Embedded System Tools Guide

Embedded Development Kit

EDK 6.1 October 6, 2003

XILINX®

Embedded System Tools Guide
EDK 6.1 October 6, 2003

The following table shows the revision history for this document.

	Version	Revision
06/24/02	1.0	Initial Xilinx EDK (Embedded Processor Development Kit) release.
08/13/02	1.1	EDK (v3.1) release.
09/02/03	6.1	EDK 6.1 release.

Chapter 3: Base System Builder

Chapter 4: Import Peripheral Wizard

Chapter 5: Platform Generator

Chapter 6: Simulation Model Generator

Chapter 7: Library Generator

Chapter 8: Platform Specification Utility

Chapter 28: Device Drivers

Chapter 29: Stand-Alone Board Support Package

About This Guide

Welcome to the Embedded Developement Kit. This kit is designed to provide designers with a rich set of design tools and a wide selection of standard peripherals required to build embedded processor systems using MicroBlaze, the industry's fastest soft processor solution, and the new and unique feature in Virtex-II Pro, the IBM ® PowerPC ® CPU.

This guide provides information about the Embedded System Tools (EST) included in the Embedded Development Kit (EDK). These tools, consisting of processor platform tailoring utilities, software application development tool, a full featured debug tool chain and device drivers and libraries, allow the developer to fully exploit the power of MicroBlaze and Virtex-II Pro.

Guide Contents

This guide discusses the following topics:

- Embedded System Tools Flow
- Processor Platform Tailoring Utilities
- Software Application Development Tools
- Debug Tool Chain
- Simulation
- Libraries
- Drivers
- Software Specification

Additional Resources

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

Resource	Description/URL
EDK Home	Embedded Development Kit home page, FAQ and tips. http://www.xilinx.com/edk
EDK Examples	A set of complete EDK examples. http://www.xilinx.com/ise/embedded/edk_examples.htm

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records http://support.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/partinfo/databook.htm
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://support.xilinx.com/support/troubleshoot/psolvers.htm
GNU Manuals	The entire set of GNU manuals http://www.gnu.org/manual

Conventions

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

Typographical

The following typographical conventions are used in this document:

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

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

Online Document

The following conventions are used in this document:

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

Chapter 1

Embedded System Tools Architecture

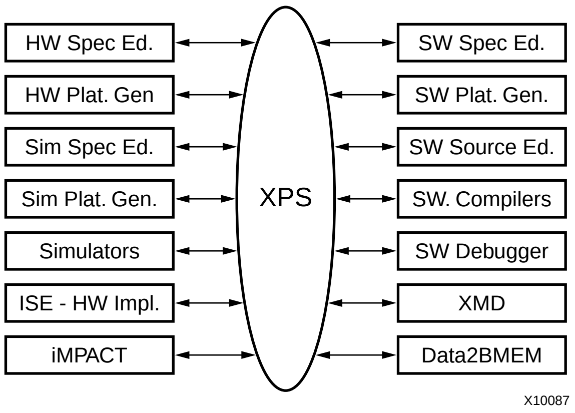
Summary

This chapter describes the Embedded System Tools (EST) architecture and flows for the Xilinx embedded processors, PowerPC 405 and MicroBlaze.

Tool Architecture Overview

Figure 1-1 depicts the embedded software tool architecture. Multiple tools based on a common framework allow the user to design the complete embedded system. System design consists of the creation of the hardware and software components of the embedded processor system, and optionally, a verification or simulation component as well. The hardware component consists of an automatically generated hardware platform that can be optionally extended to include other hardware functionality specified by the user. The software component of the design consists of the software platform generated by the tools, along with the user designed application software. The verification component consists of automatically generated simulation models targeted to a specific simulator, based on the hardware and software components.

Figure 1-1: **Embedded Software Tool Architecture**



Tool Flows

A typical embedded system design project involves the following phases:

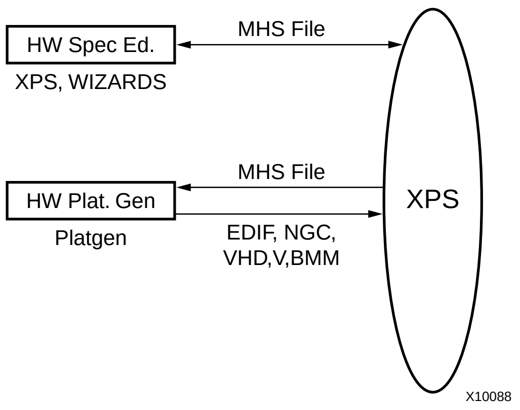
- hardware platform creation,
- hardware platform verification (simulation),
- software platform creation,
- software application creation, and
- software verification (debugging).

Xilinx provides tools to assist in all the above design phases. These tools play together with other, third-party tools such as simulators and text editors that may be used by the designers.

Hardware Platform Creation

Hardware platform creation is depicted in Figure 1-2.

Figure 1-2: **Hardware Platform Creation**



The hardware platform is defined by the MHS (Microprocessor Hardware Specification) file (see Chapter 15, “Microprocessor Hardware Specification (MHS)” for more information). The hardware platform consists of one or more processors and peripherals connected to the processor buses. Several useful peripherals are usually supplied by Xilinx, along with the EDK tools. Users can define their own peripherals and include them in the MHS by following the guidelines in Chapter 14, “Platform Specification Format (PSF)”. The MHS file is a simple text file and any text editor can be used to create this file. The XPS tool provides graphical means to create the MHS file.

The MHS file defines the system architecture, peripherals and embedded processors. The MHS file also defines the connectivity of the system, the address map of each peripheral in the system and configurable options for each peripheral. Multiple processor instances connected to one or more peripherals through one or more buses and bridges can also be specified in the MHS.

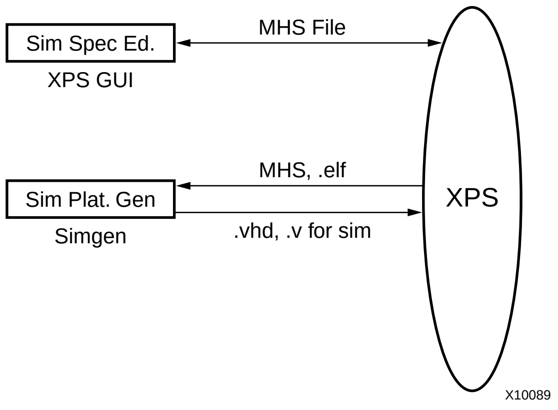
The Platform Generator tool (platgen) creates the hardware platform using the MHS file as input. Platgen creates netlist files in various formats (NGC, EDIF), as well as support files for downstream tools, and top level HDL wrappers to allow users to add other components to the automatically generated hardware platform. See Chapter 5, "Platform Generator," for more information.

Note: After running platgen, FPGA implementation tools (ISE) are run to complete the implementation of the hardware. Typically, XPS spawns off the ProjNav front end for the implementation tools, allowing full control over the implementation. See ISE documentation for more info on the ISE tools. At the end of the ISE flow, a bitstream is generated to configure the FPGA. This bitstream includes initialization information for BRAM memories on the FPGA chip. If user code or data is required to be placed on these memories at startup time, the Data2MEM tool in the ISE toolset is used to update the bitstream with code/data information obtained from the user's executable files that are generated at the end of the "Software Application Creation and Verification" flow.

Verification Platform Creation

The verification platform is based on the hardware platform. The verification specification allows the user to specify a simulation model for each processor, peripheral or other module in the hardware platform. The MHS file is processed by the Simgen tool to create simulation files (VHDL, Verilog or various compiled models) along with some command files for specific simulators supported by the tool. See Chapter 6, "Simulation Model Generator" for more information. As in the case of the hardware platform, these simulation files may be edited by the user to add other components to the automatically generated verification platform. The entire process of generating the verification platform is depicted in Figure 1-3. If the software application that runs on the hardware platform is available in executable format, it can be used to initialize memories in the verification platform. Details of this process are provided in later chapters.

Figure 1-3: **Verification Platform.**

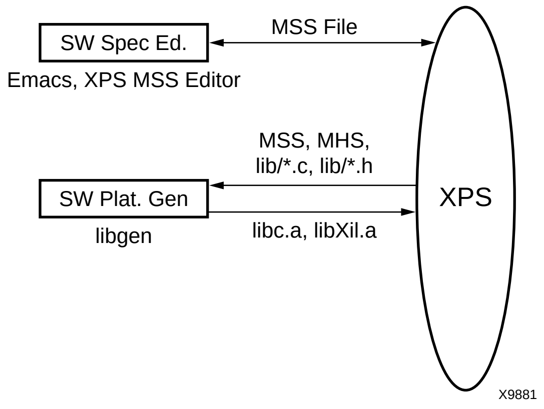


Software Platform Creation

The software platform is defined by the MSS (Microprocessor Software Specification) file (see Chapter 19, "Microprocessor Software Specification (MSS)" for more information). The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is a simple text file and any text editor can be used to create this file. The XPS tool (see Chapter 2, "Xilinx Platform Studio" for more information) provides a graphical user interface for creating the MSS file.

The MSS file is an input to the Library Generator tool (LibGen) for customization of drivers, libraries and interrupt handlers. See Chapter 7, "Library Generator" for more information. The entire process of creating the software platform is shown in Figure 1-4.

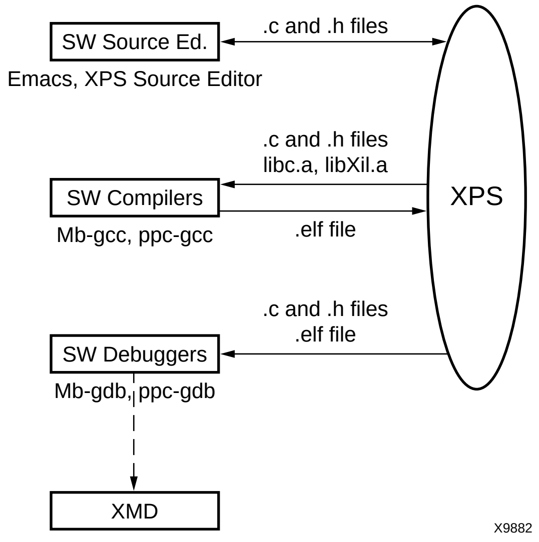
Figure 1-4: **Software Platform**



Software Application Creation and Verification

The software application is the code that runs on the hardware and software platforms. The source code for the application is written in a high level language such as C or C++, or in assembly language. XPS provides a source editor for creating these files, but any other text editor may be used here. Once the source files are created, they are compiled and linked to generate executable files in the ELF (Executable and Link Format) format. GNU compiler tools (see Chapter 11, "GNU Compiler Tools" for more information) for PowerPC and MicroBlaze are used by default but other compiler tools that support the specific processors used in the hardware platform may be used as well. XMD and the GNU debugger (GDB) are used together to debug the software application. XMD provides an instruction set simulator, and optionally connects to a working hardware platform to allow GDB to run the user application. This entire process is depicted in Figure 1-5. See Chapter 13, "Xilinx Microprocessor Debugger" for more information on XMD and Chapter 12, "GNU Debugger" for more information on GDB.

Figure 1-5: **Software Application Creation and Verification**



Some Useful Tools

Xilinx Platform Studio

The Xilinx Platform Studio (XPS) tool provides a GUI for creating the MHS and MSS files for the hardware and software flow. XPS also provides source file editor capability and project and process management capability. XPS is used for managing the complete tool flow, that is, both hardware and software implementation flows. Please see Chapter 2, “Xilinx Platform Studio” for more information.

Platform Generator

The embedded processor system in the form of hardware netlists (HDL and EDIF files) is customized and generated by the Platform Generator (platgen).

Please refer Chapter 5, “Platform Generator” for more information.

HDL Synthesis

Platgen generates hierarchal NGC netlists in the default mode. This means that each instance of a peripheral in the MHS file is synthesized. The default mode leaves the top-level HDL file untouched allowing any synthesis tool to be used. Currently, Platform Generator only supports XST (Xilinx Synthesis Technology).

ISE XST

If Platform Generator is run in the default mode, a synthesis script file for XST is created. This script can be executed under XST using the following command:

```
xst -ifn system.scr
```

Simulation Model Generator

The Simulation Platform Generation tool (simgen) generates and configures various simulation models for the hardware. It takes a Microprocessor Hardware Specification (MHS) file as input. Note: Previous versions of Simgen used a separate specification file called the MVS file. MVS files are not used in this version of the software.

Please refer Chapter 6, "Simulation Model Generator" for details.

Library Generator

XPS calls the Library Generator tool for configuring the software flow.

The Library Generator (libgen) tool configures libraries, device drivers, file systems and interrupt handlers for the embedded processor system. The input to LibGen is an MSS file.

Please see Chapter 7, "Library Generator" for more information. For more information on Libraries and Device Drivers please refer to Chapter 22, "Xilinx Microkernel (XMK)" and Chapter 28, "Device Drivers".

GNU Compiler Tools

XPS calls GNU compiler tools for compiling and linking application executables for each processor in the system.

Given a set of C source files, a Microprocessor executable is created as follows.

Microblaze

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

This command compiles and links the files into an executable that can run on the MicroBlaze processor. The output executable is in **a.out**. The **-o** flag can be used to specify a different file name for the output file.

In order to initialize memories in the hardware bitstream with this executable, the file name should have an **elf** extension.

For further information on compiler options, **mb-gcc -help** can be run on the command line. Please refer Chapter 11, "GNU Compiler Tools" for more information.

