDK peripherals are shipped with associated drivers, libraries and BSPs. Refer to the *Device Driver Programmer Guide* for more information on driver functions. A link to the guide is supplied in [Appendix E, Additional Resources](#).

The MSS file includes a driver block for each peripheral instance. The block contains a reference to the driver by name (DRIVER_NAME parameter) and the driver version (DRIVER_VER). There is no default value for these parameters.

A driver has an associated MDD file and a Tcl file.

- The driver MDD file is the data definition file and specifies all configurable parameters for the drivers.
- Each MDD file has a corresponding Tcl file which generates data that includes generation of header files, generation of C files, running DRCs for the driver, and generating executables.

You can write your own drivers. These drivers must be in a specific directory under `<YOUR_PROJECT>/<driver_name>` or `<library_name>/drivers`, as shown in [Figure 8-1 on page 86](#).

- The DRIVER_NAME attribute allows you to specify any name for your drivers, which is also the name of the driver directory.
- The source files and make file for the driver must be in the `/src` subdirectory under the `/<driver_name>` directory.
- The make file must have the targets `/include` and `/libs`.
- Each driver must also contain an MDD file and a Tcl file in the `/data` subdirectory.

Open the existing EDK driver files to get an understanding of the required structure.

Refer to the “Microprocessor Driver Definition (MDD)” chapter in the *Platform Specification Format Reference Manual* for details on how to write an MDD and its corresponding Tcl file. A link to the document is supplied in [Appendix E, Additional Resources](#).

Libraries

The MSS file includes a library block for each library. The library block contains a reference to the library name (`LIBRARY_NAME` parameter) and the library version (`LIBRARY_VER`). There is no default value for these parameters. Each library is associated with a processor instance specified using the `PROCESSOR_INSTANCE` parameter. The library directory contains C source and header files and a make file for the library.

The MLD file for each library specifies all configurable options for the libraries and each MLD file has a corresponding Tcl file.

You can write your own libraries. These libraries must be in a specific directory under `<YOUR_PROJECT>/sw_services` or `<library_name>/sw_services` as shown in [Figure 8-1 on page 86](#).

- The `LIBRARY_NAME` attribute lets you specify any name for your libraries, which is also the name of the library directory.
- The source files and make file for the library must be in the `/src` subdirectory under the `<library_name>` directory.
- The make file must have the targets `/include` and `/libs`.
- Each library must also contain an MLD file and a Tcl file in the `/data` subdirectory.

Refer to the existing EDK libraries for more information about the structure of the libraries.

Refer to the “Microprocessor Library Definition (MLD)” chapter in the *Platform Specification Format Reference Manual (UG642)* for details on how to write an MLD and its corresponding Tcl file. A link to the document is in [Appendix E, Additional Resources](#).

OS Block

The MSS file includes an OS block for each processor instance. The OS block contains a reference to the OS name (`OS_NAME` parameter), and the OS version (`OS_VER`). There is no default value for these parameters. The `bsp` directory contains C source and header files and a make file for the OS.

The MLD file for each OS specifies all configurable options for the OS. Each MLD file has a corresponding Tcl file associated with it. Refer to the “Microprocessor Library Definition (MLD)” and “Microprocessor Software Specification (MSS)” chapters in the *Platform Specification Format Reference Manual, (UG642)*. A link to the document is in [Appendix E, Additional Resources](#).

You can write your own OSs. These OSs must be in a specific directory under `<YOUR_PROJECT>/bsp` or `<library_name>/bsp` as shown in [Figure 8-1 on page 86](#).

- The `OS_NAME` attribute allows you to specify any name for your OS, which is also the name of the OS directory.
- The source files and make file for the OS must be in the `src` subdirectory under the `<os_name>` directory.
- The make file should have the targets `/include` and `/libs`.
- Each OS must contain an MLD file and a Tcl file in the `/data` subdirectory.

Look at the existing EDK OSs to understand the structures. See the “Microprocessor Library Definition (MLD)” chapter in the *Platform Specification Format Reference Manual (UG642)* for details on how to write an MLD and its corresponding Tcl file. The *Device Driver Programmer Guide* is located in the `/doc/usenglish` folder of your EDK installation, file name: `xilinx_drivers_guide.pdf`.

GNU Compiler Tools

This chapter describes the GNU compiler tools.

Overview

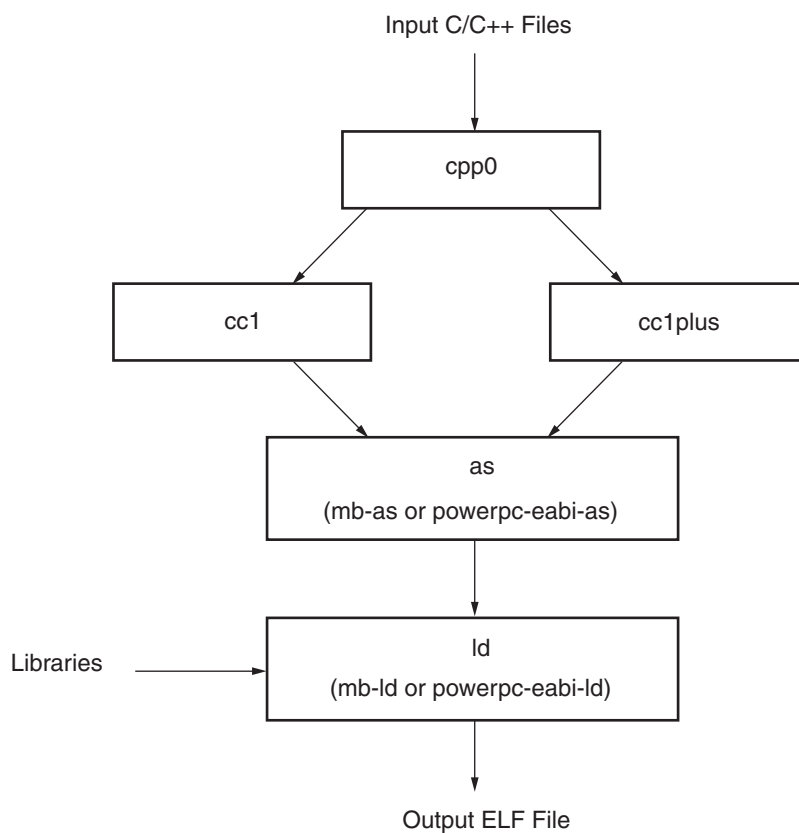
EDK includes the GNU compiler collection (GCC) for both the PowerPC® (405 and 440) processors and the MicroBlaze™ processor.

- The EDK GNU tools support both the C and C++ languages.
- The MicroBlaze GNU tools include `mb-gcc` and `mb-g++` compilers, `mb-as` assembler and `mb-ld` linker.
- The PowerPC processor tools include `powerpc-eabi-gcc` and `powerpc-eabi-g++` compilers, `powerpc-eabi-as` assembler and the `powerpc-eabi-ld` linker.
- The toolchains also include the C, Math, GCC, and C++ standard libraries.

The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The PowerPC and MicroBlaze compiler tools use the GNU binutils based on GNU version 2.16 of the sources. The concepts, options, usage, and exceptions to language and library support are described [Appendix A, “GNU Utilities.”](#)

Compiler Framework

This section discusses the common features of both the MicroBlaze and PowerPC processor compilers. [Figure 9-1](#) displays the GNU tool flow.



UG111_05_101905

Figure 9-1: GNU Tool Flow

The GNU compiler is named `mb-gcc` for MicroBlaze and `powerpc-eabi-gcc` for PowerPC. The GNU compiler is a wrapper that calls the following executables:

- Pre-processor (`cpp0`)
This is the first pass invoked by the compiler. The pre-processor replaces all macros with definitions as defined in the source and header files.
- Machine and language specific compiler
This compiler works on the pre-processed code, which is the output of the first stage. The language-specific compiler is one of the following:
 - C Compiler (`cc1`)
The compiler responsible for most of the optimizations done on the input C code and for generating assembly code.
 - C++ Compiler (`cc1plus`)
The compiler responsible for most of the optimizations done on the input C++ code and for generating assembly code.
- Assembler (`mb-as` for MicroBlaze and `powerpc-eabi-as` for PowerPC processors)
The assembly code has mnemonics in assembly language. The assembler converts these to machine language. The assembler also resolves some of the labels generated by the compiler. It creates an object file, which is passed on to the linker.
- Linker (`mb-ld` for MicroBlaze and `powerpc-eabi-ld` for PowerPC processors)
Links all the object files generated by the assembler. If libraries are provided on the command line, the linker resolves some of the undefined references in the code by linking in some of the functions from the assembler.

Executable options are described in:

- [Commonly Used Compiler Options: Quick Reference, page 99](#)
- [Linker Options, page 103](#)
- [MicroBlaze Compiler Options: Quick Reference, page 109](#)
- [MicroBlaze Linker Options, page 116](#)
- [PowerPC Compiler Options: Quick Reference, page 124.](#)

Note: From this point forward the references to GCC in this chapter refer to both the MicroBlaze compiler, `mb-gcc`, and the PowerPC processor compiler, `powerpc-eabi-gcc`, and references to G++ refer to both the MicroBlaze C++ compiler, `mb-g++`, and the PowerPC processor C++ compiler, `powerpc-eabi-g++`.

Common Compiler Usage and Options

Usage

To use the GNU compiler, type:

```
<Compiler_Name> options files...
```

where `<Compiler_Name>` is `powerpc-eabi-gcc` or `mb-gcc`. To compile C++ programs, you can use either the `powerpc-eabi-g++` or the `mb-g++` command.

Input Files

The compilers take one or more of the following files as input:

- C source files
- C++ source files
- Assembly files
- Object files
- Linker scripts

Note: These files are optional. If they are not specified, the default linker script embedded in the linker (**mb-ld** or **powerpc-eabi-ld**) is used.

The default extensions for each of these types are listed in [Table 9-1](#). In addition to the files mentioned above, the compiler implicitly refers to the libraries files `libc.a`, `libgcc.a`, `libm.a`, and `libxil.a`. The default location for these files is the EDK installation directory. When using the G++ compiler, the `libsupc++.a` and `libstdc++.a` files are also referenced. These are the C++ language support and C++ platform libraries, respectively.

Output Files

The compiler generates the following files as output:

- An ELF file. The default output file name is `a.exe` on Windows.
- Assembly file, if `-save-temps` or `-S` option is used.
- Object file, if `-save-temps` or `-c` option is used.
- Preprocessor output, `.i` or `.ii` file, if `-save-temps` option is used.

File Types and Extensions

The GNU compiler determines the type of your file from the file extension. [Table 9-1](#) lists the valid extensions and the corresponding file types. The GCC wrapper calls the appropriate lower level tools by recognizing these file types.

Table 9-1: File Extensions

| Extension | File type (Dialect) |
|-----------|---|
| .c | C file |
| .C | C++ file |
| .cxx | C++ file |
| .cpp | C++ file |
| .c++ | C++ file |
| .cc | C++ file |
| .S | Assembly file, but might have preprocessor directives |
| .s | Assembly file with no preprocessor directives |

Libraries

[Table 9-2](#) lists the libraries necessary for the `powerpc_eabi_gcc` and `mb_gcc` compilers.

Table 9-2: Libraries Used by the Compilers

| Library | Particular |
|--------------------------|---|
| <code>libxil.a</code> | Contain drivers, software services (such as XilMFS) and initialization files developed for the EDK tools. |
| <code>libc.a</code> | Standard C libraries, including functions like <code>strcmp</code> and <code>strlen</code> . |
| <code>libgcc.a</code> | GCC low-level library containing emulation routines for floating point and 64-bit arithmetic. |
| <code>libm.a</code> | Math Library, containing functions like <code>cos</code> and <code>sine</code> . |
| <code>libsupc++.a</code> | C++ support library with routines for exception handling, RTTI, and others. |
| <code>libstdc++.a</code> | C++ standard platform library. Contains standard language classes, such as those for stream I/O, file I/O, string manipulation, and others. |

Libraries are linked in automatically by both compilers. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. The `libxil.a` is modified by the Library Generator tool, Libgen, to add driver and library routines.

Language Dialect

The GCC compiler recognizes both C and C++ dialects and generates code accordingly. By GCC convention, it is possible to use either the GCC or the G++ compilers equivalently on a source file. The compiler that you use and the extension of your source file determines the dialect used on the input and output files.

When using the GCC compiler, the dialect of a program is always determined by the file extension, as listed in [Table 9-1, page 96](#). If a file extension shows that it is a C++ source file, the language is set to C++. This means that if you have compile C code contained in a `CC` file, even if you use the GCC compiler, it automatically mangles function names.

The primary difference between GCC and G++ is that G++ automatically sets the default language dialect to C++ (irrespective of the file extension), and if linking, automatically pulls in the C++ support libraries. This means that even if you compile C code in a `.c` file with the G++ compiler, it will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is said to be overloaded if the same function can perform different actions based on the arguments passed in, and can return different return values. To support this, C++ compilers encode the type of the function to be invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful about name mangling if you decide to follow a mixed compilation mode, with some source files containing C code and some others containing C++ code (or using GCC for compiling certain files and G++ for compiling others). To prevent name mangling of a C symbol, you can use the following construct in the symbol declaration.

```
#ifdef __cplusplus
extern "C" {
#endif

int foo();
int morefoo();

#ifdef __cplusplus
}
#endif
```

Make these declarations available in a header file and use them in all source files. This causes the compiler to use the C dialect when compiling definitions or references to these symbols.

Note: All EDK drivers and libraries follow these conventions in all the header files they provide. You must include the necessary headers, as documented in each driver and library, when you compile with G++. This ensures that the compiler recognizes library symbols as belonging to "C" type.

When compiling with either variant of the compiler, to force a file to a particular dialect, use the `-x lang` switch. Refer to the GCC manual on the GNU website for more information on this switch. A link to the document is provided in the [Appendix E, "Additional Resources."](#)

- When using the GCC compiler, `libstdc++.a` and `libsupc++.a` are *not* automatically linked in.
- When compiling C++ programs, use the G++ variant of the compiler to make sure all the required support libraries are linked in automatically.
- Adding `-lstdc++` and `-lsupc++` to the GCC command are also possible options.

For more information about how to invoke the compiler for different languages, refer to the GNU online documentation. A link to the documentation is provided in the [Appendix E, "Additional Resources."](#)

Commonly Used Compiler Options: Quick Reference

The summary below lists compiler options that are common to the compilers for MicroBlaze and PowerPC processors.

Note: The compiler options are case sensitive.

To jump to a detailed description for a given option, click its name.

General Options

[-E](#) [-Wp,*option*](#)
[-S](#) [-Wa,*option*](#)
[-c](#) [-Wl,*option*](#)
[-g](#) [--help](#)
[-gstabs](#) [-B directory](#)
[-On](#) [-L directory](#)
[-v](#) [-I directory](#)
[-save-temps](#) [-l library](#)
[-o filename](#)

Library Search Options

[-l libraryname](#)
[-L Lib Directory](#)

Header File Search Option

[-I Directory Name](#)

Linker Options

[-defsym _STACK_SIZE=value](#)
[-defsym _HEAP_SIZE=value](#)

General Options

-E

Preprocess only; do not compile, assemble and link. The preprocessed output displays on the standard out device.

-S

Compile only; do not assemble and link. Generates a .s file.

-c

Compile and Assemble only; do not link. Generates a .o file.

-g

This option adds DWARF2-based debugging information to the output file. The debugging information is required by the GNU debugger, `mb-gdb` or `powerpc-eabi-gdb`. The debugger provides debugging at the source and the assembly level. This option adds debugging information only when the input is a C/C++ source file.

-gstabs

Use this option for adding STABS-based debugging information on assembly (.s) files and assembly file symbols at the source level. This is an assembler option that is provided directly to the GNU assembler, `mb-as` or `powerpc-eabi-as`. If an assembly file is compiled using the compiler `mb-gcc` or `powerpc-eabi-gcc`, prefix the option with **-Wa.**

-On

The GNU compiler provides optimizations at different levels. The optimization levels in the following table apply only to the C and C++ source files.

Table 9-3: Optimizations for Values of n

| <i>n</i> | Optimization |
|-----------------|--|
| 0 | No optimization. |
| 1 | Medium optimization. |
| 2 | Full optimization |
| 3 | Full optimization. Attempt automatic inlining of small subprograms. |
| S | Optimize for size. |

Note: Optimization levels 1 and above cause code re-arrangement. While debugging your code, use of no optimization level is recommended. When an optimized program is debugged through **gdb**, the displayed results might seem inconsistent.

-v

This option executes the compiler and all the tools underneath the compiler in verbose mode. This option gives complete description of the options passed to all the tools. This description is helpful in discovering the default options for each tool.

-save-temps

The GNU compiler provides a mechanism to save the intermediate files generated during the compilation process. The compiler stores the following files:

- Preprocessor output `-input_file_name.i` for C code and `input_file_name.ii` for C++ code
- Compiler (cc1) output in assembly format `-input_file_name.s`
- Assembler output in ELF format `-input_file_name.s`

The compiler saves the default output of the entire compilation as `a.out`.

-o filename

The compiler stores the default output of the compilation process in an ELF file named `a.out`. You can change the default name using `-o output_file_name`. The output file is created in ELF format.

-Wp,option**-Wa,option****-Wl,option**

The compiler, `mb-gcc` or `powerpc-eabi-gcc`, is a wrapper around other executables such as the preprocessor, compiler (cc1), assembler, and the linker. You can run these components of the compiler individually or through the top level compiler.

There are certain options that are required by tools, but might not be necessary for the top-level compiler. To run these commands, use the options listed in the following table.

Table 9-4: Tool-Specific Options Passed to the Top-Level GCC Compiler

| Option | Tool | Example |
|-------------------|--------------|---|
| -Wp,option | Preprocessor | mb-gcc -Wp, -D -Wp, MYDEFINE ... Signal the pre-processor to define the symbol MYDEFINE with the -D MYDEFINE option. |
| -Wa,option | Assembler | powerpc-eabi-gcc -Wa, -m405... Signal the assembler to target the PowerPC 405 processor with the -m405 option. |
| -Wl,option | Linker | mb-gcc -Wl, -M ... Signal the linker to produce a map file with the -M option. |

-help

Use this option with any GNU compiler to get more information about the available options.

You can also consult the GCC manual. A link to the manual is in [Appendix E, “Additional Resources.”](#)

-B directory

Add *directory* to the C run time library search paths.

-L directory

Add *directory* to library search path.

-I directory

Add *directory* to header search path.

-l library

Search *library* for undefined symbols.

Note: The compiler prefixes “lib” to the library name indicated in this command line switch.

Library Search Options

-l libraryname

By default, the compiler searches only the standard libraries, such as `libc`, `libm`, and `libx11`. You can also create your own libraries. You can specify the name of the library and where the compiler can find the definition of these functions. The compiler prefixes `lib` to the library name that you provide.

The compiler is sensitive to the order in which you provide options, particularly the **-l** command line switch. Provide this switch only after all of the sources in the command line.

For example, if you create your own library called `libproject.a`. you can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -l project
```

Caution! If you supply the library flag `-l library_name` before the source files, the compiler does not find the functions called from any of the sources. This is because the compiler search is only done in one direction and it does not keep a list of available libraries.

-L Lib Directory

This option indicates the directories in which to search for the libraries. The compiler has a default library search path, where it looks for the standard library. Using the `-L` option, you can include some additional directories in the compiler search path.

Header File Search Option

-I Directory Name

This option searches for header files in the `<dir_name>` directory before searching the header files in the standard path.

Default Search Paths

The compilers, `mb-gcc` and `powerpc-eabi-gcc`, search certain paths for libraries and header files. The search paths on the various platforms are described below.

Library Search Procedures

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the `-L <dir_name>` option.
2. Directories are passed to the compiler with the `-B <dir_name>` option.
3. The compilers search the following libraries:
 - a. `${XILINX_EDK}/gnu/processor/platform/processor-lib/lib`
 - b. `${XILINX_EDK}/lib/processor`

Note: Processor indicates `powerpc-eabi` for the PowerPC processor and `microblaze` for MicroBlaze.

Header File Search Procedures

The compilers search header files in the following order:

1. Directories are passed to the compiler with the `-I <dir_name>` option.
2. The compilers search the following header files:
 - a. `${XILINX_EDK}/gnu/processor/platform/lib/gcc/processor/{gcc version}/include`
 - b. `${XILINX_EDK}/gnu/processor/platform/processor-lib/include`

Initialization File Search Procedures

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the **-B** *<dir_name>* option.
2. The compilers search `${XILINX_EDK}/gnu/processor/platform/processor-lib/lib`.
3. The compilers search the following libraries:
 - a. `$XILINX_EDK/gnu/<processor>/platform/<processor-lib>/lib`
 - b. `$XILINX_EDK/lib/processor`

Where:

- *<processor>* is `powerpc-eabi` for PowerPC processors, and `microblaze` for MicroBlaze processors.
- *<processor-lib>* is `powerpc-eabi` for PowerPC processors, and `microblaze-xilinx-elf` for MicroBlaze processors.

Note: *platform* indicates `lin` for Linux, `lin64` for Linux 64-bit and `nt` for Windows Cygwin.

Linker Options

-defsym _STACK_SIZE=value

The total memory allocated for the stack can be modified using this linker option. The variable `_STACK_SIZE` is the total space allocated for the stack. The `_STACK_SIZE` variable is given the default value of 100 words, or 400 bytes. If your program is expected to need more than 400 bytes for stack and heap combined, it is recommended that you increase the value of `_STACK_SIZE` using this option. The value is in bytes.

In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program tries to write in other, incorrect, sections of the program, leading to incorrect execution of the code.

Note: A minimum stack size of 16 bytes (0x0010) is required for programs linked with the Xilinx-provided C runtime (CRT) files.

-defsym _HEAP_SIZE=value

The total memory allocated for the heap can be controlled by the value given to the variable `_HEAP_SIZE`. The default value of `_HEAP_SIZE` is zero.

Dynamic memory allocation routines use the heap. If your program uses the heap in this fashion, then you must provide a reasonable value for `_HEAP_SIZE`.

For advanced users: you can generate linker scripts directly from XPS.

Memory Layout

The MicroBlaze and PowerPC processors use 32-bit logical addresses and can address any memory in the system in the range 0x0 to 0xFFFFFFFF. This address range can be categorized into reserved memory and I/O memory.

Reserved Memory

Reserved memory has been defined by the hardware and software programming environment for privileged use. This is typically true for memory containing interrupt vector locations and operating system level routines. [Table 9-5](#) lists the reserved memory locations for MicroBlaze and PowerPC processors as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

Note: In addition to these memories that are reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform that you are using to find out if any memory locations are deemed reserved.

Table 9-5: Hardware Reserved Memory Locations

| Processor Family | Reserved Memories | Reserved Purpose | Default Text Start Address |
|------------------|-----------------------------|---|----------------------------|
| MicroBlaze | 0x0 - 0x4F | Reset, Interrupt, Exception, and other reserved vector locations. | 0x50 |
| PowerPC | 0xFFFFFFFFFC - 0xFFFFFFFFFF | Reset vector location. | 0xFFFF0000 |

I/O Memory

I/O memory refers to addresses used by your program to communicate with memory-mapped peripherals on the processor buses. These addresses are defined as a part of your hardware platform specification.

User and Program Memory

User and Program memory refers to all the memory that is required for your compiled executable to run. By convention, this includes memories for storing instructions, read-only data, read-write data, program stack, and program heap. These sections can be stored in any addressable memory in your system. By default the compiler generates code and data starting from the address listed in [Table 9-5](#) and occupying contiguous memory locations. This is the most common memory layout for programs. You can modify the starting location of your program by defining (in the linker) the symbol `_TEXT_START_ADDR` for MicroBlaze and `_START_ADDR` for PowerPC processors.

In special cases, you might want to partition the various sections of your ELF file across different memories. This is done using the linker command language (refer to the [Linker Scripts](#), page 108 for details). The following are some situations in which you might want to change the memory map of your executable:

- When partitioning large code segments across multiple smaller memories
- Remapping frequently executed sections to fast memories
- Mapping read-only segments to non-volatile flash memories

No restrictions apply to how you can partition your executable. The partitioning can be done at the output section level, or even at the individual function and data level. The resulting ELF can be non-contiguous, that is, there can be “holes” in the memory map. Ensure that you do not use documented reserved locations.

Alternatively, if you are an advanced user and want to modify the default binary data provided by the tools for the reserved memory locations, you can do so. In this case, you must replace the default startup files and the memory mappings provided by the linker.

Object-File Sections

An executable file is created by concatenating input sections from the object files (.o files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following figure.

In addition to these sections, you can also create your own custom sections and assign them to memories of your choice.

Sectional Layout of an object or an Executable File

| | |
|---------|--|
| .text | Text Section |
| .rodata | Read-Only Data Section |
| .sdata2 | Small Read-Only Data Section |
| .sbss2 | Small Read-Only Uninitialized Data Section |
| .data | Read-Write Data Section |
| .sdata | Small Read-Write Data Section |
| .sbss | Small Uninitialized Data Section |
| .bss | Uninitialized Data Section |
| .heap | Program Heap Memory Section |
| .stack | Program Stack Memory Section |

X11005

Figure 9-2: Sectional Layout of an Object or Executable File

The reserved sections that you would not typically modify include: .init, .fini, .ctors, .dtors, .got, .got2, and .eh_frame.

.text

This section of the object file contains executable program instructions. This section has the *x* (executable), *r* (read-only) and *i* (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.

.rodata

This section contains read-only data. This section has the *r* (read-only) and the *i* (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.

.sdata2

This section is similar to the `.rodata` section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. You can change the size of the data going into this section with the `-G` option to the compiler. This section has the *r* (read-only) and the *i* (initialized) flags.

.data

This section contains read-write data and has the *w* (read-write) and the *i* (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.

.sdata

This section contains small read-write data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the *w* (read-write) and the *i* (initialized) flags and must be mapped to initialized RAM.

.sbss2

This section contains small, read-only un-initialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the *r* (read) flag and can be mapped to ROM.

.sbss

This section contains small un-initialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the *w* (read-write) flag and must be mapped to RAM.

.bss

This section contains un-initialized data. This section has the *w* (read-write) flag and must be mapped to RAM.

.heap

This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.

.stack

This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections might appear merged together into a section named `.bss_stack`.

.init

This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.

.fini

This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.

.ctors

This section contains a list of functions that must be invoked at program startup and the same flags as `.data` and must be mapped to initialized RAM.

.dtors

This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.

.got2/.got

This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.

.eh_frame

This section contains frame unwind information for exception handling. It contains the same flags as `.rodata`, and can be mapped to initialized ROM.

.tbss

This section holds uninitialized thread-local data that contribute to the program memory image. This section has the same flags as `.bss`, and it must be mapped to RAM.

.tdata

This section holds initialized thread-local data that contribute to the program memory image. This section must be mapped to initialized RAM.

.gcc_except_table

This section holds language specific data. This section must be mapped to initialized RAM.

.jcr

This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions. This section must be mapped to initialized RAM.

.fixup

This section contains information necessary for doing fixup, such as the fixup page table, and the fixup record table. This section must be mapped to initialized RAM.

Linker Scripts

The linker utility uses commands specified in linker scripts to divide your program on different blocks of memories. It describes the mapping between all of the sections in all of the input object files to output sections in the executable file. The output sections are mapped to memories in the system. You do not need a linker script if you do not want to change the default contiguous assignment of program contents to memory. There is a default linker script provided with the linker that places section contents contiguously.

You can selectively modify only the starting address of your program by defining the linker symbol `_TEXT_START_ADDR` on MicroBlaze processors, or `_START_ADDR` on PowerPC processors, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
```

```
powerpc-eabi-gcc <input files and flags> -Wl,-defsym  
-Wl,_TEXT_START_ADDR=0x2000
```

```
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The choices of the default script that will be used by the linker from the `$XILINX_EDK/gnu/<procname>/<platform>/<processor_name>/lib/ldscripts` area are described as follows:

- `elf32<procname>.x` is used by default when none of the following cases apply.
- `elf32<procname>.xn` is used when the linker is invoked with the `-n` option.
- `elf32<procname>.xbn` is used when the linker is invoked with the `-N` option.
- `elf32<procname>.xr` is used when the linker is invoked with the `-r` option.
- `elf32<procname>.xu` is used when the linker is invoked with the `-Ur` option.

where `<procname>` = ppc or microblaze, `<processor_name>` = powerpc-eabi or microblaze, and `<platform>` = lin or nt.

To use a linker script, provide it on the GCC command line. Use the command line option `-T <script>` for the compiler, as described below:

```
compiler -T <linker_script> <Other Options and Input Files>
```

If the linker is executed on its own, include the linker script as follows:

```
linker -T <linker_script> <Other Options and Input Files>
```

This tells GCC to use your linker script in the place of the default built-in linker script. Linker scripts can be generated for your program from within XPS and SDK.

In XPS or SDK, select **Tools > Generate Linker Script**.

This opens up the linker script generator utility. Mapping sections to memory is done here. Stack and Heap size can be set, as well as the memory mapping for Stack and Heap. When the linker script is generated, it is given as input to GCC automatically when the corresponding application is compiled within XPS or SDK.

Linker scripts can be used to assign specific variables or functions to specific memories. This is done through “section attributes” in the C code. Linker scripts can also be used to assign specific object files to sections in memory. These and other features of GNU linker scripts are explained in the GNU linker documentation, which is a part of the online `binutils` manual. A link to the GNU manuals is supplied in the [Appendix E, “Additional Resources.”](#) For a specific list of input sections that are assigned by MicroBlaze and PowerPC processor linker scripts, see “[MicroBlaze Linker Script Sections](#)” on page 117, and “[PowerPC Processor Linker Script Sections](#)” on page 126.

MicroBlaze Compiler Usage and Options

The MicroBlaze GNU compiler is derived from the standard GNU sources as the Xilinx port of the compiler. The features and options that are unique to the MicroBlaze compiler are described in the sections that follow. When compiling with the MicroBlaze compiler, the pre-processor provides the definition `__MICROBLAZE__` automatically. You can use this definition in any conditional code.

MicroBlaze Compiler

The `mb-gcc` compiler for the Xilinx™ MicroBlaze soft processor introduces new options as well as modifications to certain options supported by the GNU compiler tools. The new and modified options are summarized in this chapter.

MicroBlaze Compiler Options: Quick Reference

Click an option name below to view its description.

Processor Feature Selection Options

[-mcpu=vX.YY.Z](#)
[-mno-xl-soft-mul](#)
[-mxl-multiply-high](#)
[-mno-xl-multiply-high](#)
[-mxl-soft-mul](#)
[-mno-xl-soft-div](#)
[-mxl-soft-div](#)
[-mxl-barrel-shift](#)
[-mno-xl-barrel-shift](#)
[-mxl-pattern-compare](#)
[-mno-xl-pattern-compare](#)
[-mhard-float](#)
[-msoft-float](#)
[-mxl-float-convert](#)
[-mxl-float-sqrt](#)

General Program Options

[-msmall-divides](#)
[-mxl-gp-opt](#)
[-mno-clearbss](#)
[-mxl-stack-check](#)

Application Execution Modes

[-xl-mode-executable](#)
[-xl-mode-xmdstub](#)
[-xl-mode-bootstrap](#)
[-xl-mode-novectors](#)

MicroBlaze Linker Options

[-defsym _TEXT_START_ADDR=value](#)
[-relax](#)
[-N](#)

Processor Feature Selection Options

-mcpu=vX.YY.Z

This option directs the compiler to generate code suited to MicroBlaze hardware version `v.X.YY.Z`. To get the most optimized and correct code for a given processor, use this switch with the hardware version of the processor.

The `-mcpu` switch behaves differently for different versions, as described below:

- `Pr-v3.00.a`: Uses 3-stage processor pipeline mode. Does not inhibit exception causing instructions being moved into delay slots.
- `v3.00.a` and `v4.00.a`: Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
- `v5.00.a` and later: Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots.

-mlittle-endian / -mbig-endian

Use these options to select the endianness of the target machine for which code is being compiled. The endianness of the binary object file produced is also set appropriately based on this switch. The GCC driver passes switches to the sub tools (`as`, `cc1`, `cc1plus`, `ld`) to set the corresponding endianness in the sub tool.

The default is `-mbig-endian`.

Note: You cannot link together object files of mixed endianness.

-mno-xl-soft-mul

This option permits use of hardware multiply instructions for 32-bit multiplications.

The MicroBlaze processor has an option to turn the use of hardware multiplier resources on or off. This option should be used when the hardware multiplier option is enabled on the MicroBlaze processor. Using the hardware multiplier can improve the performance of your application. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. See the *MicroBlaze Processor Reference Guide, (UG081)*, for more details about the usage of the multiplier option in MicroBlaze. A link to the document is in [Appendix E, “Additional Resources.”](#)

-mxl-multiply-high

The MicroBlaze processor has an option to enable instructions that can compute the higher 32-bits of a 32x32-bit multiplication. This option tells the compiler to use these multiply high instructions. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is available or not. See the *MicroBlaze Processor Reference Guide, (UG081)*, for more details about the usage of the multiply high instructions in MicroBlaze. A link to the document is in [Appendix E, “Additional Resources.”](#)

-mno-xl-multiply-high

Do not use multiply high instructions. This option is the default.

-mxl-soft-mul

This option tells the compiler that there is no hardware multiplier unit on MicroBlaze, so every 32-bit multiply operation is replaced by a call to the software emulation routine `__mulsi3`. This option is the default.

-mno-xl-soft-div

You can instantiate a hardware divide unit in MicroBlaze. When the divide unit is present, this option tells the compiler that hardware divide instructions can be used in the program being compiled.

This option can improve the performance of your program if it has a significant amount of division operations. The compiler automatically defines the C pre-processor definition `HAVE_HW_DIV` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. See the *MicroBlaze Processor Reference Guide, (UG081)*, for more details about the usage of the hardware divide option in MicroBlaze. A link to the document is in [Appendix E, “Additional Resources.”](#)

-mxl-soft-div

This option tells the compiler that there is no hardware divide unit on the target MicroBlaze hardware.

This option is the default. The compiler replaces all 32-bit divisions with a call to the corresponding software emulation routines (`__divsi3`, `__udivsi3`).

-mxl-barrel-shift

The MicroBlaze processor can be configured to be built with a barrel shifter. In order to use the barrel shift feature of the processor, use the option `-mxl-barrel-shift`.

The default option assumes that no barrel shifter is present, and the compiler uses add and multiply operations to shift the operands. Enabling barrel shifts can speed up your application significantly, especially while using a floating point library. The compiler automatically defines the C pre-processor definition `HAVE_HW_BSHIFT` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether or not this feature is specified as available. See the *MicroBlaze Processor Reference Guide*, (UG081), for more details about the use of the barrel shifter option in MicroBlaze. A link to the document is in [Appendix E, "Additional Resources."](#)

-mno-xl-barrel-shift

This option tells the compiler not to use hardware barrel shift instructions. This option is the default.

-mxl-pattern-compare

This option activates the use of pattern compare instructions in the compiler.

Using pattern compare instructions can speed up boolean operations in your program. Pattern compare operations also permit operating on word-length data as opposed to byte-length data on string manipulation routines such as `strcpy`, `strlen`, and `strcmp`. On a program heavily dependent on string manipulation routines, the speed increase obtained will be significant. The compiler automatically defines the C pre-processor definition `HAVE_HW_PCMP` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Guide*, (UG081), for more details about the use of the pattern compare option in MicroBlaze. A link to the document is in [Appendix E, "Additional Resources."](#)

-mno-xl-pattern-compare

This option tells the compiler not to use pattern compare instructions. This is the default.

-mhard-float

This option turns on the usage of single precision floating point instructions (`fadd`, `fsub`, `fmul`, and `fdiv`) in the compiler.

It also uses `fcmp.p` instructions, where `p` is a predicate condition such as `le`, `ge`, `lt`, `gt`, `eq`, `ne`. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware. The compiler automatically defines the C pre-processor definition `HAVE_HW_FPU` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Guide*, (UG081), for more details about the use of the hardware floating point unit option in MicroBlaze. A link to the document is in [Appendix E, "Additional Resources."](#)

-msoft-float

This option tells the compiler to use software emulation for floating point arithmetic. This option is the default.

-mx1-float-convert

This option turns on the usage of single precision floating point conversion instructions (`fsint` and `flt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide, (UG081)*, for more details about the use of the hardware floating point unit option in MicroBlaze. A link to the document is in [Appendix E, “Additional Resources.”](#)

-mx1-float-sqrt

This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide, (UG081)*, for more details about the use of the hardware floating point unit option in the MicroBlaze processor. A link to the document is in [Appendix E, “Additional Resources.”](#)

General Program Options

-msmall-divides

This option generates code optimized for small divides when no hardware divider exists. For signed integer divisions where the numerator and denominator are between 0 and 15 inclusive, this switch provides very fast table-lookup-based divisions. This switch has no effect when the hardware divider is enabled.

-mx1-gp-opt

If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load or store operations to that address require two instructions.

The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 Kbytes of data. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value, needing only one instruction for a load or store to the small data area. This optimization can be turned on with the `-mx1-gp-opt` command line parameter. Variables of size less than a certain threshold value are stored in these areas and can be addressed with fewer instructions. The addresses are calculated during the linking stage.

Caution! If this option is being used, it must be provided to both the compile and the link commands of the build process for your program. Using the switch inconsistently can lead to compile, link, or run-time errors.

-mno-clearbss

This option is useful for compiling programs used in simulation.

According to the C language standard, uninitialized global variables are allocated in the `.bss` section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Optimizing compilers also allocate global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the above two language features can be unwanted overhead. Some simulators automatically zero the entire memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler to not allocate zero-initialized global variables in the `.bss` and instead move them to the `.data` section. This option might improve startup times for your application. Use this option with care and ensure either that you do not use code that relies on global variables being initialized to zero, or that your simulation platform performs the zeroing of memory.

-mxl-stack-check

With this option, you can check whether the stack overflows when the program runs.

The compiler inserts code in the prologue of the every function, comparing the stack pointer value with the available memory. If the stack pointer exceeds the available free memory, the program jumps to a the subroutine `_stack_overflow_exit`. This subroutine sets the value of the variable `_stack_overflow_error` to 1.

You can override the standard stack overflow handler by providing the function `_stack_overflow_exit` in the source code, which acts as the stack overflow handler.

Application Execution Modes

-xl-mode-executable

This is the default mode used for compiling programs with `mb-gcc`. This option need not be provided on the command line for `mb-gcc`. This uses the startup file `crt0.o`.

-xl-mode-xmdstub

The Xilinx Microprocessor Debugger (XMD) allows debugging of applications in a software-intrusive manner, known as XMDSTUB mode. Compile programs being debugged in such a manner with this switch. In such programs, the address locations 0x0 to 0x800 are reserved for use by XMDSTUB. Using `-xl-mode-xmdstub` has two effects:

- The start address of your program is set to 0x800. You can change this address by overriding the `_TEXT_START_ADDR` in the linker script or through linker options. For more details about linker options, refer to [Linker Options, page 103](#). If the start address is defined to be less than 0x800, XMD issues an address overlap error.
- `crt1.o` is used as the initialization file. The `crt1.o` file returns the control back to the XMDStub when your program execution is complete.

Note: Use `-xl-mode-xmdstub` for designs when XMDStub is part of the bitstream. Do not use this mode when the system is compiled for No Debug or when "Hardware Debugging" is turned ON. For more details on debugging with XMD, refer to [Chapter 11, GNU Debugger](#).

-x1-mode-bootstrap

This option is used for applications that are loaded using a bootloader. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If a normal executable is loaded by this bootloader, the application reset vector overwrites the reset vector of the bootloader. In such a scenario, on a processor reset, the bootloader does not execute first (it is typically required to do so) to reload this application and do other initialization as necessary.

To prevent this, you must compile the bootloaded application with this compiler flag. On a processor reset, control then reaches the bootloader instead of the application.

Using this switch on an application that is deployed in a scenario different from the one described above will not work. This mode uses `crt2.o` as a startup file.

-x1-mode-novectors

This option is used for applications that do not require any of the MicroBlaze vectors. This is typically used in standalone applications that do not use any of the processor's reset, interrupt, or exception features. Using this switch leads to smaller code size due to the elimination of the instructions for the vectors. This mode uses `crt3.o` as a startup file.

Caution! Do not use more than one mode of execution on the command line. You will receive link errors due to multiple definition of symbols if you do so.

Position Independent Code

The GNU compiler for MicroBlaze supports the `-fPIC` and `-fpic` switches. These switches enable Position Independent Code (PIC) generation in the compiler. This feature is used by the Linux operating system only for MicroBlaze to implement shared libraries and relocatable executables. The scheme uses a Global Offset Table (GOT) to relocate all data accesses in the generated code and a Procedure Linkage Table (PLT) for making function calls into shared libraries. This is the standard convention in GNU-based platforms for generating relocatable code and for dynamically linking against shared libraries.

MicroBlaze Application Binary Interface

The GNU compiler for MicroBlaze uses the Application Binary Interface (ABI) defined in the *MicroBlaze Processor Reference Guide (UG081)*. Refer to the ABI documentation for register and stack usage conventions as well as a description of the standard memory model used by the compiler. A link to the document is provided in [Appendix E, "Additional Resources."](#)

MicroBlaze Assembler

The `mb-as` assembler for the Xilinx MicroBlaze soft processor supports the same set of options supported by the standard GNU compiler tools. It also supports the same set of assembler directives supported by the standard GNU assembler.

The `mb-as` assembler supports all the opcodes in the MicroBlaze machine instruction set, with the exception of the `imm` instruction. The `mb-as` assembler generates `imm` instructions when large immediate values are used. The assembly language programmer is never required to write code with `imm` instructions. For more information on the MicroBlaze instruction set, refer to the *MicroBlaze Processor Reference Guide (UG081)*. A link to the document is in [Additional Resources, page 279](#).

The `mb-as` assembler requires all MicroBlaze instructions with an immediate operand to be specified as a constant or a label. If the instruction requires a PC-relative operand, then the `mb-as` assembler computes it and includes an `imm` instruction if necessary.

For example, the Branch Immediate if Equal (`beqi`) instruction requires a PC-relative operand.

The assembly programmer should use this instruction as follows:

```
beqi r3, mytargetlabel
```

where `mytargetlabel` is the label of the target instruction. The `mb-as` assembler computes the immediate value of the instruction as `mytargetlabel - PC`.

If this immediate value is greater than 16 bits, the `mb-as` assembler automatically inserts an `imm` instruction. If the value of `mytargetlabel` is not known at the time of compilation, the `mb-as` assembler always inserts an `imm` instruction. Use the `relax` option of the linker remove any unnecessary `imm` instructions.

Similarly, if an instruction needs a large constant as an operand, the assembly language programmer should use the operand as is, without using an `imm` instruction. For example, the following code adds the constant 200,000 to the contents of register `r3`, and stores the results in register `r4`:

```
addi r4, r3, 200000
```

The `mb-as` assembler recognizes that this operand needs an `imm` instruction, and inserts one automatically.

In addition to the standard MicroBlaze instruction set, the `mb-as` assembler also supports some pseudo-op codes to ease the task of assembly programming. Table 9-6 lists the supported pseudo-opcodes.

Table 9-6: Pseudo-Opcodes Supported by the GNU Assembler

| Pseudo Opcodes | Explanation |
|-----------------------------|--|
| <code>nop</code> | No operation. Replaced by instruction: <code>or R0, R0, R0</code> |
| <code>la Rd, Ra, Imm</code> | Replaced by instruction: <code>addik Rd, Ra, imm; = Rd = Ra + Imm;</code> |
| <code>not Rd, Ra</code> | Replace by instruction: <code>xori Rd, Ra, -1</code> |
| <code>neg Rd, Ra</code> | Replace by instruction: <code>rsub Rd, Ra, R0</code> |
| <code>sub Rd, Ra, Rb</code> | Replace by instruction: <code>rsub Rd, Rb, Ra</code> |

MicroBlaze Linker Options

The `mb-ld` linker for the MicroBlaze soft processor provides additional options to those supported by the GNU compiler tools. The options are summarized in this section.

-defsym _TEXT_START_ADDR=value

By default, the text section of the output code starts with the base address 0x28 (0x800 in XMDStub mode). This can be overridden by using the `-defsym _TEXT_START_ADDR` option. If this is supplied to `mb-gcc` compiler, the text section of the output code starts from the given value.

You do not have to use `-defsym _TEXT_START_ADDR` if you want to use the default start address set by the compiler.

This is a linker option and should be used when you invoke the linker separately. If the linker is being invoked as a part of the `mb-gcc` flow, you must use the following option:

```
-Wl,-defsym -Wl,_TEXT_START_ADDR=value
```

-relax

This is a linker option that removes all unwanted `imm` instructions generated by the assembler. The assembler generates an `imm` instruction for every instruction where the value of the immediate cannot be calculated during the assembler phase.

Most of these instructions do not need an `imm` instruction. These are removed by the linker when the `-relax` command line option is provided.

This option is required only when linker is invoked on its own. When linker is invoked through the `mb-gcc` compiler, this option is automatically provided to the linker.

-N

This option sets the text and data section as readable and writable. It also does not page-align the data segment. This option is required only for MicroBlaze programs. The top-level GCC compiler automatically includes this option, while invoking the linker, but if you intend to invoke the linker without using GCC, use this option.

For more details on this option, refer to the GNU manuals online. A link to the manuals is in [Appendix E, "Additional Resources."](#)

The MicroBlaze linker uses linker scripts to assign sections to memory. These are listed in the following section.

MicroBlaze Linker Script Sections

Table 9-7 lists the input sections that are assigned by MicroBlaze linker scripts.

Table 9-7: Section Names and Descriptions

| Section | Description |
|------------------------------------|---|
| <code>.vectors.reset</code> | Reset vector code. |
| <code>.vectors.sw_exception</code> | Software exception vector code. |
| <code>.vectors.interrupt</code> | Hardware Interrupt vector code. |
| <code>.vectors.hw_exception</code> | Hardware exception vector code. |
| <code>.text</code> | Program instructions from code in functions and global assembly statements. |
| <code>.rodata</code> | Read-only variables. |
| <code>.sdata2</code> | Small read-only static and global variables with initial values. |
| <code>.data</code> | Static and global variables with initial values. Initialized to zero by the boot code. |
| <code>.sdata</code> | Small static and global variables with initial values. |
| <code>.sbss2</code> | Small read-only static and global variables without initial values. Initialized to zero by boot code. |
| <code>.sbss</code> | Small static and global variable without initial values. Initialized to zero by the boot code. |
| <code>.bss</code> | Static and global variables without initial values. Initialized to zero by the boot code. |
| <code>.heap</code> | Section of memory defined for the heap. |
| <code>.stack</code> | Section of memory defined for the stack. |

Tips for Writing or Customizing Linker Scripts

The following points must be kept in mind when writing or customizing your own linker script:

- Ensure that the different vector sections are assigned to the appropriate memories as defined by the MicroBlaze hardware.
- 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. Because 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.

Note: The `.bss` section boundary does not include either stack or heap.

- 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.
- ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap). The standard CRT that is provided assumes a single `.bss` section that is initialized to zero. If there are multiple `.bss` sections, this CRT will not work. You should write your own CRT that initializes all the `.bss` sections.

Startup Files

The compiler includes pre-compiled startup and end files in the final link command when forming an executable. Startup files set up the language and the platform environment before your application code executes. Start up files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to [Table 9-8, page 118](#) for details.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions, such as C++ constructors.
- Initialize the hardware sub-system. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

Similarly, end files are used to include code that must execute after your program ends. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- De-initialize the hardware sub-system. For example, if the program is being profiled, clean up the profiling sub-system.

[Table 9-8](#) lists the register names, values, and descriptions in the C-Runtime files.

Table 9-8: Register Initialization in C-Runtime Files

| Register | Value | Description |
|-----------------|-------------------------|--|
| r1 | <code>_stack-16</code> | The stack pointer register is initialized to point to the bottom of the stack area with an initial negative offset of 16 bytes. The 16 bytes can be used for passing in arguments. |
| r2 | <code>_SDA2_BASE</code> | <code>_SDA2_BASE_</code> is the read-only small data anchor address. |
| r13 | <code>_SDA_BASE_</code> | <code>_SDA_BASE</code> is the read-write small data anchor address. |
| Other registers | Undefined | Other registers do not have defined values. |

The following subsections describe the initialization files used for various application modes. This information is for advanced users who want to change or understand the startup code of their application.

For MicroBlaze, there are two distinct stages of C runtime initialization. The first stage is primarily responsible for setting up vectors, after which it invokes the second stage initialization. It also provides exit stubs based on the different application modes.

First Stage Initialization Files

crt0.o

This initialization file is used for programs which are to be executed in standalone mode, without the use of any bootloader or debugging stub such as `xmdstub`. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine `_crtinit`. On returning from `_crtinit`, it ends the program by infinitely looping in the `_exit` label.

crt1.o

This initialization file is used when the application is debugged in a software-intrusive manner. It populates all the vectors *except the breakpoint and reset vectors* and transfers control to the second-stage `_crtinit` startup routine. Upon return from `_crtinit`, program control returns back to the `XMDStub`, which signals to the debugger that the program has finished.

crt2.o

This initialization file is used when the executable is loaded using a bootloader. It populates all the vectors *except the reset vector* and transfers control to the second-stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the reset vector is not populated, on a processor reset, control is transferred to the bootloader, which can reload and restart the program.

crt3.o

This initialization file is employed when the executable does not use any vectors and wishes to reduce code size. It populates only the reset vector and transfers control to the second stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the other vectors are not populated, the GNU linking mechanism does not pull in any of the interrupt and exception handling related routines, thus saving code space.

Second Stage Initialization Files

According to the C standard specification, all global and static variables must be initialized to 0. This is a common functionality required by all the CRTs above. Another routine, `_crtinit`, is invoked. The `_crtinit` routine initializes memory in the `.bss` section of the program. The `_crtinit` routine is also the wrapper that invokes the `main` procedure. Before invoking the `main` procedure, it may invoke other initialization functions. The `_crtinit` routine is supplied by the startup files described below.

crtinit.o

This default, second stage, C startup file performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes “constructor” functions (`_init`).
4. Sets up the arguments for `main` and invokes `main`.

5. Invokes “destructor” functions (`_fini`).
6. Invokes `_program_clean` and returns.

pgcrtinit.o

This second stage startup file is used during profiling, and performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes `_profile_init` to initialize the profiling library.
4. Invokes “constructor” functions (`_init`).
5. Sets up the arguments for `main` and invokes `main`.
6. Invokes “destructor” functions (`_fini`).
7. Invokes `_profile_clean` to cleanup the profiling library.
8. Invokes `_program_clean`, and then returns.

sim-crtinit.o

This second-stage startup file is used when the `-mno-clearbss` switch is used in the compiler, and performs the following steps:

1. Invokes `_program_init`.
2. Invokes “constructor” functions (`_init`).
3. Sets up the arguments for `main` and invokes `main`.
4. Invokes “destructor” functions (`_fini`).
5. Invokes `_program_clean`, and then returns.

sim-pgcrtinit.o

This second stage startup file is used during profiling in conjunction with the `-mno-clearbss` switch, and performs the following steps in order:

1. Invokes `_program_init`.
2. Invokes `_profile_init` to initialize the profiling library.
3. Invokes “constructor” functions (`_init`).
4. Sets up the arguments for and invokes `main`.
5. Invokes “destructor” functions (`_fini`).
6. Invokes `_profile_clean` to cleanup the profiling library.
7. Invokes `_program_clean`, and then returns.

Other files

The compiler also uses certain standard start and end files for C++ language support. These are `crti.o`, `crtbegin.o`, `crtend.o`, and `crtfn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections.

Modifying Startup Files

The initialization files are distributed in both pre-compiled and source form with EDK. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the MicroBlaze GNU compiler can be found in the `<XILINX_EDK>/sw/lib/microblaze/src` directory, where `<XILINX_EDK>` is the EDK installation area.

To fulfill a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, you can assemble the files into `.o` files and place them in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory -name` command-line option while invoking `mb-gcc`.

To prevent the default startup files from being used, use the `-nostartfiles` on the final compile line.

Note: The miscellaneous compiler standard CRT files, such as `crti.o`, and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You might need to bring them in on your final link command.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you might want to eliminate all sources of overhead. This section describes how to reduce the overhead of invoking the C++ constructor or destructor code in a C program that does not require that code. You might be able to save approximately 220 bytes of code space by making the following modifications:

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, copy over the particular versions of `crti.s` and `xcertinit.s` that suit your application. For example, if your application is being bootstrapped and profiled, copy `crt2.s` and `pg-crtinit.s` from the installation area.
2. Modify `pg-crtinit.s` to remove the following lines:

```
brlid r15, __init
/* Invoke language initialization functions */
nop

and

brlid r15, __fini
/* Invoke language cleanup functions */
nop
```

This avoids referencing the extra code usually pulled in for constructor and destructor handling, reducing code size.

3. Compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the `-nostartfiles` switch to the compiler. Add the `-B directory` switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

If your application is executing in a different mode, then you must pick the appropriate CRT files based on the description in [Startup Files, page 118](#).

Compiler Libraries

The `mb-gcc` compiler requires the GNU C standard library and the GNU math library. Precompiled versions of these libraries are shipped with EDK. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze, during the execution of Libgen. To manually select the library version that you would like to use, look in the following folder:

```
$XILINX_EDK/gnu/microblaze/<platform>/microblaze-xilinx-elf/lib
```

The filenames are encoded based on the compiler flags and configurations used to compile the library. For example, `libc_m_bs.a` is the C library compiled with hardware multiplier and barrel shifter enabled in the compiler.

Table 9-9 shows the current encodings used and the configuration of the library specified by the encodings.

Table 9-9: Encoded Library Filenames on Compiler Flags

| Encoding | Description |
|------------------|-------------------------------------|
| <code>_bs</code> | Configured for barrel shifter. |
| <code>_m</code> | Configured for hardware multiplier. |
| <code>_p</code> | Configured for pattern comparator. |

Of special interest are the math library files (`libm*.a`). The C standard requires the common math library functions (`sin()` and `cos()`, for example) to use double-precision floating point arithmetic. However, double-precision floating point arithmetic may not be able to make full use of the optional, single-precision floating point capabilities in available for MicroBlaze.

The Newlib math libraries have alternate versions that implement these math functions using single-precision arithmetic. These single-precision libraries might be able to make direct use of the MicroBlaze processor hardware Floating Point Unit (FPU) and could therefore perform better.

If you are sure that your application does not require standard precision, and you want to implement enhanced performance, you can manually change the version of the linked-in library.

By default, the CPU driver copies the double-precision version (`libm*_fpd.a`) of the library into your XPS project.

To get the single precision version, you can create a custom CPU driver that copies the corresponding `libm*_fps.a` library instead. Copy the corresponding `libm*_fps.a` file into your processor library folder (such as `microblaze_0/lib`) as `libm.a`.

When you have copied the library that you want to use, rebuild your application software project.

Thread Safety

The MicroBlaze processor C and math libraries distributed with EDK are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are *not* thread-safe and will cause unrecoverable errors in the system at run-time. Use appropriate mutual exclusion mechanisms when using the EDK libraries in a multi-threaded environment.

Command Line Arguments

The MicroBlaze processor programs cannot take command-line arguments. The command line arguments `argc` and `argv` are initialized to 0 by the C runtime routines.

Interrupt Handlers

Interrupt handlers must be compiled in a different manner than normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save the volatile registers that are being used. Interrupt handlers should also store the value of the machine status register (RMSR) when an interrupt occurs.

interrupt_handler attribute

To distinguish an interrupt handler from a sub-routine, `mb-gcc` looks for an attribute (`interrupt_handler`) in the declaration of the code. This attribute is defined as follows:

```
void function_name () __attribute__ ((interrupt_handler));
```

Note: The attribute for the interrupt handler is to be given *only* in the prototype and *not* in the definition.

Interrupt handlers might also call other functions, which might use volatile registers. To maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function.

Note: Functions that have calls to other sub-routines are called *non-leaf* functions.

Interrupt handlers are defined in the Microprocessor Hardware Specification (MHS) and the Microprocessor Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions. For more information, refer to [Appendix B, "Interrupt Management."](#)

The interrupt handler uses the instruction `rtid` for returning to the interrupted function.

save_volatiles attribute

The MicroBlaze compiler provides the attribute `save_volatiles`, which is similar to the `interrupt_handler` attribute, but returns using `rtsd` instead of `rtid`.

This attribute saves all the volatiles for non-leaf functions and only the used volatiles in the case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

[Table 9-10](#) lists the attributes with their functions.

Table 9-10: Use of Attributes

| Attributes | Functions |
|--------------------------|---|
| interrupt_handler | This attribute saves the machine status register and all the volatiles, in addition to the non-volatile registers. <code>rtid</code> returns from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved. |
| save_volatiles | This attribute is similar to <code>interrupt_handler</code> , but it uses <code>rtsd</code> to return to the interrupted function, instead of <code>rtid</code> . |

PowerPC Compiler Usage and Options

PowerPC Compiler Options: Quick Reference

PowerPC Compiler Options

`-mcpu=440`
`-mfp={sp_lite, sp_full, dp_lite, dp_full, none}`
`-mppcperlib`
`-mno-clearbss`

Linker Options

`-defsym _START_ADDR=value`

PowerPC Compiler Options

The PowerPC processor GNU compiler (`powerpc-eabi-gcc`) is built out of the sources for the PowerPC processor port as distributed by GNU foundation. The compiler is customized for Xilinx purposes. The features and options that are unique to the version distributed with EDK are described in the following sections.

When compiling with the PowerPC processor compiler, the pre-processor automatically provides the definition `__PPC__`. You can use this definition in any conditional code.

`-mcpu=440`

Target code for the PowerPC 440 processor. This includes instruction scheduling optimizations, enable or disable instruction workarounds, and usage of libraries targeted for the 440 processor.

`-mfp={sp_lite, sp_full, dp_lite, dp_full, none}`

Generate hardware floating point instructions to use with the Xilinx PowerPC processor APU FPU coprocessor hardware. The instructions and code output follow the floating point specification in the PowerPC Book-E, with some exceptions tailored to the APU FPU hardware.

Book-E is available from the IBM web page. Refer to the FPU hardware documentation for more information on the architecture. Links to Book-E and to the FPU documentation are in [Appendix E, "Additional Resources."](#)

The option given to `-mfp=` determines which variant of the FPU hardware to target. The variants are:

`sp_lite`

Produces code targeted to the Single precision Lite FPU coprocessor. This version supports only single precision hardware floating point and does not use hardware divide and square root instructions. The compiler automatically defines the C preprocessor definition `HAVE_XFPU_SP_LITE` when this option is given.

`sp_full`

Produces code targeted to the Single precision Full FPU coprocessor. This version supports only single precision hardware floating point and uses hardware divide and square root instructions. The compiler automatically defines the C preprocessor definition `HAVE_XFPU_SP_FULL` when this option is given.

dp_lite

Produces code targeted to the Double precision Lite FPU coprocessor. This version supports both single and double precision hardware floating point and does not use hardware divide and square root instructions. The compiler automatically defines the C preprocessor definition, `HAVE_XFPU_DP_LITE`, when this option is given.

dp_full

Produces code targeted to the double precision full FPU coprocessor. This version supports both single and double precision hardware floating point and uses hardware divide and square root instructions. The compiler automatically defines the C preprocessor definition, `HAVE_XFPU_DP_FULL`, when this option is given.

Caution! Do not link code compiled with one variant of the `-mfpu` switch with code compiled with other variants (or without the `-mfpu` switch). You must use the switch even when you are only linking object files together. This allows the compiler to use the correct set of libraries and prevent incompatibilities.

none

Instructs the compiler to use software emulation for floating point arithmetic.

Refer to the latest APU FPU user guide for detailed information on how to optimize use of the hardware floating point co-processor. A link to the guide is in [Appendix E, Additional Resources](#).

-mppcperflib

Use the PowerPC processor performance libraries for low-level integer and floating emulation, and simple string routines. These libraries are used in the place of the default emulation routines provided by GCC and simple string routines provided by `Newlib`. The performance libraries show an average of three times increase in speed on applications that heavily use these routines. The SourceForge project web page contains more information and detailed documentation. A link to that page is in the [Appendix E, "Additional Resources."](#)

Caution! You cannot use the performance libraries in conjunction with the `-mfpu` switch. They are incompatible.

-mno-clearbss

This option is useful for compiling programs used in simulation. According to the C language standard, uninitialized global variables are allocated in the `.bss` section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Additionally optimizing compilers also allocates global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the language features can be unwanted overhead. Some simulators automatically zero the whole memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios because it:

- Causes the C startup files to not initialize the `.bss` section with zeroes.
- Internally forces the compiler not to allocate zero-initialized global variables in the `.bss` and instead move them to the `.data` section.

This option might improve startup times for your application. Use this option with care. Do not use code that relies on global variables being initialized to zero, or ensure that your simulation platform performs the zeroing of memory.

PowerPC Processor Linker

The `powerpc-eabi-ld` linker for the PowerPC processor introduces a new option in addition to those supported by the GNU compiler tools. The option is described below:

-defsym _START_ADDR=value

By default, the text section of the output code starts with the base address `0xffff0000` because this is the start address listed in the default linker script. This can be overridden by using this option or providing a linker script that lists the value for the start address.

You are not required to use `-defsym _START_ADDR`, if you want to use the default start address set by the compiler. This is a linker option. Use this option when you invoke the linker separately. If the linker is being invoked as a part of the `powerpc-eabi-gcc` flow, use the option `-Wl,-defsym -Wl,_START_ADDR=value`.

The PowerPC linker uses linker scripts to assign sections to memory. [Table 9-11](#) and the following subsection lists the script sections.

PowerPC Processor Linker Script Sections

Table 9-11: Input Sections Assigned by the PowerPC Processor Linker Scripts

| Section | Description |
|--------------------------------|---|
| <code>.boot</code> | Processor reset vector code with initial branch to <code>.boot0</code> . |
| <code>.boot0</code> | Boot code. |
| <code>.bss</code> | Static and global variables without initial values. Initialized to 0 by the boot code. |
| <code>.data</code> | Static and global variables with initial values. These variables are initialized to zero by the boot code. |
| <code>.fixup</code> | Fixup information, such as fixup record table. |
| <code>.gcc_except_table</code> | Language specific data. |
| <code>.got2</code> | Global Offset Table (GOT). The GOT is to define a place where position independent code can access global data. |
| <code>.got1</code> | Global Offset Table (GOT). The GOT defines a place where position independent code can access global data. |
| <code>.heap</code> | Section of memory defined for the heap. |
| <code>.jcr</code> | Compiler-specific. Used by compiler initialization functions. |
| <code>.rodata</code> | Read-only variables. |
| <code>.stack</code> | Section of memory defined for the stack. |
| <code>.sbss</code> | Small static and global variables without initial values. Initialized to 0 by the boot code. |
| <code>.sbss2</code> | Small read-only static and global variables with initial values. Initialized to zero by the boot code. |
| <code>.sdata</code> | Small static and global variables with initial values. |
| <code>.sdata2</code> | Small read-only static and global variables with initial values. |

Table 9-11: Input Sections Assigned by the PowerPC Processor Linker Scripts

| Section | Description |
|---------------------|---|
| <code>.text</code> | Program instructions from code in functions and global assembly statements. |
| <code>.tdata</code> | Initialized thread-local data. |
| <code>.tbss</code> | Uninitialized thread-local data. |

Tips for Writing or Customizing Linker Scripts

The following points must be kept in mind when writing or customizing your own linker script:

- The PowerPC processor linker is built with default linker scripts. These scripts:
 - Define the start address to be `0xFFFF0000`. To specify a different start address, you can convey it to the linker using either a command line assignment or an adjustment to the linker script.
 - Assume a contiguous memory starting at address `0xFFFF0000`.
 - Define `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 Libgen tool. The script
- When writing or customizing your own linker script:
 - Ensure that the `.boot` section starts at `0xFFFFFFF0`. Upon power-up, the PowerPC processor starts execution from the location `0xFFFFFFF0`.
 - The `_end` variable is defined after the `.boot0` section definition. This section is a jump to the start of the `.boot0` section. The jump is defined to be 24 bits; hence the `.boot` and `.boot0` sections should not be more than 24 bits apart. On the PowerPC 440 processor, the `.boot0` section has a fixed location of `0xFFFFF000`.
 - 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.
 - Because 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. Note that the `.bss` section boundary does not include either stack or heap.
 - 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.
 - For the PowerPC 405 processor, ensure that the `.vectors` section is aligned on a 64K boundary. The PowerPC 440 processor does not require any special alignment on the `.vectors` section. Include this section definition only when your program uses interrupts and/or exceptions.
 - Each (physical) region of memory must use a separate program header. Two discontinuous regions of memory cannot share a program header.
 - ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap.) The standard CRT provided assumes a single `.bss`

section that is initialized to zero. If there are multiple `.bss` sections, this CRT will not work. You must write your own CRT that initializes the `.bss` sections.

Startup Files

When the compiler forms an executable, it includes pre-compiled startup and end files in the final link command. Startup files set up the language and the platform environment before your application code can execute. Startup files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers as required.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions such as C++ constructors.
- Initialize the hardware sub-system. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for and invoke the main procedure.

End files include code that must execute after your program is finished. End files typically:

- Invoke language cleanup functions, such as C++ destructors.
- Clean up the hardware subsystem. For example, if the program is being profiled, clean up the profiling subsystem.

Table 9-12 lists the register initialization in the C runtime files.

Table 9-12: Register Initialization in C-Runtime Files

| Register | Value | Description |
|-----------------|-------------------------|--|
| r1 | <code>_stack-8</code> | Stack pointer register initializes the bottom of the allocated stack, offset by 16 bytes. The 16 bytes can be used for passing in arguments. |
| r2 | <code>_SDA2_BASE</code> | <code>_SDA2_BASE</code> is the read-only small data anchor address. |
| r13 | <code>_SDA_BASE</code> | <code>_SDA_BASE</code> is the read-write small data anchor address. |
| Other registers | Undefined | Other registers do not have defined values. |

The following subsection describes the initialization files. This information is for advanced users who want to change or understand the startup code of their application.

Initialization File Description

The PowerPC processor compiler uses four different CRT files: `xil-crt0.o`, `xil-pgcrt0.o`, `xil-sim-crt0.o`, and `xil-sim-pgcrt0.o`. The various CRT files perform the following steps, with exceptions as described.

1. Invoke the function `_cpu_init`. This function is provided by the board support package library and contains processor architecture specific initialization.
2. Clear the `.bss` memory regions to zero.
3. Set up registers. Refer to Table 9-12 for details.
4. Initialize the timer base register to zero.
5. Optionally, enable the floating point unit bit in the MSR.
6. Invoke the C++ language and constructor initialization function (`_init`).

7. Invoke `main`.
8. Invoke C++ language destructors (`_fini`).
9. Transfer control to `exit`.

Start-up File Descriptions

xil-crt0.o

This is the default initialization file used for programs that are to be executed in standalone mode, with no other special requirements. This performs all the common actions described above.

xil-pgcrt0.o

This initialization file is used when the application is to be profiled in a software-intrusive manner. In addition to all the common CRT actions described, it also invokes the `_profile_init` routine before invoking `main`. This initializes the software profiling library before your code executes. Similarly, upon exit from `main`, it invokes the `_profile_clean` routine, which cleans up the profiling library.

xil-sim-crt0.o

This initialization file is used when the application is compiled with the `-mno-clearbss` switch. It performs all the common CRT setup actions, except that it does not clear the `.bss` section to zero.

xil-sim-pgcrt0.o

This initialization file is used when the application is compiled with the `-mno-clearbss` switch. It performs all the common CRT setup actions, except that it does not clear the `.bss` section to zero. It also invokes the `_profile_init` routine before invoking `main`. This initializes the software profiling library before your code executes. Similarly, upon exit from `main`, it invokes the `_profile_clean` routine, which cleans up the profiling library.

Other files

The compiler also uses standard start and end files for C++ language support: `ecrti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections. The PowerPC default and generated linker scripts also make `boot.o` a startup file. This file is present in the standalone package for PowerPC (405 and 440) processors.

Modifying Startup Files

The initialization files are distributed in both pre-compiled and source form with EDK. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the PowerPC compiler can be found in the `<XILINX_EDK>/sw/lib/ppc405/src` directory, where `<XILINX_EDK>` is the EDK installation area.

Any time you need a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, they can be assembled into `.o` files and placed in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory-name` command line option while invoking `powerpc-eabi-gcc`. To prevent the default startup files being used, add `-nostartfiles` on final compile line.

Note: The compiler standard CRT files for C++ support, such as `ecrti.o` and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You might need to bring them in on your final link command if your code uses constructors and destructors.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you can eliminate all sources of overhead. This section documents how to remove the overhead of invoking the C++ constructor or destructor code in a C program that does not need them. You might be able to save approximately 500 bytes of code space by making these modifications.

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, you need to copy over the particular version of `xil-crt.s` that suits your application. For example, if your application is being profiled, copy `xil-pgcrt0.s` from the installation area, and modify the CRT file to remove the following lines:

```
/* Call _init */
bl _init

and

/* Invoke the language cleanup functions */
bl _fini
```

This avoids referencing the extra code that is usually pulled in for constructor and destructor handling, and reducing code size.

2. Either compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
3. Add the `-nostartfiles` switch to the compiler. Add the `-B <directory>` switch if you have chosen to assemble the files in a particular folder.
4. Compile your application.

Modifying Startup Files for Bootstrapping an Application

If your application is going to be loaded from a bootloader, you might not want to overwrite the processor reset vector of the bootloader with that of your application. This re-executes the bootloader on a processor reset instead of your application. To achieve this, your application must not bring in `boot.o` as a startup file. Unlike other compiler startup files, `boot.o` is not explicitly linked in by the compiler. Instead, the default linker scripts and the tools for generating the linker scripts specify `boot.o` as a startup file. You must remove the `STARTUP` directive in such linker scripts. You must also modify the `ENTRY` directive to be `_start` instead of `_boot`.

Compiler Libraries

The `powerpc-eabi-gcc` compiler requires the GNU C standard library and the GNU math library.

Precompiled versions of these libraries are shipped with EDK. These libraries are located in `$XILINX_EDK/gnu/powerpc-eabi/platform/powerpc-eabi/lib`.

Various subdirectories under this top level library directory contain customized versions of the libraries for a particular configuration. For instance, the `/double` directory contains the version of libraries for use with a double precision FPU, whereas the `/440` subdirectory contains the version of libraries suited for use with PowerPC 440 processor.

Thread Safety

The C and math libraries for the PowerPC processor distributed with EDK are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are *not* thread-safe and will cause unrecoverable errors in the system at run-time. Use appropriate mutual exclusion mechanisms when using the EDK libraries in a multi-threaded environment.

Command Line Arguments

PowerPC processor programs cannot take in command-line arguments. The command-line arguments, `argc` and `argv`, are initialized to zero by the C runtime routines.

Other Notes

C++ Code Size

The GCC toolchain combined with the latest open source C++ standard library (`libstdc++-v3`) might be found to generate large code and data fragments as compared to an equivalent C program. A significant portion of this overhead comes from code and data for exception handling and runtime type information. Some C++ applications do not require these features.

To remove the overhead and optimize for size, use the `-fno-exceptions` and/or the `-fno-rtti` switches. This is recommended only for advanced users who know the requirements of their application and understand these language features. Refer to the GCC manual for more specific information on available compiler options and their impact.

C++ programs might have more intensive dynamic memory requirements (stack and heap size) due to more complex language features and library routines.

Many of the C++ library routines can request memory to be allocated from the heap. Review your heap and stack size requirements for C++ programs to ensure that they are satisfied.

C++ Standard Library

The C++ standard defines the C++ standard library. A few of these platform features are unavailable on the default Xilinx EDK software platform. For example, file I/O is supported in only a few well-defined `STDIN/STDOUT` streams. Similarly, locale functions, thread-safety, and other such features may not be supported.

Note: The C++ standard library is not built for a multi-threaded environment. Common C++ features such as `new` and `delete` are not thread-safe. Please use caution when using the C++ standard library in an operating system environment.

For more information on the GNU C++ standard library, refer to the documentation available on the GNU website. A link to the documentation is in [Appendix E, Additional Resources](#).

Position Independent Code (Relocatable Code)

The MicroBlaze and PowerPC processor compilers support the `-fPIC` switch to generate position independent code. The PowerPC processor compiler supports the `-mrelocatable` switches to generate a slightly different form of relocatable code.

While both these features are supported in the Xilinx compiler, they are not supported by the rest of the libraries and tools, because EDK only provides a standalone platform. No loader or debugger can interpret relocatable code and perform the correct relocations at runtime. These independent code features are not supported by the Xilinx libraries, startup files, or other tools. Third-party OS vendors could use these features as a standard in their distribution and tools.

Other Switches and Features

Other switches and features might not be supported by the Xilinx EDK compilers and/or platform, such as `-fprofile-arcs`. Some features might also be experimental in nature (as defined by open source GCC) and could produce incorrect code if used inappropriately. Refer to the GCC manual for more information on specific features. A link to the document is in [Appendix E, “Additional Resources.”](#)

Xilinx Microprocessor Debugger (XMD)

The Xilinx® Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using the PowerPC® (405 or 440) processor, the MicroBlaze™ processor, or the Dual ARM Cortex-A9 MPCore processor. You can use it to debug programs on MicroBlaze, PowerPC 405, or Cortex A9 processors running on a hardware board, cycle-accurate Instruction Set Simulator (ISS).

XMD provides a Tool Command Language (Tcl) interface. This interface can be used for command line control and debugging of the target as well as for running complex verification test scripts to test a complete system.

XMD supports GNU Debugger (GDB) remote TCP protocol to control debugging of a target. Some graphical debuggers use this interface for debugging, including the PowerPC processor GDB (`powerpc-eabi-gdb`), the MicroBlaze GDB (`mb-gdb`), and the Cortex A9 GDB (`arm-xilinx-eabi-gdb`), along with the Software Development Kit (SDK), the EDK, Eclipse-based software tool. In either case, the debugger connects to XMD running on the same computer or on a remote computer on the network.

XMD reads Xilinx Microprocessor Project the (XMP) system file or system.xml file, whichever is available, to gather information about the hardware system on which the program is debugged. The information is used to perform memory range tests, determine MicroBlaze to Microprocessor Debug Module (MDM) connectivity for faster download speeds, and perform other system actions.

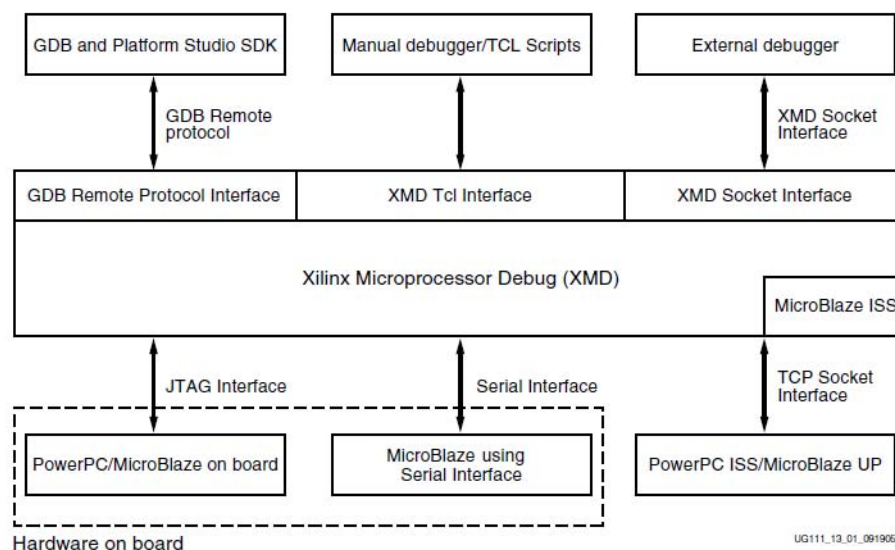


Figure 10-1: XMD Targets

XMD Usage