PowerPC

```
powerpc-eabi-gcc file1.c file2.c
```

This command compiles and links the files into an executable that can run on the PowerPC processor. The output executable is in **a.out**. The **-o** flag can be used to specify a different file name for the output file.

In order to initialize memories in the hardware bitstream with this executable, the file name should have an **elf** extension.

For further information on compiler options, **powerpc-eabi-gcc -help** can be run on the command line. Please refer Chapter 11, "GNU Compiler Tools" for more information.

Compiling with Optimization

Once you are satisfied that your program is correct, recompile your program with optimization turned on. This will reduce the size of your executable, and reduce the number of cycles it needs to execute. This is achieved by the following:

```
mb-gcc -O3 file1.c file2.c
```

Setting the Stack Size

By default, the EDK tools build the executable with a default stack size of 0x100 (256) bytes.

The stack size can be set at compile time by using:

```
mb-gcc file1.c file2.c -Wl,defsym -Wl,_STACK_SIZE=0x400
```

This will set the stack size to 0x400 (1024) bytes.

Software Debugging

You can debug your program in software (using a simulator, available for MicroBlaze only), or on a board which has a Xilinx FPGA loaded with your hardware bitstream. Refer to the XMD documentation for more information.

Debugging Using Hardware: software intrusive

Create your application executable using the compiler. For example

```
mb-gcc -g -xl-mode-xmdstub file1.c file2.c
```

This command creates the Microprocessor executable *a.out*, linked with the C runtime library crt1.o and starting at physical address 0x400, and with debugging information that can be read by **mb-gdb** (or **powerpc-eabi-gdb** if compilation was done for PowerPC).

If you want to debug your code using a board, you must specify the **DEFAULT_INIT** parameter for that processor to **XMDSTUB** in **MSS** file. This creates a data2mem script (**run_download**) file that initializes the Local Memory (LM) with the **xmdstub** executable. Next, load the bitstream representing your design onto your FPGA. Refer to XMD and Libgen documentation for more information.

Start xmd server in a new window with the following command:

```
xmd
```

Connect to use **stub** target GDB. Please see XMD documentation for more information.

Load the program in mb-gdb using the command:

```
mb-gdb a.out
```

Click on the "Run" icon and in the mb-gdb Target Selection dialog, choose

- Target: Remote / TCP
- Hostname: localhost
- Port: 1234

Now, mb-gdb's Insight GUI can be used to debug the program.

Debugging Using A Simulator: non-intrusive

If you want to debug your code using a simulator, compile programs using the following command:

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

This command creates the MicroBlaze executable file, *a.out*, with debugging information that can be accessed by mb-gdb. For PowerPC, the compiler used is powerpc-eabi-gcc.

Xilinx EDK provides two ways to debug programs in simulation.

1. Cycle-accurate simulator in XMD:

Start xmd server in a new window with the following command:

```
xmd
```

Connect using **sim** target. Please see the XMD documentation for more information.

Loading and debugging the program in mb-gdb is done the same way as for xmd in hardware mode described above.

This is the preferred mechanism to debug user programs in simulation

2. Simple ISA simulator in mb-gdb:

The xmd server is not needed in this mode. After loading the program in mb-gdb, Click on the “Run” icon and in the mb-gdb Target Selection dialog, choose “**Simulator**”.

Use this mechanism only if your program does not attempt to access any peripherals (not even via a print call).

Dumping an Object/Executable File

The mb-objdump utility lets you see the contents of an object (.o) or executable (.out) file.

To see your symbol table, the size of your file, and the names/sizes of the sections in the file, run the following:

```
mb-objdump -x a.out
```

To see a listing of the (assembly) code in your object or executable file, use

```
mb-objdump -d a.out
```

To get a list of other options, use the following command:

```
mb-objdump --help
```

Verifying Tools Setup

The environment variable ***XILINX_EDK***, needs to be set at the level of the hierarchy where the directories **doc**, **hw**, and **bin** reside.

Tools Directory Path

Ensure that the GNU tools are in your path.

For Solaris

Check the executable search path. Your path must include the following:

- ${XILINX_EDK}/gnu/microblaze/sol/bin
- ${XILINX_EDK}/gnu/powerpc-eabi/sol/bin
- ${XILINX_EDK}/bin/sol

For PC

Check the executable search path.

- %XILINX_EDK%\gnu\microblaze\nt\bin
- %XILINX_EDK%\gnu\powerpc-eabi\nt\bin
- %XILINX_EDK%\bin\nt

Xilinx Alliance Software

The system should be set up to use the Xilinx Development System. Please verify that the system is properly configured. Consult release notes and installation notes included in the Xilinx ISE software package for more information. The EDK 3.2 release supports Xilinx ISE 5.2 Tools.

Chapter 2

Xilinx Platform Studio

Overview

This chapter describes the Xilinx Platform Studio (XPS) IDE for the Xilinx Embedded Processors, MicroBlaze and PowerPC.

Xilinx Platform Studio (XPS) provides an integrated environment for creating the software, hardware specification flows for a Embedded Processor system. It also provides an editor and a project management interface to create and edit source code. XPS offers customization of tool flow configuration options. It also provides a graphical system editor for connection of processors, peripherals and buses. XPS is available on both Windows and Solaris platforms. There is also an batch mode invocation of XPS available.

Processes Supported

XPS supports the creation of the MHS (refer to the Microprocessor Hardware Specification chapter), and MSS file (refer to the Microprocessor Software Specification chapter) files needed for embedded tools flow. The MVS file used in EDK 3.2 has been discontinued and that information is stored in XPS project files. XPS also aids users in creating a MHS (refer to Microprocessor Hardware Specification chapter) through a dialog based editor and bus connection matrix, or through a graphical block diagram editor (referred to as the Platform Block Diagram editor). It supports customization of software libraries, drivers, interrupt handlers and compilation of user programs. Source management of C source files and header files for user applications is also provided by XPS. Users can also choose the simulation mode for the complete system. Users can begin a project by either importing an existing MHS file or by starting with an empty MHS file and then adding cores to it. It performs process management and dependency checking between the hardware, software and simulation tool flows by calling the tools in the correct order using the makefile mechanism. Figure 2-1 provides a detailed view of processes supported by XPS.

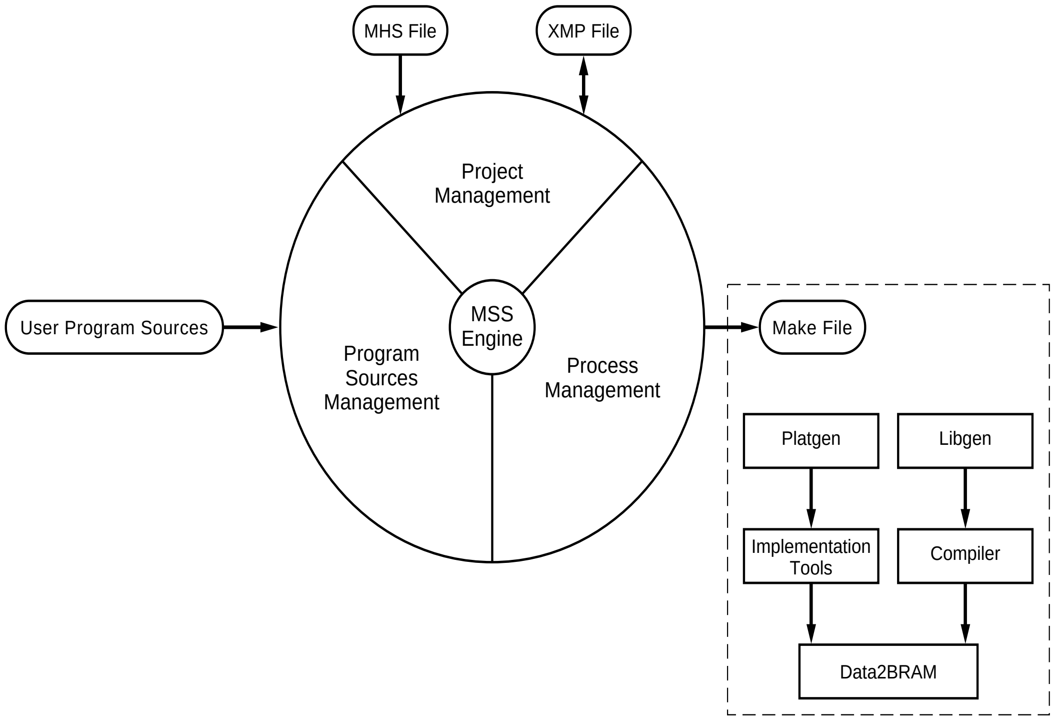


Figure 2-1: **XPS Process**

Tools Supported

Table 2-1 describes the tools that are supported in the XPS.

Table 2-1: **Tools supported in XPS**

Tool	Function	Reference/Notes
Library Generator (LibGen)	Customizes software libraries, drivers and interrupt handlers	The Library Generator Documentation
GNU Compiler Tools	Preprocess, compile, assemble and link programs	GNU tools Documentation
Platform Generator (PlatGen)	Allows to customize various options. Runs platgen with the options and the MHS file	The Platform Generator Document
Simulation Model Generator (SimGen)	Generates the hardware simulation model and the compilation script file for the complete system.	The Simulation Model Generator
Makefile	Generates a Makefile, which provides targets to run various hardware and software flow tools.	Uses gmake on Solaris.
System ACE	Generates SystemACE file	Not supported on Solaris
XMD	Opens an XMD terminal for the user for on-board debug.	XMD Documentation
Project Navigator Export and Import	Export and Import design to Project Navigator for synthesis and implementation of design.	Flow is an alternative to the XFlow mechanism in XPS.

Features

XPS has the following features

- Adding cores and editing core parameters, making bus and signal connections to generate a Microprocessor Hardware Specification (MHS)
- Generation and modification of the Microprocessor Software Specification (MSS)
- Support for all the tools described in Table 2-1.
- Graphical Block Diagram View and Editor.
- Viewing and editing of C source and header files
- Project Management
- Process and tool flow dependency management

Project Management

Project information is saved in a Xilinx Microprocessor Project (XMP) file. An XMP file consists of the location of the MHS file, the MSS file, and the C source and header files that need to be compiled into an executable for a processor. The project also includes the FPGA architecture family and the device type for which the hardware tool flow needs to be run.

Creating A New Project

A New Project is created using the **New Project** menu option in the Project submenu of the main menu. The **Base System Builder Wizard** option in the New Project menu can be used to invoke the wizard to create a basic system. Please refer to the Base System Wizard documentation for more information. The **Platform Studio** option can be used to create a new project using XPS. The **New Project** toolbar button can also be used.

For creating a new project, users need to specify the location of the **xmp** file. The name of the xmp file is take to be the project name and the directory where the xmp file resides is considered to be the project directory. All tools are invoked from the project directory. All relative paths are assumed to be relative to the project directory. Optionally, users can also specify an MHS file to be used for the project if the project is created using Platform Studio. If the specified MHS file does not exist in the project directory or does not have same name as the project name, XPS copies it into the project directory with same base name as the project name. XPS always modifies the local copy of the MHS and never refers to the original MHS.

The target architecture **must** be set before running any tool. However, choosing the device size, the package and the speed grade can be defered till implementation of the design. These options can also be set/changed later in the **Set Project Options** dialog box in **Options->Project Options** menu.

Users **must** specify all **Search Path** directories before loading the project **if**

- The MHS uses a peripheral which is not present either in the Xilinx EDK installation area or in **pcores** directory of the XPS project directory.
- The MSS uses a driver which is not present either in the Xilinx EDK installation area or in the **drivers** directory of the XPS project directory.

The concept of a Search Path directory, and its subdirectory structure is explained in detail in PlatGen and LibGen chapters. This corresponds to the **-lp option** of the tools. Please note that all the tools automatically look into the **pcores,** and **drivers** directories in the project directory and that the project directory itself should **not** be specified as the Search

Path. Multiple directories can be specified as part of search path by specifying a semicolon (;) separated list of directories.

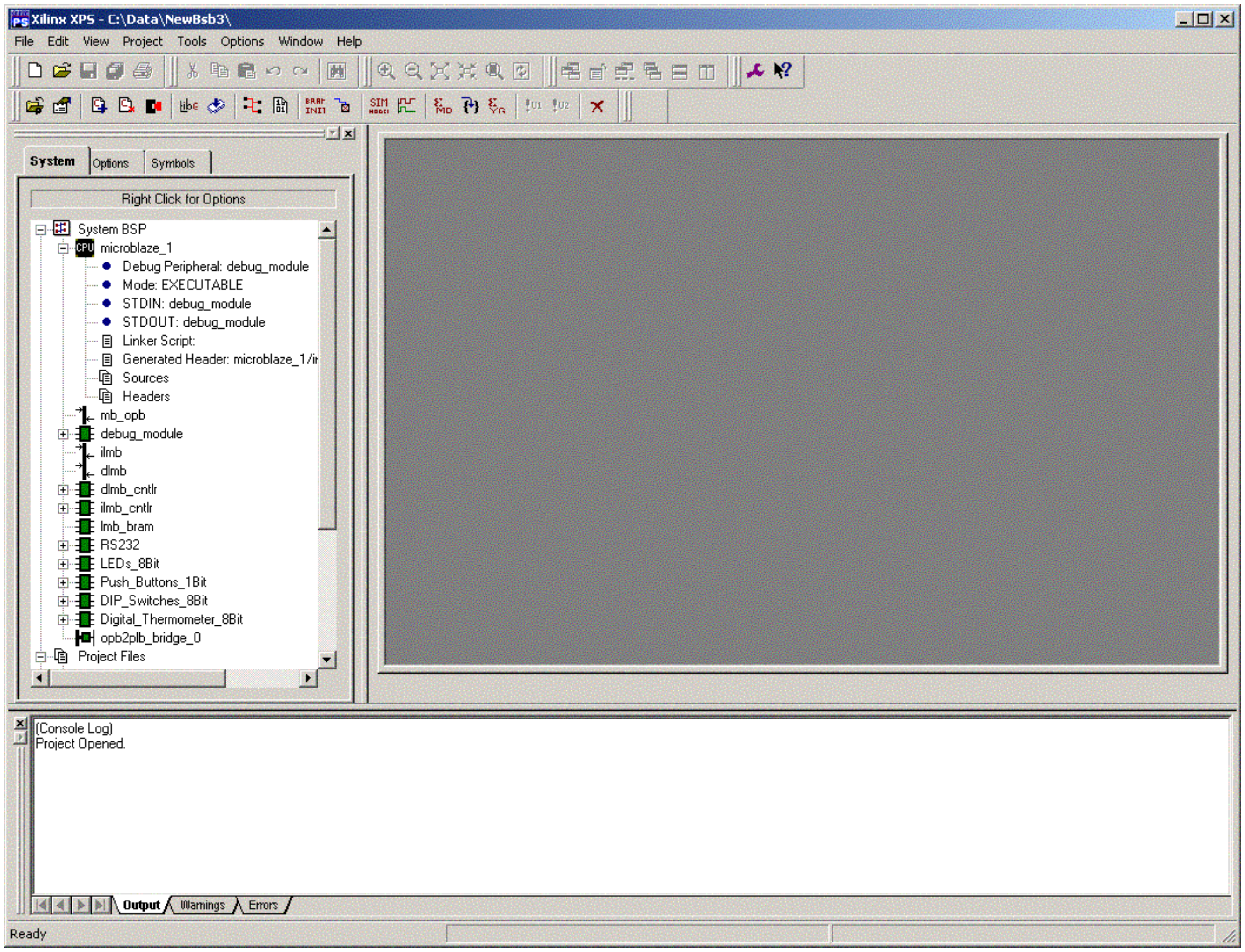
Opening An Existing Project

An existing XPS project can be opened by using the **Open Project** menu option (**File** menu) or using the Open Project button on the toolbar and specifying the existing XMP file corresponding to that project.

New source files and header files can be created, added, and deleted as described in the Source Code Management section of this chapter.

XPS does not allow multiple projects to be open simultaneously. Any open project must be closed before another project can be opened.

XPS Interface



```
[Console Log]
Project Opened.
```

Figure 2-2: **XPS Screenshot**

Figure 2-2 shows a screenshot of XPS. XPS opens three main windows by default.

Editor Workspace

The main editor workspace appears on the right in XPS in Figure 2-2. The workspace opens PBD (Platform Block Diagram) file and allows graphical editing of the system. The main workspace also functions as a C source and header file editor of XPS. Users can also view and edit other text files in the main window. Any number of text files can be opened simultaneously in the XPS main window. The PBD file can be opened by double clicking on the PBD file in the system tree view

The PBD Editor and its operation is detailed in a separate section in this document.

System Tabs

This tab is one of the three tabs that appear on the left in the XPS window in Figure 2-2. The system tab shows the system in a tree format. There are three sub-trees in this view:

- The **System BSP** tree shows system components (various cores) by their instance names. Each core can have its own sub-tree which displays information corresponding to that instance (for example base address and high address). Source and header files corresponding to a processor are listed in the sub-tree for that processor instance.
- The **Project Files** tree shows the MHS. MSS, PBD and UCF files corresponding to the project. Users can double-click on any of the file names to open it in the XPS main window.
- The **Project Options** tree shows the current value set for various project options. Users can double-click or do a Right-click on any of the fields shown in this tree to bring up the **Set Project Options** dialog box.

Transcript Window (Output)

The transcript window is the bottom window in Figure 2-2. This window acts as a console for output, warning and error messages from XPS and from other tools invoked by XPS.

Platform Management

In order to change the system specification, software settings, and simulation options, XPS supports the following features and processes.

Add Cores (Dialog)

A **Right click** on **System BSP** item in the System View tab gives a menu option to **Add Cores (dialog)** to the system. Selecting it brings up a tabbed dialog box that lists all the cores which can be instantiated in the MHS file. Multiple cores can be selected at a time for adding to the MHS file by using the ‘Shift’ or ‘Ctrl’ key. The tabs can be used to add and connect buses, connect BRAMs to BRAM controllers, add ports and connect using net names and set parameters on cores. Please refer to the MPD and MHS document for parameter information. Also the IP documentation includes parameters that can be changed for each IP.

Simulation Models

A **Right click** on **System BSP** item in the System View tab gives a menu option to set the **Simulation Model** for the system. User can choose between **Behavioral**, **Structural**, and **Timing** modes of simulation. The currently selected model has a check mark against it. This information is stored in XMP file.

View MPD

Right click on an instance name give users the option to “View MPD” for that core. If selected, the MPD file for that core is opened in the main window. If the MPD file is already open, focus is set on the file. MPD files are opened in read-only mode and can not be edited.

View MDD

Right click on an instance name gives users the option to “View MDD” for driver assigned to that core instance. This option is disabled if no driver is assigned to that core. If selected, the MDD file for that core’s driver is opened in the main window. If the MDD file is already open, focus is set on the file. MDD files are opened in read-only mode and can not be edited.

S/W Settings

In the System BSP tree, a **double click** on an instance name opens a dialog window displaying configurable software options for that peripheral. This window can also be brought up by doing a Right click on peripheral instance name and choosing the menu item **S/W Settings**.There are two different kinds of dialog windows, the **Processor Dialog Window** for cores of type **PROCESSOR** (MicroBlaze or PowerPC), and the **Peripheral Dialog Window** for all non-PROCESSOR cores. Note that **no S/w Settings** are required for cores of type **IP** and **BUS**, therefore the option is disabled for such cores. The type of a core is defined in the Microprocessor Peripheral Description (MPD) file corresponding to that core.

Peripheral Dialog Window

A **Peripheral Dialog Window** opens up when you double-click or choose **S/W Settings** menu on the instance name of a core, if the core is of type **PERIPH**, **BRIDGE**, and **BUS_ARBITER**. The options which can be set in a Peripheral Dialog Window are as follows.

Interrupt Handler Routines

The name of the interrupt handling routine is specified for any peripheral interrupt signal. If the peripheral has no interrupt port, or if those interrupt port(s) are not connected to any signal in the MHS file, then this edit box is disabled. Currently, XPS can only handle upto two interrupt ports. If there are more than 2 connected interrupt ports in the MHS file, users can edit the MSS file using the text editor.

Driver Options

There are three edit boxes which allow you to set the name of the driver, the driver version and the interface level of driver to be set for that peripheral. If you do not select any driver interface level, the default level specified in the MDD (Microprocessor Driver Definition) file for the driver is used. Please refer to the chapter on The Library Generator tool (LibGen) for definitions of these parameters.

Other MDD Parameters

Other parameters corresponding to the driver assigned to this core can be set by clicking on “**MDD Params**” button. Any parameter for a driver which can be overwritten in MSS file are specified in the **MDD** file corresponding to that driver.

Processor Dialog Window

A **Processor Dialog Window** opens up when any processor instance name is double-clicked or S/W Settings menu option is chosen for that instance in the System BSP tree. This window has the following six tabs.

Processor Property

In this tab, users can specify the driver, driver version, and driver interface level for the processor. Users can also specify which peripherals are to be used as Standard Input, Standard Output, and Debug Peripheral. The Mode for a MicroBlaze instance (XMDSTUB, or EXECUTABLE) can also be specified in this tab. Please note that Debug Peripheral can not be specified for a PowerPC instance.

Environment

The tab allows users to specify compiler and archiver to be used for compiling libraries and sources for that processor. You can also specify upto what stage the compiler should be run. Currently, XPS supports only **mb-gcc** compiler for MicroBlaze. For PowerPC, XPS supports both **powerpc-eabi-gcc** and the WindRiver **dcc** compiler. However, for the dcc compiler, certain options in other tabs can not specified (see description for individual tabs).

Optimization

This tab allows you to specify various compiler options. The degree of optimization can be specified to be 1,2, or 3. For a MicroBlaze instance, the user can also specify whether to use the hardware multiplier and whether to perform Global pointer optimizations. You can also specify whether the code should be generated in debug mode or not.

Directories

This tab allows you to specify various search directories for the **Compiler** (-B), for **Libraries** (-L) and for **Include** (-I) files. You can specify what user libraries, if any, should be used by the linker in the **Libs to Link** (-l) field. The libxil.a library is automatically picked up by gcc- based compilers. For dcc, XPS automatically adds libxil.a as a library to link in the makefile compiler options. You can also specify any **Linker script** (some times called map file) to be used. Again, the gcc based compilers pick up the default linker script from the EDK installation area if this option is not specified. You can also specify the name of the **Output ELF file** to be generated by the compiler. If these paths are not absolute, they must be relative to the project directory.

Details

This tab gives you the ability to provide **Program Start Address**, **Stack Size**, and **Heap Size** for the gcc-based compilers (mb-gcc and powerpc-eabi-gcc)**.** Please note that these options should **not be used with dcc** (they should be specified in the linker script for dcc). Heap size is only for PowerPC instance.

The user can also specify various options which the compiler should pass to the **Preprocessor** (-Wp), the **Assembler** (-Wa), and the **Linker** (-Wl). Each option is dealt in detail in the GNU Compiler Tools documentation. You do not need to type in the specific flags as XPS introduces the correct flag for each option automatically. However, if you type the flags, then XPS does not introduce them. If there are more than one option in a field, they should be separated by space.

Others

For compiling program sources, if you want to specify any Compiler Options in addition to those specified in other tabs, you can specify them in the **Program Sources Compiler Options** edit box. LibGen automatically puts default compiler options to build the library libxil. If you want to override these default options used by LibGen, you can specify them in **Compiler Flags** edit box. If you want to specify any additional options for compiling the libraries, the can be specified in **Extra Compiler Flags** options. These two edit box values are put as COMPILER_OPTIONS and EXTRA_COMPILER_OPTIONS parameters in the MSS file. Please refer to the Microprocessor Software Specification chapter for more details on these parameters.

Table 2-2 shows the options that are displayed in a processor dialog window under various tabs.

Table 2-2: **Processor Options**

Option	Value Type	Description
Debug Peripheral	Instance Name	Designates the peripheral instance as the Debug Peripheral. Here the peripheral is used to download the debug stub (xmdstub)
STDIN	Instance Name	Peripheral designated as the standard input
STDOUT	Instance Name	Peripheral designated as the standard output
Flow Option	Compiler Option	Runs the compiler flow until preprocessor, compile, assemble or link stage.
Compiler Options	Optimization Level	Choose the level of compiler optimization. Equivalent to -O option in gcc.
Global Pointer Optimization	Compiler Option	This option enables global pointer optimization in the compiler. This option is only for MicroBlaze.
Hardware Multiply	Compiler Option	Enables the use of hardware multiplier on Virtex II or VirtexIIPro architecture families. This option is only for MicroBlaze.
Debug	Compiler Option	-g option to generate debug symbols.
Search Paths	Directories	Compiler, Library and Include paths. Equivalent to -B, -L and -I option to gcc.
Libraries to Link	Linker Option	The libraries to link against while building the ELF file (-l option)
Output File	File path and name	Sets the name of the executable file. Equivalent to -o option of gcc.
Program Start Address	Hex Value	Specifies the start address of the text segment of the executable for MicroBlaze and the program start address for PPC.
Stack Size	Hex Value	Specifies the stack size in bytes for the program.
Heap Size	Hex Value	Specifies the heap size in bytes for the program. Heap size can only be specified for a PPC Instance.
Pass Options	Compiler Options	Options can also be passed to the compiler, assembler and linker. The options have to be space separated.

For more information on the options, please refer to the Library Generator chapter and Microprocessor Software Specification chapter.

Source Code Management

XPS has an integrated editor for viewing and editing C source and header files of the user program. The source code is grouped for each processor instance. You can add or delete list of source code files for each processor. All the source code files for a processor are compiled using the compiler specified for that processor.

Adding Files

Files can be added to a processor by clicking the right mouse button on the Sources or Headers child of the processor instance sub-tree in the System BSP Tree Item. The same operation can be accomplished by using the **Project->Add Program Sources** menu item in the Main menu. Multiple files are added by pressing the control key and using arrow keys (or the mouse) to select in the file selection dialog. XPS adds files to Sources or Headers subtree depending upon the file extension. All directories where the header files are present are automatically added to the Include Search Path compiler option.

Deleting Files from Project

Any file can be deleted from a processor by selecting the file in the Project View window then clicking the right mouse button on the item and choosing **Delete File**. Note that the file does not get physically deleted from the disk. It is just removed from the list of files to be compiled to generate the executable for that processor instance. The same operation can be accomplished by selecting the file to be deleted in the Project View window and then using the **Project->Delete File** menu item in the Main Menu or by pressing the **Delete key**.

Editing Files

Double clicking on the source or header file in the Project View window opens the file for editing. The editor supports basic editing functions such as cut, paste, copy and search / replace. The editor highlights basic source code syntax. It also supports file management and printing functions such as saving, printing, and print previews.

Flow Tool Settings and Required Files

XPS supports tool flows as shown in Table 2-1. The Main menu has a **Options** submenu. You can set various project and tool options, as described below for each menu item.