```
xmd [-h] [-help] [-hw <hardware_specification_file>] [-ipcpport  
<port_number>] [-nx] [-opt <optfile>] [-v] [-xmp <xmpfile>]  
[-tcl <tcl_file> <tcl_args>]
```

XMD Options

Help

Displays the usage menu and quits.

Command: **-h**, **-help**

Hardware Specification File

Specifies the XML file that describes the hardware components.

Command: **-hw** <hardware_specification_file>

Port Number

Starts the XMD server at <portnum>. Internal XMD commands can be issued over this TCP Port. If [<port_number>] is not specified, a default value, 2345, is used.

Command: **-ipcpport** <port_number>

No Initialization File

Does not source xmd.ini file on startup.

Command: **-nx**

Option File

Specifies the option file to use to connect to target. The option file contains the XMD connect command to target.

Command: **-opt** <connect_option_file>

Tcl File

Specifies the XMD Tcl script to run.

The <tclargs> are arguments to the Tcl script. This Tcl file is sourced from XMD. XMD quits after executing the script. No other option can follow **-tcl**.

Command: **-tcl** <tclfile> <tclarg>

Version

Displays the version and then quits.

Command: **-v**

XMP File

Specifies the XMP file to load.

Command: **-xmp** <xmpfile>

Upon startup, XMD does the following:

- If an XMD Tcl script is specified, XMD executes the script, then quits.
- If an XMD Tcl script is *not* specified, XMD starts in *interactive mode*. In this case, XMD:
 - a. Creates source `${HOME}/.xmдрс` file. You can use this configuration file to form custom Tcl commands using XMD commands:

| | |
|----------|---|
| -hw | loads the XML file. When -nx option is not given, sources the xmd.ini file if present in the current directory. |
| -opt | Uses Connect option file to connect to processor target. |
| -ipcport | opens XMD socket server |
| -xmp | loads system XMP file. |
 - b. Displays the `XMD%` prompt. From the `XMD Tcl` prompt, you can use XMD commands for debugging, as described in the next section, [XMD Command Reference, page 136](#).

XMD Console

The XMD console is a standard Tcl console, where you can run any available Tcl commands. Additionally, the XMD console provides command editing convenience, such as file and command name auto-fill and command history.

The available Tcl commands on which you can use auto-fill are defined in the `<EDK_Install_Area>/data/xmd/cmdlist` file. The command history is stored in `$HOME/.xmдcmdhistory`.

To use different files for available command names and command history, you can use environment variables `$XILINX_XMD_CMD_LIST` and `$XILINX_XMD_CMD_HISTORY` to overwrite the defaults.

XMD Command Reference

XMD User Command Summary

The following is a summary of XMD commands. To go to a description for a given command, click on its name.

| | |
|--|------------------------------------|
| bpl | read_uart |
| bpr | rrd |
| bps | rst |
| close_terminal | rwr |
| con | run |
| connect | safemode [options] |
| cstp | srrd |
| data_verify | stackcheck |
| debugconfig | state |
| dis | stats |
| disconnect | stop |
| dow | stp |
| dow -data | targets |
| elf_verify | terminal |
| fpga -f <bitstream> | tracestart |
| mrd <address> [<number of words half words bytes> {w h b}] | tracestop |
| mrd_var | watch |
| mwr | xload |
| profile | |

XMD User Commands

bpl

Lists breakpoints and watchpoints.

| Options | Example Usage |
|---------|---------------|
| bpl | bpl |

bpr

Removes breakpoints and watchpoints

| Options | Example Usage |
|--|----------------------------------|
| bpr {all <bp id> <address> <function>} | bpr 0x400 bpr main bpr all |

bps

Sets a software or hardware breakpoint at <address> or start of <function name>. The last downloaded ELF file is used for function lookup. Defaults to software breakpoint.

| Options | Example Usage |
|--|--------------------------|
| bps {<address> <function_name>} {sw hw} | bps 0x400 bps main hw |

close_terminal

Closes the terminal server opened by the terminal command and the MDM Uart target connection.

| Options | Example Usage |
|----------------|----------------|
| close_terminal | close_terminal |

con

Continues from current PC or optionally specified <Execute Start Address>.

If -block option is specified, the command returns when the Processor stops on breakpoint or watchpoint.

A -timeout value can be specified to prevent indefinite blocking of the command.

The -block option is useful in scripting.

| Options | Example Usage |
|---|------------------|
| con [<Execute Start Address> [-block [-timeout <Seconds>]] | con con 0x400 |

connect

Connects to *<target_type>*. Valid target types are: mb, ppc, and mdm. For additional information, refer to [“Connect Command Options” on page 153](#).

| Options | Example Usage |
|---|---|
| <code>connect <target_type(s)></code> | <code>connect mb mdm</code> <code>connect ppc</code> |

cstp

Steps through the specified number of cycles. This is supported only on ISS targets.

| Options | Example Usage |
|--|---|
| <code>cstp <number of cycles></code> | <code>cstp</code> <code>cstp 10</code> |

data_verify

Verify if the *<Binary filename>* is downloaded correctly to the target at *<Load Address>*.

| Options | Example Usage |
|---|---|
| <code>data_verify <binary_filename></code> <code><load_address></code> | <code>data_verify</code> <code>system.dat 0x400</code> |

debugconfig

Configures the debug session for the target. For additional information, refer to [“Configure Debug Session” on page 175](#).

| Options | Example Usage |
|--|--|
| <code>debugconfig</code> | <code>debugconfig</code> |
| <code>debugconfig -step_mode</code> <code>enable_interrupt</code> | <code>debugconfig -step_mode</code> <code>{disable_interrupt </code> <code>enable_interrupt}</code> |
| <code>debugconfig -memory_datawidth_</code> <code>matching enable</code> | <code>debugconfig</code> <code>-memory_datawidth_matching</code> <code>{disable enable}</code> |
| <code>debugconfig -reset_on_run system</code> <code>enable</code> | <code>debugconfig -reset_on_run</code> <code>{system enable processor</code> <code>enable disable}</code> |
| <code>debugconfig -reset_on_data_dow</code> <code>processor enable</code> | <code>debugconfig -reset_on_data_dow</code> <code>{system enable processor</code> <code>enable disable}</code> |

dis

Disassemble instruction. Supported on the MicroBlaze target only.

| Options | Example Usage |
|---|---------------------------|
| <code>dis [<address in hex>] [<number of words>]</code> | <code>dis 0x400 10</code> |

disconnect

Disconnects from the current processor target, closes the corresponding GDB server, and reverts to the previous processor target, if any.

| Options | Example Usage |
|---|---------------------------|
| <code>disconnect <target id></code> | <code>disconnect 0</code> |

dow

Downloads the given ELF or data file (with the -data option) onto the memory of the current target. If no address is provided along with the ELF file, the download address is determined from the ELF file by reading its headers.

Only those segments of the ELF file that are marked LOAD are written to memory.

| Options | Example Usage |
|--|---------------------------------------|
| <code>dow <filename.elf></code> | <code>dow executable.elf</code> |
| <code>dow <PIC filename.elf> <load_address></code> | <code>dow executable.elf 0x400</code> |

dow -data

If an address is provided with the ELF file (on MicroBlaze targets only), it is treated as Position Independent Code (PIC code) and downloaded at the specified address. Also, the R20 Register is set to the start address according to the PIC code semantics.

The R20 Register is reserved for storing a pointer to the Global Offset Table (GOT) in Position Independent Code (PIC). It is non-volatile in non-PIC code and must be saved across function calls.

When an ELF file is downloaded, the command does a reset, stops the processor at the reset location by using software breakpoints, and loads the ELF program to the memory. The reset is done to ensure that the system is in a known good state. The reset behavior can be configured using the following commands:

```
debugconfig -reset_on_run
{system enable | processor enable | disable}
```

```
debugconfig -reset_on_data_dow
{system enable | processor enable | disable}
```

Refer to the [“Configure Debug Session” on page 175](#)

| Options | Example Usage |
|---|---|
| <code>dow -data <binary_filename> <load_address></code> | <code>dow -data system.dat 0x400</code> |

elf_verify

Verify if the executable.elf is downloaded correctly to the target. If ELF file is not specified, it uses the most recent ELF file downloaded on the target.

| Options | Example Usage |
|--|--|
| <code>elf_verify [<filename.elf>]</code> | <code>elf_verify executable.elf</code> |

fpga -f <bitstream>

Loads the FPGA device bitstream. Optionally specify the cable, JTAG configuration, and debug device options.

For additional information, refer to [“Connect Command Options” on page 153](#).

| Options | Example Usage |
|---|---|
| <code>fpga -f <bitstream></code> | <code>fpga -f download.bit</code> |
| <code>fpga -f <bitstream> [-cable <cable_options>] [-configdevice <configuration_options>] [-debugdevice <device_name>]</code> | <code>fpga -f download.bit -cable type xilinx_parallel</code> |

mrd <address> [<number of words|half words|bytes> {w|h|b}]

Reads <num> memory locations starting at address. Defaults to a word (w) read.

If <Global Variable Name> name is specified, reads memory corresponding to global variable in the previously downloaded ELF file.

| Options | Example Usage |
|--|-----------------------------|
| <code>mrd <address> [<number of words half words bytes> {w h b}]</code> | <code>mrd 0x400</code> |
| <code>mrd <Global Variable Name></code> | <code>mrd 0x400 10</code> |
| | <code>mrd 0x400 10 h</code> |

mrd_var

Reads memory corresponding to global variable in the <filename.elf> or in a previously downloaded ELF file.

| Options | Example Usage |
|--|---|
| <code>mrd_var <Global Variable Name> <filename.elf></code> | <code>mrd_var global_var1 executable.elf</code> |

mwr

Writes to num memory locations starting at <address> or <Global Variable Name>. Defaults to a word (w) write

| Options | Example Usage |
|--|--|
| mwr <address> <values> [<number of words/half words/bytes> {w h b}] | mwr 0x400 0x12345678 mwr 0x400 0x1234 1 h |
| mwr <Global Variable Name> <values> [<number of words/half words/bytes> {w h b}] | mwr 0x400 {0x12345678 0x87654321} 2 |

profile

Writes a Profile output file, which can be interpreted by mb-gprof (for MicroBlaze), powerpc-eabi-gprof (for PowerPC), or arm-xilinx-eabi-gprof (for Cortex A9) to generate profiling information.

Specify the profile configuration sampling frequency in Hz, histogram bin size, and memory address for collecting profile data.

For details about Profiling using XPS, search on “Profiling” in the *Platform Studio Online Help*.

| Options | Example Usage |
|-------------------------------------|-----------------------|
| profile [-o <GMON Output filename>] | profile -o gproff.out |

read_uart

The read_uart start command redirects the output from the mdm UART interface to an optionally specified TCL channel (TCL Channel ID).

The read_uart stop command stops redirection.

A TCL channel represents an open file or a socket connection. The TCL channel should be opened prior to using the read_uart command, using appropriate TCL commands.

| Options | Example Usage |
|---|--|
| read_uart [{start stop}] [<TCL Channel ID>] | read_uart start read_uart stop read_uart start \$channel_id |

rrd

Reads all registers or reads <reg_num> register.

| Options | Example Usage |
|-----------------|--|
| rrd [<reg_num>] | rrd rrd r1 (or) rrd R1 rrd 1 |

rst

Resets the system.

If the `-processor` option is specified, the current processor target is reset.

If the processor is not in a “Running” state (use the `state` command), then the processor will be stopped at the processor reset location on reset.

| Options | Example Usage |
|-------------------------------|--|
| <code>rst [-processor]</code> | <code>rst</code> <code>rst - processor</code> |

rwr

Registers writes from a `<register_number>`, `<register_name>`, or `<hex_value>`.

| Options | Example Usage |
|--|---------------------------|
| <code>rwr <register_number> <register_name> <Hex_value></code> | <code>rwr pc 0x400</code> |

run

Runs program from the program start address. The command does a “reset”, stops the processor at the reset location by using breakpoints and loads the ELF program data sections to the memory. Loading the ELF program data sections ensures that the static variables are properly initialized and “reset” is done so the system is in a “known good” state.

The “reset” behavior can be configured using the following commands:

```
debugconfig -reset_on_run
{system enable | processor enable | disable}
```

```
debugconfig -reset_on_data_dow
{system enable | processor enable | disable}
```

Refer to [“Configure Debug Session”](#) on page 175.

| Options | Example Usage |
|------------------|------------------|
| <code>run</code> | <code>run</code> |

safemode [options]

Enables, disables, configures, and specifies files to be read in safemode. The following safemode options are available.

| Options | Description | Example Usage |
|---|--|---|
| <code>safemode [-config <mode> <exception_mask>]</code> | Changes the current safemode configuration. | <code>safemode -config <mode> <exception_mask></code> |
| <code>safemode [{on off}]</code> | Enables and disables safemode. | <code>safemode on</code> <code>safemode off</code> |
| <code>safemode [-config <exception_id> <exception_addr>]</code> | Changes exception handler ID and/or addresses. | <code>safemode -config <exception_id></code> |
| <code>safemode[-info]</code> | Displays the safemode information. | <code>safemode -info</code> |
| <code>safemode [-elf <elf_file>]</code> | Specifies the ELF file to be debugged. | <code>safemode -elf <elf_file></code> |

srrd

Reads special purpose registers or reads `<reg_name>` register.

| Options | Example Usage |
|---|----------------------|
| <code>srrd</code> | <code>srrd</code> |
| <code>srrd [<register_name>]</code> | <code>srrd pc</code> |

stackcheck

Gives the stack usage information of the program running on the current target. The most recent ELF file downloaded on the target is taken into account for stack check.

| Options | Example Usage |
|-------------------------|-------------------------|
| <code>stackcheck</code> | <code>stackcheck</code> |

state

When no target id is specified, the command displays the current state of all targets.

When a `<target_id>` is specified, state of that target is displayed.

When `-system <system_id>` is specified the current state of all the targets in the system is displayed.

| Options | Example Usage |
|--|--|
| <code>state</code> | <code>state</code> |
| <code>state [<target_id>]</code> | <code>state <target_id></code> |
| <code>state -system <system_id></code> | <code>state -system <system_id></code> |

stats

Displays execution statistics for the ISS target. The *<filename>* is the trace output from trace collection.

| Options | Example Usage |
|-----------------------------------|-----------------|
| stats | stats |
| stats [<i><filename></i>] | stats trace.txt |

stop

Stops the target. For MicroBlaze, if the program is stalled at memory or FSL access, it is stopped forcibly.

| Options | Example Usage |
|---------|---------------|
| stop | stop |

stp

Steps through the specified number of instructions.

| Options | Example Usage |
|---|---------------|
| stp | stp |
| stp <i><number of instructions></i> | stp 10 |

targets

Lists information about all current targets or changes the current target.

| Options | Example Usage |
|--|-------------------|
| targets | targets |
| targets <i><target_id></i> | targets 0 |
| targets -system <i><system_id></i> | targets -system 1 |

terminal

JTAG-based hyperterminal to communicate with mdm UART interface. The UART interface should be enabled in the mdm.

If the `-jtag_uart_server` option is specified, a TCP server is opened at `<port_no>`. Use any hyperterminal utility to communicate with opb_mdm UART interface over TCP sockets.

The `<port_number>` default value is 4321.

The `<baudrate>` determines the rate at which the JTAG UART port reads the data. This option can have the values `low`, `med`, or `high`. The default setting is `med`.

Increasing the baud rate might affect other debug operations, because XMD is busy polling for data on the JTAG UART port.

| Options | Example Usage |
|--|---|
| <code>terminal</code> | <code>terminal</code> |
| <code>terminal [-jtag_uart_server] [<port_number>] [<baudrate>]</code> | <code>terminal -jtag_uart_server 4321 high</code> |

tracestart

Starts collecting instruction and function trace information to `<filename>`.

Trace collection can be stopped and started any time the program runs.

`<filename>` is specified on first tracestart only.

`<pc_trace_filename>` defaults to `isstrace.out`.

`<func_trace_filename>` defaults to `fntrace.out`.

Note: This is supported on ISS targets only.

| Options | Example Usage |
|---|---|
| <code>tracestart</code> | <code>tracestart</code> |
| <code>tracestart [<pc_trace_filename> [-function_name <func_trace_filename>]</code> | <code>tracestart pctrace.txt</code> <code>tracestart pctrace.txt -function_name fntrace.txt</code> |

tracestop

Stops collecting trace information. The `done` option signifies the end of tracing.

Note: This is supported on ISS targets only.

| Options | Example Usage |
|-------------------------------|-----------------------------|
| <code>tracestop</code> | <code>tracestop</code> |
| <code>tracestop [done]</code> | <code>tracestop done</code> |

watch

Sets a read or write watchpoint at *address*. If the value compares to data, stop the processor. Address and Data can be specified in hex 0x format or binary 0b format.

Don't care values are specified using X.

Addresses can be of contiguous range only.

Default value of data is 0XXXXXXXX. That is, it matches any value.

For the PowerPC processor, only absolute values are supported.

| Options | Example Usage |
|---|---|
| <code>watch {r w} <address> [<data>]</code> | <pre>watch r 0x400 0x1234 watch r 0x40X 0x12X4 watch r 0b01000000XXXX 0b00010010XXXX0100 watch r 0x40X</pre> |

xload

Loads hardware specification XML file. XMD reads the XML file to gather instruction and data memory address maps of the processor. This information is used to verify the program and data downloaded to processor memory. XPS generates the hardware specification file during the Export to SDK process.

| Options | Example Usage |
|--|----------------------------------|
| <code>xload hw <hw_spec_file></code> | <code>xload hw system.xml</code> |

Special Purpose Register Names

MicroBlaze Special Purpose Register Names

The following special register names are valid for MicroBlaze processors:

| | | | | |
|-------|-------|------|------|-----|
| pc | msr | ear | esr | zpr |
| fsr | btr | pvr0 | pvr1 | zpr |
| pvr2 | pvr3 | pvr4 | pvr5 | zpr |
| pvr6 | pvr7 | pvr8 | pvr9 | |
| pvr10 | pvr11 | edr | pid | |

For additional information, descriptions, and usage of MicroBlaze special register names, refer to the "Special Purpose Registers" section of the "MicroBlaze Architecture" chapter in the *MicroBlaze Processor Reference Guide*. A link to the document is supplied in [Appendix E, Additional Resources](#).

Note: When MicroBlaze is debugged in XMDSTUB mode, only PC and MSR registers are accessible.

PowerPC 405 Processor Special Purpose Register Names

[Table 10-1](#) lists the special register names that are valid for PowerPC 405 processors:

Table 10-1: Special Register Names for PowerPC 405 Processors

| | | | | | | |
|-------|-----|-----|-----|--------|-------|--------|
| ccr0 | f0 | f11 | f22 | iac1 | pvr | su0r |
| cr | f1 | f12 | f23 | iac2 | sgr | tbl |
| ctr | f2 | f13 | f24 | iac4 | sler | tbu |
| dac1 | f3 | f14 | f25 | iccr | sprg0 | tcr |
| dac2 | f4 | f15 | f26 | icdbdr | sprg1 | tsr |
| dbcr0 | f5 | f16 | f27 | lr | sprg2 | usprg0 |
| dbcr1 | f6 | f17 | f28 | msr | sprg3 | xer |
| dbsr | f7 | f18 | f29 | pc | sprg4 | zpr |
| dccr | f8 | f19 | f30 | pid | sprg5 | su0r |
| dcwr | f9 | f20 | | pit | sprg6 | tbl |
| dear | f10 | f21 | | iac1 | sprg7 | tbu |
| dvc1 | | | | iac2 | srr0 | |
| dvc2 | | | | | srr1 | |
| esr | | | | | srr2 | |
| evpr | | | | | srr3 | |

Note: XMD always uses 64-bit notation to represent the Floating Point Registers (f0-f31). In the case of a Single Precision floating point unit, the 32-bit Single Precision value is extended to a 64-bit value.

For additional information about PowerPC 405 processor special register names, refer to the *PowerPC 405 Processor Block Reference Guide*. A link to the document is supplied in [Appendix E, Additional Resources](#).

PowerPC 440 Processor Special Purpose Register Names

[Table 10-2](#) lists the special register names that are valid for PowerPC 440 processors:

Table 10-2: PowerPC 440 Processor Special Purpose Register Names

| | | | | | |
|---------|---------|---------|---------|---------|--------|
| pc | msr | cr | lr | ctr | xer |
| fpscr | pvr | sprg0 | sprg1 | sprg2 s | prg3 |
| srr0 | srr1 | tbl | tbu | icbdr | esr |
| dear | ivpr | tsr | tr | dec | csrr0 |
| csrr1 | dbsr | dbcr0 | iac1 | iac2 | dac1 |
| dac2 | pir | rstcfg | mmucr | pid | ccr1 |
| dbdr | ccr0 | dbcr1 | dvc1 | dvc2 | iac3 |
| iac4 | dbcr2 | sprg4 | sprg5 | sprg6 | sprg7 |
| decar | usprg0 | ivor0 | ivor1 | ivor2 | ivor3 |
| ivor4 | ivor5 | ivor6 | ivor7 | ivor8 | ivor9 |
| ivor10 | ivor11 | ivor12 | ivor13 | ivor14 | ivor15 |
| inv0 | inv1 | inv2 | inv3 | itv0 | itv1 |
| itv2 | itv3 | dnv0 | dnv1 | dnv2 | dnv3 |
| dtv0 | dtv1 | dtv2 | dtv3 | dvlim | ivlim |
| dcdbtrl | dcdbtrh | icdbtrl | icdbtrh | mcsr | mcsrr0 |
| mcsrr1 | f0 | f1 | f2 | f3 | f4 |
| f5 | f6 | f7 | f8 | f9 | f10 |
| f11 | f12 | f13 | f14 | f15 | f16 |
| f17 | f18 | f19 | f20 | f21 | f22 |
| f23 | f24 | f25 | f26 | f27 | f28 |
| f29 | f30 | f31 | | | |

Note: XMD always uses 64-bit notation to represent the Floating Point Registers (f0-f31). In the case of a Single Precision floating point unit, the 32-bit Single Precision value is extended to a 64-bit value.

For additional information about PowerPC 440 processor special register names, refer to the “Register Set Summary” section of the *PowerPC 440 Processor Block Reference Guide*. A link to the document is supplied in [Appendix E, Additional Resources](#).

Cortex A9 Special Purpose Registers

Cortex A9 has sets of coprocessor registers. The different groups are listed here.

Table 10-3: Cortex A9 Special Purpose Register Names

| | | |
|------|-----|-----|
| ctrl | dma | tcn |
| id | etc | vfp |

The examples section of this chapter has details about these registers. Refer also to the ARM documentation for additional information.

XMD Reset Sequence

When the `rst` command is issued, XMD resets the processor or system to bring them back to known states. Following are the sequences of operation that `rst` does for each type of processors.

PowerPC 405 Processors

1. Disable virtual addressing.
2. If reset address (`0xFFFFFFFFC`) is writable and not on OCM, write a branch-to-self instruction at the reset location. If the reset address is not writable, XMD cannot put the processor into a known state.
3. Set DBCR0 to `0x81000000`.
4. Issue reset signal (either system reset or processor reset) through JTAG Debug Control Register (DCR). The processor starts running.
5. Stop the processor.
6. Restore the original instruction at reset address.

PowerPC 440 Processors

1. Set DBCR0 to `0x81000000`.
2. Set register MMUCR to 0.
3. Set DBCR1 and DBCR2 to 0.
4. Set up TLB so that virtual addresses are the same as real addresses.
5. Synchronize with the shadow TLB.
6. If reset address (`0xFFFFFFFFC`) is writable, write a branch-to-self instruction at the reset location. If the reset address is not writable, XMD cannot put the processor into a known state.
7. Issue reset signal (either system reset or processor reset) through JTAG DCR. The processor starts running.
8. Stop the processor.
9. Restore the original instruction at reset address.

MicroBlaze

1. Set a hardware breakpoint at reset location (`0x0`).
2. Issue reset signal (system reset or processor reset). The processor starts running.
3. After processor is stopped at reset location, remove the breakpoint.

Recommended XMD Flows

The following are the recommended steps in XMD for debugging a program and debugging programs in a multi-processor environment, and running a program in a debug session.

Debugging a Program

To debug a program:

1. Connect to the processor.
2. Download the ELF file.
3. Set the required breakpoints and watchpoints.
4. Start the processor execution using the **con** command or step through the program using the **stp** command.
5. Use the **state** command to check the processor status.
6. Use **stop** command to stop the processor if needed.
7. When the processor is stopped, read and write registers and memory.
8. To re-run the program, use the **run** command.

Debugging Programs in a Multi-Processor Environment

For debugging programs in a multi-processor environment:

1. Connect to processor1.
2. Use the **debugconfig** command to configure the reset behavior, which depends on your system architecture. Refer to the [“Configure Debug Session” on page 175](#).
3. Download the ELF file.
4. Set the required breakpoints and watchpoints.
5. Start the processor execution using the **con** command or step through the program using the **stp** command.
6. Connect to processor2.
7. Use the **debugconfig** command to configure the reset behavior, which depends on your system architecture. Refer to the [“Configure Debug Session” on page 175](#).
8. Download the ELF file.
9. Set the required Breakpoints and Watchpoints.
10. Start the processor execution using the **con** command or step through the program using the **stp** command.
11. Use the **targets** command to list the targets in the system. Each target is associated with a *<target id>*; an asterisk (*) marks the active target.
12. Use **targets <target id>** to switch between targets.
13. Use the **state** command to check the processor status.
14. Use the **stop** command to stop the processor.
15. When the processor is stopped, read and write the registers and memory.
16. To re-run the program use the **run** command.

Running a Program in a Debug Session

1. Connect to the processor.
2. Download the ELF file.
3. Set the Breakpoint at the `<exit>` function.
4. Start the processor execution using the **con** command.
5. Use the **state** command to check the processor status.
6. Use the **stop** command to stop the processor.
7. When the processor is stopped, read and write the registers and memory.
8. To re-run the program use the **run** command.

Using Safemode for Automatic Exception Trapping

XMD allows you to trap exceptions in your program when errors occur. Such errors include the execution of illegal instructions and bus errors. Use the following steps:

1. Download the program.
2. Run the **safemode on** command.
3. Start the program with the **con** command.

The program stops when an exception occurs. This feature is more useful when working with the GUI debugger (either Insight GDB or SDK).

- When using SDK, check the **Enable Safemode** checkbox box in the Initialization tab before running the program.
- When using GDB, download the program and run the **safemode on** command in XMD console before running the program in GDB.

When the exception occurs the program stops and the GUI shows the line of code that triggered the exception.

Processor Default Exception Settings

Table 10-4 and Table 10-5, page 152 show the factory default settings for exception trapping settings by processor types:

Table 10-4: PowerPC Processor Exception Settings

| Exception_id | Trap | Exception_Name |
|--------------|------|---|
| 0 | No | External critical-interrupt exception. |
| 1 | Yes | External bus error exception. |
| 2 | Yes | Data storage exception. |
| 3 | Yes | Instruction storage exception. |
| 4 | No | External noncritical-interrupt exception. |
| 5 | No | Unaligned data access exception. |
| 6 | Yes | Illegal op-code exception. |
| 7 | Yes | FPU non-available exception. |
| 8 | No | System call instruction. |

Table 10-4: PowerPC Processor Exception Settings (Cont'd)

| Exception_id | Trap | Exception_Name |
|--------------|------|--|
| 9 | Yes | APU non-available exception. |
| 10 | No | Time out exception on programmable interval timer. |
| 11 | No | Time out exception on fixed interval timer. |
| 12 | No | Time out exception on watchdog timer. |
| 13 | No | Data TLB miss exception. |
| 14 | No | Instruction TLB miss exception. |
| 15 | No | Debug event exception. |
| 16 | Yes | Assertion failure. |
| 17 | Yes | Program exit. |

Table 10-5: MicroBlaze Exception Settings

| Exception_id | Trap | Exception_Name |
|--------------|------|-----------------------------------|
| 0 | Yes | Fast Simplex Link exception. |
| 1 | No | Unaligned data access exception. |
| 2 | Yes | Illegal op-code exception. |
| 3 | Yes | Instruction bus error exception. |
| 4 | Yes | Data bus error exception. |
| 5 | Yes | Divide by zero exception. |
| 6 | Yes | Floating point unit exception. |
| 7 | Yes | Privileged instruction exception. |
| 8 | Yes | Data storage exception. |
| 9 | Yes | Instruction storage exception. |
| 10 | Yes | Data TLB miss exception. |
| 11 | Yes | Instruction TLB miss exception. |
| 12 | Yes | Assertion failure. |
| 13 | Yes | Program exit. |

Overwriting Exception Settings

There are two methods to overwrite the default exception settings:

1. Use the command **xmdconfig** [-mb_trap_mask|-ppc_trap_mask] [MASK]
This sets the mask for all targets in the current XMD session. To define your own default setting for all XMD sessions, you can write that command in the .xmdrc file which is located at your home directory.
2. Use the command **safemode -config mode** [MASK]
This sets the mask for current target only. While debugging a program, this is a convenient way to change the trap settings.

Note: The current target is destroyed when you disconnect from the target.

Viewing Safemode Settings

You can view the current safemode setting with the **safemode -info** command.

In safe mode, XMD sets the breakpoint at the exception handlers that you want to trap.

- For MicroBlaze processors, all exceptions take PC to 0x20.
- For PowerPC processors, XMD detects the exception handler locations from the ELF file.

The detection works on most Standalone or Xilkernel projects. If another software platform is used, the detection might fail. In such cases, set the exception handler address with the **safemode -config** <exception_id> <exception_handler_addr> command:

Connect Command Options

XMD can debug programs on different targets (processor or peripheral.)

- When communicating with a target, XMD connects to the target and a unique target ID is assigned to each target after connection.
- When connecting to a processor, the gdbserver starts, enabling communication with GDB or SDK.

Usage

```
connect {mb | ppc | mdm | arm} <Connection_Type> [Options]
```

Table 10-6: Connect Command Options

| Option | Description |
|-------------------|--|
| ppc | Connects to PowerPC processor |
| mb | Connects to MicroBlaze processor |
| mdm | Connects to MDM peripheral |
| arm | Connects to Cortex- A9 processor and to the Coresight. |
| <Connection_Type> | Connection method, target dependent |
| [Options] | Connection options |

The following sections describe connect options for different targets.

PowerPC Processor Targets

Xilinx Virtex[®] series devices can contain one or two PowerPC (405 and 440) processor cores. XMD can connect to these PowerPC processor targets over a JTAG connection on the board. XMD also communicates over a TCP socket interface to an IBM PowerPC 405 Processor Instruction Set Simulator (ISS).

Use the **connect ppc** command to connect to the PowerPC processor target and start a remote GDB server. When XMD is connected to the PowerPC processor target, `powerpc-eabi-gdb` or SDK can connect to the processor target through XMD, and debugging can proceed.

Note: XMD does not support Virtual Addressing. Debugging is only supported for Programs running in Real Mode.

PowerPC Processor Hardware Connection

When connecting to a PowerPC processor hardware target, XMD detects the JTAG chain automatically, and the PowerPC processor type and processors in the system, and connects to the first processor. You can override or provide information using the following options.

Usage

```
connect ppc hw [-cable <JTAG Cable options>] {[-configdevice <JTAG chain options>]} [-debugdevice <PowerPC options>]
```

JTAG Cable Options

You can use the options listed in [Table 10-7](#) to specify the Xilinx JTAG cable used to connect to a target.

Table 10-7: JTAG Cable Options

| Option | Description |
|---|---|
| esn <USB cable ESN> | Specify the Electronic Serial Number (ESN) of the USB cable connected to the host machine. Use this option to uniquely identify a USB cable when multiple cables are connected to the host machine. To read the ESN of the USB cable, connect the cable and use the xrcableesn command. |
| fname <filename.svf> | Filename for creating the Serial Vector Format (SVF) file. |
| frequency <cable speed in Hz> | Specify the cable clock speed in Hertz. Valid Cables speeds are: <ul style="list-style-type: none"> For Parallel 4: 5000000 (default), 2500000, 200000 For Platform USB: 24000000, 12000000, 6000000 (default), 3000000, 1500000, 750000 |

Table 10-7: JTAG Cable Options (Cont'd)

| Option | Description |
|---------------------------|--|
| port <port name> | Specify the port. Valid arguments for port are: lpt1, lpt2,usb21, usb22, .. |
| type <cabl_e_type> | Specify the cable type. Valid cable types are: <ul style="list-style-type: none"> • diligent <ul style="list-style-type: none"> -cable type xilinx_plugin modulename <name> modulearg <args> (7 series only) • xilinx_parallel3 • xilinx_parallel4 • xilinx_platformusb • xilinx_svffile In the case of xilinx_svffile, the JTAG commands are written into a file specified by the fname option. |

JTAG Chain Options

Table 10-8 lists the options that let you specify device information of non-Xilinx devices in the JTAG chain. Refer to “[Example Showing Special JTAG Chain Setup for Non-Xilinx Devices](#)” on page 162.

Table 10-8: JTAG Chain Options

| Option | Description |
|---|---|
| devicenr <device position> | The position of the device in the JTAG chain. The device position number starts from 1. |
| irlength <length of the JTAG Instruction Register> | The length of the IR register of the device. This information can be found in the device BSDL file. |
| idcode <device idcode> | JTAG ID code of the device. If the PowerPC processor JTAG pins are connected directly to FPGA user IO pins, the irlength should be 4. |
| partname <device name> | The name of the device. |

PowerPC Processor Options

The following options allow you to specify the FPGA device to debug and the processor number in the device. You can also map special PowerPC processor features such as ISOCM, Caches, TLB, and DCR registers to unused memory addresses, and then access them from the debugger as memory addresses. This is helpful for reading and writing to these registers and memory from GDB or XMD.

Table 10-9 lists the PowerPC processor options

Note: These options *do not* create any real memory mapping in hardware.

Table 10-9: PowerPC Processor Options

| Option | Description |
|--|---|
| cpunr <i><CPU Number></i> | PowerPC processor number to be debugged in a Virtex device with multiple PowerPC processors. It starts from 1. |
| dcachestartadr <i><D-Cache start address></i> | Start address for reading or writing the data cache contents. |
| dcrstartadr <i><DCR start address></i> | Start address for reading and writing the Device Control Registers (DCR). Using this option, the entire DCR address space (210 addresses) can be mapped to addresses starting from the <i><DCR start address></i> for debugging from XMD and GDB. |
| devicenr <i><PowerPC device position></i> | Position in the JTAG chain of the Virtex device containing the PowerPC processor. The device position number starts from 1. |
| dtagstartadr <i><D-Cache start address></i> | Start address for reading or writing the data cache tags. |
| fputype {sp dp} | XMD does not automatically look for a Floating Point Unit (FPU) in the PowerPC processor system. To force XMD to detect a FPU, specify this option with the FPU type in the system. Options: sp = Single Precision dp = Double Precision |
| icachestartadr <i><I-Cache start address></i> | Start address for reading or writing the instruction cache contents. |
| isocmdcrstartadr <i><ISOCM (in Bytes) DCR address></i> | DCR address corresponding to the ISOCM interface specified using the C_ISOCM_DCR_BASEADDR parameter on PowerPC 405 processors. |
| isocmstartadr <i><ISOCM start address></i> | Start address for the Instruction Side On-Chip Memory (ISOCM). Only for PowerPC 405 processor. |
| isocmsize <i><ISOCM size in Bytes></i> | Size of the ISBRAM memory connected to the ISOCM interface. Only for PowerPC 405 processor. |
| itagstartadr <i><I-Cache start address></i> | Start address for reading or writing the instruction cache tags. |
| romemstartadr <i><ROM start address></i> | Start address of Read-Only Memory. This can be used to specify flash memory range. XMD sets hardware breakpoints instead of software breakpoints. |
| romemsize <i><ROM size in bytes></i> | Size of Read-Only Memory (ROM). |
| tlbstartadr <i><TLB start address></i> | Start address for reading and writing the Translation Look-aside Buffer (TLB). |

PowerPC Processor Target Requirements

There are two possible methods for XMD to connect to the PowerPC processors over a JTAG connection. The requirements for each of these methods are described in the following subsections.

Debug connection using the JTAG port of a Virtex FPGA

If the JTAG ports of the PowerPC processors are connected to the JTAG port of the FPGA internally using the `JTAGPPC` primitive, then XMD can connect to any of the PowerPC processors inside the FPGA, as shown in the following figure. Refer to the *PowerPC 405 Processor Block Reference Guide* and the *PowerPC 440 Processor Block Reference Guide* for more information. A link to the document is supplied in [Appendix E, Additional Resources](#).

Debug connection using I/O pins connected to the JTAG port of the PowerPC Processor

If the JTAG ports of the PowerPC processors are brought out of the FPGA using I/O pins, then XMD can directly connect to the PowerPC processor for debugging. In this mode XMD can only communicate with one PowerPC processor. If there are two PowerPC processors in your system, you cannot chain them, and the JTAG ports to each processor should be brought out to use FPGA I/O pins. Refer to the *PowerPC 405 Processor Block Reference Guide* and the *PowerPC 440 Processor Block Reference Guide* for more information about this debug setup. A link to the document is supplied in [Appendix E, Additional Resources](#).

Figure 10-2, page 157 illustrates the PowerPC processor target.

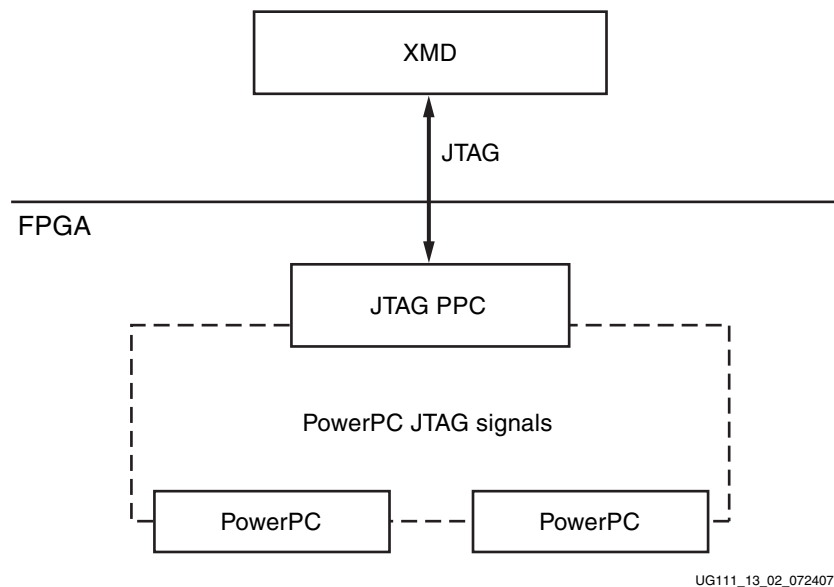


Figure 10-2: PowerPC Processor Target

Example Debug Sessions

Example Using a PowerPC 405 Processor Target

The following example demonstrates a simple debug session with a PowerPC 405 processor target. Basic XMD-based commands are used after connecting to the PowerPC processor target using the **connect ppc hw** command.

At the end of the session, `powerpc-eabi-gdb` is connected to XMD using the GDB remote target. Refer to [Chapter 11, “GNU Debugger,”](#) for more information about connecting GDB to XMD.