Compiler Options

This menu opens the same dialog box as one opened by double-clicking on a processor instance name (excluding the Processor Property tab). If there is a single processor in user's system, it will automatically open the dialog box corresponding to the instance, otherwise, user will be asked which processor you want the options to be set for. User can set various compiler options in the processor dialog box which opens, as explained earlier in Processor Dialog Box section.

Project Options

Menu item **Options->Project Options** opens a dialog box which allows user to specify various project options. The same dialog can be brought up by clicking on the Project Options button in the toolbar or by double-clicking on any item in the Project Options tree in the Project View window. There are three tabs in this dialog box.

Device and Repository

The target device for the project can be changed here. There are four different items: Architecture, Device Size, Package, and Speed Grade.

Users can specify the **Search Path** directories here. However, if this option is changed, users must close the project immediately. If this option is changed here, the changes will be effective only if the project is closed and loaded again.This option corresponds to the **-lp option** of various batch tools. See LibGen and PlatGen documentation for more information.

Hierarchy and Flow

This tab allows user to specify the design hierarchy, whether the processor design being done in XPS is the top level module or if it is just a sub-module in the entire hierarchy. If this design is a sub-module, the Top Instance edit box allows you to specify the instance name used to instantiate this module in the top-level design. This corresponds to the **-iobuf** and **-ti** options of PlatGen tool.

In EDK 6.1, XPS only supports modular (hierarchical) design mode. The Flat mode is not supported.User can also choose whether to run the Xilinx Synthesis Tool (XST).

Users can also specify the flow to use for running the Xilinx implementation tools. The available options are XPS (Xflow) and ISE (Project Navigator) flow. Note that if the design is a sub-module, users must use the ISE flow. Please see the ISE Project Navigator Interface section described later for details on how to add design components and files to ProjNav project using XPS.

Simulation

This tab allows the user to specify the HDL (VHDL or Verilog) to be used by PlatGen and SimGen. Users can also specify the location of various simulation libraries. For details on simulation libraries, please refer to SimGen tool. These options are saved into the XMP file.

Required Files

If XPS (Xflow) is chosen to run the implementation tools, XPS expects a certain directory structure in the project directory. For each project, the user must provide User Constraints File (UCF). The file should reside in **data** directory in the project directory and should have the name **<proj_name>.ucf**. Users are also expected to provide an **iMPACT** script file. This file should reside in **etc** directory and should be called **download.cmd**. If these files do not exist, XPS will prompt the user to provide these files and will not run XFlow.To run Xilinx Implementation tools, XPS uses two more files, **bitgen.ut** and **fast_runtime.opt** from **etc** directory. However, if the two files are not present, XPS copies the default version of these two files into that directory from the EDK installation directory. To change options for Xilinx implementation tools, the user can modify the two files. Note that when a new project is created, if the data and etc directories do not exists, XPS creates these empty directories in the prjoect directory.

Tool Invocation

After all options for the compiler and library generator are set, the tools can be invoked from the **Run** submenu in the Main menu. The main toolbar also contains buttons to invoke these tools.

There are two different flows in the EDK platform buildig flow, the hardware flow and the software flow.

Software Flow

The software flow involves buiding up the software part of the embedded system. There are two important steps:

1. **Generate Libraries:** This button invokes the library building tool LibGen with the correct MSS file as input.
2. **Compile Program Sources:** This button invokes the compiler for each processor instance to compile corresponding program sources. It builds the executable files for each processor. If LibGen has not been executed, this button first invokes LibGen.

Hardware Flow

The hardware flow involves buiding up the hardware part of the embedded system. There are two important steps:

1. **Generate Netlist:** This button calls the platform building tool PlatGen with the correct MHS file and produces the netlist files in NGC format.
2. **Generate Bitstream:** If using XPS for implementation tools, this button calls the tool xflow with the fast_runtime.opt and bitgen.ut files residing in the etc. directory in the project directory. XFlow in turn calls the Xilinx iSE Implementation tools. If using ProjNav for the implementation flow, the button is greyed out. User must use Tools->Export to ProjNav menu to add the XPS files into ProjNav project, run the complete flow in ProjNav and then use Tools->Import from ProjNav menu to import bitstream and bmm files back into the flow.

Merging Hardware and Software Flows and Downloading

1. **Update Bitstream:** This button invokes the tool data2bram. This is the stage where the hardware and the software flows come together. This button also calls hardware and software flow tools if required. At the end of this stage, users get **download.bit** file which contains information regarding both the software and the hardware part of the design.
2. **Generate SystemACE File:** This menu item generates a SystemACE file. This option is available only when you have single powerpc in your system. This option is available only in windows platform in this release. Note that there is no toolbar button for this option.
3. **Download Bitstream:** This button downloads the download.bit file onto the target board using the Xilinx iMPACT tool in batch mode. XPS uses the file etc/download.cmd for downloading the bitstream.

XPS generates a makefile in the **__xps** directory inside project directory and calls the corresponding target. The dependencies between various tools being run is take care of by the Makefile.

When LibGen is invoked, an MSS file is created for the software specification. When the user exits the application, a prompt to save the current project appears.

ISE Project Navigator Interface

If ISE (ProjNav) is chosen for implementation flow in the Project Options dialog box, then user must specify the ProjNav project (NPL) file. ProjNav will run implementation tools in the directory where this ProjNav project file is created. Default NPL file location is **<proj_dir>/projnav/<proj_name>.npl**. It is recommended not to use **implementation** directory for ProjNav flow since XPS clean mechanism deletes this directory**.** To run the

ProjNav flow, user can create a new ProjNav project file or specify an already existing ProjNav project file.

Menu option **Tools->Export ProjNav Project** adds the required vhdl and bmm files to the ProjNav project. It also sets the ProjNav option **Macro Search Path** to **<proj_dir>/implementation** so that implementation tools can locate ngc files generated by PlatGen or XST.

Menu option **Tools->Import ProjNav Project** gives user the option to import a bitstream and a bmm file back into the XPS Project. The bit file should be the one generated by bitgen at the end of implementation tools. The bmm file should also be the one generated by bitgen, which has BRAM placement information. XPS copies the bit and bmm files into implementation directory as **<mhsbasename>.bit** and **<mhsbasename>_bd.bmm** respectively.

Debug and Simulation

Users can debug the hardware and the software part of the design either by simulation or by running it on the hardware itself. XPS provides support for invoking the corresponding tools to perform the job.

- **Xilinx Microprocessor Debug (XMD)**: Invoke the XMD tool to debug the application software. The XMD-button on the XPS toolbar opens up a XMD shell in the project directory.
- **Software Debugger:** The debug button invokes the software debugger corresponding to the compiler being used for the processor. If there are more than one processor in the design, XPS prompts to choose the processor whose program sources the user wants to debug.
- **Hardware Simulation Model Generator (SimGen):** Invoke the SimGen tool to generate various simulation models for the components instantiated in MHS File. Depending on the simulation model to be used (Behavioral, Structural or Timing), XPS calls SimGen with appropriate options to generate the simulation models and initialize memory. Then XPS compiles those models for ModelTech's ModelSim simulator and starts the simulator with the compiled files.

PBD Editor

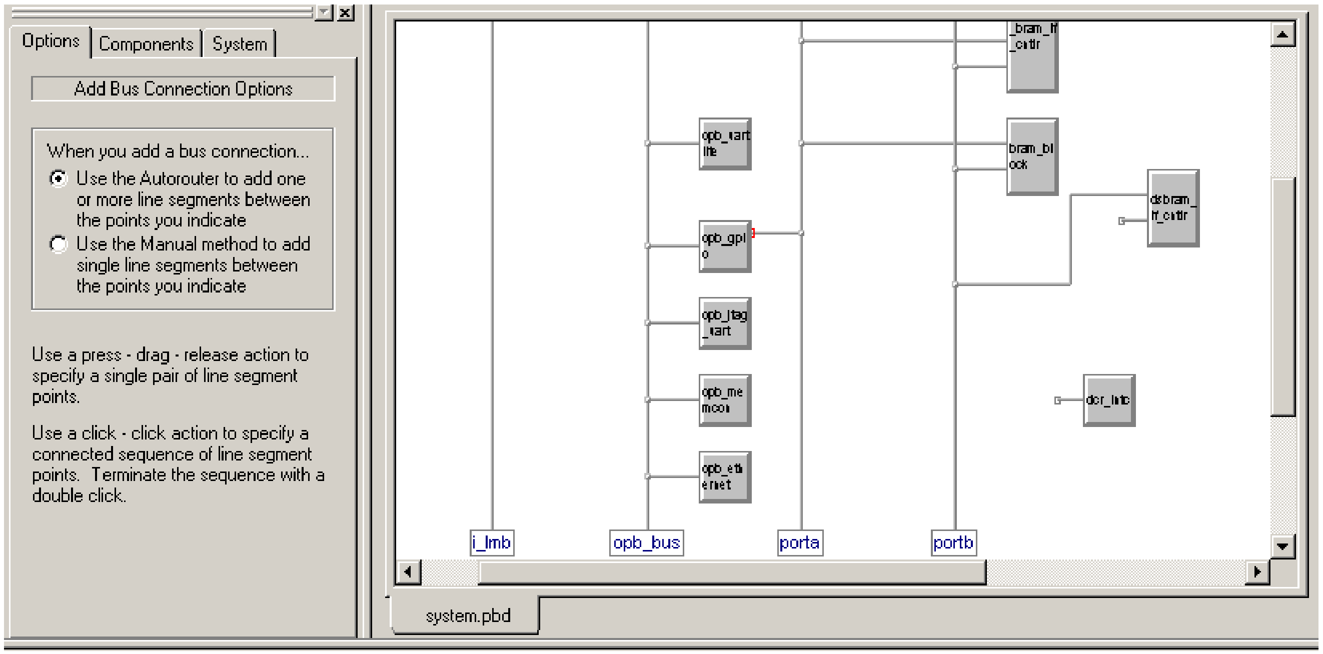
The Processor Block Diagram Editor (PBD Editor) allows you to read, create, modify and save a description of an FPGA Platform that references Hardware (HW) components. The HW components comprise, in part, microprocessors, buses and bus arbiters, and peripheral devices.

The PBD Editor block diagram supplies the hardware platform information written into the MHS file.

PBD Editor Interface

The PBD Editor interface is shown in Figure 2-4. These areas comprise the interface:

- The workspace
- The system tabs



Options | Components | System

Add Bus Connection Options

When you add a bus connection...

[✓ Use the Autorouter to add one or more line segments between the points you indicate]

Use the Manual method to add single line segments between the points you indicate

Use a press - drag - release action to specify a single pair of line segment points.

Use a click - click action to specify a connected sequence of line segment points. Terminate the sequence with a double click.

Figure 2-3: **The PBD Editor**

PBD Editor Workspace

The PBD Editor workspace is the upper right window in the XPS (see Figure 2-4). The workspace contains the block diagram describing the system hardware.

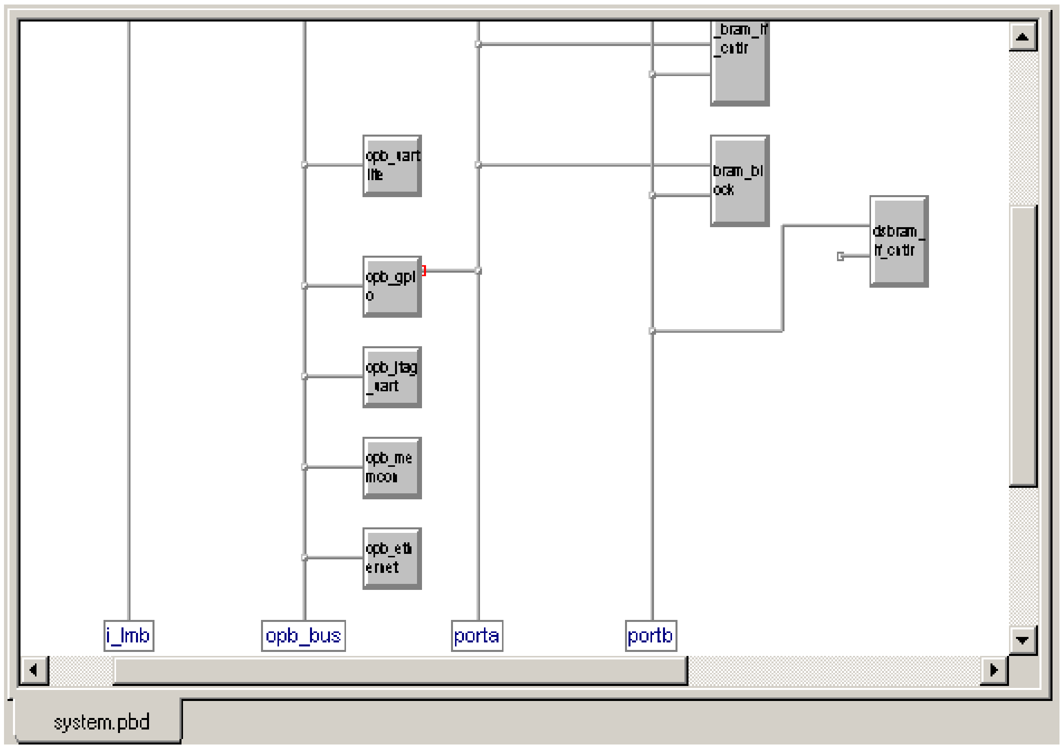


Figure 2-4: **PBD Editor Workspace**

System Tabs

The system tabs are in the upper left of the XPS window (see Figure 2-5). Two of the tabs in the window are used in the PBD Editor operation.

- The Options tab changes according to the tool that you are using and allows you to set options related to the tool, such as how the Add Bus Connection tool should operate.
- The Components tab allows you to select a component (a CPU, Bus Infrastructure component, or peripheral) to instantiate into your system. The components are Xilinx cores.

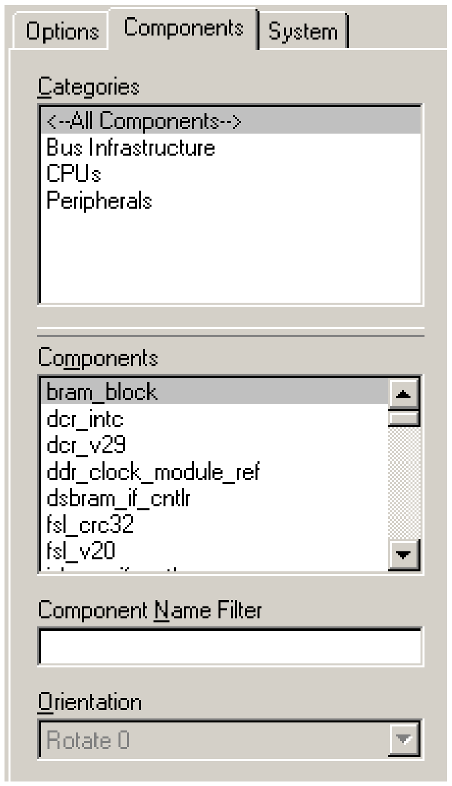


Figure 2-5: **System Tabs**

Creating the Hardware Block Diagram

The following procedures are used to create the hardware platform in the PBD Editor.

Adding a Component Instance to the System

Component instances are Xilinx cores (IP) instantiated in the hardware design. The components you add to the system may be:

- CPUs
- Bus components
- Peripherals

To add a component instance to the system:

1. Select the ***project_name*.pbd** tab in the workspace to display the system block diagram.

2. Select **Add** → **Component** or click the Add Component toolbar button.

3. In the **Components** tab, use the **Categories** and **Components** lists to specify the component you are adding.

   The component you select is attached to the mouse cursor.

   Note: To make the component selection easier, type the first letter or letters of the component in the **Component Name Filter** field. The **Components** list box shows only the components that begin with those letters. A regular expression can also be used to filter components. For eg. typing ".*uart" will list all components with "uart" in the name. A "." stands for a character and "*" means "zero or more".

4. Click where you want the component instance to appear in the workspace.

Component instance notes:

- The PBD Editor assigns the new component instance the default name *corename_number*. The *number* is incremented each time another instance is added.
- To rename a component instance, see , "Naming an Instance".
- If a bus pin on the component symbol touches a bus, and if the pin is compatible with the bus type, the symbol pin is connected to the bus when the component instance is placed in the block diagram.

Naming an Instance

When you add a component to the system, the PBD Editor assigns the new component instance the default name *corename_number*, and the *number* is incremented each time another instance is added. You can leave the machine-generated names as is. However, it is usually easier to debug the design using your own names.

To rename an instance.

1. Double-click the instance in the workspace.
2. In the Object Properties dialog box, change the **Instance Name**.

Setting Component Instance Parameters

You set parameters to customize the instantiated IP for your design. Parameters may be set for CPUs, bus components, or peripherals. The properties you set depend on the type of component and the IP (core) from which the component was instantiated.

IP parameters are described in the data sheets for the cores instantiated in the design. Data sheets can be accessed from the Xilinx IP Center page at http://www.xilinx.com/ipcenter.

To set parameters for a customizable component instance:

1. Double-click the component instance in the workspace.
2. In the Properties dialog box, click the **Parameters** entry in the tree view on the left side of the dialog box.
3. To override a value displayed in the **Default Parameter Values** table:
   a. Select the parameterin the **Default Parameter Values** table.
   b. Clicking **Add**.
   c. Change the parameter **Value** in the **Explicit Parameter Values** table.

d. Click **Apply**.

The value entered in the **Explicit Parameter Values** table overrides the value displayed in the **Default Parameter Values** table.

Setting Symbol Properties

Symbol properties determine the appearance of an instance's block in the workspace. You can modify the size of the symbol drawing or the location of the bus pins on the symbol.

Some components (the MicroBlaze processor, for example) have a large number of bus interfaces, only a few of which may be used in the block diagram. You can hide the bus interface pins that are not in use, thus reducing the size of the symbol and making the diagram easier to read.

To set symbol properties:

1. Double-click component instance in the workspace.
2. In the Properties dialog box, click the **Symbol** entry in the tree view on the left side of the dialog box.
3. To change the size of the symbol:
   a. Enter a value in the **Min Width** and / or **Min Height** fields.
   b. Click **Add**.
4. To change the orientation (top, bottom, left, or right) of a symbol pin:
   a. Select the pin in the **Available Pins** table.
   b. Click **Add**.
   c. At the top of the **Pins on Symbol** area, select the orientation you want (**Top**, **Bottom**, **Left**, or **Right**).
   d. Click **Apply**.

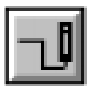
The symbol in the workspace is updated to reflect the change.

Connecting a Component Bus Pin to a Bus

When you connect a component bus pin to a compatible bus, connection lines are drawn from the pin to show the bus connection. All of the signals represented by the bus pin are connected to the bus.

To connect a component bus pin to a bus:

1. Select **Add** → **Bus Connection** or click the Add Bus Connection toolbar button.



2. Select the bus pin on the component instance you wish to connect to the bus.

   To select the pin, move the cursor near the end of the pin until four squares appear to help you locate the exact point. When the cursor is in the correct position to select the pin, a box appears with information about the component instance and the type of pin you are selecting.
3. Click anywhere on the bus to which you will connect the pin.

   If the type of bus is compatible with the type of pin, connection lines are drawn to show the bus connection.

Connecting Ports

You can create nets to connect ports on component instances. To create a net, you assign the same net name to all of the ports you want to connect.

Port connections **cannot** be seen as nets drawn on the block diagram. All of the nets shown on the block diagram are bus connections.

To connect ports on two component instances:

Note: This procedure describes how to connect a port on one component instance to a port on another component instance. Using a similar procedure, you can connect ports on more than two component instances, connect multiple ports at the same time, or create system ports.

1. Double-click one of the component instances you want to connect.
2. In the Properties dialog box, click the **Ports** entry in the tree view on the left side of the dialog box.
3. In the box under **Show Ports**, choose the type of ports appearing in the ports list (**With No Default Nets**, **With Default Nets**, **All Ports**, or **New Filter**).
4. Note that ports With Default Nets need not be connected, they will be automatically connected by PlatGen. The user needs to connect these ports only when the connection is not desired.
5. In the **Show Ports** list, select the a port to which you will assign a net.
6. Click **Add**.

   The selected port is copied to the **Explicit Port Assignments** list.
7. In the **Explicit Port Assignments** list, modify the fields describing the port connection (**Polarity**, **Range**, etc.) and assign the net connected to the port a **Net Name**.
8. Perform Steps 1 through 6 for the second component instance. If you assign the same **Net Name** to a port on each component instance, the ports are connected.

Viewing and Editing System Ports

You can view and edit the all of the system ports (that is, all of the ports designated **External**) in a single dialog box. Using this dialog box, you can also add power and ground ports to the system.

To view and edit system ports:

1. Double-click an area in the workspace that does not contain any objects.
2. If you want to add power or ground system ports to the design:
   a. Click Add.
   b. In the Add External Port dialog box, enter a **Port Name** and select **GND (net_gnd)** or **VCC (net_vcc)**.
   c. In the Add External Port dialog box, Click **OK**.
3. Edit the entries in the **System Ports** table as desired.

   Some notes about the table:

   ♦ Fields that the you can edit are displayed in white; read-only fields are displayed in grey.

   ♦ If you click the heading of a column, the entries in the column are displayed in alphabetical order. If the click the column heading again, the entries in the column are displayed in reverse alphabetical order.

- You can remove a system port by selecting it and clicking **Remove**.

4. When you have finished your edits, click **OK**.

Viewing and Editing All of the Ports in the System

You can view and edit the all of the ports in the system (internal and external) in a single dialog box. Using this dialog box, you can also print a port list or export the ports as a CSV (Comma Separated Value) file formatted for the PBD Editor or for the Xilinx PACE (Pinout and Area Constraints Editor) tool.

To view and edit all of the ports in the system:

1. Select **Add** → **Ports** or click the Add Ports toolbar button.
2. If you want to print the **System Ports** table, click **Print**.
3. If you want to export the ports to a CSV file:
   a. If you only want to export selected ports, select the ports to export.
   b. Click Export.
   c. In the Export Ports dialog box, enter a **CSV File Name**, select an **Output Format** of **PBD Editor** or **PACE**, and specify whether you want to export **All Ports** or **Selected Ports**.
   d. In the Export Ports dialog box, Click **OK**.
4. Edit the entries in the **System and Component Ports** table as desired.

   Some notes about the table:

   - Fields that the you can edit are displayed in white; read-only fields are displayed in grey.
   - If you click the heading of a column, the entries in the column are displayed in alphabetical order. If the click the column heading again, the entries in the column are displayed in reverse alphabetical order.
5. When you have finished your edits, click **OK**.

Viewing and Editing Interrupts

You can view and edit the interrupts driving a component. Not all components have interrupt ports, and most components that use interrupts have only one interrupt port.

An interrupt may be driven by more than one net. If an interrupt is driven by multiple nets, you must specify the priority of each net driving the interrupt.

To edit the interrupts driving a component instance:

1. Double-click the component instance in the workspace.
2. In the Properties dialog box, click the **Interrupts** entry in the tree view on the left side of the dialog box.
3. In the Component Interrupts dialog box, select the Interrupt you wish to configure in the **Interrupt Port** box.
4. In the **Possible Interrupt Nets** box, select the nets that will drive the internet.

To select multiple nets, click the first net name, then press the `Ctrl` key and click the additional net names.

Note: If the interrupt port is a scalar port (that is, its range is blank) then only one net may be selected to drive the interrupt. An interrupt controller must be used in such a case to manage the interrupts, and the controller's output port should be used as the single input to the component with the scaler interrupt port.

5. Click **Add** to move the nets to the **Interrupt Drivers** box.
6. In the **Interrupt Drivers** box, use the **Move Up** and **Move Down** buttons to list the nets in priority order.

   Nets higher in the list will be serviced before nets lower in the list.
7. Click **OK**.

Editing the Block Diagram

Selecting Objects

To Select objects in the workspace:

1. Select **Edit** → **Select Object(s)**, or click the Select toolbar button.

   The **Options** tab shows the **Select Options**.
2. In the **Options** tab, set the following options:
   - Click **Select the entire bus** or **Select the line segment** to specify whether the bus or just the line is selected when you click a bus line.
   - Click **Keep the connections to other objects** or **Break the connections to other objects** to specify whether connections to other objects are retained when you move an object.
   - Click **Are enclosed by the area** or **Intersect the area** to specify which objects to select when you drag a bounding box around an area. **Are enclosed by the area** selects only those object that are completely enclosed in the bounding box.
3. Click the object to select it.

The PBD Editor also has these extended selections:

- If you hold the **Shift** key while you select an object, it is added to the current selections
- If you hold the **Ctrl** key while you select an object, its status is toggled (that is, it will be selected if it was not selected and deselected if it was selected).
- **Edit** → **Select** All selects all objects on the current sheet.
- **Edit** → **Unselect All** unselects all objects on the current sheet.

Viewing Object Information

To view information about an object in the workspace, place the cursor over the object. A box appears supplying information about the object (name, IP name, bus pin type, etc.).

Zooming in the Workspace

You can use menu commands to zoom the display in the workspace.

Zooming Behavior	Menu Command	Toolbar Icon
Zoom in	Select **View** → **Zoom** → **In**, or click the Zoom In toolbar button.	
Zoom out	Select **View** → **Zoom** → **Out**, or click the Zoom Out toolbar button.	
Zoom to display the entire schematic or symbol in the workspace	Select **View** → **Zoom** → **Full View**, or click the Zoom Full View toolbar button.	
Zoom to an area you select	Select **View** → **Zoom** → **To Box**, or click the Zoom To Box toolbar button. Zoom in or out as follows: • To zoom in, draw a bounding box around the area from the top left corner of the area to the bottom right corner. • To zoom out, draw a bounding box from the bottom right corner to the top left corner.	
Zoom to display selected objects at the highest magnification	1. Select the objects you want to center in the workspace. 2. Select **View** → **Zoom** → **To Selected**, or click the Zoom To Selected toolbar button	

Drawing Non-Electrical Objects

Non-Electrical Objects are graphic only and have no electrical meaning in the block diagram. You can draw these non-electrical objects in the PBD Editor:

- Arcs
- Circles
- Lines
- Rectangles
- Text

To draw a non-electrical object:

1. In the **Add** menu, select the object (**Arc**, **Circle**, **Line**, **Rectangle**, or **Text**) you want to draw, or select the toolbar icon for the object.

Object	Toolbar Icon
Arc	◝
Circle	○
Line	╲
Rectangle	□
Text	A

2. If any options appear in the Options tab, select the appropriate options for the object.
3. Click to start drawing the object.
4. Drag the cursor until the object is the appropriate size.
5. If necessary, move the cursor to adjust the object.

   For example, when you draw an arc you must move the cursor until the arc appears as you want it to display.

You can draw as many objects as you want until you select another command.