```
XMD% connect ppc hw
JTAG chain configuration
-----
Device      ID Code      IR Length    Part Name
1           0a001093      8            System_ACE
2           f5059093      16           XCF32P
3           01e58093      10           XC4VFX12
4           49608093      8            xc95144x1

PowerPC405 Processor Configuration
-----
Version.....0x20011430
User ID.....0x00000000
No of PC Breakpoints.....4
No of Read Addr/Data Watchpoints....1
No of Write Addr/Data Watchpoints...1
User Defined Address Map to access Special PowerPC Features using XMD:
    I-Cache (Data).....0x70000000 - 0x70003fff
    I-Cache (TAG).....0x70004000 - 0x70007fff
    D-Cache (Data).....0x78000000 - 0x78003fff
    D-Cache (TAG).....0x78004000 - 0x78007fff
    DCR.....0x78004000 - 0x78004fff
    TLB.....0x70004000 - 0x70007fff
Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
XMD% rrd
    r0: ef0009f8      r8: 51c6832a      r16: 00000804      r24: 32a08800
    r1: 00000003      r9: a2c94315      r17: 00000408      r25: 31504400
    r2: fe008380      r10: 00000003     r18: f7c7dfcd      r26: 82020922
    r3: fd004340      r11: 00000003     r19: fbcbefce      r27: 41010611
    r4: 0007a120      r12: 51c6832a     r20: 0040080d      r28: fe0006f0
    r5: 000b5210      r13: a2c94315     r21: 0080040e      r29: fd0009f0
    r6: 51c6832a      r14: 45401007     r22: c1200004      r30: 00000003
    r7: a2c94315      r15: 8a80200b     r23: c2100008      r31: 00000003
    pc: ffff0700      msr: 00000000

XMD% srrd
    pc: ffff0700      msr: 00000000      cr: 00000000      lr: ef0009f8
    ctr: ffffffff      xer: c000007f      pvr: 20010820      sprg0: ffffe204
    sprg1: ffffe204    sprg2: ffffe204    sprg3: ffffe204    srr0: ffff0700
    srr1: 00000000      tbl: a06ea671      tbu: 00000010      icdbdr: 55000000
    esr: 88000000      dear: 00000000     evpr: ffff0000      tsr: fc000000
    tcr: 00000000      pit: 00000000      srr2: 00000000      srr3: 00000000
    dbcr0: 00000300    dbcr1: 81000000    iac1: ffffe204     iac2: ffffe204
    dac1: ffffe204     dac2: ffffe204     dccr: 00000000      iccr: 00000000
    zpr: 00000000      pid: 00000000      sgr: ffffffff      dcwr: 00000000
    ccr0: 00700000     dbcr1: 00000000    dvc1: ffffe204     dvc2: ffffe204
```

```

    iac3: fffffe204    iac4: fffffe204    sler: 00000000    sprg4: fffffe204
    sprg5: fffffe204    sprg6: fffffe204    sprg7: fffffe204    su0r: 00000000
    usprg0: fffffe204
XMD% rst
Sending System Reset
Target reset successfully
XMD% rwr 0 0xAAAAAAAA
XMD% rwr 1 0x0
XMD% rwr 2 0x0
XMD% rrd
    r0: aaaaaaaaa      r8: 51c6832a      r16: 00000804      r24: 32a08800
    r1: 00000000      r9: a2c94315      r17: 00000408      r25: 31504400
    r2: 00000000      r10: 00000003     r18: f7c7dfcd      r26: 82020922
    r3: fd004340      r11: 00000003     r19: fbcbefce      r27: 41010611
    r4: 0007a120      r12: 51c6832a      r20: 0040080d      r28: fe0006f0
    r5: 000b5210      r13: a2c94315      r21: 0080040e      r29: fd0009f0
    r6: 51c6832a      r14: 45401007      r22: c1200004      r30: 00000003
    r7: a2c94315      r15: 8a80200b      r23: c2100008      r31: 00000003
    pc: ffffffff      msr: 00000000
XMD% mrd 0xFFFFFFF0
FFFFFFF0: 4BFFFC74
XMD% stp
ffffffc70:
XMD% stp
ffffffc74:
XMD% mrd 0xFFFFC000 5
FFFC000: 00000000
FFFC004: 00000000
FFFC008: 00000000
FFFC00C: 00000000
FFFC010: 00000000
XMD% mwr 0xFFFFC004 0xabcd1234 2
XMD% mwr 0xFFFFC010 0xa5a50000
XMD% mrd 0xFFFFC000 5
FFFC000: 00000000
FFFC004: ABCD1234
FFFC008: ABCD1234
FFFC00C: 00000000
FFFC010: A5A50000
XMD%
XMD%

```

Example Connecting to PowerPC440 Processor Target

To connect to the PowerPC 440 processor target use the **connect ppc hw** command.

XMD automatically detects the processor type and connects to the PowerPC 440 processor.

Use powerpc-eabi-gdb to debug software program remotely. Refer to [Chapter 11, “GNU Debugger,”](#) for more information about connecting the GNU Debugger to XMD.

```
XMD% connect ppc hw
JTAG chain configuration
-----
Device      ID Code          IR Length    Part Name
  1         f5059093             16      XCF32P
  2         f5059093             16      XCF32P
  3         59608093              8      xc95144xl
  4         0a001093              8      System_ACE
  5         032c6093             10      XC5VFX70T_U

PowerPC440 Processor Configuration
-----
Version.....0x7ff21910
User ID.....0x00f00000
No of PC Breakpoints.....4
No of Read Addr/Data Watchpoints....1
No of Write Addr/Data Watchpoints...1
User Defined Address Map to access Special PowerPC Features using XMD:
    I-Cache (Data).....0x70000000 - 0x70007fff
    I-Cache (TAG).....0x70008000 - 0x7000ffff
    D-Cache (Data).....0x78000000 - 0x78007fff
    D-Cache (TAG).....0x78008000 - 0x7800ffff
    DCR.....0x78020000 - 0x78020fff
    TLB.....0x70020000 - 0x70023fff

Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
XMD% targets
-----
System(0) - Hardware System on FPGA(Device 5) Targets:
-----
    Target(0) - PowerPC440(1) Hardware Debug Target*
XMD%
```

Example with a Program Running in ISOCM Memory and Accessing DCR Registers

This example demonstrates a simple debug session with a program running on ISOCM memory of the PowerPC 405 processor target. The ISOCM address parameters can be specified during the **connect** command. If the XMP file is loaded, XMD infers the ISOCM address parameters of the system from the MHS file.

Note: In a Virtex-4 device, ISOCM memory is readable. This enables better debugging of a program running from ISOCM memory.

```
XMD% connect ppc hw -debugdevice \
isocmstartadr 0xFFFFE000 isocmsize 8192 isocmdcrstartadr 0x15 \
dcrstartadr 0xab000000
JTAG chain configuration
-----
Device      ID Code          IR Length    Part Name
  1         0a001093             8    System_ACE
  2         f5059093            16    XCF32P
  3         01e58093            10    XC4VFX12
  4         49608093             8    xc95144x1

PowerPC405 Processor Configuration
-----
Version.....0x20011430
User ID.....0x00000000
No of PC Breakpoints.....4
No of Read Addr/Data Watchpoints....1
No of Write Addr/Data Watchpoints...1
ISOCM.....0xfffffe000 - 0xffffffff
User Defined Address Map to access Special PowerPC Features using XMD:
    I-Cache (Data).....0x70000000 - 0x70003fff
    I-Cache (TAG).....0x70004000 - 0x70007fff
    D-Cache (Data).....0x78000000 - 0x78003fff
    D-Cache (TAG).....0x78004000 - 0x78007fff
    DCR.....0xab000000 - 0xab000fff
    TLB.....0x70004000 - 0x70007fff

XMD% stp
ffffe21c:
XMD% stp
ffffe220:
XMD% bps 0xFFFFE218
Setting breakpoint at 0xfffffe218
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
8
Processor stopped at PC: 0xfffffe218
XMD%
XMD% mrd 0xab000060 8
AB000060:  00000000
AB000064:  00000000
AB000068:  FF000000 <--- DCR register : ISARC
AB00006c:  81000000 <--- DCR register : ISCNTL
AB000070:  00000000
AB000074:  00000000
AB000078:  FE000000 <--- DCR register : DSARC
AB00007c:  81000000 <--- DCR register : DSCNTL
XMD%
```

Example Showing Special JTAG Chain Setup for Non-Xilinx Devices

This example demonstrates the use of the **-configdevice** option to specify the JTAG chain on the board in the event that XMD is unable to detect the JTAG chain automatically.

Automatic detection in XMD can fail for non-Xilinx devices on the board for which the JTAG IRLengths are not known. The JTAG (Boundary Scan) IRLength information is usually available in Boundary-Scan Description Language (BSDL) files provided by device vendors. For these unknown devices, IRLength is the only critical information; the other fields such as `partname` and `idcode` are optional.

The options used in the following example are:

- Xilinx Parallel cable (III or IV) connection is done over the LPT1 parallel port.
- The two devices in the JTAG chain are explicitly specified.
- The `IRLength`, `partname`, and `idcode` of the PROM are specified.
- The **debugdevice** option explicitly specifies to XMD that the FPGA device of interest is the second device in the JTAG chain.

In Virtex devices it is also explicitly specified that the connection is for the first PowerPC processor, if there is more than one.

```
XMD% connect ppc hw -cable type xilinx_parallel port LPT1 -configdevice
devicenr 1 partname PROM_XC18V04 irlength 8 idcode 0x05026093
-configdevice devicenr 2 partname XC2VP4 irlength 10 idcode 0x0123e093
-debugdevice devicenr 2 cpunr 1
```

Adding Non-Xilinx Devices

You can add a non-Xilinx device either on the command line using the **connect** command using the JTAF Chain options or by specifying it in the GUI. See [Connect Command Options, page 153](#) and [JTAG Chain Options, page 155](#) and for more information.

PowerPC Processor Simulator Target

XMD can connect to one or more PowerPC 405 processor ISS targets through socket connection. Use the **connect ppc sim** command to start the PowerPC 405 processor ISS on a local host, connect to that host, and start a remote GDB server.

You can also use **connect ppc sim** to connect to a PowerPC 405 processor ISS running on localhost or other machine.

When XMD is connected to the PowerPC 405 processor target, `powerpc-eabi-gdb` can connect to the target through XMD and debugging can proceed.

Note: XMD does not support PowerPC 440 processor ISS targets.

Running PowerPC Processor ISS

XMD starts the ISS with a default configuration.

- The ISS executable file is located in the `${XILINX_EDK}/third_party/bin/<platform>/` directory.
- The PowerPC 405 processor configuration file used is `${XILINX_EDK}/third_party/data/iss405.icf`.

You can run ISS with different configuration options and XMD can connect to the ISS target. Refer to the *IBM Instruction Set Simulator User Guide* for more details. A link to the document is supplied in [Appendix E, Additional Resources](#).

The following are the default configurations for ISS.

- Two local memory banks
- Connect to XMD Debugger
- Debugger port at 6470...6490
- Data cache size of 16 K
- Instruction cache size of 16 K
- Non-deterministic multiply cycles
- Processor clock period and timer clock period of 5 ns (200 MHz).

[Table 10-10](#) lists the Local Memory Banks.

Table 10-10: Local Memory Banks

| Name | Start Address | Length | Speed |
|------|---------------|---------|-------|
| Mem0 | 0x0 | 0x80000 | 0 |
| Mem1 | 0xff80000 | 0x80000 | 0 |

[Figure 10-3](#) illustrates a PowerPC processor ISS target.

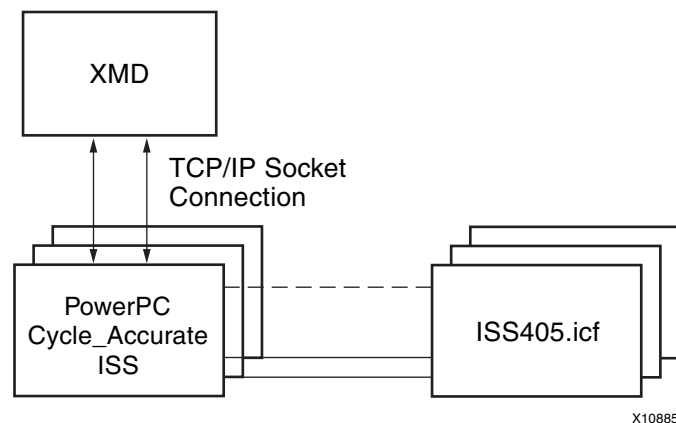


Figure 10-3: PowerPC Processor ISS Target

Usage

```
connect ppc sim [-icf <Configuration File>] [-ipcport IP:<port>]
```

| Option | Description |
|-------------------------------------|---|
| -icf <configuration file> | Uses the given ISS configuration file instead of the default configuration file. You can customize the PowerPC ISS features such as cache size, memory address map, and memory latency. |
| -ipcport: IP:<port> | Specifies the IP address and debug port of a PowerPC processor ISS that you have started. XMD does not spawn a ISS, you must start the ISS. |

Example Debug Session for PowerPC Processor ISS Target

```
XMD% connect ppc sim
Instruction Set Simulator (ISS)
PPC405,
Version 1.9 (1.76)
(c) 1998, 2005 IBM Corporation
Waiting to connect to controlling interface (port=6470,
protocol=tcp)....
[XMD] Connected to PowerPC Sim
Controlling interface connected....
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD% dow dhry2.elf
XMD% bps 0xffff09d0
XMD% con
Processor started. Type "stop" to stop processor

RUNNING>
```

DCR, TLB, and Cache Address Space and Access

The XMD sets up address space for you to access TLB entries and Cache entries. These address spaces can be specified with `tlbstartadr`, `icachestartadr`, and `dcachestartadr` as options to the connection command. If the TLB and Cache address space is not specified, XMD uses a default unused address space for this purpose. When connected, these address spaces are displayed in the XMD console. For example:

```
I-Cache (Data).....0x70000000 - 0x70007fff
I-Cache (TAG).....0x70008000 - 0x7000ffff
D-Cache (Data).....0x78000000 - 0x78007fff
D-Cache (TAG).....0x78008000 - 0x7800ffff
DCR.....0x78020000 - 0x78020fff
TLB.....0x70020000 - 0x70023fff
```

TLB Access

Each TLB entry is represented by a 4-word entry. Table 10-11 shows the 4-word entries available for PPC405 and PPC440.

Table 10-11: PPC405 and PPC440 TLB Entries

| Word | PPC405 | PPC440 |
|------|-----------------|---------------------------|
| 1 | PID | PID |
| 2 | TLBHI | TLB Word0 (excluding PID) |
| 3 | TLBLO | TLB Word1 |
| 4 | Padded with 0's | TLB Word2 |

The total 64 TLB entries can be read from (or written to) the 256 words starting from the TLB starting address.

Cache Word Access

The cache entries are mapped to the address space in a way-by-way manner. Using the provided example, if the cache line size is 32 byte and each way has 16 sets, then 0x70000000~0x700001FF is mapped to I-Cache way 0 and 0x70000200~0x700003FF is mapped to I-Cache way 1.

Cache Tag and Parity Access

The cache tag address space contains the tag, status, and parity information of the cache entries for the corresponding cache address space. In the provided example, the tag information for I-Cache entry at 0x70000100 is available at 0x70008100 and the tag information for the D-Cache entry at 0x78000600 is available at 0x78008600.

The PowerPC 405 processor uses one word to store the tag and status of one cache line and one word to store parities.

The PowerPC 440 processor also uses two words (first word is tag low and second word is tag high) to store the tag of one cache line. For more information on how to translate the tag bits, refer to the `icread` and `dcread` instructions in the respective *PowerPC405 User Manual* or *PowerPC440 User Manual*. A link to these documents can be found in [Appendix E, Additional Resources](#). Because the cacheline size is 32 bytes, the tag values repeat within the same cacheline.

DCR Address Spaces

Although the DCR bus is not in the same address domain as the PLB bus, you can access the DCR bus in XMD through the PLB address map. Each DCR address corresponds to one DCR register, which has 4 bytes. When it is mapped to the PLB address, it needs 4 bytes of address range. In the example shown in [Example Debug Session for PowerPC Processor ISS Target, page 164](#), the address mappings are:

| DCR Address | Mapped Address |
|-------------|----------------|
| 0x0 | 0x78020000 |
| 0x1 | 0x78020004 |
| 0x2 | 0x78020008 |
| ... | ... |
| 0x10 | 0x78020040 |

Advanced PowerPC Processor Debugging Tips

Support for Running Programs from ISOCM and ICACHE

There are restrictions on debugging programs from PowerPC 405 processor ISOCM memory and instruction caches (ICACHES). One such restriction is that you cannot use software breakpoints. In such cases, XMD can set hardware breakpoints automatically 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, refer to the “*Xilinx Platform Studio Help*”.

The special features of the PowerPC processor can be accessed from XMD by specifying the appropriate options to the **connect** command in the XMD console.

Debugging Setup for Third-Party Debug Tools

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 processor (TCK, TMS, TDI, and TDO,) out of the FPGA as User IO to appropriate debug connectors on the hardware board.

You must also bring the DBG405DEBUGHALT and C405JTGTDOEN signals out of the FPGA as User IO.

In the case of multiple PowerPC processors, Xilinx recommends that you chain the PowerPC processor JTAG signals inside the FPGA. For more information about connecting the PowerPC processor JTAG port to FPGA User IO, refer to the JTAG port sections of the *PowerPC 405 Processor Block Reference Guide*, and the *PowerPC 440 Processor Block Reference Guide*. A link to the document is supplied in [Appendix E, Additional Resources](#).

Note: DO NOT use the JTAGPowerPC module while bringing the PowerPC processor JTAG signals out as User IO.

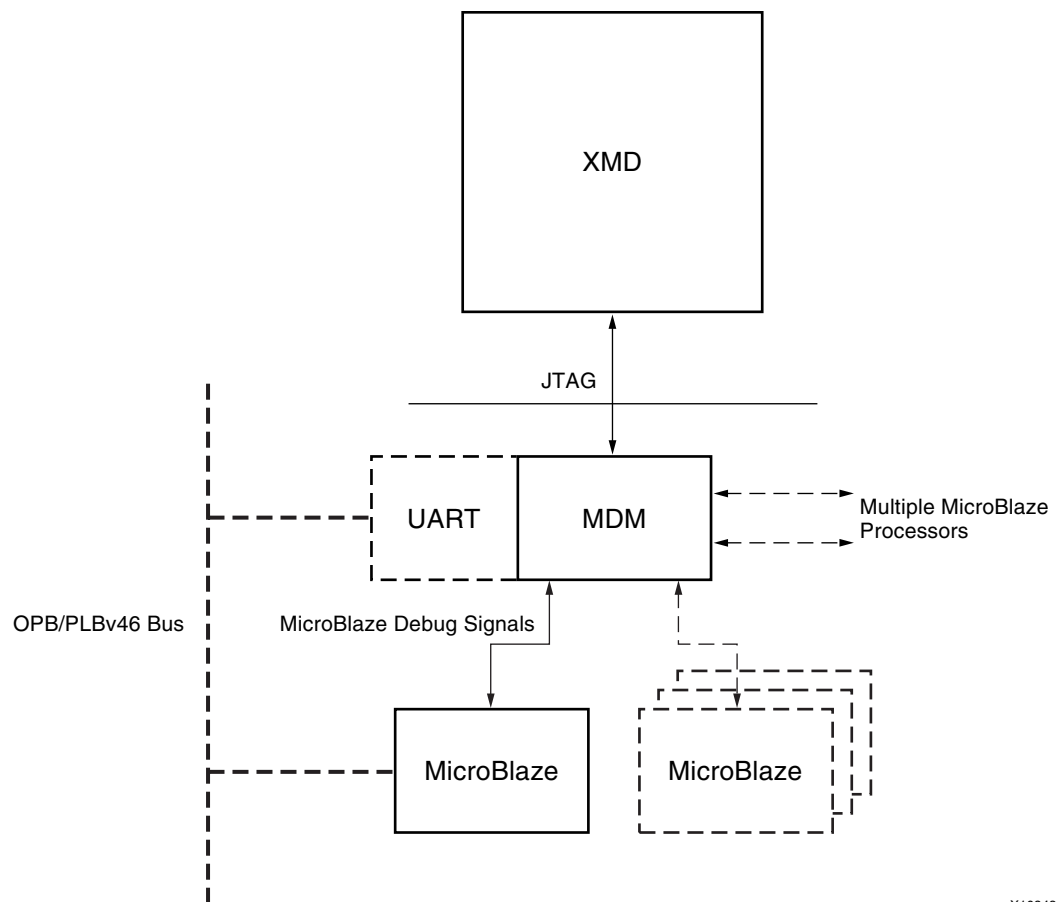
MicroBlaze Processor Target

XMD can connect through JTAG to one or more MicroBlaze processors using the MDM peripheral. XMD can communicate with a ROM monitor such as XMDStub through a JTAG or serial interface. You can also debug programs using built-in, cycle-accurate MicroBlaze ISS. The following sections describe the options for these targets.

MicroBlaze MDM Hardware Target

Use the command **connect mb mdm** to connect to the MDM target and start the remote GDB server. The MDM target supports non-intrusive debugging using hardware breakpoints and hardware single-step, without the need for a ROM monitor.

Figure 10-4, page 167 illustrates the MicroBlaze MDM target.



X10843

Figure 10-4: MicroBlaze MDM Target

When no option is specified to the **connect mb mdm**, XMD detects the JTAG cable automatically and chains the FPGA device containing the MicroBlaze-MDM system.

If XMD is unable to detect the JTAG chain or the FPGA device automatically, you can explicitly specify them using the following options:

Usage

```
connect mb hw [-cable <JTAG Cable options>] {[-configdevice <JTAG
chain options>]} [-debugdevice <MicroBlaze options>]
```

JTAG Cable Options and JTAG Chain Options

For JTAG cable and chain option descriptions, refer to [Table 10-7, JTAG Cable Options on page 154](#), and [Table 10-8, JTAG Chain Options on page 155](#), respectively.

MicroBlaze Options

[Table 10-12](#) describes the MicroBlaze options.

Table 10-12: MicroBlaze Options

| Option | Description |
|---|--|
| cpunr <CPU Number> | Specific MicroBlaze processor number to be debugged in an FPGA containing multiple MicroBlaze processors connected to MDM. The processor number starts from 1. |
| devicenr <MicroBlaze device position> | Position in the JTAG chain of the FPGA device containing the MicroBlaze processor. The device position number starts from 1. |
| romemstartadr <ROM start address> | Start address of Read-Only Memory. Use this to specify flash memory range. XMD sets hardware breakpoints instead of software breakpoints. |
| romemsize <ROM Size in Bytes> | Size of Read-Only Memory. |
| tlbstartadr <TLB start address> | Start address for reading and writing the Translation Look-aside Buffer (TLB). |

MicroBlaze MDM Target Requirements

1. To use the hardware debug features on MicroBlaze, such as hardware breakpoints and hardware debug control functions like stopping and stepping, the hardware debug port must be connected to the MDM.
2. To use the UART functionality in the MDM target, you must set the `C_USE_UART` parameter while instantiating the MDM core in a system.

Note: Unlike the MicroBlaze stub target, programs should be compiled in executable mode and NOT in XMDSTUB mode while using the MDM target. Consequently, you do *not* need to specify an `XMDSTUB_PERIPHERAL` for compiling the XMDStub.

Example Debug Sessions

Example Using a MicroBlaze MDM Target

This example demonstrates a simple debug session with a MicroBlaze MDM target. Basic XMD-based commands are used after connecting to the MDM target using the `connect mb mdm` command. At the end of the session, `mb-gdb` connects to XMD using the GDB remote target. Refer to [Chapter 11, “GNU Debugger,”](#) for more information about connecting GDB to XMD.

```
XMD% connect mb mdm
JTAG chain configuration
-----
Device      ID Code          IR Length    Part Name
  1         0a001093             8    System_ACE
  2         f5059093            16    XCF32P
  3         01e58093            10    XC4VFX12
  4         49608093             8    xc95144xl

MicroBlaze Processor Configuration:
-----
Version.....7.00.a
Optimisation.....Performance
Interconnect.....PLBv46
No of PC Breakpoints.....3
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Exceptions Support.....off
FPU Support.....off
Hard Divider Support.....off
Hard Multiplier Support.....on - (Mul32)
Barrel Shifter Support.....off
MSR clr/set Instruction Support....on
Compare Instruction Support.....on
PVR Supported.....on
PVR Configuration Type.....Base

Connected to MDM UART Target
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD% rrd
      r0: 00000000      r8: 00000000      r16: 00000000      r24: 00000000
      r1: 00000510      r9: 00000000      r17: 00000000      r25: 00000000
      r2: 00000140      r10: 00000000      r18: 00000000      r26: 00000000
      r3: a5a5a5a5      r11: 00000000      r19: 00000000      r27: 00000000
      r4: 00000000      r12: 00000000      r20: 00000000      r28: 00000000
      r5: 00000000      r13: 00000140      r21: 00000000      r29: 00000000
      r6: 00000000      r14: 00000000      r22: 00000000      r30: 00000000
      r7: 00000000      r15: 00000064      r23: 00000000      r31: 00000000
      pc: 00000070      msr: 00000004

<--- Launching GDB from XMD% console --->
XMD% start mb-gdb microblaze_0/code/executable.elf
XMD%
<--- From GDB, a connection is made to XMD and debugging is done from
the GDB GUI --->
XMD: Accepted a new GDB connection from 127.0.0.1 on port 3791
XMD%
XMD: GDB Closed connection
```

```

XMD% stp
BREAKPOINT at
    114:  F1440003  sbi      r10, r4, 3
XMD% dis 0x114 10
    114:  F1440003  sbi      r10, r4, 3
    118:  E0E30004  lbui     r7, r3, 4
    11C:  E1030005  lbui     r8, r3, 5
    120:  F0E40004  sbi      r7, r4, 4
    124:  F1040005  sbi      r8, r4, 5
    128:  B800FFCC  bri      -52
    12C:  B6110000  rtsd     r17, 0
    130:  80000000  Or       r0, r0, r0
    134:  B62E0000  rtid     r14, 0
    138:  80000000  Or       r0, r0, r0
XMD% dow microblaze_0/code/executable.elf
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING> stop
XMD% Info:User Interrupt, Processor Stopped at 0x0000010c
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING> rrd pc
pc : 0x000000f4 <--- With the MDM, the current PC of MicroBlaze can be
                        read while the program is running

RUNNING> rrd pc
pc : 0x00000110 <--- Note: the PC is constantly changing, as the
                        program is running

RUNNING> stop
Info:Processor started. Type "stop" to stop processor
XMD% rrd
    r0: 00000000      r8: 00000065      r16: 00000000      r24: 00000000
    r1: 00000548      r9: 0000006c      r17: 00000000      r25: 00000000
    r2: 00000190      r10: 0000006c     r18: 00000000      r26: 00000000
    r3: 0000014c      r11: 00000000     r19: 00000000      r27: 00000000
    r4: 00000500      r12: 00000000     r20: 00000000      r28: 00000000
    r5: 24242424      r13: 00000190     r21: 00000000      r29: 00000000
    r6: 0000c204      r14: 00000000     r22: 00000000      r30: 00000000
    r7: 00000068      r15: 0000005c     r23: 00000000      r31: 00000000
    pc: 0000010c      msr: 00000000
XMD% bps 0x100
Setting breakpoint at 0x00000100
XMD% bps 0x11c hw
Setting breakpoint at 0x0000011c
XMD% bpl
SW BP: addr = 0x00000100, instr = 0xe1230002 <-- Software Breakpoint
HW BP: BP_ID   0 : addr = 0x0000011c      <--- Hardware Breakpoint
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x00000100
Info:Processor stopped. Type "start" to start processor
XMD% con
Info:Processor started. Type "stop" to stop processor
RUNNING>
Info:Processor started. Type "stop" to stop processor

```

MicroBlaze Stub Hardware Target

To connect to a MicroBlaze target, use the XMDStub (a ROM monitor running on the target) and start a GDB server for the target. XMD connects to XMDStub through a JTAG or serial interface. The default option connects using a JTAG interface.

MicroBlaze Stub-JTAG Target Options

Usage

```
connect mb stub -comm jtag [-cable {<JTAG Cable options>}]
[-configdevice {<JTAG chain options>}] [-debugdevice {<MicroBlaze
options>}]
```

JTAG Cable Options and JTAG Chain Options

For JTAG cable and chain option descriptions, refer to [Table 10-7, JTAG Cable Options on page 154](#) and [Table 10-8, JTAG Chain Options on page 155](#), respectively.

MicroBlaze Option

| Option | Description |
|---|--|
| devicenr <MicroBlaze device position> | The position in the JTAG chain of the FPGA device containing MicroBlaze. |

MicroBlaze Stub-Serial Target Options

Usage

```
connect mb stub -comm serial {<Serial Communication options>}
```

Serial Communication Options

[Table 10-13](#) lists the options that specify the MicroBlaze stub-serial target.

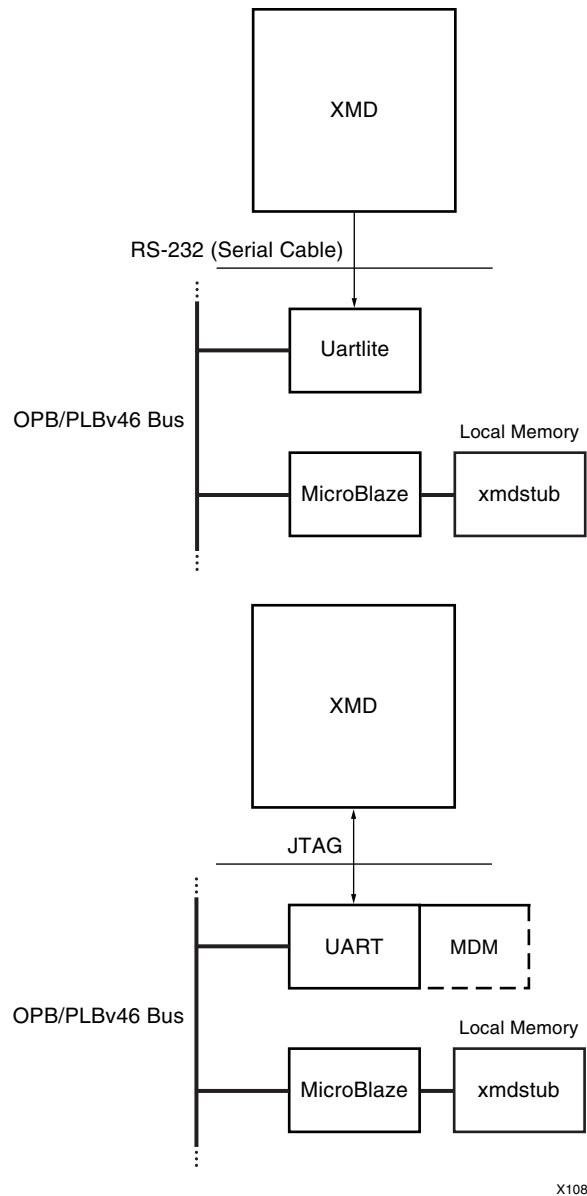
Table 10-13: MicroBlaze Stub-Serial Target Options

| Option | Description |
|---|---|
| -baud <serial port baud rate> | Specifies the serial port baud rate in bits per second (bps). The default value is 19200 bps. |
| -port <serial port> | Specifies the serial port to which the remote hardware is connected when XMD communication is over the serial cable. The default serial ports are: " /dev/ttyS0 on Linux " Com1 on Windows |
| -timeout <timeout in secs> | Timeout period while waiting for a reply from XMDStub for XMD commands. |

Note: If the program has any I/O functions such as `print()` or `putnum()` that write output onto the UART or MDM UART, it is printed on the console or terminal in which XMD was started. Refer to

Chapter 8, “Library Generator (Libgen),” for more information about libraries and I/O functions.

Figure 10-5, page 172 illustrates a MicroBlaze sub target with an MDM UART and a UARTlite.



X10844

Figure 10-5: MicroBlaze Stub Target with MDM UART and UARTlite

Stub Target Requirements

To debug programs on the hardware board using XMD, the following requirements must be met:

- XMD uses a JTAG or serial connection to communicate with XMDStub on the board. Therefore, an mdm or a UART designated as XMDSTUB_PERIPHERAL in the MSS file is necessary on the target MicroBlaze system.

Platform Generator can create a system that includes a mdm or a UART, if specified in its MHS file. The JTAG cables supported with the XMDStub mode are:

- Xilinx parallel cable
- Platform USB cable
- XMDStub on the board uses the MDM or UART to communicate with the host computer; therefore, it must be configured to use the MDM or UART in the MicroBlaze system.

The Library Generator (Libgen) can configure the XMDStub to use the XMDSTUB_PERIPHERAL in the system. Libgen generates an XMDStub configured for the XMDSTUB_PERIPHERAL and puts it in `code/xmdstub.elf` as specified by the XMDStub attribute in the MSS file. For more information, refer to [Chapter 8, “Library Generator \(Libgen\).”](#)

- The XMDStub executable must be included in the MicroBlaze local memory at system startup.

Data2MEM can populate the MicroBlaze memory with XMDStub. Libgen generates a Data2MEM script file that can be used to populate the block RAM contents of a bitstream containing a MicroBlaze system. It uses the executable specified in `DEFAULT_INIT`.

- For any program that must be downloaded on the board for debugging, the program start address must be higher than 0x800 and the program must be linked with the startup code in `crt1.o`.

`mb-gcc` can compile programs satisfying the above two conditions when it is run with the option `-x1-mode-xmdstub`.

Note: For source level debugging, programs should also be compiled with the `-g` option. While initially verifying the functional correctness of a C program, it is advisable to not use any `mb-gcc` optimization option such as `-O2` or `-O3`, as `mb-gcc` performs aggressive code motion optimizations which might make debugging difficult to follow.

MicroBlaze Simulator Target

You can use `mb-gdb` and XMD to debug programs on the cycle-accurate simulator built in to XMD.

Usage

```
connect mb sim [-memsize <size>]
```

MicroBlaze Simulator Option

| Option | Description |
|-----------------------|--|
| memsize <size> | The width of the memory address bus allocated in the simulator. Programs can access the memory range from 0 to $(2^{\text{size}}) - 1$. The default memory size is 64 KB. |

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, you must compile programs for debugging and link them with the startup code in `crt0.o`.

The `mb-gcc` can compile programs with debugging information when it is run with the option `-g`, and by default, `mb-gcc` links `crt0.o` with all programs.

The option is **-x1-mode-executable**.

The program memory size must not exceed 64 K and must begin at address 0. The program must be stored in the first 64KB of memory.

Note: XMD with a simulator target does not support the simulation of OPB peripherals.

MDM Peripheral Target

You can connect to the `mdm` peripheral and use the UART interface for debugging and collecting information from the system.

Usage

```
connect mdm -uart
```

MDM Target Requirements

To use the UART functionality in the MDM target, you must set the `C_USE_UART` parameter while instantiating the `mdm` in a system.

UART input can also be provided from the host to the program running on MicroBlaze using the `xuart w <byte>` command. You can use the `terminal` command to open a hyperterminal-like interface to read and write from the UART interface. The `read_uart` command provides interface to write to STDIO or to file.

Configure Debug Session

Configure the debug session for a target using the **debugconfig** command. You can configure the behavior of instruction stepping and memory access method of the debugger.

Usage

```
debugconfig [-step_mode {disable_interrupt | enable_interrupt}]
[-memory_datawidth_matching {disable | enable}]
[-reset_on_run {system enable | processor enable | disable}]
[-reset_on_data_dow {system enable | processor enable | disable}]
```

Table 10-14, page 175 lists the debug configuration options.

Table 10-14: Debug Configuration Options

| Option | Description |
|--|---|
| No Option | Lists the current debug configuration for the current session. |
| -step_mode {disable_interrupt enable_interrupt} | Configures how XMD handles instruction stepping. disable_interrupt is the default mode. The interrupts are disabled during step. enable_interrupt enables interrupts during step. If an interrupt occurs during step, the interrupt is handled by the registered interrupt handler of the program. |
| -memory_datawidth_matching {disable enable} | Configures how XMD handles memory read and write. By default, the data width matching is set to enable. All data width (byte, half, and word) accesses are handled using the appropriate data width access method. This method is especially useful for memory controllers and flash memory, where the datawidth access should be strictly followed. When data width matching is set to disable, XMD uses the best possible method, such as word access. |
| -reset_on_run [system enable processor enable disable] | Configures how XMD handles reset on program execution. A reset brings the system to a known consistent state for program execution. This ensures correct program execution without any side effects from a previous program run. By default, XMD performs system reset on run (on program download or program run). To enable different reset types, specify: debugconfig -reset_on_run processor enable debugconfig -reset_on_run system enable To disable reset, specify: debugconfig -reset_on_run disable |

Table 10-14: Debug Configuration Options (Cont'd)

| Option | Description |
|--|--|
| -reset_on_data_dow [system enable processor enable disable] | Changes how XMD handles reset on data download. A reset brings the system to a known consistent state for program execution. This ensures correct program execution without any side effects from a previous program run. By default, XMD performs system reset on run (on program download or program run). To enable different reset types, specify: debugconfig -reset_on_data_dow processor enable debugconfig -reset_on_data_dow system enable To disable reset, specify: debugconfig -reset_on_data_dow disable |
| -run_poll_interval <i><time in millisec></i> | When the processor is run using either the run or con command, XMD monitors the processor state at regular intervals (100 ms). If you want XMD to poll less frequently, use this option to specify the poll interval. |

Configuring Instruction Step Modes

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
```

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 appropriate data width accesses when performing memory read and write operations. You can use the `debugconfig` command for configuring XMD to match the data width of the 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... Enabled

XMD% debugconfig -memory_datawidth_matching disable
XMD% debugconfig
Debug Configuration for Target 0
-----
Step Mode..... Interrupt Disabled
Memory Data Width Matching... Disabled
```

Configuring Reset for Multiprocessing Systems

By default, XMD performs a system reset upon download of a program to a processor. This behavior ensures a clean processor state before running the program. However, in multiprocessing systems, downloading and running programs to the various processors happens in sequence.

Depending upon the system architecture, a system reset performed during download of a program could cause programs downloaded to other processors, earlier in the sequence, to get reset. This may or may not be desirable; consequently, use the `debugconfig` command to disable system reset and or enable processor reset only on the various processors.

The following are example use cases:

Example 1: One Master Processor and Multiple Slave Processors

In this scenario, the program on the master processor gets downloaded and run first, followed by the other processors. In this case, the user wants to enable system reset on download to the master processor and only a processor reset (or no reset) on the other processors.

Example 2. Peer Processors

In this case, the download sequence could be arbitrary and the user wants to enable only processor reset (or no reset) at both the processors. This will ensure that downloading a program to one of the peer processors, does not affect the system state for the other peers.

Refer the `proc_sys_reset` IP module documentation for more information on how the reset connectivity and sequencing works through this module.

Cortex A9 Processor Target

```
XMD% connect arm hw
```

```
JTAG chain configuration
```

```
-----
Device      ID Code          IR Length    Part Name
  1         4ba00477             4    Cortex-A9
  2         03727093             6    XC7Z020
```

```
CortexA9 Processor Configuration
```

```
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....1
```

```
Connected to "arm" target. id = 64
```

```
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
```

```
srrd [reg name]
```

Special Register Read. For CortexA9, read a set of CoProcessor regs, identified by [reg name].

[reg name] can be any of ctrl, debug, dma, tcm, id, etc, vfp. (default: ctrl)

```
mrc <CPx> <op1> <CRn> <CRm>
<op2>
```

Cortex A9 CoProcessor Register Read

```
rwr <register> <word>
```

Register Write

```
mcr <CPx> <op1> <CRn> <CRm>
<op2> <word>
```

Cortex A9 CoProcessor Register Write

```
mrd <address> [num] [w|h|b]
```

Memory Read (default: 'w'ord)

```
mrd_var <variable name> [ELF
file]
```

Read memory at global variable

```
mrd_phys <address> [num] [w|h|b]
```

Cortex A9 Memory Read through AHB AP (default: 'w'ord)

```
[-force]
```

Read from OCM at 0x0 (iff DDR is not remapped to 0x0)

```
mwr <address> <values> [<num>
<w|h|b>]
```

Memory Write (default: 'w'ord)

```
mwr_phys <address> <values>
[<num> <w|h|b>]
```

Cortex A9 Memory Write through AHB AP (default: 'w'ord)

```
[-force]
```

Write to OCM at 0x0 (iff DDR is not remapped to 0x0)

```
srrd ctrl
```

```
Control: 08c50879
```

```
Auxiliary Control: 00000045
```

```
Coprocessor Access Control: 00f00000
```

```
Secure Configuration: 00000000
```

```

Secure Debug Enable: 00000000
Non-Secure Access Control: 00000000
Translation Table Base 0: 00108059
Translation Table Base 1: 00108059
Translation Table Base Control: 00000002
Domain Access Control: ffffffff
Data Fault Status: 00000000
Instruction Fault Status: 00000000
Fault Address: 00000000
Watchpoint Fault Address: 00000000
Instruction Fault Address: 00000000
Secure or Non-secure Vector Base Address: 00100000
Monitor Vector Base Address: 00000000
Interrupt Status: 00000000
FCSE PID: 00000000
Context ID: : 00000000
User Read/Write Thread and Process ID: 00000000
User Read-only Thread and Process ID: 00000000
Privileged Only Thread and Process ID: 00000000
Peripheral Port Memory Remap: 00000000