XPS “No Window” Mode

XPS “no window” mode can be invoked by typing the command **xps -nw** at the command prompt. It provides limited functionality to generate MSS file. It also provides mechanism to generate makefile. Users can also create an XMP project file or load an XMP project file created by the XPS GUI.

When invoking the batch mode for XPS, users can specify a tcl script along with **-scr** option. XPS sources this Tcl script and then provides a command prompt to the user. Users can also provide an existing project (XMP) file as input to xps. XPS will load the project before presenting the command prompt to the user.

Available Commands

XPS-Batch provides you a Tcl shell interface. You can use the commands in Table 2-3.

Table 2-3: **XPS-Batch commands**

Command	Description
load [mhs|xmp|new|mss|] <filename>	Loads the MHS/XMP file and opens/creates XPS project. Updates project with MSS file. Input <filename> is optional when loading MSS. Users can create an empty project with suboption new
save [mss|xmp|make|proj]	Saves the corresponding file. Option proj will save all files
xset option [value]	This command sets the value of a field (corresponding to option) to the given value. Refer to Section "Setting Project Options".
xget option	This command displays the current value of the field (corresponding to option). Refer to Section "Setting Project Options".
run option	Executes makefile with appropriate target. Refer to Section "Executing Flow Commands"
exit	Closes the project and exits out the XPS

Creating A New Empty Project

For creating a new project with no components, use the command

load new <basename>.xmp.

XPS will create a project with an empty MHS file and will also create the corresponding MSS file. All the files have same basename as the xmp file. If XPS finds an existing project in the directory with same basename, then the XMP file is overwritten. However, if MHS,or MSS file with same name is found, then they are read in as part of the new project.

Creating A New Project With Given MHS

For creating a new project, use the command

load mhs <basename>.mhs.

XPS will read in the MHS file and create the new project. The project name will be same as MHS basename. All the files generated will have the same name as MHS.

After reading in the MHS file, XPS will also assign various default drivers to each of the peripheral instance, if a driver is known and available to XPS.

Opening An Existing Project

If you already have a XMP project file, you can load that file using command

load xmp <basename>.xmp.

XPS will read in the XMP file and load the project. Project name will be same as XMP basename. Note that XPS will take the name of MSS file from the XMP file, if specified. Otherwise, it will assume these files based on the XMP file name. If XMP file does not refer to an MSS file, but the file exists in the projct directory, XPS will read that MSS file. If the file does not exist, then XPS will create a new MSS file.

Reading MSS File

You can read in a MSS file using command

load mss <filename>.

Note that if user does not specify <filename>, it is assumed to be the file associated with this project. Loading an MSS file will override any earlier settings. For example, if you specify a new driver for a peripheral instance in the MSS file, the old driver for that peripheral will be over ridden.

Saving Files and Project

Users can save MSS, XMP and make files for your project using the command

save [mss | xmp | make | proj].

Command **save proj** will save all the files.

Setting Project Options

Users can set various project options and other fields in XPS using the xset command. Users can also display the current value of those fields by using xget commands. The various options taken by the two commands are shown in Table 2-4.

xset option [value]

xget option

Table 2-4: **Options for command xset and xget**

arch	Set target device architecture
dev	Set target part name
package	Set package of the target device
speedgrade	Set speedgrade of the target device
searchpath [dirs]	Set the Search Path as semicolon separated list of directories
hier [top|sub]	Set the design hierarchy
topinst [instname]	Set the name by which processor design in instantiated (if submodule)
hdl [vhdl|verilog]	Set HDL language to be used
sim_model [structural|behavioral |timing]	Set current simulation mode
simulator [mti|none]	Set simulator for which you want simulation scripts generated
sim_x_lib sim_edk_lib	Set the simulation library paths. For details, please refer to SimGen chapter
pnproj [nplfile]	Set the ProjNav Project file where design will be exported

Table 2-4: **Options for command xset and xget**

addtonpl	If NPL file exists, specify whether XPS should add to that file or should overwrite it
synproj	Set the synthesis tool to be *xst* or *none*
intstyle	Set instyle value
usercmd1	Set user command 1
usecmd2	Set user command 2
pn_import_bit_file	Set the bit file to be imported from ProjNav
pn_import_bmm_file	Set the bmm file to be imported from ProjNav
reload_pbde	Set GUI option to reload PBDE or recreate everytime
main_mhs_editor	Set GUI option about main_mhs_editor

Executing Flow Commands

Users can run various flow tools by using the run command with appropriate option. XPS will create a makefile for the project and run that makefile with appropriate target. Note that XPS generates the makefile everytime run command is executed. Valid options for run command are shown in Table 2-5.

run option

Table 2-5: **Options for command run**

netlist	Generate netlist
bits	Run Xilinx Implementation tools flow and generate bitstream
libs	Generate software libraries
bsp	Generate VxWorks bsp for given ppc405 system
program	Compile user program into ELF file(s)
init_bram	Update bitstream with BRAM initialization information
ace	Generate SystemACE file after .bit file is updated with BRAM info
simmodel	Generate simulation models (does not run simulator)
sim	Generate simulation models and run simulator
download	Download bitstream onto the FPGA
exporttopn	Export the processor design to ProjNav
importfrompn	Import .bit and .bmm files from ProjNav
netlistclean	Delete ngc/edn netlist
bitsclean	Delete .bit, .ncd, and .bmm files in implementation directory
hwclean	Delete implementation directory
libsclean	Delete software libraries
programclean	Delete ELF file(s)
swclean	Calls libsclean and programclean
simclean	Delete simulation directory

Table 2-5: **Options for command run**

clean	Delete all tool generated files and directories
resync	Updates any MHS file changes into the memory
assign_default_ drivers	Assigns Default drivers to all peripherals in the MHS file and saves to MSS file.

Closing A Project and Exiting

For closing the project, you can use the command

exit.

This will also save the project and close XPS. Thus, you can only work on a single project during a single execution of the batch mode version of XPS.

Limitations And Workarounds

MSS Changes

XPS-batch supports limited MSS editing. So, if user wants to make any changes in these files, he/she will have to hand-edit the file, make the changes and load it in to XPS. Note that user does not have to close the project. S/he can save the MSS file, edit it and then just re-load it into the project by using load mss command.

XMP Changes

It is not recommended to change the XMP file by hand. XPS-batch supports changing of project options through commands. However, it does not support adding of source and header files to a processor, and setting any compiler options. Users must do such changes from the XPS GUI.

Chapter 3

Base System Builder

Overview

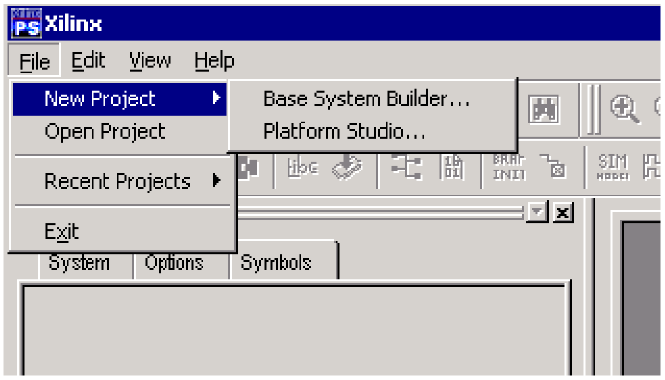
The Base System Builder (BSB) is a software tool that helps the user quickly build a working hardware system for a specific development board. For a list of board currently supported by BSB, user only needs to input minimum information, while BSB automatically fills in the rest based on a board description file and intelligent defaults. After it is done, the user can go back to XPS and further modify and enhance the system.

BSB Flow

The user can start BSB when he creates a new project in XPS.

Invoke BSB

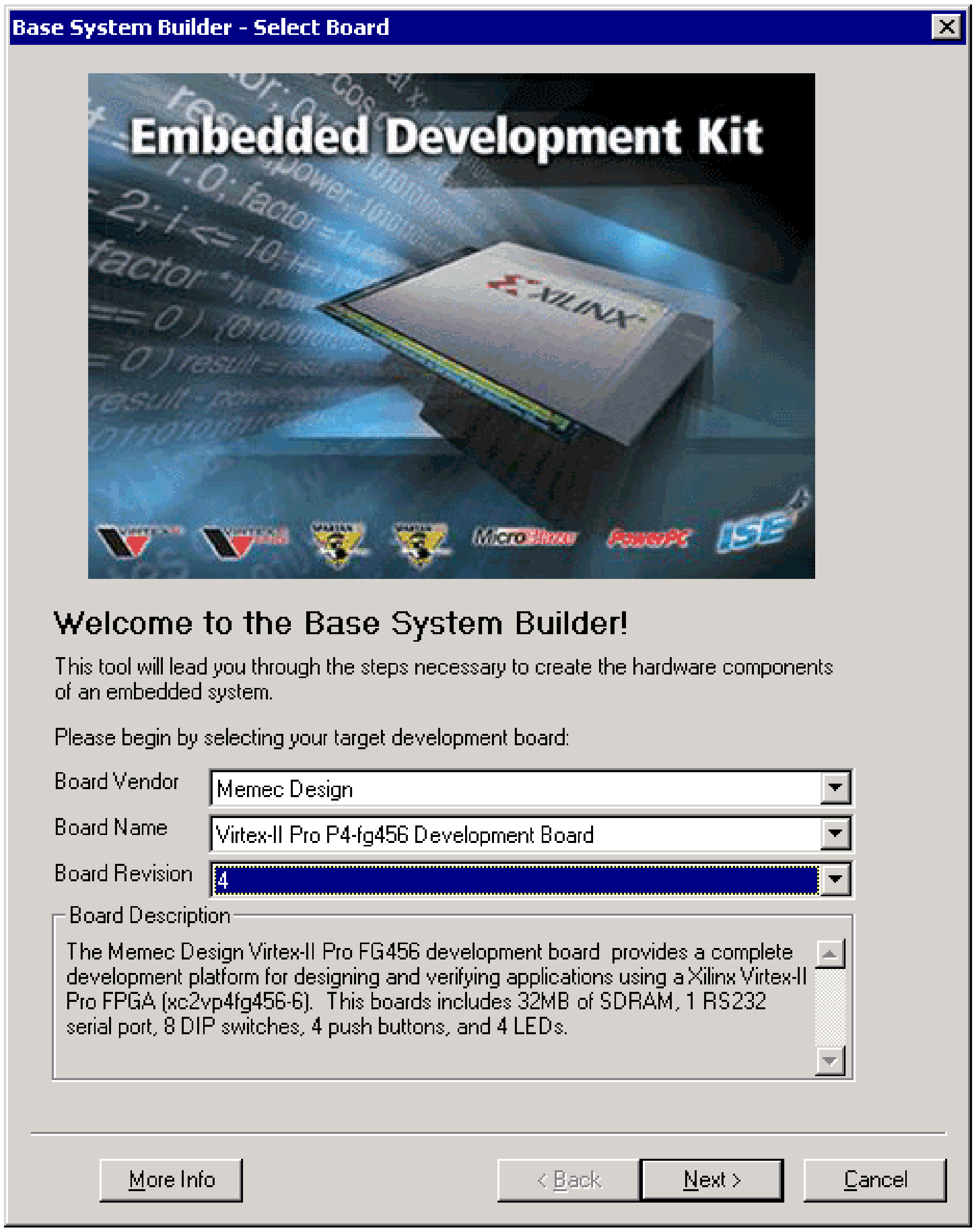
When the user creates a new project, he may use BSB to create a base system as a good starting point, or directly go to XPS with an empty new project. Select *Base System Builder* to start BSB.



Select a target board

First the user selects a development board by choosing vendor, board name and revision. A brief description of the selected board shows the Xilinx FPGA device and a list of standard IO devices on the board.

BSB only recoganizes a list of boards supported by Xilinx and third-party vendors. For the present, if the user uses a board that is yet to be supported, he may select the board that provides similar features. Also he needs to modify data/system.ucf created by BSB to reflect the pin locations of his own board.



Welcome to the Base System Builder!

This tool will lead you through the steps necessary to create the hardware components of an embedded system.