XMD% srrd dbg
Unknown CortexA9 Register name dbg
XMD% srrd debug
Debug ID: 35137030
Debug Status and Control: 02086003
Data Transfer: f8000008
Watchpoint Fault Address: 00000000
Vector Catch: 00000000
Debug State Cache Control: 00000000
Debug State MMU Control: 00000000
Breakpoint Value 0: 00100000
Breakpoint Value 1: 00000000
Breakpoint Value 2: 00000000
Breakpoint Value 3: 00000000
Breakpoint Value 4: 00000000
Breakpoint Value 5: 00000000
Breakpoint Control 0: 004001e6
Breakpoint Control 1: 00000000
Breakpoint Control 2: 00000000
Breakpoint Control 3: 00000000
Breakpoint Control 4: 00000000
Breakpoint Control 5: 00000000
Watchpoint Value 0: 00000000
Watchpoint Value 1: 00000000
Watchpoint Control 0: 00000000
Watchpoint Control 1: 00000000

DMA Identification and Status present: 00100001
DMA Identification and Status queued: 00000000
DMA Identification and Status running: 00000000
DMA Identification and Status interrupting: 00000000
DMA User Accessibility: 00000000
DMA Channel Number: 00000000
DMA Control: 00000000
DMA Internal Start Address: 00000000
DMA External Start Address: 00000000
DMA Internal End Address: 00000000
DMA Channel Status: 00000000

```

```
DMA Context ID: 00000000

Data Cache Lockdown: 00000000
Instruction Cache Lockdown: 00000000
Data TCM Region: 00000000
Instruction TCM Region: 00000000
Data TCM Non-secure Control Access: 00000000
Instruction TCM Non-secure Control Access: 00000000
TCM Selection: 00000000
Cache Behavior Override: 00000000

Main ID: 413fc090
Cache Type: 83338003
TCM status: 00000000
TLB Type: 00000402
Processor Feature 0: 00001231
Processor Feature 1: 00000011
Debug Feature 0: 00010444
Auxiliary Feature 0: 00000000
Memory Model Feature 0: 00100103
Memory Model Feature 1: 20000000
Memory Model Feature 2: 01230000
Memory Model Feature 3: 00102111
Instruction Set Feature Attribute 0: 00101111
Instruction Set Feature Attribute 1: 13112111
Instruction Set Feature Attribute 2: 21232041
Instruction Set Feature Attribute 3: 11112131
Instruction Set Feature Attribute 4: 00011142
Instruction Set Feature Attribute 5: 00000000

XMD% srrd etc
PA: 00000000
Cache Dirty Status: 00000000
TLB Lockdown: 00000000
Primary Region Memory Remap: 00098aa4
Normal Region Memory Remap: 44e048e0
Secure User and Non-secure Access Validation Control: 00000000
Performance Monitor Control: 00000000
Cycle Counter: 00000000
Count 0: 00000000
Count 1: 00000000
Reset Counter: 00000000
Interrupt Counter: 00000000
Fast Interrupt Counter: 00000000
System Validation Cache Size Mask: 00000000
TLB Lockdown Index: 00000000
TLB Lockdown VA: 00000000
TLB Lockdown PA: 000000c6
TLB Lockdown Attributes: 00000000

XMD% srrd vfp
Floating-Point System ID: 41033094
Floating-Point Status And Control: 00000000
Floating-Point Exception: 40000000
Floating-Point Instruction: 40000000
Floating-Point Instruction 2: 40000000
Media and VFP Feature 0: 10110222
Media and VFP Feature 1: 01111111
```

XMD Internal Tcl Commands

In the Tcl interface mode, XMD starts a Tcl shell augmented with XMD commands. All XMD Tcl commands start with **x**, and you can list them from XMD by typing **x?**.

Xilinx recommends using the Tcl wrappers for these internal commands as described in [XMD Options, page 134](#). The Tcl wrappers print the output of most of these commands and provide more options. While the Tcl wrappers are backward-compatible, the **x<name>** commands will be deprecated.

The following Tcl command subsections are:

- [Program Initialization Options](#)
- [Register/Memory Options](#)
- [Program Control Options](#)
- [Program Trace and Profile Options](#)
- [Miscellaneous Commands](#)

Program Initialization Options

Table 10-15: Program Initialization Option

| Option | Description |
|---|--|
| xconnect <target> {mb ppc mdm} <connect type> {options} | Connects to a processor or a peripheral target. Valid target types are mb, ppc, and mdm. Refer to Connect Command Options, page 153 for more information on options. |
| xdebugconfig <target id> [-step_mode <Step Type>] [-memory_datawidth_matching {disable enable}] [-reset_on_run {system enable processor enable disable}] [-reset_on_data_dow {system enable processor enable disable}] [run_poll_interval <time in millisec>] | Configures the debug session for the target. For additional information, refer to the Configure Debug Session, page 175 . |
| xdisconnect [<target id>] [-cable] | Disconnects from the target. Use the -cable option command to disconnect from cable and all targets. |

Table 10-15: Program Initialization Option (Cont'd)

| Option | Description |
|--|--|
| xdownload <target_id> <filename> [load address] xdownload <target_id> -data <filename> <load_address> | <p>Downloads the given ELF or data file, using the -data option, onto the memory of the current target.</p> <p>If no address is provided along with ELF file, the download address is determined from the ELF file headers.</p> <p>Otherwise, it is treated as Position Independent Code (PIC code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics.</p> <p>XMD does not perform bounds checking, with the exception of preventing writes into the XMDSTUB area (address 0x0 to 0x800).</p> |
| xrcableesn | Returns the ESN values of USB cables connected to the host machine. |
| xrjtagchain [-cable <options>] | Returns the Jtag Device Chain information of the board connected to the host machine. |
| xfpga -f <bitstream> [-cable <options>] [-configdevice <configuration_options>] [-debugdevice <device_name>] | Loads the FPGA device bitstream and, optionally, the cable configuration and debug device options. |
| xload_sysfile hw <hw_spec_file> | Loads the hardware specification file. |
| xrut [Session ID] | Authenticates the XMD session when communicating over XMD sockets interface. The session ID is first assigned and subsequent calls return the session ID. |
| xtargets -listSysID xtargets -system <system_ID> [-print] [-listTgtID] xtargets -target <target_ID> {-print -prop} | <p>Provides system and target information in the current XMD session.</p> <p>-listSysID returns a list of existing systems.</p> <p>-system <system_ID> provides information on the specified system.</p> <p>-print prints the different targets in the system</p> <p>-listTgtID returns a list of existing targets in the system.</p> <p>-target <target_ID> provides information on the specified target. The options:</p> <p>-print prints the target information</p> <p>-prop returns the target properties</p> |

Register/Memory Options

Table 10-16: Register/Memory Options

| Option | Description |
|--|---|
| xdata_verify <target id> <Binary filename> <load address> | Verifies if the <Binary filename> was downloaded correctly at <load address> memory. |
| xdisassemble <inst> | Disassembles and displays one 32-bit instruction. |
| xelf_verify <target id> [<filename>.elf] | Verifies if the <filename>.elf is downloaded correctly to memory. If <filename>.elf is not specified, verifies the last downloaded ELF file to target. |
| xrmem <target id> <address> {<number of bytes> half word} {b h w} xrmem <target id> -var <Global Variable Name> | Reads <number of bytes> of memory locations from the specified memory address. Defaults to byte (b) read. Returns a list of data values. The data type depends on the data-width of memory access. |
| xwmem <target id> <address> {<number of bytes> half word} {b h w} <value list> xwmem <target id> -var <Global Variable Name> <value list> | Writes <number of bytes> data value from the specified memory address. Defaults to byte (b) write. |
| xrreg <target id> [reg] | Reads all registers or only register number <reg>. |
| xwreg <target id> [reg] [value] | Writes a 32-bit value into register number <reg>. |
| xstack_check <target id> | Gives the stack usage information of the program running on the current target. The most recent ELF file downloaded on the target is taken into account for stack check. |

Program Control Options

Table 10-17: Program Control Options

| Option | Description |
|--|--|
| xbreakpoint <target id> {addr function name} {sw hw} | Sets a breakpoint at the given address or start of function. Note: Breakpoints on instructions immediately following an IMM instruction can lead to undefined results for an XMDStub target. |
| xcontinue <target id> [<Execute Start Address>] [-block] | Continues from current PC or optionally specified <Execute Start Address>. If -block option is specified, the command returns when the Processor stops on breakpoint or watchpoint. The -block option is useful in scripting. |
| xcycle_step <target id> [cycles] | Cycle steps through one clock cycle of PowerPC processor ISS. If cycles is specified, then step cycles number of clock cycles. ^a |
| xlist <target id> | Lists all of the breakpoint addresses. |

Table 10-17: Program Control Options (Cont'd)

| Option | Description |
|--|---|
| xremove <target id> {<addr> <function name> <bp id> all} | Removes one or more breakpoints or watchpoints. |
| xreset <target id> [reset type] | Resets target. Optionally, provide target-specific reset types such as the signals mentioned in Table 10-18 on page 184 . |
| xrun <target id> | Runs program from the program start address. |
| xstate <target id> | Returns the processor target state; running or stopped. |
| xstep <target id> | Single steps one MicroBlaze instruction. If the PC is at an IMM instruction, the next instruction also runs. During a single step, interrupts are disabled by keeping the BIP flag set. Use xcontinue with breakpoints to enable interrupts while debugging. |
| xstop <target id> | Stops the program execution. |
| xwatch <target id> {r w} <address> [<data value>] | Sets read/write watchpoints at a given <address> and, optionally, check for <data value>. If <data value> is not specified, watchpoints match any value. The address and value can be specified in hex or binary format. |

a. This command is for Simulator targets only.

XMD MicroBlaze Hardware Target Signals

Table 10-18: XMD MicroBlaze Hardware Target Signals

| Signal Name (Value) | Description |
|---------------------------|---|
| Non-maskable Break (0x10) | Similar to the Break signal, but works even while the BIP flag is already set. Refer the <i>MicroBlaze Processor Reference Guide</i> for more information about the BIP flag. A link to the document is supplied in Appendix E, Additional Resources . |
| Processor Break (0x20) | Raises the Brk signal on MicroBlaze using the JTAG UART Ext_Brk signal. It sets the Break-in-Progress (BIP) flag on MicroBlaze and jumps to address 0x18. |
| Processor Reset (0x80) | Resets MicroBlaze using the JTAG UART Debug_Rst signal. |
| System Reset (0x40) | Resets the entire system by sending an OPB Rst using the JTAG UART Debug_SYS_Rst signal. |

Program Trace and Profile Options

Table 10-19: Program Trace/Profile Options

| Option | Description |
|---|--|
| xprofile <target id> [-o <GMON Output File>] xprofile <target id> -config[sampling_freq_hw <value>] [binsize <value>] [profile_mem <start addr>] | Generates profile output that can be read by mb-gprof (MicroBlaze), powerpc-eabi-gprof (PowerPC), or arm-xilinx-eabi-gprof (Cortex A9). Specify the profile configuration sampling frequency in Hz, Histogram binary size, and memory address for collecting profile data. |
| xstats <target id> {options} | Displays the simulation statistics for the current session. Use the reset option to reset the simulation statistics. ^a |
| xtracestart <target id> | Starts collecting trace information. |
| xtracestop <target id> | Stops collecting trace information. ^(a) |

a. This command is for ISS targets only.

Miscellaneous Commands

Table 10-20: Miscellaneous Commands

| Command | Description |
|-----------------|--|
| xclean | Cleans up any Xilinx resources that are using the cable. |
| xhelp | Lists the XMD commands. |
| xverbose | Toggles verbose mode on and off. Dumps debugging information from XMD. |

GNU Debugger

This chapter describes the general usage of the Xilinx® GNU debugger (GDB) for the MicroBlaze™ processor and the PowerPC® (405 and 440) processors.

Overview

GDB is a powerful and flexible tool that provides a unified interface for debugging and verifying MicroBlaze and PowerPC (405 and 440) systems during various development phases. It uses Xilinx Microprocessor Debugger (XMD) as the underlying engine to communicate to processor targets.

Tool Overview

Tool Usage

MicroBlaze GDB usage:

```
mb-gdb <options> executable-file
```

PowerPC GDB usage:

```
powerpc-eabi-gdb <options> executable-file
```

Tool Options

The following options are the most common in the GNU debugger:

-command=FILE

Execute GDB commands from the specified file. Used for debugging in batch and script mode.

-batch

Exit after processing options. Used for debugging in batch and script mode.

-nx

Do not read initialization file `.gdbinit`. If you have issues connecting to XMD (GDB connects and disconnects from XMD target), launch GDB with this option or remove the `.gdbinit` file.

-nw

Do not use a GUI interface.

-w

Use a GUI interface (Default).

Debug Flow using GDB

1. Start XMD from XPS.
2. Connect to the Processor target. This action opens a GDB server for the target.
3. Start GDB from XPS.
4. Connect to Remote GDB Server on XMD.
5. Download the Program and Debug application.

MicroBlaze GDB Targets

The MicroBlaze GNU Debugger and XMD tools support remote targets. Remote debugging is done through XMD. The XMD server program can be started on a host computer with the Simulator target or the Hardware target.

The Cycle-Accurate Instruction Set Simulator (ISS) and the hardware interface provide powerful debugging tools for verifying a complete MicroBlaze system. The debugger `mb-gdb` connects to XMD using the GDB remote protocol over TCP/IP socket connection.

Simulator Target

The XMD simulator is a cycle-accurate ISS of the MicroBlaze system which presents the simulated MicroBlaze system state to GDB.

Hardware Target

With the hardware target, XMD communicates with Microprocessor Debug Module (`mdm`) debug core or an `XMDSTUB` program running on a hardware board through the serial cable or JTAG cable, and presents the running MicroBlaze system state to GDB.

For more information about XMD, refer to [Chapter 10, Xilinx Microprocessor Debugger \(XMD\)](#).

Compiling for Debugging on MicroBlaze Targets

To debug a program, you must generate debugging information when you compile the program. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The `mb-gcc` compiler for the Xilinx MicroBlaze soft processor includes this information when the appropriate modifier is specified.

The `-g` option in `mb-gcc` allows you to perform debugging at the source level. The debugger `mb-gcc` adds appropriate information to the executable file, which helps in debugging the code. The debugger `mb-gdb` provides debugging at source, assembly, and mixed source and assembly.

Note: While initially verifying the functional correctness of a C program, do not use any `mb-gcc` optimization option like `-O2` or `-O3` as `mb-gcc` does aggressive code motion optimizations which might make debugging difficult to follow.

Note: For debugging with XMD in hardware mode using `XMDSTUB`, specify the `mb-gcc` option `-x1-mode-xmdstub`. Refer to [Chapter 10, Xilinx Microprocessor Debugger \(XMD\)](#) for more information about compiling for specific targets.

PowerPC 405 Targets

Debugging for the PowerPC 405 processor is supported by `powerpc-eabi-gdb` and XMD through the GDB Remote TCP protocol. XMD supports two remote targets: PowerPC 405 Hardware and Cycle-Accurate PowerPC Instruction Set Simulator (ISS).

To connect to a PowerPC 405 target:

1. Start XMD and connect to the board using the **connect ppc** command as described in [Chapter 10, Xilinx Microprocessor Debugger \(XMD\)](#).
2. Select **Run > Connect to target** from GDB.
3. In the GDB target selection dialog box, specify the following:
 - Target: **Remote/TCP**
 - Hostname: **localhost**
 - Port: **1234**
4. Click **OK**.

The debugger `powerpc-eabi-gdb` attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD started.

At this point, the debugger is connected to XMD and controls the debugging. The GUI can be used to debug the program and read and write memory and registers.

PowerPC 440 Targets

Debugging for the PowerPC 440 processor is supported by `powerpc-eabi-gdb` and XMD through the GDB Remote TCP protocol.

XMD supports two remote targets: PowerPC 440 Hardware and Cycle-Accurate PowerPC Instruction Set Simulator (ISS).

To connect to a PowerPC 440 target:

1. Start XMD and connect to the board using the **connect ppc** command as described in [Chapter 10, Xilinx Microprocessor Debugger \(XMD\)](#).
2. From GDB select **Run > Connect to target**.
3. In the GDB target selection dialog box, specify the following:
Target: **Remote/TCP**
Hostname: **localhost**
Port: **1234**
4. Click **OK**.
5. The debugger `powerpc-eabi-gdb` attempts to make a connection to XMD, and if successful, prints a message to the shell window where XMD was invoked.
6. Select **View > Console** to open the console window.
7. On the console type:
set arch powerpc:440 to set the architecture to a PowerPC 440 processor.

At this point, the debugger is connected to XMD in PowerPC 440 mode and controls the debugging. The user interface can be used to debug the program and read and write memory and registers.

Console Mode

To start `powerpc-eabi-gdb` in the console mode, type the following:

```
xilinx > powerpc-eabi-gdb -nw executable.elf
```

In the console mode, type the following two commands to connect to the board through XMD:

```
(gdb) target remote localhost:1234
(gdb) load
```

The following text displays:

```
Loading section .text, size 0xfcc lma 0xffff8000
Loading section .rodata, size 0x118 lma 0xffff8fd0
Loading section .data, size 0x2f8 lma 0xffff90e8
Loading section .fixup, size 0x14 lma 0xffff93e0
Loading section .got2, size 0x20 lma 0xffff93f4
Loading section .sdata, size 0xc lma 0xffff9414
Loading section .boot0, size 0x10 lma 0xffffa430
Loading section .boot, size 0x4 lma 0xfffffff0
Start address 0xfffffff0, load size 5168
Transfer rate: 41344 bits/sec, 323 bytes/write.
(gdb) c
Continuing
```

For the console mode, these two commands can also be placed in the GDB startup file `gdb.ini` in the current working directory.

GDB Command Reference

For help on using mb-gdb, select **Help > Help Topics** in the XPS main dialog box or type **help** in the console mode.

To open a console window from the GBD main dialog box, select **View > Console**.

For comprehensive online documentation on using GDB, refer to the GNU web site. For information about the mb-gdb Insight GUI, refer to the Red Hat Insight website. Links to these documents are provided in the [Additional Resources, page 191](#).

[Table 11-1](#) describes the commonly used mb-gdb console commands. The equivalent GUI versions can be identified in the mb-gdb GUI window icons. Some of the commands, such as info target and monitor info, might be available only in the console mode.

Table 11-1: Commonly Used GDB Console Commands

| Command | Description |
|-----------------------|--|
| load <program> | Load the program into the target. |
| b main | Set a breakpoint in function main. |
| c | Continue after a breakpoint. Note: Do not use the <code>run</code> command |
| l | View a listing of the program at the current point. |
| n | Steps one line, stepping over function calls. |
| s | Step one line, stepping into function calls. |
| stepi | Step one assembly line. |
| info reg | View register values. |
| info target | View the number of instructions and cycles executed for the built-in simulator only. |
| p <xyz> | Print the value of <code>xyz</code> data. |
| hbreak main | Set hardware breakpoint in function <code>main()</code> . |
| watch <gvar1> | Set Watchpoint on Global Variable <code>gvar1</code> . |
| rwatch <gvar1> | Set Read Watchpoint on Global Variable <code>gvar1</code> . |

Additional Resources

- GNU website: <http://www.gnu.org>
- Red Hat Insight website: <http://sources.redhat.com/insight>.

Bitstream Initializer (BitInit)

Overview

BitInit initializes the instruction memory of processors on the FPGA, which is stored in block RAMs in the FPGA. This utility reads an Microprocessor Hardware Specification (MHS) file, and invokes the Data2MEM utility provided in Xilinx® ISE® to initialize the FPGA block RAMs.

Tool Usage

To invoke the BitInit tool, type the following:

```
% bitinit <mhsfile> [options]
```

Note: You must specify <mhsfile> before specifying other tool options.

Tool Options

Table 12-1 lists the supported options in BitInit.

Table 12-1: BitInit Syntax Options

| Option | Command | Description |
|-----------------------|----------------------|--|
| Input BMM file | -bm | Specifies the input BMM file which contains the address map and the location of the instruction memory of the processor. Default: implementation/<sysname>_bd.bmm |
| Bitstream file | -bt | Specifies the input bitstream file that does not have its memory initialized. Default: implementation/<sysname>.bit |
| Display Help | -h | Displays the usage menu and then quits. |
| Log file name | -log | Specifies the name of the log file to capture the log. Default: bitinit.log |
| Libraries path | -lp | Specifies the path to repository libraries. This option can be repeated to specify multiple libraries. |
| Output bitstream file | -o | Specifies the name of the output file to generate the bitstream with initialized memory. Default: implementation/download.bit |
| Part name | -p <partname> | Uses the specified part type to implement the design. |

Table 12-1: BitInit Syntax Options (Cont'd)

| Option | Command | Description |
|---|---------------|--|
| Specify the Processor Instance name and list of ELF files | -pe | Specifies the name of the processor instance in associated ELF file that forms its instruction memory. This option can be repeated once for each processor instance in the design. Only one ELF per processor can be initialized into block RAM. |
| Quiet mode | -quiet | Runs the tool in quiet mode. In this mode, it does not print status, warning, or informational messages while running. It prints only error messages on the console. |
| Display version | -v | Displays the version and then quits. |

BitInit also produces a file named `data2mem.dmr`, which is the log file generated during invocation of the Data2MEM utility.

System ACE File Generator (GenACE)

This chapter describes the steps to generate Xilinx® System ACE™ technology configuration files from an FPGA bitstream and Executable Linked Format (ELF) data files. The generated ACE file can be used to:

- Configure the FPGA
- Initialize block RAM
- Initialize external memory with valid program or data
- Bootup the processor in a production system

EDK provides a Tool Command Language (Tcl) script, `genace.tcl`, which uses Xilinx Microprocessor Debug (XMD) commands to generate ACE files. ACE files can be generated for PowerPC® (405 and 440) processors and the MicroBlaze™ processor with Microprocessor Debug Module (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:

- a `genace.tcl` file
- XMD
- iMPACT (from ISE®)

GenACE Features

GenACE:

- Supports PowerPC (405 and 440) processor and the MicroBlaze processor with MDM targets.
- Generates ACE files from hardware (Bitstream) and software (ELF and data) files.
- Initializes external memories on PowerPC (405 and 440) processors and MicroBlaze systems.
- Supports multi-processor systems.
- Supports single and multiple FPGA device systems.

GenACE Model

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. System ACE CF configures devices using Boundary-Scan (JTAG) instructions and a Boundary-Scan Chain. The generated System ACE files support the System ACE CF family of configuration solutions. The System ACE file is generated from a Serial Vector Format (SVF) file, which is a text file that contains 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 an FPGA using iMPACT
- Connecting to the processor target
- Downloading the program and running the program from XMD

These actions 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 block RAM or external memory.
5. Download multiple executable files to block RAM 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 to SVF, ELF to SVF, binary data to SVF, SVF to ACE file.

The following section describes the options available in the `genace.tcl` script.

The Genace.tcl Script

The genace.tcl script uses Xilinx Microprocessor Debug (XMD) commands to generate ACE files. This script is located in the `${XILINX_EDK}/data/xmd/` directory.

Some non-supported boards might require some customization, such as changing the delay of programming after FPGA configuration or modifying the processor reset sequence. For these boards, copy the script to the local directory, make the required changes, and then use it to generate the ACE file.

[Table 13-1](#) list the genace.tcl script command options.

Syntax

```
xmd -tcl genace.tcl [-ace <ACE_file>] [-board <board_type>] [-data
<data_files> <load_address>] [-elf <elf_files>] [-hw <bitstream_file>]
[-jprog {true|false}] [-opt <genace_options_file>] |
[-target <target_type> {ppc_hw|mdm}]
```

Table 13-1: genace.tcl Script Command Options

| Options | Default | Description |
|--|---------|--|
| -ace <ACE_file> | none | The output ACE file. The file prefix should not match any of input files (bitstream, elf, data files) prefix. |
| -board <board_type> [supported_board_list] | none | 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. You must specify the <code>-configdevice</code> and <code>-debugdevice</code> option in the OPT file. Refer to the <code>genace.opt</code> file for details. For Supported board type refer to “Supported Target Boards in Genace.tcl Script” on page 200 . |
| -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. |
| -elf <list_of_Elf_Files> | none | List of ELF files to download. If an SVF file is specified, it is used. |
| -hw <bitstream_file> | none | The bitstream file for the system. If an SVF file is specified, it is used. |
| -jprog [true false] | false | Clear the existing FPGA configuration. This option should not be specified if performing runtime configuration. |

Table 13-1: **genace.tcl** Script Command Options (Cont'd)

| Options | Default | Description |
|---|---------|--|
| -opt <genace_options_file> | none | GenACE options are read from the options file. |
| -target <target_type> [ppc_hw mdm] | ppc_hw | Target to use in the system for downloading ELF or Data file. Target types are: ppc_hw to connect to a PowerPC (405 and 440) processor system. mdm to connects to a MicroBlaze processor system. This assumes the presence of mdm in the system. |

The options can be specified in an options file and passed to the GenACE script. The options syntax is described in [Table 13-2](#).

Table 13-2: **GenACE** File Options

| Options | Default | Description |
|---|---------|--|
| # <Some Text> | none | The line starting with # is treated as a comment. |
| -ace <ACE_file> | none | The output ACE file. The file prefix should not match any input file (bitstream, elf, data files) prefix. |
| -board <board_type> [<user> <supported_board_list>] | none | 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: user for user-specific board. You must also specify the -configdevice and -debugdevice option in the OPT file. Refer to the genace.opt file for details. For a list of supported board types refer to “ Supported Target Boards in Genace.tcl Script ” on page 200. |
| -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 must be specified in the order in which they are connected in the JTAG chain on the board. This option is not available on the command line. Use in OPT file only. |
| -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. |

Table 13-2: GenACE File Options (Cont'd)

| Options | Default | Description |
|---|--|--|
| -debugdevice <i><XMD debug device options></i> [cpu_version <i><version></i>] [mdm_version <i><version></i>] | MB v7 MDM v1 | The device containing either PowerPC (405 or 440) processor or MicroBlaze to debug or configure in the JTAG chain. Specify the <i><XMD debug device options></i> such as: position on the chain (<i>devicenr</i>) number of processors (<i>cpunr</i>) processor options (such as OCM, Cache addresses). For a MicroBlaze system, the script assumes the MicroBlaze v7 processor and MDM v1 versions. The additional options for MicroBlaze versions are: cpu_version {microblaze_v5 microblaze_v6 microblaze_v7 microblaze_v72} The additional MDM versions are: mdm_version {mdm_v1 mdm_v2 mdm_v3} |
| -elf <i><list of Elf or SVF files></i> | none | List of ELF files to download. If an SVF file is specified, it is used. |
| -hw <i><bitstream file></i> | none | The bitstream file for the system. If an SVF file is specified, it is used. |
| -jprog | false | Clear the existing FPGA configuration. Do not specify this option if performing runtime configuration. |
| -start_address <i><processor run address></i> | Start Address of the last ELF file (if ELF file is specified): else none. | Specify the address at which to start processor execution. This is useful when a data file is being loaded and processor should execute from load address. |
| -target <i><target type></i> | ppc_hw | Target to use in the system for downloading ELF/Data file. Target types are: ppc_hw to connect to a PowerPC (405 or 440) processor system mdm to connect to a MicroBlaze system. This assumes the presence of mdm in the system. |

Usage

```
xmd -tcl genace.tcl -jprog -target mdm -hw <implementation/
download.bit> -elf executable1.elf executable2.svf
-data image.bin 0xfe000000 -board ml507 -ace system.ace
```

Preferred genace.opt file:

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board ml507
-target mdm
-elf executable1.elf executable2.svf
-data image.bin 0xfe000000
```

Supported Target Boards in Genace.tcl Script

Table 13-3 lists the boards supported in the `genace.tcl` script.

Table 13-3: Supported Target Boards

| Board name | Board type | Devices in the JTAG Chain |
|----------------------|------------|----------------------------------|
| ML401 | ml401 | XCF32P > XC4VLX25 > XC95144XL |
| ML401 with V4LX25 ES | ml401_es | XCF32P > XC4VLX25-ES > XC95144XL |
| ML402 | ml402 | XCF32P > XC4VSX35 > XC95144XL |
| ML403 | ml403 | XCF32P > XC4VFX12 > XC95144XL |
| ML405 | ml405 | XCF32P > XC4VFX20 > XC95144XL |
| ML410 | ml410 | XC4FX60 |
| ML411 | ml411 | XC4FX100 |
| ML501 | ml501 | XC5vLX50 |
| ML505 | ml505 | XC5vLX50T |
| ML506 | ml506 | XC5vSX50T |
| ML507 | ml507 | XC5VFX70T |
| ML510 | ml510 | XC5VFX130T |

For a custom board, use the **-configdevice** option to specify the JTAG chain and use an OPT file.

Generating ACE Files

System ACE files can be generated for the scenarios in the following subsections. An example OPT file is given for each. Specify the use of the OPT file as follows:

```
xmd -tcl genace.tcl -opt genace.opt
```

For Custom Boards

If your board is not listed in the [Supported Target Boards in Genace.tcl Script, page 200](#), the JTAG Chain configuration of the board can be specified using the **-configdevice** option. The options file in this are:

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board user <= Note: The Board type is user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
devicenr 2 idcode 0x1266093 irlength 14 partname XC2VP20 <= Note: The
JTAG Chain is specified here
-target ppc_hw
-elf executable.elf
```

Single FPGA Device

Hardware and Software Configuration

The options file for hardware and software configuration is:

```
-jprog  
-hw implementation/download.bit  
-ace system.ace  
-board ml501  
-target mdm  
-elf executable1.elf executable2.elf
```

Hardware and Software Partial Reconfiguration

The options file for hardware and software partial reconfiguration is:

```
-hw implementation/download.bit  
-ace system.ace  
-board ml501  
-target mdm  
-elf executable1.elf executable2.elf
```

Hardware Only Configuration

The options file for hardware only configuration is:

```
-jprog  
-hw implementation/download.bit  
-ace system.ace  
-board ml401
```

Hardware Only Partial Reconfiguration

The options file for hardware only partial reconfiguration is:

```
-hw implementation/download.bit  
-ace system.ace  
-board ml501
```

Software Only Configuration

The options file for software only configuration is:

```
-jprog  
-ace system.ace  
-board ml501  
-target mdm  
-elf executable1.elf
```

Generating ACE for a Single Processor in Multi-Processor System

Many of the Virtex[®] family designs contain two PowerPC processors (405 and 440) or the system might contain multiple MicroBlaze processors. To generate an 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 two. The options file for this configuration is:

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
-debugdevice devicenr 1 cpunr 2 <= Note: The cpunr is 2
-target ppc_hw
-elf executable1.elf executable2.elf
```

Multi-Processor System Configuration

The assumed configuration is with two PowerPC processors and a MicroBlaze processor, each loaded with a single ELF file. The board configuration is specified in the options file.

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
# Options for PowerPC Processor 1 - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 1
-target ppc_hw
-elf executable1.elf
# Options for PowerPC Processor 2 - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 2
-target ppc_hw
-elf executable2.elf
# Options for MicroBlaze Processor - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 1
-target mdm
-elf executable3.elf
```

Note: When multi-processors are specified in an OPT file, processor-specific options such as target type, ELF/data files should follow **-debugdevice** option for that processor. The **cpunr** of the processor is inferred from **-debugdevice** option.

Multiple FPGA Devices

The assumed configuration is with two FPGA devices, each with a single processor and 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.

1. Generate an SVF file for the first FPGA device. The options file contains the following:

```
-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 file for the second FPGA device. The options file contains the following:

```
-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`.
Use the following SCR file:

```
svf2ace -wtck -d -m 16776192 -i final_system.svf -o final_system.ace
quit
```

On some boards; for example, the ML561, the FPGA DONE pins are all connected together. For these boards, the FPGAs on the board must be configured with the hardware bitstream at the same time, followed by software configuration. The following are the steps to generate the ACE file for such an configuration. This procedure uses an ML561 board as an example only:

To generate an SVF file for hardware configuration for all FPGAs.

1. Create a SCR file (`impact_download.scr`) with the following contents and invoke the **impact -batch impact_download.scr** command.

```
setMode -cf
setPreference -pref KeepSVF:True
addCollection -name Temp
addDesign -version 0 -name config0
addDeviceChain -index 0
setCurrentDeviceChain -index 0
setCurrentCollection -collection Temp
setCurrentDesign -version 0
addDevice -position 1 -file "ML561_FPGA1_Download.bit"
addDevice -position 2 -file "ML561_FPGA2_Download.bit"
addDevice -position 3 -file "ML561_FPGA3_Download.bit"
generate
quit
```

This generates the SVF file, `config0.svf`.

2. Generate an SVF file for the software on the first FPGA device. The options file contains the following:

```
-jprog
-ace fpga1_sw.ace
-board user
-configdevice devicenr 1 idcode 0x22a96093 irlength 10 partname
xc5vlx50t
-configdevice devicenr 2 idcode 0x22a96093 irlength 10 partname
xc5vlx50t
-configdevice devicenr 3 idcode 0x22a96093 irlength 10 partname
xc5vlx50t
-debugdevice devicenr 1 cpunr 1
-target mdm
-elf executable1.elf
```

This generates the SVF file, `fpga1_sw.svf`.

3. Generate an SVF file for the software on the second FPGA device. The options file contains the following:

```
-jprog
-ace fpga2_sw.ace
-board user
-configdevice devicenr 1 idcode 0x22a96093 irlength 10
partname xc5vlx50t
-configdevice devicenr 2 idcode 0x22a96093 irlength 10
partname xc5vlx50t
-configdevice devicenr 3 idcode 0x22a96093 irlength 10
partname xc5vlx50t
-debugdevice devicenr 2 cpunr 1
-target mdm
-elf executable2.elf
```

This generates the SVF file, `fpga2_sw.svf`.

4. Generate an SVF file for the software on the third FPGA device. The options file contains the following:

```
-jprog  
-ace fpga3_sw.ace  
-board user  
-configdevice devicenr 1 idcode 0x22a96093 irlength 10  
partname xc5vlx50t  
-configdevice devicenr 2 idcode 0x22a96093 irlength 10  
partname xc5vlx50t  
-configdevice devicenr 3 idcode 0x22a96093 irlength 10  
partname xc5vlx50t  
-debugdevice devicenr 3 cpunr 1  
-target mdm  
-elf executable3.elf
```

This generates the SVF file, `fpga3_sw.svf`.

5. Concatenate the files in the following order: `config0.svf`, `fpga1_sw.svf`, `fpga2_sw.svf`, and `fpga3_sw.svf` to `final_system.svf`.
6. Generate the ACE file by calling **impact -batch svf2ace.scr**. Use the following SCR file:

```
svf2ace -wtck -d -i final_system.svf -o final_system.ace  
quit
```

Related Information

CF Device Format

To have the System ACE controller read the CF device, do the following:

1. Format the CF device as FAT16.
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 Help*.

Flash Memory Programming

Overview

You can program the following in flash:

- Executable or bootable images of applications
- Hardware bitstreams for your FPGA
- File system images, data files such as sample data and algorithmic tables

The executable or bootable images of applications is the most common use case. When the processor in your design comes out of reset, it starts executing code stored in block RAM at the processor reset location. Typically, block RAM size is only a few kilobytes or so and is too small to accommodate your entire software application image. You can store your software application image (typically, a few megabytes-worth of data) in flash memory. A small bootloader is then designed to fit in block RAM. The processor executes the bootloader on reset, which then copies the software application image from flash into external memory. The bootloader then transfers control to the software application to continue execution.

The software application you build from your project is in Executable Linked Format (ELF). When bootloading a software application from flash, ELF images should be converted to one of the common bootloadable image formats, such as Motorola S-record (SREC). This keeps the bootloader smaller and more simple. EDK provides interface and command line options for creating bootloaders in SREC format. See the *Xilinx Platform Studio Help* for instructions on creating a flash bootloader and on converting ELF images to SREC. The [Appendix E, Additional Resources](#) contains a link the help.

Flash Programming from XPS

The Xilinx® Platform Studio (XPS) interface includes dialog boxes from which you can program external Common Flash Interface (CFI) compliant parallel flash devices on your board, connected through the external memory controller (EMC) IP cores. The programming solution is designed to be generic and targets a wide variety of flash hardware and layouts.

The programming is achieved through the debugger connection to a processor in your design. XPS downloads and executes a small in-system flash programming stub on the target processor. The in-system programming stub requires a minimum of 8 KB of memory to operate. A host Tcl script drives the in-system flash programming stub with commands and data and completes the flash programming. The flash programming tools do not process or interpret the image file to be programmed, and the tools routinely program the file as-is onto flash memory. Your software and hardware application setup must infer the contents of the file being programmed.

Supported Flash Hardware

The flash programmer uses the Common Flash Interface (CFI) to query the flash devices, so it requires that the flash device be CFI compliant. The layout of the flash devices to form the total memory interface width is also important. The following table lists the supported flash layouts and configurations. If your flash layout does not match a configuration in [Table 14-1](#) you must then customize the flash programming session. Refer to [“Customizing Flash Programming” on page 209](#).

Table 14-1: Supported Flash Configurations

| |
|--|
| x8 only capable device forming an 8-bit data bus |
| x16/x8 capable device in x8 mode forming an 8-bit data bus |
| x32/x8 capable device in x8 mode forming an 8-bit data bus |
| x16/x8 capable device in x16 mode forming a 16-bit data bus |
| Paired x8 only capable devices forming a 16-bit data bus |
| Quad x8 only capable devices forming a 32-bit data bus |
| Paired x16 only capable devices in x16 mode, forming a 32-bit data bus |
| x32 /x8 capable device in x32 mode, forming a 32-bit data bus |
| x32 only capable device forming a 32-bit data bus |

The physical layout, geometry information, and other logical information, such as command sets, are determined using the CFI. The flash programmer can be used on flash devices that use the CFI-defined command sets only. The CFI-defined command sets are listed in [Table 14-2](#).

Table 14-2: CFI Defined Command Sets

| CFI Vendor ID | OEM Sponsor | Interface Name |
|---------------|-------------|----------------------------------|
| 1 | Intel/Sharp | Intel/Sharp Extended Command Set |
| 2 | AMD/Fujitsu | AMD/Fujitsu Standard Command Set |
| 3 | Intel | Intel Standard Command Set |
| 4 | AMD/Fujitsu | AMD/Fujitsu Extended Command Set |

By default, the flash programmer supports only flash devices which have a sector map that matches what is stored in the CFI table. Some flash vendors have top-boot and bottom-boot flash devices; the same common CFI table is used for both. The field that identifies the boot topology of the current device is not part of the CFI standard. Consequently, the flash programmer encounters issues with such flash devices.

Refer to [“Customizing Flash Programming” on page 209](#) for more information about how to work around the boot topology identification field.

The following assumptions and behaviors apply to programming flash hardware:

- Flash hardware is assumed to be in a reset state when programming is attempted by the flash programming stub.
- Flash sectors are assumed to be in an unprotected state.

The flash programming stub does not attempt to unlock or initialize the flash, and reports an error if the flash hardware is not in a ready and unlocked state.

Note: The flash programmer does not currently support dual-die flash devices which require every flash command to be offset with a Device Base Address (DBA) value. Examples of such dual-die devices are the 512 Mbit density devices in the Intel StrataFlash® Embedded Memory (P30) family of flash memory.

Flash Programmer Performance

The following factors determine the speed at which an image can be programmed:

- The flash programmer communicates with the in-system programming stub using JTAG. Consequently, the inherent bandwidth of the JTAG cable is, in most cases, the bottleneck in programming flash.
- When it is available on the system, it is best to use external memory as scratch memory. This will allow the debugger to download the flash image data without having to stream it in multiple iterations.
- It is desirable to implement the fastest configuration possible when using the MicroBlaze soft processor. You can improve programming speed by turning on features such as the barrel shifter and multiplier.

Customizing Flash Programming

Hardware incompatibilities, flash command set incompatibilities, or memory size constraints are considerations when programming flash. This section briefly describes the flash programming algorithm, so that, if necessary, you can plug in and replace elements of the flow to customize it for your particular setup.

When you click the **Program Flash** button and select a hardware platform project, the following sequence of events occurs:

1. A `flash.tcl` file is written out to the `<hardware platform project>/settings` folder. This contains parameters that describe the flash programming session and is used by the flash programmer Tcl file.
2. XPS launches XMD with the flash programmer Tcl script, executing it with a command such as `xmd -nx -hw <hardware platform project>/system.xml -tcl flashwriter.tcl <hardware platform project>/settings/flash.tcl`. This flash programmer host Tcl comes from the installation. You can replace the default `flashwriter.tcl` with your own driver Tcl to run when you click the **Program Flash** button by placing a copy of the `flashwriter.tcl` file in your `<hardware platform project>/tmp` directory. XMD searches for the specified file in your project directory before looking for it in the installation.
3. The flash programmer Tcl script copies the flash programmer application source files from the installation to the `<hardware platform project>/tmp/` folder. It compiles the application locally to execute from the scratch memory address you specified in the dialog box. You can compile your own flash writer sources by modifying your local copy of the `flashwriter.tcl` script to compile your own sources instead of those from the installation.

4. The script downloads the flash programmer to the processor and communicates with the flash programmer through mailboxes in memory.
In other words, it writes parameters to the memory locations corresponding to variables in the flash programmer address space and lets the flash programmer execute.
5. The script waits for the flash programmer to invoke a callback function at the end of each operation and stops the application at the callback function by setting a breakpoint at the beginning of the function. When the flash programmer stops, the host Tcl processes the results and continues with more commands as required.
6. While running, the flash programmer erases only as many flash blocks as required in which to store the image.
7. The flashwriter allocates a streaming buffer (based on the amount of scratch pad memory available) and iteratively stream programs the image file. The stream buffer is allocated within the flashwriter. If there is enough scratch memory to hold the entire image, the programming can be completed quickly.
8. When the programming is done, the flash programmer Tcl sends an exit command to the flash programmer and terminates the XMD session.

The following is an example set of steps to perform for a custom flow:

XPS stores the flash settings and temporary files in the hardware platform project directory. If multiple hardware projects exist in the workspace, the flash programmer dialog box prompts you to select the hardware platform. In the following procedure, *<XPS project>* refers to this hardware platform project.

1. Create a new subdirectory called tmp under the *<XPS project>* directory.
Note: If this folder already exists, skip this step.
2. Copy `flashwriter.tcl` from `<edk_install>/data/xmd/flashwriter.tcl` to your *<XPS project>/tmp* directory.
3. Create a `sw_services` directory within your project.
4. Copy the `<edk_install>/data/xmd/flashwriter` directory to the `/sw_services` directory.
5. Change the following line in the `flashwriter.tcl` file copy:

```
set flashwriter_src [file join $xilinx_edk "data" "xmd" "flashwriter"
"src"]
```

to

```
set flashwriter_src [file join "." "sw_services" "flashwriter" "src"]
```

From this point when you use the **Program Flash Memory** dialog box in XPS, the flash programming tools use the script and the sources you copied into the `sw_services` directory. You can customize these as required.

If you prefer to not have the GUI overwrite the *<XPS project>/settings/flash.tcl* file, run the command `xmd -nx -hw system.xml -tcl tmp/flashwriter.tcl settings/flash.tcl` on the command line to use only the values that you specify in the `flash.tcl` file.

Table 14-3 lists the available parameters in the `<XPS project>` directory.

Table 14-3: Flash Programming Parameters

| Variable | Function |
|-----------------------|---|
| EXTRA_COMPILER_FLAGS | For MicroBlaze, specify any compiler flags required to turn on support for hardware features. For example, if you have the hardware multiplier enabled, add <code>-mno-xl-soft-mul</code> here. Do <i>not</i> set this variable for the PowerPC processors. |
| FLASH_BASEADDR | The base address of the flash memory bank. |
| FLASH_BOOT_CONFIG | Refer to “ Handling Flash Devices with Conflicting Sector Layouts ” on page 212. |
| FLASH_FILE | A string containing the full path of the file to be programmed. |
| FLASH_PROG_OFFSET | The offset within the flash memory bank at which the programming should be done. |
| PROC_INSTANCE | The instance name of the processor used for programming. |
| SCRATCH_BASEADDR | The base address of the scratch memory used during programming. |
| SCRATCH_LEN | The length of the scratch memory in bytes. |
| TARGET_TYPE | The type of the processor instance used for programming: MicroBlaze or PowerPC® (405 or 440) processor. |
| XILINX_PLATFORM_FLASH | To enable use of the Xilinx Platform Flash XL flash device. |
| XMD_CONNECT | The connect command used in XMD to connect to the processor. |

Manual Conversion of ELF Files to SREC for Bootloader Applications

If you want to create SREC images of your ELF file manually instead of using the auto-convert feature in XPS you can use the command line tools. For example, to create a final software application image named `myexecutable.elf`, navigate in the console of your operating system (Cygwin on Windows platforms) to the folder containing this ELF file and type the following:

```
<platform>-objcopy -O srec myexecutable.elf myexecutable.srec
```

where `<platform>` is **powerpc-eabi** if your processor is a PowerPC 405 or 440 processor, or **mb** if your processor is a MicroBlaze.

This creates an SREC file that you can then use as appropriate. The utilities `mb-objcopy` and `powerpc-eabi-objcopy` are GNU binaries that ship with EDK.

For information about creating a bootloader from within a GUI, see the *Xilinx Platform Studio Help*. [Appendix E, Additional Resources](#) contains links to the help.

Operational Characteristics and Workarounds

Handling Xilinx Platform Flash Modes

Xilinx Platform Flash memory devices initialize in synchronous mode. You must set these devices to asynchronous mode before performing device operations. When using the Xilinx Software Development Kit, you can select a check box to inform the Flash programming interface to treat the target device as Xilinx Platform Flash. This setting enables an internal workaround in the programmer that sets the device to asynchronous mode before programming.

Handling Flash Devices with 0xF0 as the Read-Reset Command

The CFI specification defines the read-reset command as 0xFF / 0xF0. By default the flash programmer uses the 0xFF read-reset command. Certain devices require 0xF0 as the read-reset command, however, the flash programmer is unable to determine this automatically. Consequently, you might encounter issues when programming newer devices.

In that event of an error occurring follow the documented steps in [Customizing Flash Programming, page 209](#), then modify the `#define FRR_CMD 0xFF` in the `cfi.c` file to `#define FRR_CMD 0xF0`.

Handling Flash Devices with Conflicting Sector Layouts

Some flash vendors store a different sector map in the CFI table and another (based on the boot topology of the flash device) in hardware. Because the boot topology information is not standardized in CFI, the flash programmer cannot determine the layout of your particular flash device.

If your flash hardware has a sector layout that is different from the one specified in the CFI table for the device, then you must create a custom flash programming flow. You must determine whether the device is a top-boot or a bottom-boot flash device.

In a top-boot flash device, the smallest sectors are the last sectors in the flash. In a bottom-boot flash device, the smallest sectors are the first sectors in the flash layout.

After you determine the flash device type, you must copy over the files to create a custom programming flow.

- If you have a bottom-boot flash, add the following line in your `/etc/flash_params.tcl` file:

```
set FLASH_BOOT_CONFIG BOTTOM_BOOT_FLASH
```
- If you have a top-boot flash, add the following line in your `/etc/flash_params.tcl` file:

```
set FLASH_BOOT_CONFIG TOP_BOOT_FLASH
```

Next, run the flash programming from the command line with the following command:

```
xmd -tcl flashwriter.tcl
```

Internally, these variables cause the flash programmer to rearrange the sector map according to the boot topology.

Data Polling Algorithm for AMD/Fujitsu Command Set

The DQ7 data polling algorithm is used during erasure and programming operations on flash hardware that supports the AMD/Fujitsu command set.

Certain flash devices are known to use a configuration register to control the behavior of the data polling DQ7 bit. Some known flash devices that offer this configuration register feature are: AT49BV322A(T), AT49BV162A(T), and AT49BV163A(T).

It is required that DQ7 output 0 during an erase operation and 1 at the end of the operation. Similarly, DQ7 must output inverted data during programming and the actual data after programming is done. If your flash hardware has a different configuration when using the Program Flash Memory dialog box, then the programming could fail.

Refer to your flash hardware datasheet for information about how to reset the configuration so that DQ7 has the appropriate outputs upon erasure and ending.

Version Management Tools (revup)

When you open an older project with the current version of EDK, the Format Revision Tool (revup) automatically performs format changes to an existing EDK project and makes that project compatible with the current version.

The revup tool performs backups of existing files, such as Xilinx® Microprocessor Project (XMP), Microprocessor Hardware Specification (MHS), and Microprocessor Software Specification (MSS) applying format changes. The tool stores these backup files in the /revup folder of the project directory.

Updates to IP and drivers, if any, are handled by the Version Management wizard, which launches after the revup tool runs. The format revision tool does not modify the IPs used in the MHS design; it only updates syntax, so the project can be opened with the new tools.

The revup tool creates a backup of your files and a file name extension that specifies the EDK release number. For example, EDK 13.1 files are saved with a .131 extension and then modified for EDK 14.x tools.

Command Line Option for the Format Revision Tool

Run the Format Revision tool from the command line as follows:

```
revup system.xmp
```

The revup tool supports the **-h** (Help) option, which displays the usage menu and then quits.

The Version Management Wizard

When revup runs it opens the Version Management wizard. The Version Management wizard:

- Outlines any modifications to IP cores that were obsoleted or updated since the project was last updated.
- Provides the option to automatically upgrade to the latest backward-compatible revision or provides more information on how to upgrade to the latest version of the core.
- Provides the option to make similar updates for drivers, if required.

You can choose to cancel the wizard at any time without modifying the files, but, as a result, it might not be possible to run the project with the current EDK version.

Microprocessor Peripheral Definition Translation tool (MPDX)

XBD2

The Xilinx® Base Description (XBD) file defines the supported interfaces of a given board, system, or sub-system. XBD enables you to create a system-level design through the Base System Builder (BSB) in Xilinx Platform Studio (XPS), without the requirement of reading a board schematic or making pin constraint assignments. The following information is included for a given board: FPGA architecture/family/speed grade, I/O list, I/O configuration, and peripheral constraints.

The BSB reads IP-XACT natively when targeting Advanced eXtensible Interface (AXI) designs. The IP-XACT-based board file set is referenced as XBD2. XBD2 models the FPGA device in IP-XACT as a component XML description which defines the interfaces available on the board. This allows designers familiar with IP-XACT to define a data-driven mechanism leveraging the BSB system data to assemble designs.

For board designers not familiar with IP-XACT, the board description can be captured in an ASCII text file similar to the Microprocessor Peripheral Definition (MPD) format defined to capture a pcore description. This MPD file is known as the Board-MPD. It includes a translation tool, MPDX, which generates the IP-XACT files on disk for the BSB repository.

Constraints are captured in a Comma Separated Value (CSV) file and a Tcl file that you provide. EDK provides the CSV file to capture pin constraints and the Tcl file to capture more complex constraints such as timing constraints.

Note: Throughout the document, any reference to Board-MPD is the input to MPDX translation tool, and reference to XBD2 (IP-XACT) is the output of MPDX.

The XBD2 file contains a number of `spirit:busInterface` elements, each corresponding to a hardware module on the board. The type of the module is specified using the `Vendor|Library|Name|reVersion` reference of the `spirit:busDefinition`. The VLNV string is used to match an IP that can communicate with this module.

[Table 16-1, page 218](#) defines the migration of XBD `IO_INTERFACE` definitions to IP-XACT bus definition XML equivalents. All XBD2 component XML files reference the bus definitions outlined in this table. A `V|L|N|V` (VLNV) reference is provided, where:

- V is the vendor.
- L is the library catalog of the vendor.
- N is the name of the board.
- V is the board revision number.

Table 16-1: IO_INTERFACE Details

| XBD Feature | IOTYPE XBD2IP-XACT Equivalent Vendor Lib Board_Name Version | Comments |
|--------------------|--|-------------------------|
| XIL_CLOCK_V1 | xilinx.com bsb_lib.rtl_busdefs clock 1.0 | Clock |
| XIL_RESET_V1 | xilinx.com bsb_lib.rtl_busdefs reset 1.0 | Reset |
| XIL_TEMAC_V1 | xilinx.com bsb_lib.rtl_busdefs gmii 1.0 | GMII |
| XIL_IIC_V1 | xilinx.com bsb_lib.rtl_busdefs i2c 1.0 | IIC |
| XIL_MEMORY_V1 | xilinx.com bsb_lib.rtl_busdefs ddr3_sdram 1.0 | DDR3 SDRAM |
| XIL_MEMORY_V1 | xilinx.com bsb_lib.rtl_busdefs ddr2_sdram 1.0 | DDR2 SDRAM |
| XIL_PCI_ARBITER_V1 | Not supported | PCI - arbitration_group |
| XIL_PCIE_V1 | Not supported | PCI Express |
| XIL_CPUDEBUG_V1 | Not supported | JTAG |
| XIL_TRACE_V1 | N/A | |
| XIL_ETHERNET_V1 | xilinx.com bsb_lib.rtl_busdefs mii 1.0 | MII |
| XIL_GPIO_V1 | xilinx.com bsb_lib.rtl_busdefs gpio 1.0 | GPIO |
| XIL EMC_V1 | xilinx.com bsb_lib.rtl_busdefs flash_nor 1.0 | NOR flash |
| XIL_PS2 _V1 | N/A | |
| XIL_SPI_V1 | xilinx.com bsb_lib.rtl_busdefs spi 1.0 | |
| XIL_SYSACE_V1 | xilinx.com bsb_lib.rtl_busdefs sysace 1.0 | |
| XIL_TFT_V1 | N/A | |
| XIL_UART_V1 | xilinx.com bsb_lib.rtl_busdefs uart 1.0 | UART |

MPDX

Given a Board MPD input file, MPDX generates the IP-XACT equivalent repository files for BSB.

BSB requires two IP-XACT files to capture the design requirements. One file is the RTL description of the IO interfaces that capture the port direction, port width, and port names. The RTL filename is `<board>.xml`.

Use the following command to generate the file:

```
% mpdx -mpd_data board -ipx_data rtl board.mpd
```

The other file is `BSB_Component.xml`, which captures a high-level representation of the system.

Use the following command to generate the file:

```
% mpdx -mpd_data board -ipx_data hurri board.mpd
```

Board MPD

The following are detailed descriptions and examples of each element in the Board MPD.

On parameters and ports, a logical to physical mapping is defined with the `IO_IS` and `IO_IF` tags. The mapping names are listed in the IP-XACT description of the high level components. Some naming conventions are listed here.

- Parameter names: `<interfaceName>_paramName`
- Port names: `<interfaceName>_portName`
- `IO_IF` tag: `<interfaceName>`
- The `IO_IS` is taken from the `spirit:id` defined for the parameter within the high level component.
- The high level IP-XACT component files reside in `$XILINX_EDK/data/wizards/ipxact/hurri/xilinx.com/components/`.

Board Options

The `VLNV` reference is used as follows:

- `V` is the name of the vendor. Tools use this element to sort various board files based on vendor name.
- `L` is the library catalog of the vendor.
- `N` is the name of the board. This is the name the tools display for you when a board is selected.
- `V` is the board revision number.

An example is:

```
OPTION VLNV = xilinx.com|bsb_lib.boards|sp605|C
```

Reference Clock

```
IO_INTERFACE IO_IF = gclk, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|clock|1.0
PARAMETER refclk_frequency_0 = 200000000, DT = LONG,
ASSIGNMENT=CONSTANT, IO_IF =
clock_0, IO_IS = frequency
PORT GCLK = "", DIR = I, IO_IF = gclk, IO_IS = CLK, SIGIS=CLK,
ASSIGNMENT=REQUIRE
```

Reference Reset

```
IO_INTERFACE IO_IF = rst_1, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|reset|1.0
PARAMETER reset_polarity = 1, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = reset_0, IO_IS = RST_POLARITY
PORT RESET_N = "", DIR = I, IO_IF = rst_1, IO_IS = RESET, SIGIS=RST,
ASSIGNMENT=REQUIRE
```

UART

```

IO_INTERFACE IO_IF = RS232_Uart_1, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|uart|1.0
PORT RS232_Uart_1_sout = "", DIR = O, IO_IF = RS232_Uart_1, IO_IS =
sout
PORT RS232_Uart_1_sin = "", DIR = I, IO_IF = RS232_Uart_1, IO_IS =
sin

```

GPIO

```

IO_INTERFACE IO_IF = DIP_Switches_8Bits, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|gpio|1.0
PARAMETER DIP_Switches_8Bits_GPIO_WIDTH_ID = 8, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = DIP_Switches_8Bits, IO_IS =
C_GPIO_WIDTH
PARAMETER DIP_Switches_4Bits_ALL_INPUTS_ID = 1, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = DIP_Switches_8Bits, IO_IS =
C_ALL_INPUTS
PARAMETER DIP_Switches_4Bits_IS_DUAL_ID = 0, DT = STRING, IO_IF =
DIP_Switches_4Bits, IO_IS = C_IS_DUAL
PORT DIP_Switches_8Bits_TRI_I = "", DIR = I, VEC = [7:0], IO_IF =
DIP_Switches_8Bits, IO_IS = TRI_I

```

DDR2 SDRAM

```

IO_INTERFACE IO_IF = MCB_DDR2, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|ddr2_sdram|1.0
PARAMETER C_MEM_PARTNO_ID = EDE1116AXXX-8E, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS = C_MEM_PARTNO
PARAMETER C_BYPASS_CORE_UCF_ID = 0, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS = C_BYPASS_CORE_UCF
PARAMETER C_MEM_TRAS_ID = 45000, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TRAS
PARAMETER C_MEM_TRCD_ID = 12500, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TRCD
PARAMETER C_MEM_TRFC_ID = 127500, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TRFC
PARAMETER C_MEM_TRP_ID = 12500, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TRP
PARAMETER C_MEM_TRP_ID = 12500, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TRP
PARAMETER C_MEM_TYPE_ID = DDR2, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_TYPE
PARAMETER C_MEM_BURST_LEN_ID = 4, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MEM_BURST_LEN
PARAMETER C_MEM_CAS_LATENCY_ID = 5, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS = C_MEM_CAS_LATENCY
PARAMETER C_MEM_DDR2_RTT_ID = 500HMS, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS = C_MEM_DDR2_RTT
PARAMETER C_MEM_DDR2_DIFF_DQS_EN_ID = YES, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS =
C_MEM_DDR2_DIFF_DQS_EN
PARAMETER C_MCB_RZQ_LOC_ID = L6, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = MCB_DDR2, IO_IS = C_MCB_RZQ_LOC
PARAMETER C_MCB_ZIO_LOC_ID = C2, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS =

```

```

C_MCB_ZIO_LOCPARAMETER MEMORY_0_BASEADDR_ID = 0x00000000, DT =
STRING, ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS =
MEMORY_0_BASEADDR
PARAMETER MEMORY_0_HIGHADDR_ID = 0x07ffffff, DT = STRING,
ASSIGNMENT=CONSTANT, IO_IF = MCB_DDR2, IO_IS = MEMORY_0_HIGHADDR
PORT mcbx_dram_clk = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = clk
PORT mcbx_dram_clk_n = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = clk_n
PORT mcbx_dram_cke = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = cke
PORT mcbx_dram_odt = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = odt
PORT mcbx_dram_ras_n = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = ras_n
PORT mcbx_dram_cas_n = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = cas_n
PORT mcbx_dram_we_n = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = we_n
PORT mcbx_dram_ldm = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = ldm
PORT mcbx_dram_udm = "", DIR = I, IO_IF = MCB_DDR2, IO_IS = udm
PORT mcbx_dram_ba = "", DIR = I, VEC = [2:0], IO_IF = MCB_DDR2,
IO_IS = ba
PORT mcbx_dram_addr = "", DIR = I, VEC = [12:0], IO_IF = MCB_DDR2,
IO_IS = addr
PORT mcbx_dram_dq = "", DIR = IO, VEC = [15:0], IO_IF = MCB_DDR2,
IO_IS = dq
PORT mcbx_dram_dqs = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS = dqs
PORT mcbx_dram_dqs_n = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS = dqs_n
PORT mcbx_dram_udqs = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS = udqs
PORT mcbx_dram_udqs_n = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS =
udqs_n
PORT rzq = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS = rzq
PORT zio = "", DIR = IO, IO_IF = MCB_DDR2, IO_IS = zio

```

NOR FLASH

```

IO_INTERFACE IO_IF = Linear_Flash, IO_TYPE =
xilinx.com|bsb_lib.rtl_busdefs|flash_nor|1.0
PARAMETER Linear_Flash_PHY_TYPE_0 = Linear_Flash, DT = STRING, ASSIGNMENT=CONSTANT,
IO_IF = Linear_Flash, IO_IS = PHY_TYPE
PARAMETER Linear_Flash_MEM_WIDTH_0 = 16, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = MEM_WIDTH
PARAMETER Linear_Flash_MEM_SIZE_0 = 33554432, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = MEM_SIZE
PARAMETER Linear_Flash_TCEDV_PS_0 = 130000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = TCEDV_PS
PARAMETER Linear_Flash_TAVDV_PS_0 = 130000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = TAVDV_PS
PARAMETER Linear_Flash_THZCE_PS_0 = 35000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = THZCE_PS
PARAMETER Linear_Flash_THZOE_PS_0 = 7000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = THZOE_PS
PARAMETER Linear_Flash_TWC_PS_0 = 13000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = TWC_PS
PARAMETER Linear_Flash_TWP_PS_0 = 70000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = TWP_PS
PARAMETER Linear_Flash_TLZWE_PS_0 = 35000, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = TLZWE_PS
PARAMETER Linear_Flash_EXCLUSIVE = SPI_FLASH, DT = STRING, ASSIGNMENT=CONSTANT, IO_IF =
Linear_Flash, IO_IS = EXCLUSIVE
PORT Linear_Flash_address = "", DIR = O, VEC = [0:23], IO_IF = Linear_Flash, IO_IS =
address
PORT Linear_Flash_data = "", DIR = IO, VEC = [0:15], IO_IF = Linear_Flash, IO_IS = data
PORT Linear_Flash_ce_n = "", DIR = O, IO_IF = Linear_Flash, IO_IS = ce_n
PORT Linear_Flash_oe_n = "", DIR = O, IO_IF = Linear_Flash, IO_IS = oe_n
PORT Linear_Flash_we_n = "", DIR = O, IO_IF = Linear_Flash, IO_IS = we_n
PORT Linear_Flash_reset = "", DIR = O, IO_IF = Linear_Flash, IO_IS = reset
PORT Linear_Flash_adv_n = "", DIR = O, IO_IF = Linear_Flash, IO_IS = adv_n

```

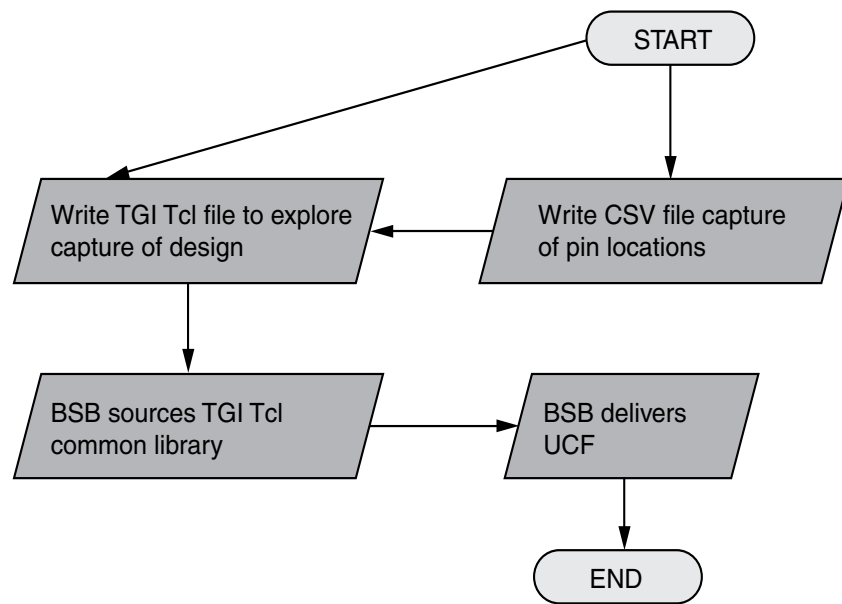
Define Constraints

Constraints are captured in a user-provided CSV file and a Tcl file. The CSV file contains pin constraints and the Tcl file contains more complex constraints like timing constraints. The file names are:

- CSV file: `<board>_pins.csv`
- Tcl file: `<board>.tcl`

XBD2 constraint specification is done with TGI calls to IP-XACT data model that explores the topology of the design. Pin location constraints are associated with the port element within the model capture of component XML. Constraints are in UCF format.

Figure 16-1 illustrates the constraint delivery model.



X12096

Figure 16-1: Constraint Delivery Model

CSV Pin File

Designers often use a CSV file during FPGA design to capture pin locations. The BSB framework uses the following as a standard:

- The CSV defines two mandatory columns: `Pin Name`, and `Pin Index`.
- Other columns, such as `LOC`, are optional.
- You can add or remove additional pin properties by adding or removing a column.

The `Pin Name` must be present and the field must match the name used in the Board MPD file. The `Pin Index` column must be present and can have an empty field.

```
Pin Name,Pin Index,LOC,DRIVE,IOSTANDARD,SLEW,TIG
CLK_P,,K15,,,,
CLK_N,,K16,,,,
RESET,,N4,,,,TIG
RS232_Uart_1_sout,,L12,,,,
RS232_Uart_1_sin,,K14,,,,
RS232_Uart_1_ctsN,,U10,,,,
RS232_Uart_1_rtsN,,T5,,,,
DIP_Switches_4Bits_TRI_I,0,D14,,LVCMOS25,,
DIP_Switches_4Bits_TRI_I,1,E12,,LVCMOS25,,
DIP_Switches_4Bits_TRI_I,2,F12,,LVCMOS25,,
DIP_Switches_4Bits_TRI_I,3,V13,,LVCMOS25,,
```

TCL

The BSB framework supports TGI calls through Tcl. TGI calls are defined in IP-XACT documentation.

Tcl and ConstraintMan

```
bsb::definePinAttribute { nCHandle strPinName strAttName strAttValue }
```

Where:

- `nCHandle`—Is the instance pointer of ConstraintManager.
- `strPinName`—Is a concatenation of `Pin Name` and `Pin Index` fields defined in the CSV. For example, "DIP_Switches_4Bits_TRI_I[0].
- `strAttName`—Is the attribute name. For example: `LOC`.
- `strAttValue`—Is the attribute value.

```
bsb:: getRepoDirPath { nCHandle }
```

Where:

- `nCHandle`—Is the instance pointer of ConstraintManager.

```
bsb::readPinData { strCsvPinFile }
```

Where:

- `strCsvPinFile`—Is the file that defines the CSV used for pins.

```
bsb::registerPinData { nCHandle nComIdXbd nDesignId strCsvFilePath }
```

- `nCHandle` —Is the instance pointer of ConstraintManager.
- `nComIdXbd`—Is the board MPD in memory.
- `nDesignID`—Is the HURRI design constructed by BSB.
- `strCsvFilePath`—Is the path to the CSV file.

```
bsb::registerRawUcfFile { nCHandle strUcfFilePath }
-   nCHandle—Is the instance pointer of ConstraintManager.
-   strUcfFilePath—Is the path to the UCF.

bsb::registerRawUcfFileForBusIf { nCHandle nDesignId vecBusIf }
-   nCHandle —Is the instance pointer of ConstraintManager.
-   nDesignID—Is the HURRI design constructed by BSB.
```

Example

The following example shows how to use the CSV pin file in a script:

```
# nCHandle is instance pointer of ConstraintManager
# nComIdXbd is the SP605
# nDesignID is the HURRI design

proc RunUcfConstraintGen { nCHandle nComIdXbd nDesignId } {
    set nResult 0

    if { $nCHandle eq "" } {
        return $nResult
    }

    if { $nComIdXbd eq "" } {
        return $nResult
    }

    if { $nDesignId eq "" } {
        return $nResult
    }

    set bApiStatus [ tgi::init "1.0" "fail" "Client connected" ]
    if { $bApiStatus == 0 } {
        return 1
    }

    # Repository path
    set strRepoDirPath [ bsb::getRepoDirPath $nCHandle ]

    # Pin Constraints
    set strCsvFilePath [ file join $strRepoDirPath "sp605_pins.csv" ]

    set nResult [ \
        bsb::registerPinData $nCHandle $nComIdXbd $nDesignId
        $strCsvFilePath \
    ]
    if { $nResult != 0 } {
        return $nResult
    }