Please begin by selecting your target development board:

Board Vendor	Memec Design
Board Name	Virtex-II Pro P4-fg456 Development Board
Board Revision	4

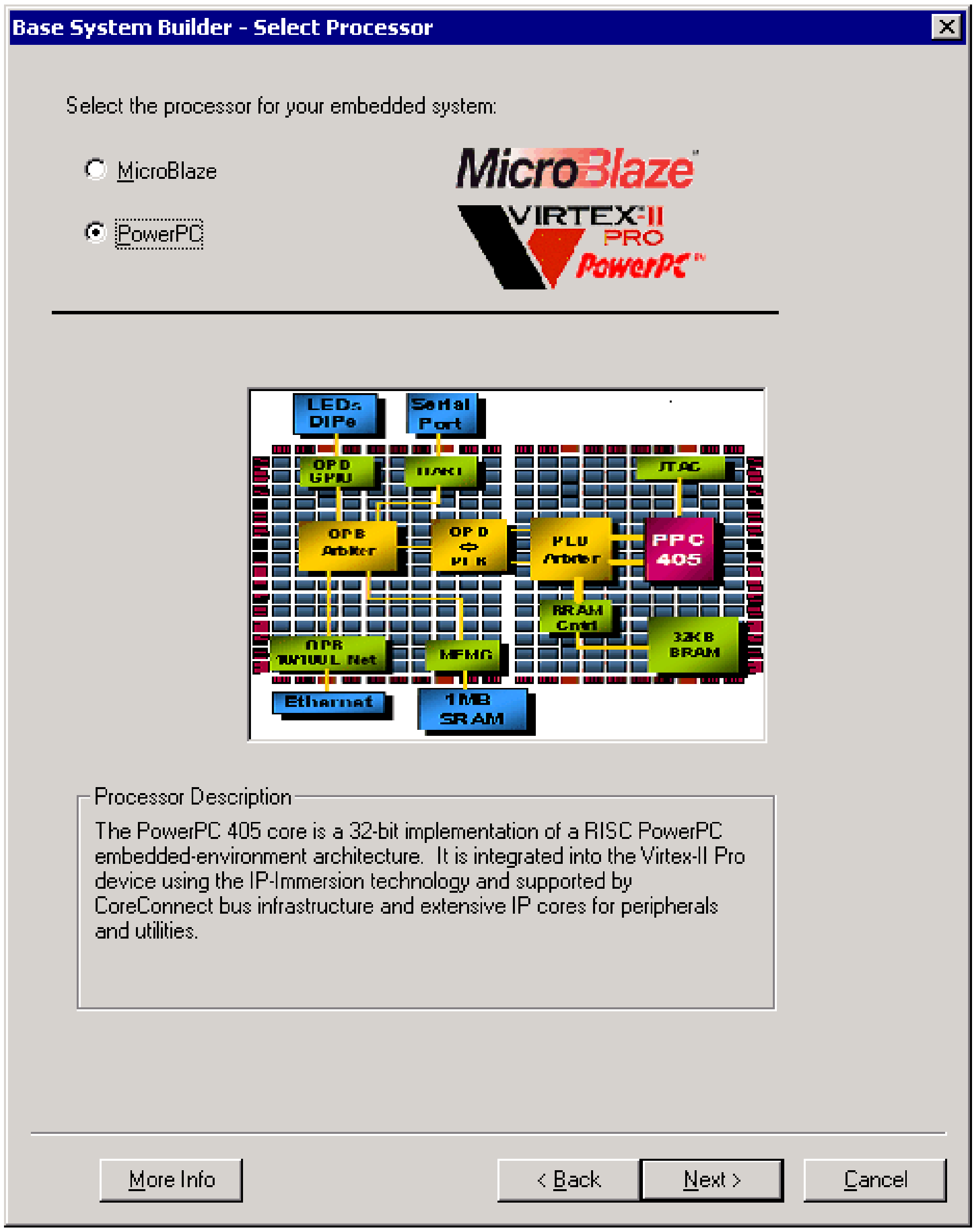
Board Description

The Memec Design Virtex-II Pro FG456 development board provides a complete development platform for designing and verifying applications using a Xilinx Virtex-II Pro FPGA (xc2vp4fg456-6). This boards includes 32MB of SDRAM, 1 RS232 serial port, 8 DIP switches, 4 push buttons, and 4 LEDs.

Select processor

Then the user selects either Microblaze soft processor or PowerPC 405 hard processor. If the target board is not a VirtexII-Pro board, PowerPC is disabled and the user may only select Microblaze.

BSB also shows a typical system diagram and a brief description of the processor based on user's selection.



Select the processor for your embedded system:

( ) MicroBlaze

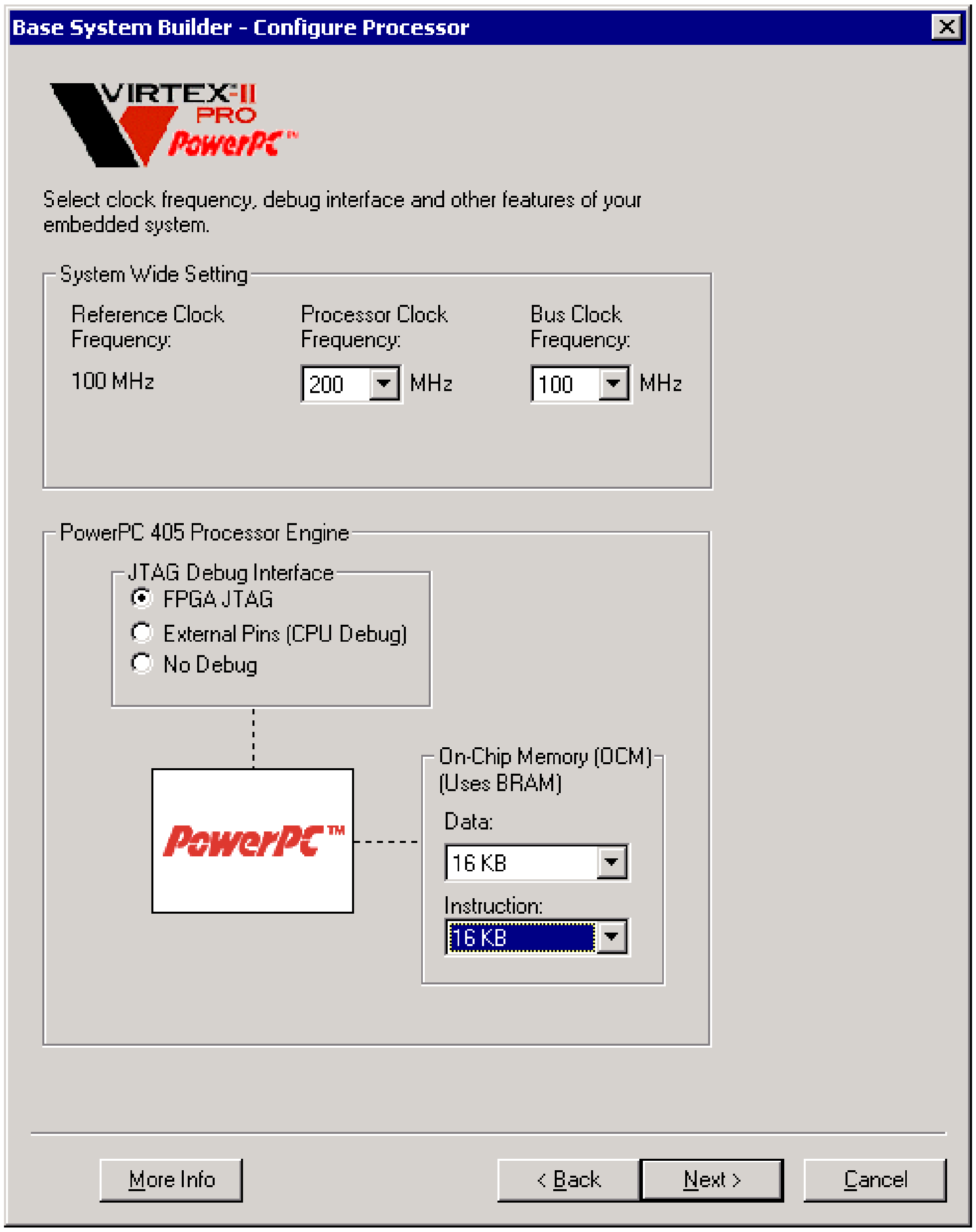
(•) PowerPC

Processor Description

The PowerPC 405 core is a 32-bit implementation of a RISC PowerPC embedded-environment architecture. It is integrated into the Virtex-II Pro device using the IP-Immersion technology and supported by CoreConnect bus infrastructure and extensive IP cores for peripherals and utilities.

Configure Processor System

In this page, the user can configure the processor system that he just selected, including operating clock frequency, debug interface, on-chip memory configuration. Please click More Info button to see detailed explanation on each feature.


Base System Builder - Configure Processor

Select clock frequency, debug interface and other features of your embedded system.

System Wide Setting

Reference Clock Frequency: 100 MHz

Processor Clock Frequency: 200 MHz

Bus Clock Frequency: 100 MHz

PowerPC 405 Processor Engine

JTAG Debug Interface

[✓ FPGA JTAG] External Pins (CPU Debug) No Debug

On-Chip Memory (OCM) (Uses BRAM)

Data: 16 KB

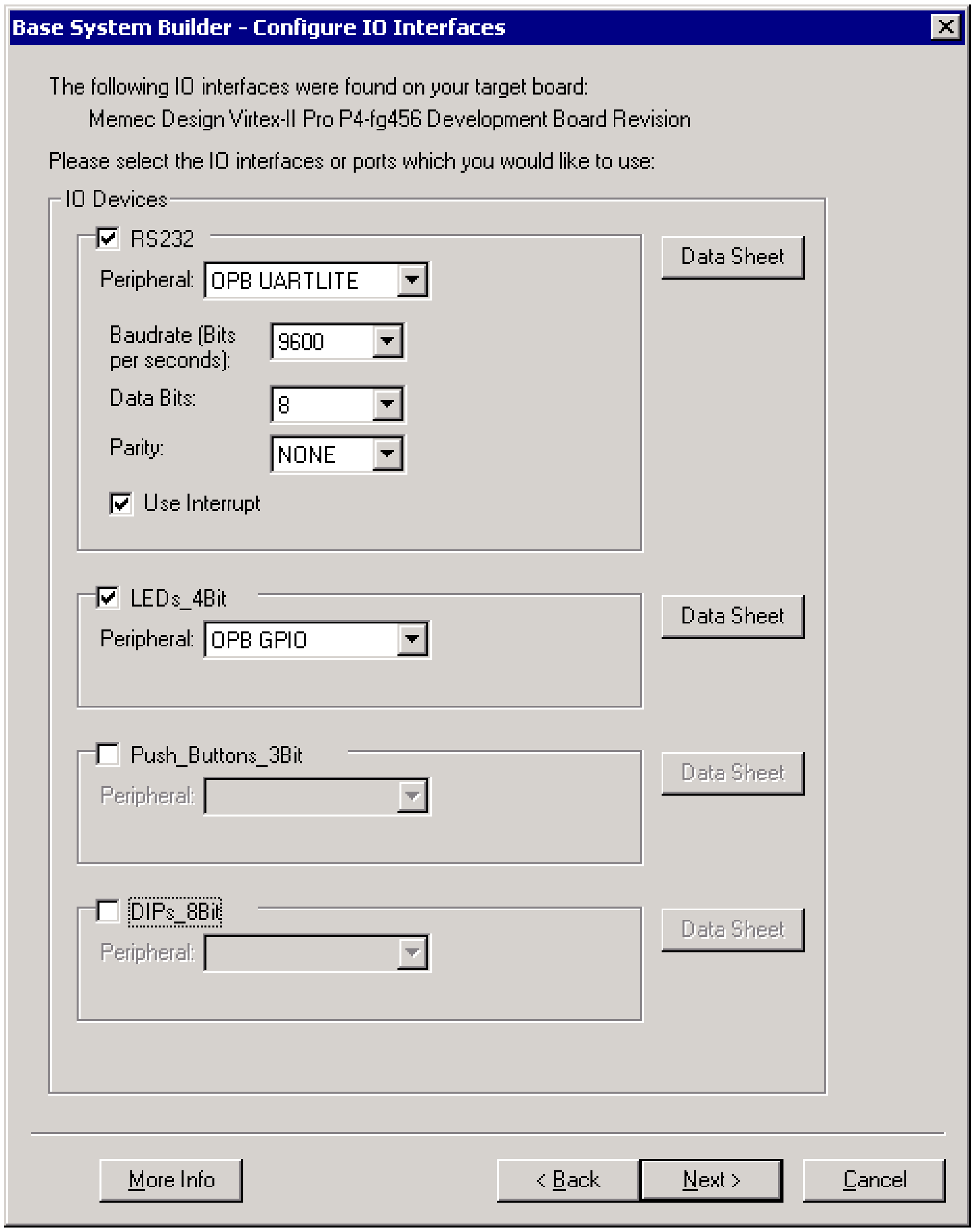
Instruction: 16 KB

Select IO interfaces

In this page, or subsequent pages if there are too many IO interfaces to fit in one page, the user selects and configure the IO interfaces to other devices connected to Xilinx FPGA.

The user can deselect the ones he does not want to be included in his system. For those interfaces in the system, he then selects the Xilinx cores that communicates with corresponding IO devices. For each selected core, the number of parameters is kept minimum to ensure ease-of-use by using fixed settings from the board description file and intelligent defaults. The user may change and override any setting in XPS once BSB is done.

To learn more about the core, click "Data Sheet" button to view the core specification.