    return $nResult
}
```


GNU Utilities

This appendix describes the GNU utilities available for use with EDK.

General Purpose Utility for MicroBlaze and PowerPC

cpp

Pre-processor for C and C++ utilities. The preprocessor is invoked automatically by GNU Compiler Collection (GCC) and implements directives such as file-include and define.

gcov

This is a program used in conjunction with GCC to profile and analyze test coverage of programs. It can also be used with the `gprof` profiling program.

Note: The `gcov` utility is not supported by XPS or SDK, but is provided as is for use if you want to roll your own coverage flows.

Utilities Specific to MicroBlaze and PowerPC

Utilities specific to MicroBlaze™ have the prefix “mb-,” as shown in the following program names. The PowerPC® processor versions of the programs are prefixed with “powerpc-eabi.”

mb-addr2line

This program uses debugging information in the executable to translate a program address into a corresponding line number and file name.

mb-ar

This program creates, modifies, and extracts files from archives. An archive is a file that contains one or more other files, typically object files for libraries.

mb-as

This is the assembler program.

mb-c++

This is the same cross compiler as `mb-gcc`, invoked with the programming language set to C++. This is the same as `mb-g++`.

mb-c++filt

This program performs name demangling for C++ and Java function names in assembly listings.

mb-g++

This is the same cross compiler as `mb-gcc`, invoked with the programming language set to C++. This is the same as `mb-c++`.

mb-gasp

This is the macro preprocessor for the assembler program.

mb-gcc

This is the cross compiler for C and C++ programs. It automatically identifies the programming language used based on the file extension.

mb-gdb

This is the debugger for programs.

mb-gprof

This is a profiling program that allows you to analyze how much time is spent in each part of your program. It is useful for optimizing run time.

mb-ld

This is the linker program. It combines library and object files, performing any relocation necessary, and generates an executable file.

mb-nm

This program lists the symbols in an object file.

mb-objcopy

This program translates the contents of an object file from one format to another.

mb-objdump

This program displays information about an object file. This is very useful in debugging programs, and is typically used to verify that the correct utilities and data are in the correct memory location.

mb-ranlib

This program creates an index for an archive file, and adds this index to the archive file itself. This allows the linker to speed up the process of linking to the library represented by the archive.

mb-readelf

This program displays information about an Executable Linked Format (ELF) file.

mb-size

This program lists the size of each section in the object file. This is useful to determine the static memory requirements for utilities and data.

mb-strings

This is a useful program for determining the contents of binary files. It lists the strings of printable characters in an object file.

mb-strip

This program removes all symbols from object files. It can be used to reduce the size of the file, and to prevent others from viewing the symbolic information in the file.

Other Programs and Files

The following Tcl and Tk shells are invoked by various front-end programs:

- cygitclsh30
- cygitkwish30
- cygtclsh80
- cygwish80
- tix4180

Interrupt Management

This appendix describes how to set up interrupts in a Xilinx® embedded hardware system. Also, this appendix describes the software flow of control during interrupts and the software APIs for managing interrupts. To benefit from this description, you need to have an understanding of hardware interrupts and their usefulness.

Hardware Setup

You must first wire the interrupts in your hardware so the processor receives interrupts.

The MicroBlaze™ processor has a single external interrupt port called `Interrupt`. The PowerPC® 405 processor and the PowerPC 440 processor each have two ports for handling interrupts. One port generates a *critical* category external interrupt and the other port generates a *non-critical* category external interrupt, the difference between the two categories being the priority level over other competing interrupts and exceptions in the system. The critical category has the highest priority.

- On the PowerPC 405 processor, the critical and non-critical interrupt ports are named `EICC405CRITINPUTIRQ` and `EICC405EXTINPUTIRQ` respectively.
- On the PowerPC 440 processor, the critical and non-critical interrupt ports are named `EICC440CRITIRQ` and `EICC440EXTIRQ` respectively.

There are two ways to wire interrupts to a processor:

- The interrupt signal from the interrupting peripheral is directly connected to the processor interrupt port. In this configuration, only one peripheral can interrupt the processor.
- The interrupt signal from the interrupting peripheral is connected to an interrupt controller core which in turn generates an interrupt on a signal connected to the interrupt port on the processor. This allows multiple peripherals to send interrupt signals to a processor. This is the more common method as there are usually more than one peripheral on embedded systems that require access to the interrupt function.

Figure B-1, page 232 illustrates the interrupt configurations.

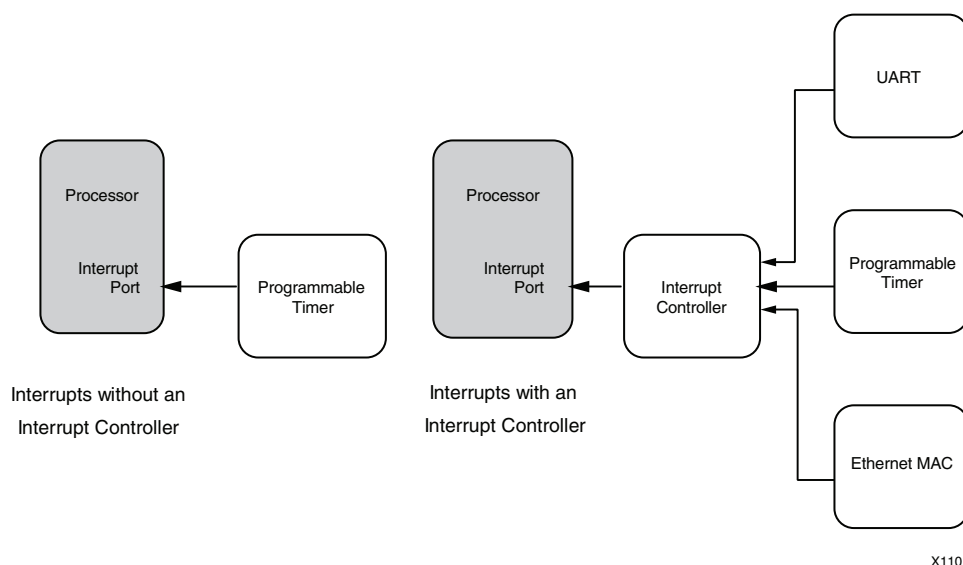


Figure B-1: Interrupt Configurations

Software Setup and Interrupt Flow

Interrupts are typically vectored through multiple levels in the software platform before the application interrupt handlers are executed. The Xilinx software platforms (Standalone and XilKernel) follow the interrupt flow shown in Figure B-2.

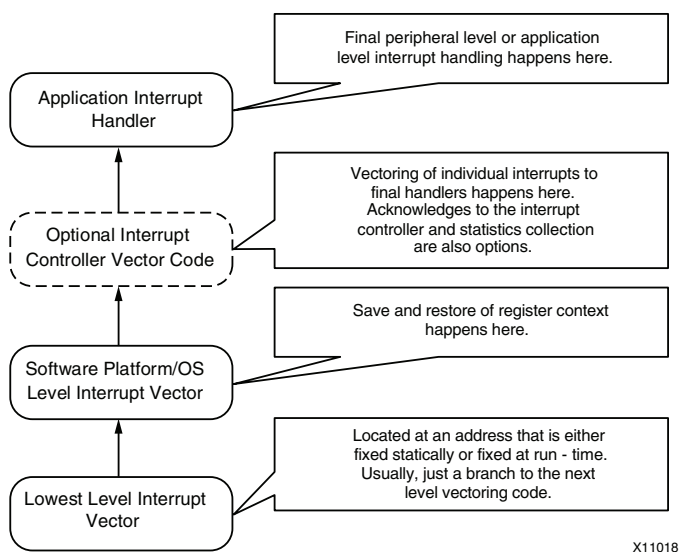


Figure B-2: Interrupt Flow

Interrupt Flow for MicroBlaze Systems

MicroBlaze interrupts go through the following flow:

1. Interrupts have to be enabled on MicroBlaze by setting appropriate bits in the Machine Status Registers (MSR).
 2. Upon an external interrupt signal being raised, the processor first disables further interrupts. Then, the processor jumps to an absolute, fixed address `0x0000_0010`.
 3. The software platform or OS provides vectoring code at this address which transfers control to the main platform interrupt handler.
 4. The platform interrupt handler saves all of the processor registers (that could be clobbered further down) onto the current application stack.
 5. The handler then transfers control to the next level handler. Because the next level handler can be dependent on whether there is an interrupt controller in the system or not, the handler consults an internal interrupt vectoring table to determine the function address of the next level handler. It also consults the vectoring table for a callback value that it must pass to the next level handler. Finally, the actual call is made.
 - On systems with an interrupt controller, the next level handler is the handler provided by the interrupt controller driver. This handler queries the interrupt controller for all active interrupts in the system. For each active interrupt, it consults its internal vector table, which contains the user registered handler for each interrupt line. If the user has not registered any handler, a default do-nothing handler is registered. The registered handler for each interrupt gets invoked in turn (in interrupt priority order).
 - On systems without an interrupt controller, the next handler is the final interrupt handler that the application wishes to execute.
 6. The final interrupt handler for a particular interrupt typically queries the interrupting peripheral and determines the cause for the interrupt. It does a series of actions that are appropriate for the given peripheral and the cause for the interrupt. The handler is also responsible for acknowledging the interrupt at the interrupting peripheral. After the interrupt handler is finished, it returns back and the interrupt stack gets unwound all the way back to the software platform level interrupt handler.
 7. The platform level interrupt handler restores the registers it saved on the stack and returns control back to the Program Counter (PC) location where the interrupt occurred. The return instruction also enables interrupts again on the MicroBlaze processor. The application resumes normal execution at this point.
- Xilinx recommends that interrupt handlers be kept to a short duration and the bulk of the work be left to the application to handle. This prevents long lockouts of other (possibly higher priority) interrupts and is considered good system design.

Figure B-3, page 234 shows a MicroBlaze interrupt flow without an interrupt controller, and Figure B-4, page 234 shows a MicroBlaze interrupt flow with an interrupt controller.

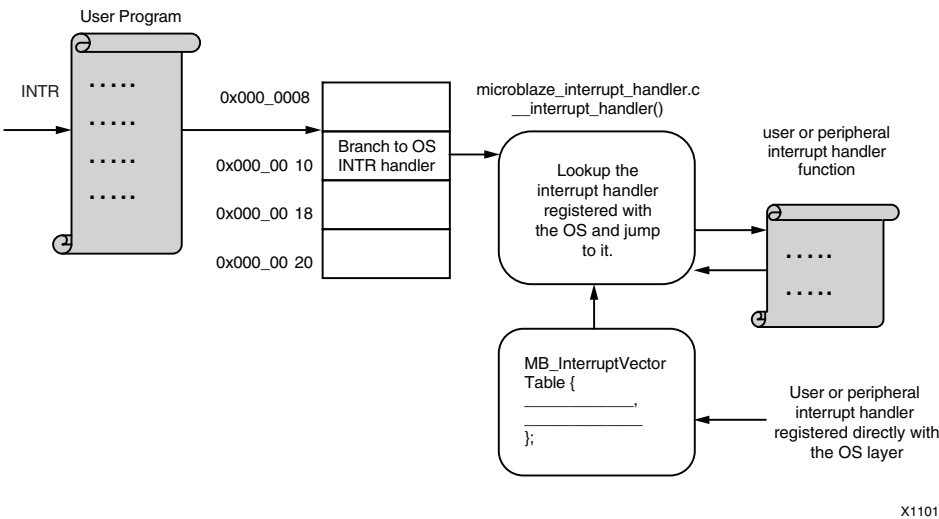


Figure B-3: MicroBlaze Interrupt Flow without Interrupt Controller

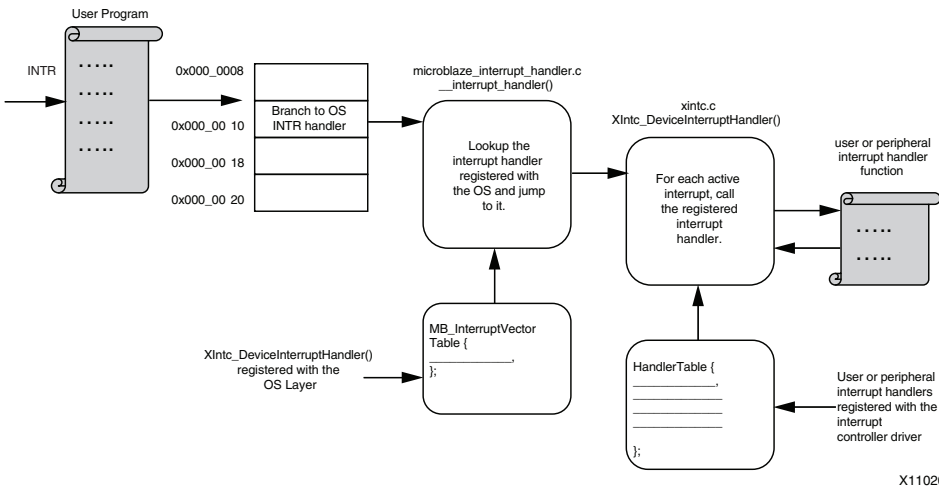


Figure B-4: MicroBlaze Interrupt Flow with Interrupt Controller

Interrupt Flow for PowerPC Systems

Interrupts on the PowerPC processors go through the following flow:

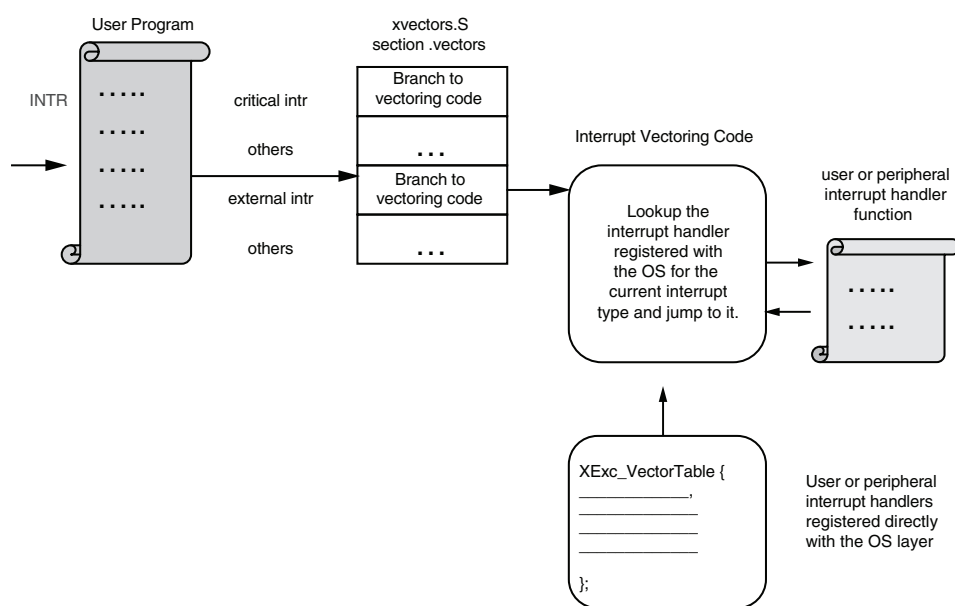
1. Interrupts must be enabled on the PowerPC processor by setting appropriate bits in the Machine Status Registers (MSR). Depending on whether critical or non-critical (or both) interrupts are being used, appropriate bits must be set.
2. Upon the external interrupt signal being raised, the processor first disables further interrupts. The processor then calculates an address for the interrupt type and jumps to that address. The calculation varies between the PowerPC 405 processor and the PowerPC 440 processor.
 - The PowerPC 405 processor consults the software-set value of the Exception Vector Prefix Register (EVPR) and adds a constant offset to this value (depending on the interrupt type) to determine the final physical address where the vector code is placed.
 - The PowerPC 440 processor has independent offset registers for each interrupt type (labeled `IVOR0-IVOR15`). Each offset register contains a value that is appended to the Interrupt Vector Prefix register (IVPR) to obtain the final physical address of the interrupt vector code.
3. The processor jumps to the calculated interrupt vector code address.
4. Each interrupt vector location contains a platform interrupt handler that is appropriate for the interrupt type:
 - For external critical and non-critical interrupts, the handler saves all of the processor registers (that could be clobbered further down) onto the current application stack.
 - The handler then transfers control to the next level handler. Because this can be dependent on whether there is an interrupt controller in the system, the handler consults an internal interrupt vectoring table to determine the function address of the next level handler.
 - The handler also consults the vectoring table for a callback value that it must pass to the next level handler. Then, the handler makes the actual call.
 - On systems with an interrupt controller, the next level handler is the handler provided by the interrupt controller driver. This handler queries the interrupt controller for all active interrupts in the system. For each active interrupt, it consults its internal vector table, which contains the user-registered handler for each interrupt line.
If no handler is registered, a default do-nothing handler is registered. The registered handler for each interrupt gets invoked in turn (in interrupt priority order).
 - On systems without an interrupt controller, the next handler is the final interrupt handler that is executed by the application.
5. The final interrupt handler for a particular interrupt typically queries the interrupting peripheral and determines the cause for the interrupt. It usually does a series of actions that are appropriate for the given peripheral and the cause for the interrupt. The handler is also responsible for acknowledging the interrupt at the interrupting peripheral. When the interrupt handler completes its activity, it returns back and the interrupt stack gets unwound back to the software platform level interrupt handler.

The platform level interrupt handler restores the registers that it saved on the stack and returns control back to the Program Counter (PC) location where the interrupt occurred.

The return instruction also enables interrupts again on the PowerPC processor. The application resumes normal execution at this point.

It is recommended that interrupt handlers be of a short duration and that the bulk of the interrupt work be done by application. This prevents long lockouts of other (possibly higher priority) interrupts and is considered good system design.

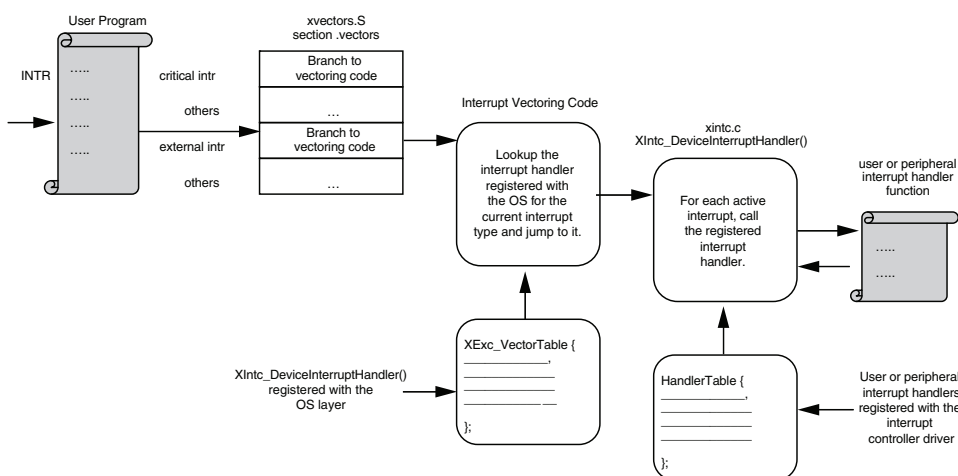
Figure B-5 shows a PowerPC processor interrupt flow without an interrupt controller.



X11021

Figure B-5: PowerPC Processor Interrupt Flow without Interrupt Controller

Figure B-6 shows a PowerPC processor interrupt flow with an interrupt controller.



X11022

Figure B-6: PowerPC Processor Interrupt Flow with Interrupt Controller

Software APIs

This section provides an overview of the software APIs involved in handling and managing interrupts, lists the available Software APIs by processor type, and provides examples of interrupt management code.

Note: This chapter is not meant to cover the APIs comprehensively. Refer to the interrupt controller device driver documentation as well as the reference documentation for the Standalone platform to for all the details of the APIs.

Interrupt Controller Driver

The Xilinx interrupt controller supports the following features:

- Enabling and disabling specific individual interrupts
- Acknowledging specific individual interrupts
- Attaching specific callback function to handle interrupt source
- Enabling and disabling the master
- Sending a single callback per interrupt or handling all pending interrupts for each interrupt of the processor

The acknowledgement of the interrupt within the interrupt controller is selectable, either prior to calling the device handler or after the handler is called. Interrupt signal inputs are either edge or level signal; consequently, support for those inputs is required:

- Edge-driven interrupt signals require that the interrupt is acknowledged prior to the interrupt being serviced to prevent the loss of interrupts which are occurring close together.
- Level-driven interrupt input signals require the interrupt to be acknowledged after servicing the interrupt to ensure that the interrupt only generates a single interrupt condition.

API Descriptions

```
int XIntc_Initialize (XIntc * InstancePtr, u16 DeviceId)
```

| | |
|-------------|---|
| Description | Initializes a specific interrupt controller instance or driver. All the fields of the <code>XIntc</code> structure and the internal vectoring tables are initialized. All interrupt sources are disabled. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the <code>XIntc</code> instance.</p> <p><i>DeviceId</i> is the unique id of the device controlled by this <code>XIntc</code> instance (obtained from <code>xparameters.h</code>). Passing in a <i>DeviceId</i> associates the generic <code>XIntc</code> instance to a specific device, as chosen by the caller or application developer.</p> |

```
int XIntc_Connect (XIntc * InstancePtr, u8 Id, XInterruptHandler
  Handler, void * CallBackRef)
```

| | |
|-------------|---|
| Description | Makes the connection between the <i>Id</i> of the interrupt source and the associated handler that is to be run when the interrupt occurs. The argument provided in this call as the <i>CallBackRef</i> is used as the argument for the handler when it is called. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the XIntc instance.</p> <p><i>Id</i> contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.</p> <p><i>Handler</i> is the handler for that interrupt.</p> <p><i>CallBackRef</i> is the callback reference, usually the instance pointer of the connecting driver</p> <p>The handler provided as an argument overwrites any handler that was previously connected.</p> |

```
void XIntc_Disconnect (XIntc* InstancePtr, u8 Id)
```

| | |
|-------------|--|
| Description | Disconnects the XIntc instance. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the XIntc instance.</p> <p><i>Id</i> contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.</p> |

```
Void XIntc_Enable (XIntc * InstancePtr, u8 Id)
```

| | |
|-------------|--|
| Description | Enables the interrupt source provided as the argument <i>Id</i> . Any pending interrupt condition for the specified <i>Id</i> occurs after this function is called. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the XIntc instance.</p> <p><i>Id</i> contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.</p> |

```
void XIntc_Disable (XIntc * InstancePtr, u8 Id)
```

| | |
|-------------|--|
| Description | Disables the interrupt source provided as the argument <i>Id</i> , such that the interrupt controller does not cause interrupts for the specified <i>Id</i> . The interrupt controller continues to hold an interrupt condition for the <i>Id</i> , but does not cause an interrupt. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the XIntc instance.</p> <p><i>Id</i> contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.</p> |

```
int XIntc_Start (XIntc * InstancePtr, u8 Mode)
```

| | |
|-------------|--|
| Description | Starts the interrupt controller by enabling the output from the controller to the processor. Interrupts can be generated by the interrupt controller after this function is called. |
| Parameters | <p><i>InstancePtr</i> is a pointer to the <code>XIntc</code> instance.</p> <p><i>Mode</i> determines if software is allowed to simulate interrupts or if real interrupts are allowed to occur. Modes are mutually exclusive. The interrupt controller hardware resets in a mode that allows software to simulate interrupts until this mode is exited. It cannot be re-entered after it has been exited. Mode is one of the following values:</p> <p><code>XIN_SIMULATION_MODE</code> enables simulation of interrupts only.</p> <p><code>XIN_REAL_MODE</code> enables hardware interrupts only.</p> <p>This function must be called after <code>XIntc</code> initialization is completed.</p> |

```
void XIntc_Stop (XIntc * InstancePtr)
```

| | |
|-------------|--|
| Description | Stops the interrupt controller by disabling the output from the controller so that no interrupts are caused by the interrupt controller. |
| Parameters | <i>InstancePtr</i> is a pointer to the <code>XIntc</code> instance. |

Hardware Abstraction Layer APIs

The following is a summary of exception functions, which can run on MicroBlaze, PowerPC 405, and PowerPC 440 processors.

Header File

```
#include "xil_exception.h"
```

Typedef

```
typedef void(* Xil_ExceptionHandler)(void *Data)
```

This typedef is the exception handler function pointer.

```
void Xil_ExceptionDisable()
```

| | |
|-------------|---|
| Description | Disable Exceptions. On PowerPC 405 and PowerPC 440 processors, this function only disables non-critical exceptions. |
|-------------|---|

```
void Xil_ExceptionEnable()
```

| | |
|-------------|---|
| Description | Enable Exceptions. On PowerPC 405 and PowerPC 440 processors, this function only enables non-critical exceptions. |
|-------------|---|

```
void Xil_ExceptionInit()
```

| | |
|-------------|---|
| Description | Initialize exception handling for the processor. The exception vector table is set up with the stub handler for all exceptions. |
|-------------|---|

```
void Xil_ExceptionRegisterHandler(u32 Id, Xil_ExceptionHandler Handler, void *Data)
```

| | |
|-------------|---|
| Description | Make the connection between the ID of the exception source and the associated handler that runs when the exception is recognized. Data is used as the argument when the handler is called. |
| Parameters | <p>Parameters:</p> <p><i>Id</i> contains the identifier (ID) of the exception source. This should be <code>XIL_EXCEPTION_INT</code> or be in the range of 0 to <code>XIL_EXCEPTION_LAST</code>. Refer to the <code>xil_exception.h</code> file for further information.</p> <p><i>Handler</i> is the handler for that exception.</p> <p><i>Data</i> is a reference to data that is passed to the handler when it is called.</p> |

```
void Xil_ExceptionRemoveHandler(u32 Id)
```

| | |
|-------------|--|
| Description | Remove the handler for a specific exception ID. The stub handler is then registered for this exception ID. |
| Parameters | <i>Id</i> contains the ID of the exception source. It should be <code>XIL_EXCEPTION_INT</code> or in the range of 0 to <code>XIL_EXCEPTION_LAST</code> . Refer to the <code>xil_exception.h</code> file for further information. |

Interrupt Setup Example

```

/***** Include Files *****/

#include "xparameters.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "xil_exception.h"

/***** Constant Definitions *****/
/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are only defined here such that a user can
 * easily change all the needed parameters in one place.
 */
#define TMRCTR_DEVICE_ID XPAR_TMRCTR_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#define TMRCTR_INTERRUPT_ID XPAR_INTC_0_TMRCTR_0_VEC_ID

/*
 * The following constant determines which timer counter of the device
 * that is used for this example, there are currently 2 timer counters
 * in a device and this example uses the first one, 0, the timer numbers
 * are 0 based
 */

```

```

*/
#define TIMER_CNTR_0 0

/*
 * The following constant is used to set the reset value of the timer
 * counter, making this number larger reduces the amount of time this
 * example consumes because it is the value the timer counter is loaded
 * with when it is started
 */
#define RESET_VALUE 0xF0000000

/***** Function Prototypes *****/

int TmrCtrIntrExample(XIntc* IntcInstancePtr,
                    XTmrCtr* InstancePtr,
                    u16 DeviceId,
                    u16 IntrId,
                    u8 TmrCtrNumber);

void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber);

/***** Variable Definitions *****/
XIntc InterruptController; /* The instance of the Interrupt Controller */

XTmrCtr TimerCounterInst; /* The instance of the Timer Counter */

/*
 * The following variables are shared between non-interrupt processing
 * and interrupt processing such that they must be global.
 */
volatile int TimerExpired;

/*****
**
 * This function is the main function of the Tmrctr example using
 * Interrupts.
 *
 * @paramNone.
 *
 * @returnXST_SUCCESS to indicate success, else XST_FAILURE to indicate
 * a Failure.
 *
 * @noteNone.
 *
 *****/

int main(void)
{
    int Status;

    /*
     * Run the Timer Counter - Interrupt example.
     */
    Status = TmrCtrIntrExample(&InterruptController,
                              &TimerCounterInst,
                              TMRCTR_DEVICE_ID,

```

```

        TMRCTR_INTERRUPT_ID,
        TIMER_CNTR_0);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;

}

/*****
/**
 * This function does a minimal test on the timer counter device and
 * driver as a design example. The purpose of this function is to
 * illustrate how to use the XTmrCtr component. It initializes a timer
 * counter and then sets it up in compare mode with auto reload such that
 * a periodic interrupt is generated.
 *
 * This function uses interrupt driven mode of the timer counter.
 *
 * @paramIntcInstancePtr is a pointer to the Interrupt Controller
 * driver Instance
 * @paramTmrCtrInstancePtr is a pointer to the XTmrCtr driver Instance
 * @paramDeviceId is the XPAR_<TmrCtr_instance>_DEVICE_ID value from
 * xparameters.h
 * @paramIntrId is
 * XPAR_<INTC_instance>_<TmrCtr_instance>_INTERRUPT_INTR
 * value from xparameters.h
 * @paramTmrCtrNumber is the number of the timer to which this
 * handler is associated with.
 *
 * @returnXST_SUCCESS if the Test is successful, otherwise XST_FAILURE
 *
 * @noteThis function contains an infinite loop such that if interrupts
 * are not working it may never return.
 *
 *****/
int TmrCtrIntrExample(XIntc* IntcInstancePtr,
                    XTmrCtr* TmrCtrInstancePtr,
                    u16 DeviceId,
                    u16 IntrId,
                    u8 TmrCtrNumber)
{
    int Status;
    int LastTimerExpired = 0;

    /*
     * Initialize the timer counter so that it's ready to use,
     * specify the device ID that is generated in xparameters.h
     */
    Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Initialize the interrupt controller driver so that
     * it's ready to use, specify the device ID that is generated in
     * xparameters.h

```

```

    */
    Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
    * Connect a device driver handler that will be called when an
    * interrupt for the device occurs, the device driver handler performs
    * the specific interrupt processing for the device
    */
    Status = XIntc_Connect(IntcInstancePtr, IntrId,
        (XInterruptHandler)XTmrCtr_InterruptHandler,
        (void *)TmrCtrInstancePtr);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
    * Start the interrupt controller such that interrupts are enabled for
    * all devices that cause interrupts, specific real mode so that
    * the timer counter can cause interrupts thru the interrupt
    * controller.
    */
    Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
    * Enable the interrupt for the timer counter
    */
    XIntc_Enable(IntcInstancePtr, IntrId);

    /*
    * Initialize the exception table.
    */
    Xil_ExceptionInit();

    /*
    * Register the interrupt controller handler with the exception table.
    */
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)
        XIntc_InterruptHandler,
        IntcInstancePtr);

    /*
    * Enable exceptions.
    */
    Xil_ExceptionEnable();
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
    * Setup the handler for the timer counter that will be called from the
    * interrupt context when the timer expires, specify a pointer to the

```

```

    * timer counter driver instance as the callback reference so the
    * handler is able to access the instance data
    */
XTmrCtr_SetHandler(TmrCtrInstancePtr,
    TimerCounterHandler,
    TmrCtrInstancePtr);

/*
 * Enable the interrupt of the timer counter so interrupts will occur
 * and use auto reload mode such that the timer counter will reload
 * itself automatically and continue repeatedly, without this option
 * it would expire once only
 */
XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
    XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

/*
 * Set a reset value for the timer counter such that it will expire
 * earlier than letting it roll over from 0, the reset value is loaded
 * into the timer counter when it is started
 */
XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);

/*
 * Start the timer counter such that it's incrementing by default,
 * then wait for it to timeout a number of times
 */
XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);

while (1) {
    /*
     * Wait for the first timer counter to expire as indicated by the
     * shared variable which the handler will increment
     */
    while (TimerExpired == LastTimerExpired) {
    }
    LastTimerExpired = TimerExpired;

    /*
     * If it has expired a number of times, then stop the timer counter
     * and stop this example
     */
    if (TimerExpired == 3) {

        XTmrCtr_Stop(TmrCtrInstancePtr, TmrCtrNumber);
        break;
    }
}

/*
 * Disable the interrupt for the timer counter
 */
XIntc_Disable(IntcInstancePtr, DeviceId);

return XST_SUCCESS;
}

/*****
**

```

```

* This function is the handler which performs processing for the timer
* counter. It is called from an interrupt context such that the amount
* of processing performed should be minimized. It is called when the
* timer counter expires if interrupts are enabled.
*
* This handler provides an example of how to handle timer counter
* interrupts but is application specific.
*
* @param CallbackRef is a pointer to the callback function
* @param TmrCtrNumber is the number of the timer to which this
* handler is associated with.
*
* @return None.
*
* @note None.
*
*****/
void TimerCounterHandler(void *CallbackRef, u8 TmrCtrNumber)
{
    XTmrCtr *InstancePtr = (XTmrCtr *)CallbackRef;

    /*
     * Check if the timer counter has expired, checking is not necessary
     * since that's the reason this function is executed, this just shows
     * how the callback reference can be used as a pointer to the instance
     * of the timer counter that expired, increment a shared variable so
     * the main thread of execution can see the timer expired
     */
    if (XTmrCtr_IsExpired(InstancePtr, TmrCtrNumber)) {
        TimerExpired++;
        if(TimerExpired == 3) {
            XTmrCtr_SetOptions(InstancePtr, TmrCtrNumber, 0);
        }
    }
}

```


EDK Tcl Interface

This appendix describes the various Tool Command Language (Tcl) Application Program Interfaces (APIs) available in EDK tools and methods for accessing information from EDK tools using Tcl APIs.

Introduction

Each time EDK tools run, they build a runtime data structure of your design. The data structure contains information about user design files, such as Microprocessor Hardware Specification (MHS), or library data files, such as Microprocessor Peripheral Definition (MPD), Microprocessor Driver Definition (MDD), and Microprocessor library Definition (MLD). Access to the data structure is given as Tcl APIs. Based on design requirements, IP, driver, library, and OS writers that provide the corresponding data files can access the data structure information to add some extra steps in the tools processing. EDK tools also use Tool Command Language (Tcl) to perform various Design Rule Checks (DRCs), and to update the design data structure in a limited manner.

Understanding Handles

The tools provide access points into the data structure through a set of API functions. Each API function requires an argument in the form of system information, which is called a *handle*.

For example, an IP defined in the Microprocessor Hardware Specification (MHS) file could serve as a handle. Handles can be of various types, based on the kind of data to which they are providing access. Data types include instance names, driver names, hardware parameters, or hardware ports. From a given handle, you can get information associated with that handle, or you can get other, associated handles.

Data Structure Creation

EDK tools provide access to two basic types of run-time information:

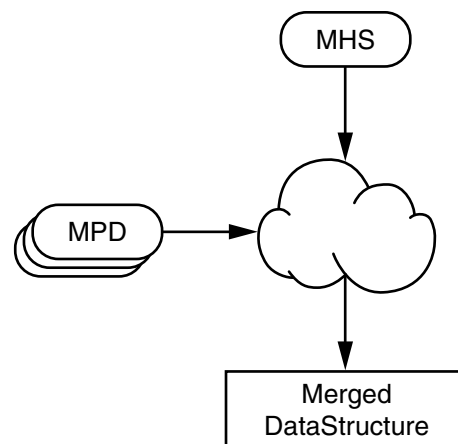
- The original design and library datafile data structure:

The original data structure provides access only to the information present in various data files. You can get a handle to such files as the MHS, MPD, MDD, and MLD. These handles allow you to query the contents of the files with which they are associated.

- The merged data structure:

When EDK tools run, the information in the design files (MHS) is combined with the corresponding information from library files (MPD) to create *merged data structures*: *hardware merged datastructure* (also referred to as the hardware merged object). During the process of creating the merged data structure, the tools also analyze various design characteristics (such as connectivity or address mapping), and that information is also stored in the merged data structures. A merged data structure provides an easy way to access this analyzed information. For example, an instance of an IP in the MHS file is merged with its corresponding MPD. Using the merged instances, complete information can be obtained from one handle; it is not necessary to access the IP instance and MPD handles separately.

Figure C-1 shows a merged hardware data structure creation.



X10582

Figure C-1: Merged Hardware Data Structure Creation

Tcl Command Usage

General Conventions

There are two kinds of Tcl APIs, which differ based on the type of data they return. Tcl APIs return either:

- A handle or a list of handles to some objects.
- A value or a list of values.

The common rules followed in all Tcl APIs are:

- An API returns a NULL handle when an expected handle to another object is not found.
- An API returns an empty string when a value is either empty or that value cannot be determined.

Before You Begin

When you use XPS in non-GUI mode (**xps -nw**), you must first initialize the internal tool database (the runtime datastructure) by loading the project with the **xload** command:

```
xload <filetype> <filename>.{MHS/XMP}
```

Refer to [Chapter 5, Command Line Mode](#) for more detail regarding **xload**.

To gain access to either the MHS Handle or the merged MHS Handle, use one of the following commands after loading the project:

```
XPS% set original_mhs_handle [xget_handle mhs]
```

or

```
XPS% set merged_mhs_handle [xget_handle merged_mhs]
```

The following section provides the nomenclature of the EDK Hardware Tcl commands in more detail.

EDK Hardware Tcl Commands

Overview

This section provides a list of Tcl APIs available in the EDK hardware data structure. The description of these commands uses certain terms, which are defined in the following subsections.

Original MHS Handle (`original_mhs_handle`)

The handle that points to the MHS information only. This handle does not contain any MPD information. If an IP parameter has not been specified in the MHS, this handle does not contain that parameter.

Merged MHS Handle (`merged_mhs_handle`)

The handle that points to both the MHS and MPD information. A hardware datastructure/merged object is formed when the tools merge the MHS and MPD information.

Note: Various Tcl procedures are also called within batch tools such as Platgen, Libgen, and Simgen. Handles provided through batch tools always refer to the merged MHS handle. You do not have access to the original MHS handle from the batch tools. The original MHS handle is needed only when you must modify the design using the provided APIs so that the generated MHS design file can be updated.

Original IP Instance Handle (`original_IP_handle`)

A handle to an IP instance obtained from the original MHS handle that contains information present only in the MHS file.

Merged IP Instance Handle (`merged_IP_handle`)

Refers to the IP handle obtained from the merged MHS handle. The merged IP instance handle contains both MHS and MPD information.

Note: Batch tools such as Platgen provide access to the merged IP instance handle only and not the original IP instance handle. Consequently, the various property handles (the parameter and port handles, for example) are merged handles and not the original handles.

Hardware Read Access APIs

The following sections contain a summary table and descriptions of defined hardware read access APIs. To go to the API descriptions, which are provided in the following section, click on a summary link.

API Summary

```

xget_hw_busif_value <handle> <busif_name>
xget_hw_bus_slave_addrpairs <merged_bus_handle>
xget_hw_busif_handle <handle> <busif_name>
xget_hw_connected_busifs_handle <merged_mhs_handle> <businst_name> <busif_type>
xget_hw_connected_ports_handle <merged_mhs_handle> <connector_name> <port_type>
xget_hw_ioif_handle <handle> <ioif_name>
xget_hw_ioif_value <handle> <ioif_name>
xget_hw_ipinst_handle <mhs_handle> <ipinst_name>
xget_hw_mpd_handle <ipinst_handle>
xget_hw_name <handle>
xget_hw_option_handle <handle> <option_name>
xget_hw_option_value <handle> <option_name>
xget_hw_parameter_handle <handle> <parameter_name>
xget_hw_parameter_value <handle> <parameter_name>
xget_hw_pcore_dir_from_mpd <mpd_handle>
xget_hw_pcore_dir <ipinst_handle>
xget_hw_port_connectors_list <ipinst_handle> <portName>
xget_hw_parent_handle <handle>
xget_hw_port_connectors_list <ipinst_handle> <portName>
xget_hw_port_handle <handle> <port_name>
xget_hw_port_value <handle> <port_name>
xget_hw_proj_setting <prop_name>
xget_hw_proc_slave_periphs <merged_proc_handle>
xget_hw_subproperty_handle <property_handle> <subprop_name>
xget_hw_subproperty_value <property_handle> <subprop_name>
xget_hw_value <handle>

```

Hardware API Descriptions

xget_hw_busif_handle *<handle> <busif_name>*

Description Returns a handle to the associated bus interface.

Arguments *<handle>* is the handle to the MPD, original IP instance, or merged IP instance.
<busif_name> is the name of the bus interface whose handle is required. If *<busif_name>* is specified as an asterisk (*), the API returns a list of bus interface handles. To access an individual bus interface handle, you can iterate over the list in Tcl.

xget_hw_busif_value *<handle> <busif_name>*

| | |
|-------------|--|
| Description | Returns the value of the specified bus interface. The value is typically the instance name of the bus to which the bus interface is connected. For a transparent bus interface, the value is the connector (which is not a bus instance name.) |
| Arguments | <i><handle></i> the handle to the MPD, original IP instance or merged IP instance. <i><busif_name></i> is the name of the bus interface whose value is required. |

xget_hw_bus_slave_addrpairs *<merged_bus_handle>*

| | |
|-------------|---|
| Description | Returns a list of slave addresses associated with the specified bus handle. The returned value is a list of integers where: The first value is the base address of any connected peripherals. The second value is the associated high address. The following values are paired base and high addresses of other peripherals. |
| Arguments | <i><merged_bus_handle></i> is a handle to a merged IP instance pointing to a bus instance. |

xget_hw_connected_busifs_handle *<merged_mhs_handle> <businst_name> <busif_type>*

| | |
|-------------|--|
| Description | Returns a list of handles to bus interfaces that are connected to a specified bus. |
| Arguments | <i><merged_mhs_handle></i> is a handle to the merged MHS. <i><businst_name></i> is the name of the connected bus instance. <i><busif_type></i> is one of the following: MASTER, SLAVE, TARGET, INITIATOR, ALL. |

xget_hw_connected_ports_handle *<merged_mhs_handle> <connector_name> <port_type>*

| | |
|-------------|---|
| Description | Returns a list of handles to ports associated with a specified connector. The valid handle type is the merged MHS. |
| Arguments | <i><merged_mhs_handle></i> is the handle to the merged MHS. <i><connector_name></i> is the name of the connector. <i><port_type></i> is <i>source</i> , <i>sink</i> , or <i>all</i> . This API returns a list of handles to ports based on the <i><port_type></i> , where: <i>source</i> is a list of ports that are driving the given signal. <i>sink</i> is a list of ports that are being driven by the given signal. <i>all</i> is a list of all ports connected to the given signal. |

xget_hw_ioif_handle *<handle>* *<ioif_name>*

Description Returns the handle to an I/O interface associated with the handle.

Arguments *<handle>* is the handle to an MPD or a merged IP instance.
 If an original IP instance handle is provided, this API returns a NULL.
<ioif_name> is the name of the I/O interface whose handle is required. If *<ioif_name>* is specified as an asterisk (*), the API returns a list of I/O interface handles. To access an individual I/O interface handle, you can iterate over the list in Tcl.

xget_hw_ioif_value *<handle>* *<ioif_name>*

Description Returns the value of the I/O interface. The value is specified in the MPD file and cannot be overwritten in MHS.

Arguments *<handle>* is the handle to an MPD or a merged IP instance.
<ioif_name> is the name of the I/O interface whose value is required.

xget_hw_ipinst_handle *<mhs_handle>* *<ipinst_name>*

Description Returns the handle of the specified IP instance.

Arguments *<mhs_handle>* is the handle to either an original MHS or a merged MHS.
<ipinst_name> is the name of the IP instance whose handle is required. If *<ipinst_name>* is specified as an asterisk (*), the API returns a list of IP instance handles. To access an individual IP instance handle, you can iterate over the list in Tcl.

xget_hw_mpd_handle *<ipinst_handle>*

Description Returns a handle to the MPD object associated with the specified IP instance.

Arguments *<ipinst_handle>* is a handle to the merged IP instance.

xget_hw_name *<handle>*

Description Returns the name of the specified handle.

Arguments *<handle>* is of specified type.
 If *<handle>* is of type IP instance, its name is the instance name of that IP. For example, if the handle refers to an instance of MicroBlaze called `mymb` in the MHS file, the value the API returns is `mymb`. Similarly, to get the name of a parameter from a parameter handle, you can use the same command.

xget_hw_option_handle *<handle>* *<option_name>*

| | |
|-------------|--|
| Description | Returns a handle to the associated option. |
| Arguments | <i><handle></i> is the associated option. <i><option_name></i> is the name of the option whose value is required. If specified as an asterisk (*), the API returns a list of option handles. To access an individual option handle, you can iterate over the list in Tcl. |

xget_hw_option_value *<handle>* *<option_name>*

| | |
|-------------|--|
| Description | Returns the value of the option. The value is specified in the MPD file and cannot be overwritten in MHS |
| Arguments | <i><handle></i> the handle to an MPD or a merged IP instance. <i><option_name></i> is the name of the option whose value is required. |

xget_hw_parameter_handle *<handle>* *<parameter_name>*

| | |
|-------------|--|
| Description | Returns the handle to an associated parameter |
| Arguments | <i><handle></i> is the handle to the MPD, original IP instance, or merged IP instance. <i><parameter_name></i> is the name of the associated parameter whose handle is required. If <i><parameter_name></i> is specified as an asterisk (*), a list of parameter handles is returned. To access an individual parameter handle, you can iterate over the list in Tcl. |

xget_hw_parameter_value *<handle>* *<parameter_name>*

| | |
|-------------|--|
| Description | Returns the value of the specified parameter |
| Arguments | <i><handle></i> is the handle to the MPD, original IP instance, or merged IP instance. <i><parameter_name></i> is the name of the associated parameter whose value is required. |

xget_hw_parent_handle *<handle>*

| | |
|-------------|---|
| Description | Returns the handle to the parent of the specified handle. The type of parent handle is determined by the specified handle type. If the specified handle is a merged handle, the parent obtained through this API will also be a merged handle. |
| Arguments | <p><i><handle></i> is one of the following:</p> <ul style="list-style-type: none"> PARAMETER, the parent is the MPD, IP instance, or the merged IP instance object. PORT, the parent is the MPD, IP instance, the merged IP instance, or the MHS object. BUS_INTERFACE, the parent is the MPD, IP instance, or the merged IP instance object. IO_INTERFACE, the parent is the MPD or the merged IP instance object. OPTION, the parent is the MPD or the merged IP instance object. IPINST, the parent is the MHS or the merged MHS object. For MHS or MPD, the parent is a NULL handle. |

xget_hw_pcore_dir_from_mpd *<mpd_handle>*

| | |
|-------------|---|
| Description | Returns the pcore directory path for the MPD. |
| Arguments | <i><mpd_handle></i> is the handle to the MPD. |

xget_hw_pcore_dir *<ipinst_handle>*

| | |
|-------------|--|
| Description | Returns the pcore directory for the given IP instance. |
| Arguments | <i><ipinst_handle></i> is the handle to the IP instance. |

xget_hw_port_connectors_list *<ipinst_handle>* *<portName>*

| | |
|-------------|--|
| Description | If the value (connector) of the port is within an & separated list, this API splits that list and returns a list of strings (connector names). |
| Arguments | <p><i><ipinst_handle></i> is the handle to the IP instance (merged or original).</p> <p><i><portName></i> is the name of the port whose connectors are needed.</p> |

xget_hw_port_handle *<handle>* *<port_name>*

| | |
|-------------|--|
| Description | Returns the handle to a port associated with the handle. If a handle is of type MHS, the returned handle points to a global port of the given name. |
| Arguments | <p><i><handle></i> is the handle to the MPD, original IP instance, merged IP instance, original MHS or merged MHS.</p> <p><i><port_name></i> is the name of the port whose handle is required.</p> <p>If <i><port_name></i> is specified as an asterisk (*), a list of port handles is returned. To access an individual port handle, you can iterate over the list in Tcl.</p> <p>If a handle is of type MHS (original or merged), the returned handle points to a global port with the given name.</p> |

xget_hw_port_value *<handle>* *<port_name>*

| | |
|-------------|---|
| Description | Returns the value of the specified port. The value of a port is the signal name connected to that port. |
| Arguments | <p><i><handle></i> is the handle to the MPD, original IP instance, merged IP instance, original MHS or merged MHS.</p> <p><i><port_name></i> is the name of the port whose value is required.</p> |

xget_hw_proj_setting *<prop_name>*

| | |
|-------------|--|
| Description | Returns the value of the property specified by <i>prop_name</i> . |
| Arguments | <i><prop_name></i> is the name of the property whose value is needed. Options are: <i>fpga_family</i> , <i>fpga_subfamily</i> , <i>fpga_partname</i> , <i>fpga_device</i> , <i>fpga_package</i> , <i>fpga_speedgrade</i> |

xget_hw_proc_slave_periphs *<merged_proc_handle>*

| | |
|-------------|--|
| Description | Returns a list of handles to slaves that can be addressed by the specified processor |
| Arguments | <p><i><merged_proc_handle></i> is a handle to the merged IP instance, pointing to a processor instance. This returned list includes slaves that are not directly connected to the processor, but are accessed across a bus-to-bus bridge (for example, <i>opb2plb_bridge</i>).</p> <p>The input handle must be an IP instance handle to a processor instance, obtained from the merged MHS only (not from the original MHS).</p> |

xget_hw_subproperty_handle *<property_handle> <subprop_name>*

| | |
|-------------|--|
| Description | Returns the handle to a subproperty associated with the specified <i><property_handle></i> . |
| Arguments | <p><i><property_handle></i> is a handle to one of the following: PARAMETER, PORT, BUS_INTERFACE, IO_INTERFACE, or OPTION.</p> <p><i><subprop_name></i> is the name of the subproperty whose handle is required. For a list of sub-properties, please refer to “Microprocessor Peripheral Definition” “Microprocessor Peripheral Definition (MPD)” in the <i>Platform Specification Format Reference Manual (UG642)</i> and “Additional Keywords in the Merged Hardware Datastructure” on page 271.</p> |

xget_hw_subproperty_value *<property_handle> <subprop_name>*

| | |
|-------------|--|
| Description | Returns the value of a specified subproperty. |
| Arguments | <p><i><property_handle></i> is one of the following: PARAMETER, PORT, BUS_INTERFACE, IO_INTERFACE, or OPTION.</p> <p><i><subprop_name></i> is the name of the subproperty whose value is required. For a list of sub-properties, refer to “Microprocessor Peripheral Definition (MPD)” in the <i>Platform Specification Format Reference Manual (UG642)</i> and Additional Keywords in the Merged Hardware Datastructure, page 271</p> |

xget_hw_value *<handle>*

| | |
|-------------|---|
| Description | Gets the value associated with the specified handle. |
| Arguments | <p><i><handle></i> is of specified type.</p> <p>If <i><handle></i> is of type IP instance, its value is the IP module name. For example, if the handle refers to the MicroBlaze™ instance in the MHS file, the value the API returns is the name of the IP, that is, <code>microblaze</code>. Similarly, to get the value of a parameter from a parameter handle, you can use the same command.</p> |

Tcl Example Procedures

The following are example Tcl procedures that use some of the hardware API Tcl commands.

Example 1

This procedure explains how to get a list of IPs of a particular IPTYPE. Each IP provided in the EDK repository has a corresponding IP type specified by the IPTYPE option, in the MPD file. The merged_mhs_instance has the information from both the MHS file and the MPD file. The process for getting a list of IPs of a particular IPTYPE is:

1. Using the merged_mhs_handle, get a list of all IPs.
2. Iterate over this list and for each IP, get the value of the OPTION IPTYPE and compare it with the given IP type.

The following code snippet illustrates how to get the IPTYPE of specific IPs.

```
## Procedure to get a list of IPs of a particular IPTYPE
proc xget_ipinst_handle_list_for_ipdtype {merged_mhs_handle ipdtype}
{
  ##Get a list of all IPs
  set ipinst_list [xget_hw_ipinst_handle $merged_mhs_handle ""]
  set ret_list ""
  foreach ipinst $ipinst_list {
    ## Get the value of the IPTYPE Option.
    set curiptype [xget_hw_option_value $ipinst "IPTYPE"]
    ##if curiptype matches the given ipdtype, then add it to      ## the
    list that this proc returns.
    if {[string compare -nocase $curiptype $ipdtype] == 0}{
      lappend ret_list $ipinst
    }
  }
  return $ret_list
}
```

Example 2

The following procedure explains how to get the list of cores that are memory controllers in a design. Memory controller cores have the tag, ADDR_TYPE = MEMORY, in their address parameter.

```
## Procedure to get a list of memory controllers in a design.
proc xget_hw_memory_controller_handles { merged_mhs } {
    set ret_list ""

    # Gets all MhsInsts in the system
    set mhsinsts [xget_hw_ipinst_handle $merged_mhs "*" ]

    # Loop through each MhsInst and determine if it has
    # "ADDR_TYPE = MEMORY" in the parameters.

    foreach mhsinst $mhsinsts {

        # Gets all parameters of the IP
        set params [xget_hw_parameter_handle $mhsinst "*"]

        # Loop through each param and find tag "ADDR_TYPE = MEMORY"
        foreach param $params {
            if {$param == 0} {
                continue
            } elseif {$param == ""} {
                continue
            }
            set addrTypeValue [xget_hw_subproperty_value $param "ADDR_TYPE"]

            # Found tag! Add MhsInst to list and break to go to next MhsInst
            if {[string compare -nocase $addrTypeValue "MEMORY"] == 0} {
                lappend ret_list $mhsinst
                break
            }
        }
    }

    return $ret_list
}
```

Advanced Write Access APIs

Advance Write Access APIs modify the MHS object in memory. These commands operate on the original MHS handle and handles obtained from the MHS handle. The Write Access APIs can be used to create the project only. They are disabled during the Platgen flow.

Advance Write Access Hardware API Summary

The following table provides a summary of the Advance Write Access APIs. To go to the API descriptions, which are provided in the following section, click on a summary link.

Table C-1: Hardware Advanced Write Access APIs

Add Commands

```
xadd_hw_hdl_srcfile <ipinst_handle> <fileuse> <filename> <hdl_lang>
xadd_hw_ipinst_busif <ipinst_handle> <busif_name> <busif_value>
xadd_hw_ipinst_port <ipinst_handle> <port_name> <connector_name>
xadd_hw_ipinst <mhs_handle> <inst_name> <ip_name> <hw_ver>
xadd_hw_ipinst_parameter <ipinst_handle> <param_name> <param_value>
xadd_hw_subproperty <prop_handle> <subprop_name> <subprop_value>
xadd_hw_toplevel_port <mhs_handle> <port_name> <connector_name> <direction>
```

Delete Commands

```
xdel_hw_ipinst <mhs_handle> <inst_name>
xdel_hw_ipinst_busif <ipinst_handle> <busif_name>
xdel_hw_ipinst_port <ipinst_handle> <port_name>
xdel_hw_ipinst_parameter <ipinst_handle> <param_name>
xdel_hw_subproperty <prop_handle> <subprop_name>
xdel_hw_toplevel_port <mhs_handle> <port_name>
```

Modify Commands

```
xset_hw_parameter_value <busif_handle> <busif_value>
xset_hw_port_value <port_handle> <port_value>
xset_hw_busif_value <busif_handle> <busif_value>
```

Advance Write Access Hardware API Descriptions

Add Commands

xadd_hw_hdl_srcfile *<ipinst_handle> <fileuse> <filename> <hdl_lang>*

| | |
|-------------|--|
| Description | <p>Adds HDL files on the fly to the PAO. This API should only be used in batch tools like platgen/simgen and not in xps batch as a design entry mechanism.</p> <p>When adding VHDL files, those files are expected to be an instance-specific customization and, consequently are added to a logical library called <i><instname>_<wrapper>_<hwver></i>.</p> <p>VHDL files must be generated in the <i><projdir>/hdl/elaborate/<instname>_<wrapper>_<hwver></i> directory.</p> <p>While Verilog does not use libraries, the files must still be generated in the specified directory structure and location.</p> |
| Arguments | <p><i><ipinst_handle></i> is the handle of the IP instance.</p> <p><i><fileuse></i> is {lib synlib simlib}.</p> <p><i><filename></i> is the specified filename.</p> <p><i><hdl_lang></i> is {vhdl verilog}.</p> |
| Example | <pre>xadd_hw_hdl_srcfile \$ipinst_handle lib xps_central_dma.vhd vhdl</pre> |

xadd_hw_ipinst_busif *<ipinst_handle> <busif_name> <busif_value>*

| | |
|-------------|--|
| Description | <p>Creates and adds a bus interface specified by <i><busif_name></i> and <i><busif_value></i> to the IP instance specified by the <i><ipinst_handle></i>. This API returns a handle to the newly created bus interface, if successful, and NULL otherwise.</p> |
| Arguments | <p><i><ipinst_handle></i> is the handle to the IP instance to which the bus interface has to be added.</p> <p><i><busif_name></i> is the name of the bus interface.</p> <p><i><busif_value></i> is the value of the bus interface.</p> |
| Example | <p>Connect the ILMB bus interface from MicroBlaze to the ilmb_0 bus:</p> <pre>xadd_hw_ipinst_busif \$mb_handle ILMB ilmb_0</pre> |

```
xadd_hw_ipinst <mhs_handle> <inst_name> <ip_name> <hw_ver>
```

Description Adds a new MHS instance to the MHS specified by <mhs_handle>. Returns a handle to the newly created instance if successful, and NULL otherwise.

Arguments <mhs_handle> is the handle to the MHS in which this mhs instance has to be added.
 <inst_name> is the instance name of the IP instance that needs to be added.
 <ip_name> is the name of the IP that needs to be added.
 <hw_ver> is the version of the IP that needs to be added.

Example Add a MicroBlaze v7.00.a IP with the instance name mblaze to the MHS:

```
xadd_hw_ipinst $mhs_handle mblaze microblaze 7.00.a
```

```
xadd_hw_ipinst_port <ipinst_handle> <port_name> <connector_name>
```

Description Creates and adds a port specified by <port_name> and <connector_name> to the IP instance specified by the <ipinst_handle>.

This API returns a handle to the newly created port, if successful, and NULL otherwise.

Arguments <inst_handle> is the handle to the IP instance to which the port has to be added.
 <port_name> is the name of the port.
 <connector_name> is the name of the connector.

Example Add a clock port on a MicroBlaze instance and connect it to the sys_clk_s signal:

```
xadd_hw_ipinst_port $mb_handle Clk sys_clk_s
```

```
xadd_hw_ipinst_parameter <ipinst_handle> <param_name> <param_value>
```

Description Creates and adds a parameter specified by <param_name> and <param_value> to the IP instance specified by the <ipinst_handle>. This API returns a handle to the newly created parameter, if successful, and NULL otherwise.

Arguments <ipinst_handle> is the handle to the IP instance to which the parameter is to be added.
 <param_name> is the name of the parameter.
 <param_value> is the parameter value.

Example Add the C_DEBUG_ENABLED parameter to a MicroBlaze instance and set its value to 1:

```
xadd_hw_ipinst_parameter $mb_handle C_DEBUG_ENABLED 1
```

```
xadd_hw_subproperty <prop_handle> <subprop_name> <subprop_value>
```

| | |
|-------------|--|
| Description | Adds a subproperty to a property (parameter, port or bus interface). |
| Arguments | <p><prop_handle> is a handle to the parameter, port or bus interface.</p> <p><subprop_name> is the name of the sub-property.</p> <p><subprop_value> is the value of the sub-property. For a list of sub-properties, refer to "Microprocessor Peripheral Definition (MPD)" in the <i>Platform Specification Format Reference Manual</i> (UG642) and "Additional Keywords in the Merged Hardware Datastructure" on page 271.</p> |
| Example | <p>Add DIR to a port:</p> <pre>xadd_hw_subproperty \$port_handle DIR I</pre> |

```
xadd_hw_toplevel_port <mhs_handle> <port_name> <connector_name>  
<direction>
```

| | |
|-------------|--|
| Description | Adds a new top-level port to the MHS specified by <mhs_handle>. Returns a handle to the newly created port if successful, and NULL otherwise. |
| Arguments | <p><mhs_handle> is the handle to the MHS in which this top-level port has to be added.</p> <p><port_name> is the name of the port that needs to be added.</p> <p><connector_name> is the name of the connector.</p> <p><direction> is the direction of the port (I, O, or IO).</p> |
| Example | <p>Add a top-level input port sys_clk_pin with connector dcm_clk_s:</p> <pre>xadd_hw_toplevel_port \$mhs_handle sys_clk_pin dcm_clk_s I</pre> |

Delete Commands

```
xdel_hw_ipinst <mhs_handle> <inst_name>
```

| | |
|-------------|--|
| Description | deletes the IP instance with a specified instance name. |
| Arguments | <p><mhs_handle> is the handle to the original MHS.</p> <p><inst_name> is the name of the instance to be deleted.</p> |
| Example | <p>Delete an instance called mymb:</p> <pre>xdel_hw_ipinst \$mhs_handle mymb</pre> |

xdel_hw_ipinst_busif *<ipinst_handle>* *<busif_name>*

| | |
|-------------|---|
| Description | Deletes a specified bus interface on an IP instance handle. |
| Arguments | <i><ipinst_handle></i> is the handle of the IP instance. <i><busif_name></i> is the name of the bus interface that is to be deleted. |
| Example | Delete the ILMB bus interface from a MicroBlaze instance: xdel_hw_ipinst_busif \$mb_handle ILMB |

xdel_hw_ipinst_port *<ipinst_handle>* *<port_name>*

| | |
|-------------|---|
| Description | Deletes a specified port on an IP instance handle. |
| Arguments | <i><ipinst_handle></i> is the handle of the IP instance. <i><port_name></i> is the name of the port to be deleted. |
| Example | Delete a Clk port on a given MicroBlaze instance: xdel_hw_ipinst_port \$mb_handle Clk |

xdel_hw_ipinst_parameter *<ipinst_handle>* *<param_name>*

| | |
|-------------|---|
| Description | Deletes a specified parameter on an IP instance handle. |
| Arguments | <i><ipinst_handle></i> is a handle to the IP instance. <i><param_name></i> is the name of the parameter to be deleted. |
| Example | Delete the C_DEBUG_ENABLED parameter from a MicroBlaze instance: xdel_hw_ipinst_parameter \$mb_handle C_DEBUG_ENABLED |

xdel_hw_subproperty *<prop_handle>* *<subprop_name>*

| | |
|-------------|---|
| Description | Deletes a specified subproperty from a property handle |
| Arguments | <i><prop_handle></i> is a handle to a parameter, port, or bus interface. <i><subprop_name></i> is the name of the subproperty. |
| Example | Delete SIGIS subproperty from a given port: xdel_hw_subproperty \$port_handle SIGIS |

```
xdel_hw_toplevel_port <mhs_handle> <port_name>
```

| | |
|-------------|---|
| Description | Deletes a top-level port with the specified name. |
| Arguments | <i><mhs_handle></i> is the handle to the original MHS. <i><port_name></i> is the name of the port to be deleted. |
| Example | Delete a top-level port called <code>sys_clk_pin</code> : xdel_hw_toplevel_port \$mhs_handle sys_clk_pin |

Modify Commands

```
xset_hw_parameter_value <busif_handle> <busif_value>
```

| | |
|-------------|--|
| Description | Sets the value of the parameter to the given value. |
| Arguments | <i><port_handle></i> is the handle to the port whose value must be set. <i><port_value></i> is the value to be set. |
| Example | Set the value of a parameter to 2: xset_hw_parameter_value \$param_handle 2 |

```
xset_hw_port_value <port_handle> <port_value>
```

| | |
|-------------|--|
| Description | Sets the value of the port to the given value. |
| Arguments | <i><port_handle></i> is the handle to the port whose value must be set. <i><port_value></i> is the value to be set. |
| Example | Set the value of a port to "my_connection": xset_hw_port_value \$port_handle my_connection |

```
xset_hw_busif_value <busif_handle> <busif_value>
```

| | |
|-------------|---|
| Description | Sets the value of the bus interface to the given value. |
| Arguments | <i><busif_handle></i> is the handle to the bus interface whose value must be set. <i><busif_value></i> is the value to be set. |
| Example | Set the value of a bus interface to "my_bus": xset_hw_busif_value \$busif_handle my_bus |

Tcl Flow During Hardware Platform Generation

Input Files

Platgen, Simgen, Libgen and other tools that create the hardware platform work with the MHS design file and the IP data files (MPD). Internally, the tools create the system view based on these files. Each of the IP in the design has an MPD associated with it. Optionally, it can have an associated Tcl file. Tcl files can contain DRC procedures, procedures to automate calculation of parameters, or they can perform other tasks. The Tcl files that are used during the hardware platform generation are present in the individual core directories along with the MPD files. For Xilinx-supplied cores, the Tcl files are in the `<EDK install area>/hw/XilinxProcessorIPLib/pcores/<corename>/data/` directory.

Tcl Procedures Called During Hardware Platform Generation

Platgen (and many EDK batch tools, such as Libgen, Simgen, and Bitinit) run a few predefined Tcl procedures related to each IP to perform DRCs and to compute values of certain parameters on the IP. For information on the Tcl file for a given IP, see the *Platform Format Specification Reference Manual* (UG642). A link to the document is supplied in [Appendix E, Additional Resources](#).

This section lists the Tcl procedures and describes how they can be called for user IP. Tcl procedures can be classified based on:

- The action performed in that Tcl procedure.
 - DRC

These procedures perform DRCs on the system but do not modify the state of the system itself. The return code provided by these procedures is captured by Platgen. Hence, if there is any error status returned by a DRC procedure, Platgen captures the error and stops execution at an appropriate time.
 - UPDATE

These procedures assume the system to be in a correct state and query the design data structure using Tcl APIs to compute the values of certain parameters. The tool uses the string these procedures return to update the design with the Tcl-computed value.
- The stage during hardware platform creation at which they are invoked.
 - IPLEVEL

These procedures are invoked early in processing performed within the tools. These procedures assume that no design analysis has been performed and, therefore, none of the system-level information is available.
 - SYSLEVEL

These procedures are invoked later in processing, when the tool has performed some system-level analysis of the design and has updated certain parameters. For a list of such parameters, refer to the “Reserved Parameters” section of Chapter 2, “Platform Specification Utility (PsfUtility).” Also note that some parameters may be updated by Tcl procedures of IPs. Such parameters are governed solely by IP Tcl and are therefore not listed in the MPD documentation.

Each Tcl procedure takes one argument. The argument is a handle of a certain type in the data structure. The handle type depends on the object type with which the Tcl procedure is associated. Tcl procedures associated with parameters are provided with a handle to that parameter as an argument.

Tcl procedures associated with the IP itself are provided with a handle to a particular instance of the IP used in the design as an argument. The following is a list of the Tcl procedures that can be called for an IP instance.

Note: The MPD tag name that specifies the Tcl procedure name indicates the category to which the Tcl procedure belongs.

Each of the following tags is a name-value pair in the MPD file, where the value specifies the Tcl procedure associated with that tag. You must ensure that such a Tcl procedure exists in the Tcl file for that IP.

- Tool-specific Tcl calls
 - You can specify calls specific to either Platgen or Simgen.

Order of Execution for Tcl Procedures in the MPD

The Tcl procedures specified in the MPD are executed in the following order during hardware platform generation:

1. IPLEVEL_UPDATE_VALUE_PROC (on parameters)
2. IPLEVEL_DRC_PROC (on parameters)
3. IPLEVEL_DRC_PROC (on the IP, specified on options)
4. SYSLEVEL_UPDATE_VALUE_PROC (on parameters)
5. SYSLEVEL_UPDATE_PROC (on the IP, specified on options)
6. SYSLEVEL_DRC_PROC (on parameters, ports)
7. SYSLEVEL_DRC_PROC (on the IP, specified on options)
8. FORMAT_PROC (on parameters)
9. Helper core Tcl Procedures

UPDATE Procedure for a Parameter Before System Level Analysis

You can use the parameter subproperty IPLEVEL_UPDATE_VALUE_PROC to specify the Tcl procedure that computes the parameter value, based on other parameters on the same IP. The input handle associates with the parameter object of a particular instance of that IP.

```
## MPD snippet
PARAMETER C_PARAM1 = 4, ...,
PARAMETER C_PARAM2 = 0, ..., IPLEVEL_UPDATE_VALUE_PROC = update_param2

## Tcl computes value based on other parameters on the IP
## Argument param_handle points to C_PARAM2 because the Tcl is
## associated with C_PARAM2
proc update_param2 {param_handle} {
    set retval 0;
    set mhsinst [xget_hw_parent_handle $param_handle]
    set param1val [xget_hw_param_value $mhsinst "C_PARAM1"]
    if {$param1val >= 4} {
        set retval 1;
    }
    return $retval
}
```

DRC Procedure for a Parameter Before System Level Analysis

You can use the parameter subproperty `IPLEVEL_DRC_PROC` to specify the Tcl procedure that performs DRCs specific to that parameter. These DRCs should be independent of other `PARAMETER` values on that IP.

For example, this DRC can be used to ensure that only valid values are specified for that parameter. The input handle is a handle to the parameter object for a particular instance of that IP.

```
## MPD snippet
PARAMETER C_PARAM1 = 0, ..., IPLEVEL_DRC_PROC = drc_param1

## Tcl snippet
## Argument param_handle points to C_PARAM1 since the Tcl is
## associated with C_PARAM1
proc drc_param1 {param_handle} {
    set param1val [xget_hw_value $param_handle]
    if {$param1val >= 5} {
        error "C_PARAM1 value should be less 5"
        return 1;
    } else {
        return 0;
    }
}
```

DRC Procedure for the IP Before System Level Analysis

You can use the `OPTION IPLEVEL_DRC_PROC` to specify the Tcl procedure that performs this DRC. The procedure should be used to perform DRCs at `IPLEVEL` (for example, consistency between two parameter values). The DRCs performed here should be independent of how that IP has been used in the system (MHS) and should only use parameter, bus interface, and port settings used on that IP. The input handle is a handle to an instance of the IP.

```
## MPD Snippet
OPTION IPLEVEL_DRC_PROC = iplevel_drc
BUS_INTERFACE BUS = SPLB, BUS_STD = PLB, BUS_TYPE = SLAVE
PORT MYPORT = "", DIR = I

## Tcl snippet
proc iplevel_drc {ipinst_handle} {
    set splb_handle [xget_hw_busif_handle $ipinst_handle "SPLB"]
    set splb_conn [xget_hw_value $splb_handle]
    set myport_handle [xget_hw_port_handle "MYPORT"]
    set myport_conn [xget_hw_value $myport_handle]
    if {$splb_conn == "" || $myport_conn == ""} {
        error "Either busif SPLB or port MYPORT must be connected in the design"
        return 1;
    }
    else {
        return 0;
    }
}
```

UPDATE Procedure for a Parameter After System Level Analysis

You can use the parameter subproperty `SYSLEVEL_UPDATE_VALUE_PROC` to specify the Tcl procedure that computes the parameter value, based on other parameters of the same IP. The input handle is a handle to the parameter object of a particular instance of that IP. Note that when this procedure is called, system level parameters computed by Platgen (for example, `C_NUM_MASTERS` on a bus) are already updated with the correct values.

```
## MPD snippet
PARAMETER C_PARAM1 = 5, ..., SYSLEVEL_UPDATE_VALUE_PROC =
sysupdate_param1

## Tcl snippet
proc sysupdate_param1 {param_handle} {
    set retval [somehow_compute_param1]
    return $retval;
}
```

UPDATE Procedure for the IP Instance After System-Level Analysis

You can use the OPTION `SYSLEVEL_UPDATE_PROC` to perform certain actions associated with a specific IP. This procedure is associated with the complete IP and not with a specific parameter, so it cannot be used to update the value of a specific parameter.

For example, you can use this procedure to copy certain files associated with the IP in a particular directory. The input handle is a handle to an instance of the IP:

```
## MPD Snippet
OPTION SYSLEVEL_UPDATE_PROC = syslevel_update_proc
## Tcl snippet
Proc myip_syslevel_update_proc {ipinst_handle} {
    ## do something
    return 0;
}
```

DRC Procedure for a Parameter After System Level Analysis

Use the tag `SYSLEVEL_DRC_PROC` to specify Tcl procedure that performs DRC on the complete IP, based on how the IP has been used in the system. Input is a handle to the parameter object of a particular instance of that IP.

```
PARAMETER C_MYPARAM = 5, ..., SYSLEVEL_DRC_PROC = sysdrc_myparam
```

DRC Procedure for the IP After System Level Analysis

Use the `OPTION SYSLEVEL_DRC_PROC` to specify the Tcl procedure that performs DRC after Platgen updates system level information. The input handle is a handle to an instance of the IP. For example, if this particular IP has been instantiated, the procedure can check to limit the number of instances of this IP, check that this IP is always used in conjunction with another IP, or check that this IP is never used along with another IP.

```
## MPD Snippet
OPTION SYSLEVEL_DRC_PROC = syslevel_drc
BUS_INTERFACE BUS = SPLB, BUS_STD = PLB, BUS_TYPE = SLAVE
PORT MYPORT = "", DIR = 0
## Tcl snippet
proc syslevel_drc {ipinst_handle} {
    set myport_conn [xget_hw_port_value $ipinst_handle "MYPORT"]
    set mhs_handle [xget_hw_parent_handle $ipinst_handle]
    set sink_ports [xget_hw_connected_ports_handle $mhs_handle
$myport_conn "SINK"]
    if {[llength $sink_ports] > 5} {
        error "MYPORT should not drive more than 5 signals"
        return 1;
    }
    else {
        return 0;
    }
}
```

Platgen-specific Call

The `OPTION PLATGEN_SYSLEVEL_UPDATE_PROC` is called after all the common Tcl procedures have been invoked. If you want certain actions to occur only when Platgen runs and not when other tools run, this procedure can be used.

```
## MPD Snippet
OPTION PLATGEN_SYSLEVEL_UPDATE_PROC = platgen_syslevel_update
```

Simgen-specific Call

The `OPTION SIMGEN_SYSLEVEL_UPDATE_PROC` is called after all the common Tcl procedures have been invoked. If you want certain actions to occur when Simgen runs and not when other tools run, this procedure can be used.

```
## MPD Snippet
OPTION SIMGEN_SYSLEVEL_UPDATE_PROC = simgen_syslevel_update
```

FORMAT_PROC

The `FORMAT_PROC` keyword defines the Tcl entry point that allows you to provide a specialized formatting procedure to format the value of the parameter.

The EDK tools deliver output files of two HDL types: Verilog and VHDL. Each format semantic requires that the parameter values be normalized to adhere to a stylized representation suitable for processing. For example, Verilog is case-sensitive and does not have string manipulation functions. When developing an IP, you can use this Tcl entry point to specify procedures to format string values based on the HDL requirements. Refer to the *Platform Specification Format Reference Manual (UG642)* for further details, and examples. [Appendix E, Additional Resources](#) contains a link to the document.

Helper Core Tcl Procedures

All the illustrated Tcl procedures must be specified in the top-level cores. If a top-level core is using helper or library cores, you can execute Tcl procedures specific to those helper cores, by using one of two procedures: `SYSLEVEL_GENERIC_PROC` and `SYSLEVEL_ARCHSUPPORT_PROC`. These tcl procedures must be specified in the `/data` directory of the helper core and must follow the same naming conventions as the other PSF files. (For example: a Tcl file for the `proc_common_v1_00_a` core, must be named in a corresponding nomenclature - `proc_common_v2_1_0.tcl`.)

- The `SYSLEVEL_GENERIC_PROC` procedure is a generic procedure used to print any message.
- The `SYSLEVEL_ARCHSUPPORT_PROC` procedure is used to notify users of deprecated helper cores.

For example, if the `proc_common_v1_00_a` core is deprecated, the core developer can print a message in the tools every time this core is used within a non-deprecated top-level core, by including this procedure in the tcl file of the helper core in the `proc_common_v2_1_0.tcl` file of the `proc_common_v1_00_a` core as follows:

```
proc syslevel_archsupport_proc { mhsinst } {
    print_deprecated_helper_core_message $mhsinst proc_common_v1_00_a
}
```

The `PRINT_DEPRECATED_HELPER_CORE_MESSAGE` procedure is provided by EDK tools to generate a standard message for deprecated cores. It takes the handle to the top-level core and the name of deprecated helper core as arguments.

Additional Keywords in the Merged Hardware Datastructure

Some keywords (sub-properties) that are created optionally on parameters, ports, and bus interfaces in the merged hardware datastructure. These are used internally by tools and can also be used by Tcl for DRCs. These additional keywords are:

- **MHS_VALUE:** When the merged object is created, it combines information from both MHS and MPD. The default value is present in the MPD. However, these properties can be overridden in the MHS. The tools have conditions when some values are auto-computed and that auto-computed value will override the values in MHS also. The original value specified in MHS is then stored in the `MHS_VALUE` sub-property.
- **MPD_VALUE:** When the merged object is created, it combines information from both MHS and MPD. The default value is present in the MPD. However, these properties can be overridden in the MHS. The tools have conditions when some values are auto-computed and that auto-computed value will override the values in MHS also. The value specified in MPD is consequently stored in the `MPD_VALUE` sub-property.
- **CLK_FREQ_HZ:** The frequency of every clock port in the merged hardware datastructure, if available, is stored in a sub-property called `CLK_FREQ_HZ` on that port. This is an internal sub-property and the frequency value is always in Hz.
- **RESOLVED_ISVALID:** If a parameter, port, or bus interface has the sub-property `ISVALID` defined in the MPD, then the tools evaluate the expression to true (1) or false (0) and store the value in an internal sub-property called `RESOLVED_ISVALID` on that property.
- **RESOLVED_BUS:** If a port or parameter in an IP has a colon separated list of buses (specified in the `BUS` tag) that it can be associated with in the MPD file, the tools analyze the connectivity of that IP and determine to which of those buses the IP is connected, and store the name of that bus interface in the `RESOLVED_BUS` tag..

Interconnect Settings and Parameter Automations for AXI Designs

The MPD and MHS Chapters of the [Platform Specification Format Reference Manual\(UG642\)](#) describe the INTERCONNECT-related parameters that are captured on the end-point master and slave bus interfaces. These parameters usually contain the C_INTERCONNECT_<BusIf>_ prefix, where <BusIf> is the actual name of the bus interface (such as "M_AXI_DP" in MicroBlaze).

Allowed Parameters in Master and Slave Interfaces

The following parameters are allowed in Master and Slave interfaces. These are described in more detail in the MPD chapter of the *Platform Specification Format Reference Manual*.

For Master Interfaces, the allowed parameters are:

- C_INTERCONNECT_<BusIf>_BASE_ID
- C_INTERCONNECT_<BusIf>_IS_ACLK_ASYNC
- C_INTERCONNECT_<BusIf>_ACLK_RATIO
- C_INTERCONNECT_<BusIf>_ARB_PRIORITY
- C_INTERCONNECT_<BusIf>_AW_REGISTER
- C_INTERCONNECT_<BusIf>_AR_REGISTER
- C_INTERCONNECT_<BusIf>_W_REGISTER
- C_INTERCONNECT_<BusIf>_R_REGISTER
- C_INTERCONNECT_<BusIf>_B_REGISTER
- C_INTERCONNECT_<BusIf>_WRITE_FIFO_DEPTH
- C_INTERCONNECT_<BusIf>_READ_FIFO_DEPTH
- C_INTERCONNECT_<BusIf>_WRITE_ISSUING
- C_INTERCONNECT_<BusIf>_READ_ISSUING

For Slave Interfaces, the allowed parameters are:

- C_INTERCONNECT_<BusIf>_MASTERS
- C_INTERCONNECT_<BusIf>_IS_ACLK_ASYNC
- C_INTERCONNECT_<BusIf>_ACLK_RATIO
- C_INTERCONNECT_<BusIf>_SECURE
- C_INTERCONNECT_<BusIf>_AW_REGISTER
- C_INTERCONNECT_<BusIf>_AR_REGISTER

- C_INTERCONNECT_<BusIf>_W_REGISTER
- C_INTERCONNECT_<BusIf>_R_REGISTER
- C_INTERCONNECT_<BusIf>_B_REGISTER
- C_INTERCONNECT_<BusIf>_WRITE_FIFO_DEPTH
- C_INTERCONNECT_<BusIf>_READ_FIFO_DEPTH
- C_INTERCONNECT_<BusIf>_WRITE_ACCEPTANCE
- C_INTERCONNECT_<BusIf>_READ_ACCEPTANCE

These parameters are:

- NON_HDL parameters, meaning that they do not affect the behavior of the end point IP.
- Not present in the MPD of the IPs.

However, XPS tools allow these parameters to be specified as parameters in the MHS instances of the peripherals connected to the AXI Interconnect (end point IPs) in the MHS file.

In the context of the system as a whole, the AXI Interconnect needs to know about certain properties of the IP interfaces to which that are connected. It is simpler to capture these values on the end-point IPs. The main advantages to this approach are:

- The AXI Interconnect has vectored parameters to capture the values of parameters. Because the interconnect allows up to 16 masters and 16 slaves to be connected to it, the value of each forms part of a vectored value. Although it is possible to design a smart interface to capture the values of these parameters in a non-vectored fashion, it is inefficient to enter vectored values in the MHS by hand.
- IP information resides in a single location, so you can view core details, including some system-level settings, at one place in the MHS.
- When you need to move a core from one AXI Interconnect to another, you need only to change the bus interface name on the core. All AXI Interconnect-related settings are preserved by the tools. As long as the other AXI Interconnect is an AXI Interconnect with the same version, you do not need to specify the settings again.

The IP Configuration dialog boxes of the end point IPs include the **Interconnect Settings for BUSIF** tab, which captures the values of these parameters. At runtime, the XPS tools gather the values of these parameters from all the end-point IPs and transfer them onto the AXI Interconnect.

C_<BusIf>_AXI_ID_WIDTH

The AXI Interconnect appends the BASE ID bits to transactions. The slaves must know how many bits are appended by the AXI Interconnect, which is specified in the C_<BusIf>_AXI_ID_WIDTH parameter. Regardless of whether the tools computed the BASE ID values or you specified them, the tools compute the maximum number of bits necessary to make the masters unique and set that value as the AXI_ID_WIDTH on the AXI Interconnect and the connected slaves.

C_INTERCONNECT_<BusIf>_ACLK_RATIO

This parameter determines whether the frequency of the clock port of the master/slave interface is at an integer ratio with respect to the frequency of the clock port of the interconnect.

The tools trace the IP clocks in the design to identify the value of the frequency of the clock port. They do this based on the CLK_FREQ_HZ sub-property on the clock port (identified by SIGIS = CLK tag in the MPD). If this sub-property does not exist, the tools create the sub-property by tracing the clock port connection through bus interfaces, clock generator, external ports, etc.

Once the clock frequencies are determined, the tools then compute the values of C_INTERCONNECT_<BusIf>_ACLK_RATIO parameters. To compute this parameter, the tools look at each interconnect in the design and identify the lowest clock frequency of all the masters and slaves connected to that interconnect. That lowest frequency is then considered as base 1. All the ratios are then computed with respect to that frequency.

For example, consider an AXI Interconnect (axi_0) with the following:

- Three masters, M1, M2, and M3 with frequencies of 200 MHz, 100 MHz, and 100 MHz, respectively, on their M_AXI interfaces
- Two slaves, S1 and S2, with frequencies of 100 MHz and 50 MHz, respectively, on their S_AXI interfaces
- A clock frequency of 100 MHz on the AXI Interconnect

In this case, the tools compute the ratios to be:

- Lowest clock frequency is on S2 => ratio = 1
- Ratio of M1 with respect to S2 = 200:50 => 4
- Ratio of axi_0 with respect to S2 = 100:50 => 2 and so on.

The C_INTERCONNECT_<BusIf>_ACLK_RATIO parameters in the above example have the values as shown below:

- For M1 (Master) - parameter C_INTERCONNECT_M_AXI_ACLK_RATIO = 4
- For M2 (Master) - parameter C_INTERCONNECT_M_AXI_ACLK_RATIO = 2
- For M3 (Master) - parameter C_INTERCONNECT_M_AXI_ACLK_RATIO = 2
- For axi_0 - parameter C_INTERCONNECT_ACLK_RATIO = 2
- For S1 (Slave) - parameter C_INTERCONNECT_S_AXI_ACLK_RATIO = 2
- For S2 (Slave) - parameter C_INTERCONNECT_S_AXI_ACLK_RATIO = 1

These values are automatically updated by the tools and you cannot override them.

C_INTERCONNECT_<BusIf>_IS_ACLK_ASYNC

This parameter is used to specify whether the frequency of the clock port of the bus interface (master/slave) is asynchronous with respect to the frequency of the clock port of the interconnect. Whenever the interface is asynchronous with respect to the interconnect, the interconnect inserts some additional logic to handle that situation.

As mentioned in the MHS Chapter of the [Platform Specification Format Reference Manual \(UG642\)](#), tools require that all IPs in the design be connected to a clock port. So, when tools identify the clock frequencies of different interfaces, they compute the ratio parameters. If the ratio is a non-integer ratio, the IS_ACLK_ASYNC parameter is set to 1. Otherwise, if the ratio is an integer ratio, the tools set the value of that parameter to 0.

To make a particular clock asynchronous with respect to the interconnect, you can override the value of this parameter in the MHS. The tools will not update that parameter.

Note: If you override the C_INTERCONNECT_<BusIf>_IS_ACLK_ASYNC parameter for any interface, the tools ignore that frequency when trying to identify the lowest clock for determining clock ratios. Also, tools do not compute the ratio for that particular interface, as it is marked *asynchronous*.

C_<BusIf>_SUPPORTS_NARROW_BURST

If this parameter is present on the AXI slave interfaces, the tools automatically update it to optimize the design. The tools analyze the design at run time. When there are no masters connected to the interconnect that can generate narrow bursts, they set this parameter (on the slave to '0') to disable narrow burst support logic and save resources.

C_<BusIf>_SUPPORTS_READ

If this parameter is present on the AXI slave interfaces, the tools automatically update it to optimize the design. The tools analyze the design at run time. If there are no masters connected to the interconnect that use the AR and R channels, they set this parameter (on the slave to 0) to disable AR and R channels and save resources.

C_<BusIf>_SUPPORTS_WRITE

If this parameter is present on the AXI slave interfaces, it is automatically updated by the tools to optimize the design. The tools analyze the design at run time. If there are no masters connected to the interconnect that use the AW and W channels, they set this parameter (on the slave to '0') to disable AW and W channels and save resources.

User Signal Width parameters on the AXI interconnect

The tools analyze the design at runtime and check the value of the user signal widths of masters and slaves connected to the AXI Interconnect, then compute the maximum value of the channel width for the AW, AR, and W channels between the masters and set those values on the AXI Interconnect. Similarly, they compute the maximum value of the channel width for the R and B channels between the slaves and set those values on the AXI Interconnect.

Additional Resources

Xilinx Resources

- **Device User Guides:**
http://www.xilinx.com/support/documentation/user_guides.htm
- **Glossary of Terms:** <http://www.xilinx.com/company/terms.htm>
- **Xilinx Design Tools: Installation and Licensing Guide (UG798):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/iil.pdf
- **Xilinx Design Tools: Release Notes Guide (UG631):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/irn.pdf
- **Product Support and Documentation:** <http://www.xilinx.com/support>

EDK Documentation

You can also access the entire documentation set online at:

http://www.xilinx.com/support/documentation/dt_edk_edk14-1.htm

Individual documents are linked below.

- **EDK Concepts, Tools, and Techniques (UG683):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/edk_ctt.pdf
- **EDK Profiling Guide (UG448):**
http://www.xilinx.com/support/documentation/xilinx14_1/edk_prof.pdf
- **MicroBlaze Processor User Guide (UG081):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf
- **Platform Specification Format Reference Manual (UG642):**
http://www.xilinx.com/support/documentation/xilinx14_1/psf_rm.pdf
- **PowerPC 405 Processor Block Reference Guide (UG018):**
http://www.xilinx.com/support/documentation/user_guides/ug018.pdf
- **PowerPC 405 Processor Reference Guide (UG011):**
http://www.xilinx.com/support/documentation/user_guides/ug011.pdf
- **PowerPC 440 Embedded Processor Block in Virtex-5 FPGAs (UG200):**
http://www.xilinx.com/support/documentation/user_guides/ug200.pdf
- **Zynq™ Concepts, Tools, and Techniques (UG873):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug873_zynq_ctt.pdf
- **Zynq-7000 Software Developers Guide (UG821):**
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug821-zynq-7000-swdev.pdf

EDK Additional Resources

- **EDK Tutorials website:**
http://www.xilinx.com/support/documentation/dt_edk_edk14-1_tutorials.htm
- **Platform Studio and EDK website:**
http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
- **XPS/EDK Supported IP website:**
http://www.xilinx.com/ise/embedded/edk_ip.htm