Base System Builder - Configure IO Interfaces

The following IO interfaces were found on your target board:
Memec Design Virtex-II Pro P4-fg456 Development Board Revision

Please select the IO interfaces or ports which you would like to use:

IO Devices

[✓] RS232
Peripheral: OPB UARTLITE
Baudrate (Bits per seconds): 9600
Data Bits: 8
Parity: NONE
[✓] Use Interrupt

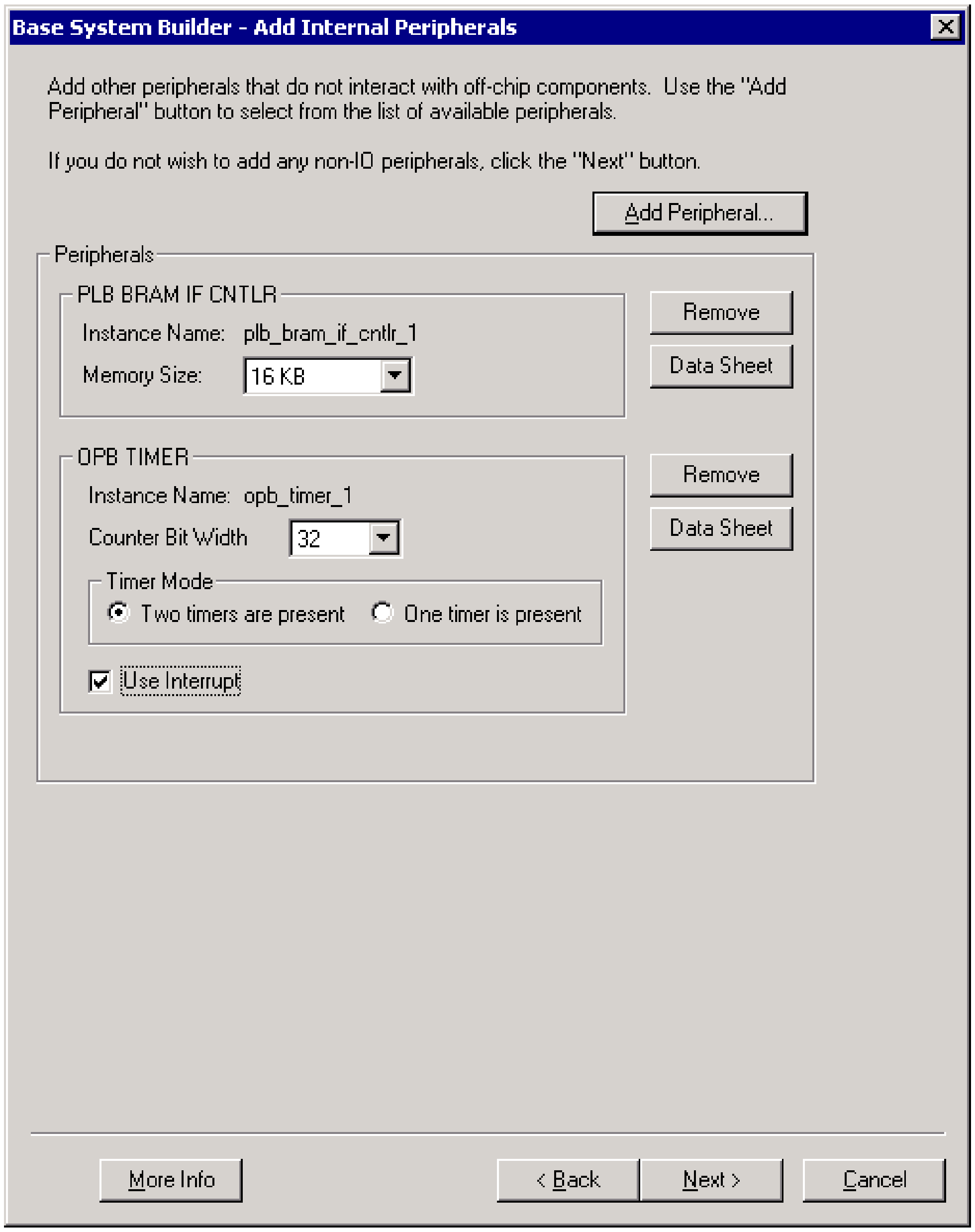
[✓] LEDs_4Bit
Peripheral: OPB GPIO

[ ] Push_Buttons_3Bit
Peripheral:

[ ] DIPs_8Bit
Peripheral:

Add internal peripherals

In this page, the user may add internal peripherals that do not directly communicate with IO devices on the board. Such peripherals include BRAM controller, timer, etc.



Add other peripherals that do not interact with off-chip components. Use the "Add Peripheral" button to select from the list of available peripherals.

If you do not wish to add any non-IO peripherals, click the "Next" button.

Peripherals

PLB BRAM IF CNTLR

Instance Name: plb_bram_if_cntlr_1

Memory Size: 16 KB

OPB TIMER

Instance Name: opb_timer_1

Counter Bit Width: 32

Timer Mode: [✓ Two timers are present] One timer is present

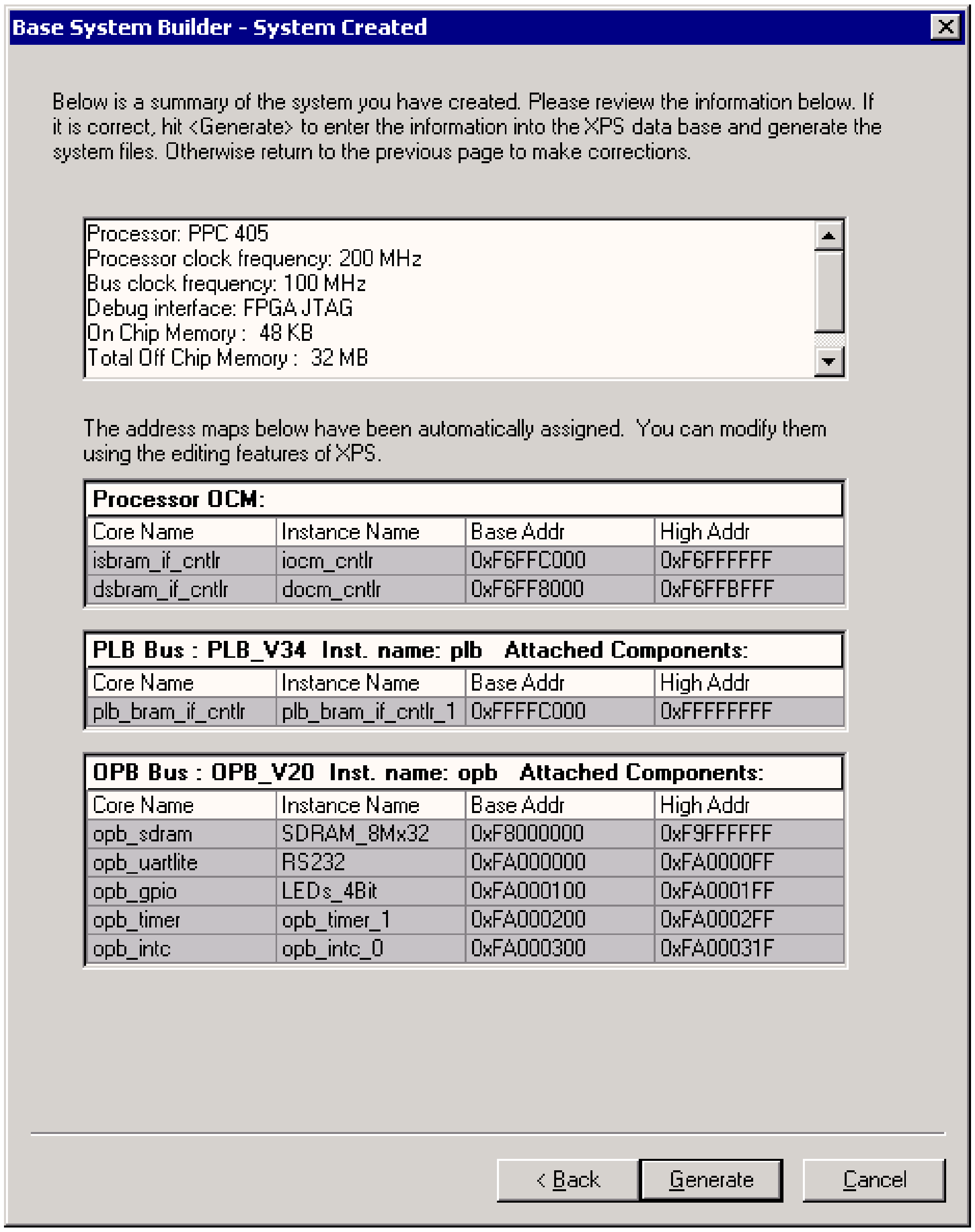
[✓] Use Interrupt

Generate system

Upon completion, BSB generates a memory map for the entire system and the following files under user project directory:

- system.mhs: Hardware description file for EDK
- data/system.ucf: Xilinx user constraint file which contains constraints such as timing, pin location, pull up/down and IO standard.
- etc/fast_runtime.opt: Xilinx implementation setup file for EDK.
- etc/download.cmd: Xilinx download command file, in which BSB uses command *identify* to automatically initialize BSCAN chain. The user must install all necessary

device data files for the board. Or he may modify download.cmd to explicitly add devices and load data files, i.e. BSDL file, bit file, etc.



Below is a summary of the system you have created. Please review the information below. If it is correct, hit <Generate> to enter the information into the XPS data base and generate the system files. Otherwise return to the previous page to make corrections.

```
Processor: PPC 405
Processor clock frequency: 200 MHz
Bus clock frequency: 100 MHz
Debug interface: FPGA JTAG
On Chip Memory :  48 KB
Total Off Chip Memory :  32 MB
```

The address maps below have been automatically assigned. You can modify them using the editing features of XPS.

Processor OCM:

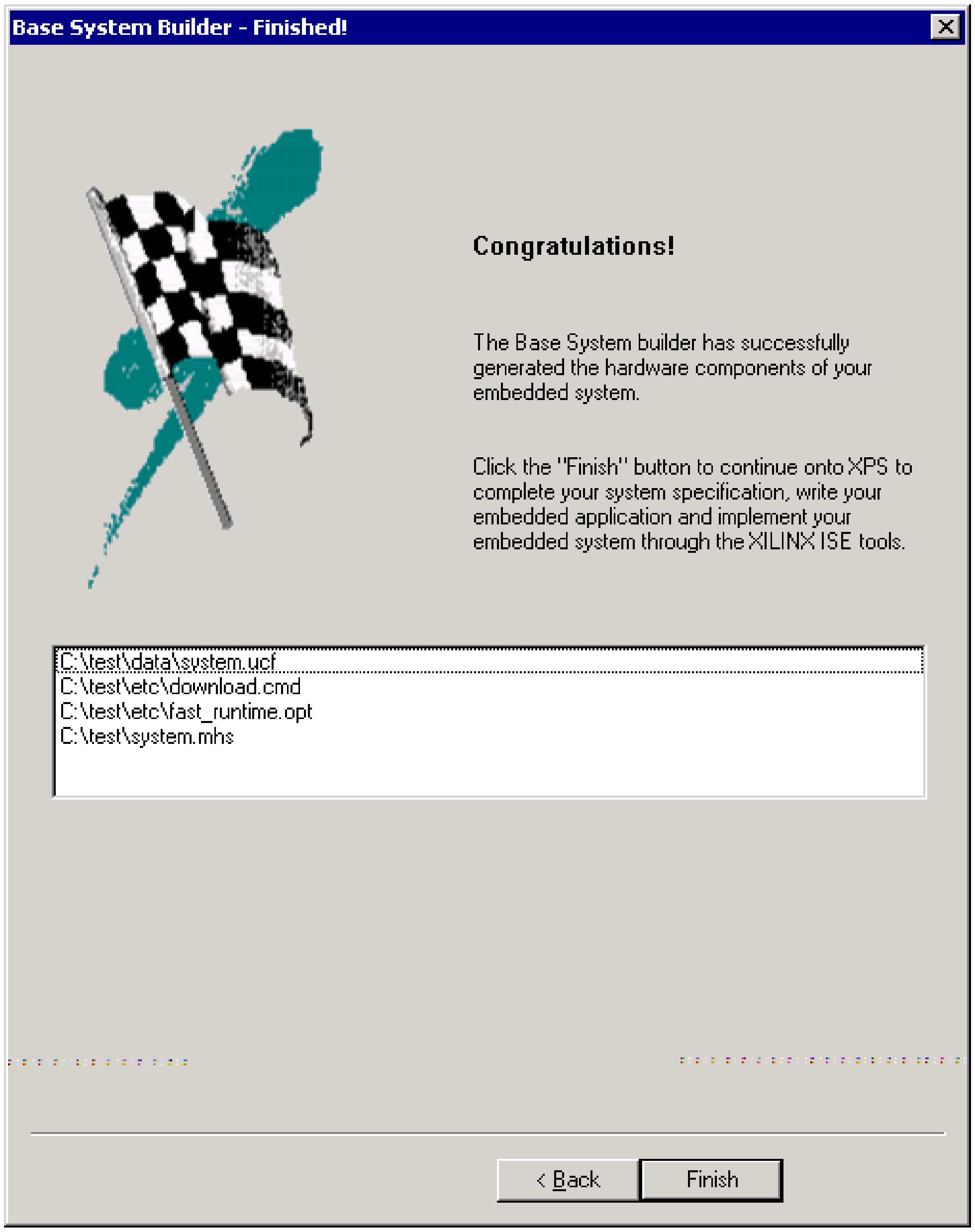
Core Name	Instance Name	Base Addr	High Addr
isbram_if_cntlr	iocm_cntlr	0xF6FFC000	0xF6FFFFFF
dsbram_if_cntlr	docm_cntlr	0xF6FF8000	0xF6FFBFFF

PLB Bus : PLB_V34 Inst. name: plb Attached Components:

Core Name	Instance Name	Base Addr	High Addr
plb_bram_if_cntlr	plb_bram_if_cntlr_1	0xFFFFC000	0xFFFFFFFF

OPB Bus : OPB_V20 Inst. name: opb Attached Components:

Core Name	Instance Name	Base Addr	High Addr
opb_sdram	SDRAM_8Mx32	0xF8000000	0xF9FFFFFF
opb_uartlite	RS232	0xFA000000	0xFA0000FF
opb_gpio	LEDs_4Bit	0xFA000100	0xFA0001FF
opb_timer	opb_timer_1	0xFA000200	0xFA0002FF
opb_intc	opb_intc_0	0xFA000300	0xFA00031F



Congratulations!

The Base System builder has successfully generated the hardware components of your embedded system.

Click the "Finish" button to continue onto XPS to complete your system specification, write your embedded application and implement your embedded system through the XILINX ISE tools.

```
C:\test\data\system.ucf
C:\test\etc\download.cmd
C:\test\etc\fast_runtime.opt
C:\test\system.mhs
```

Limitations

This tool was designed with the novice user in mind. As such it focusses on ease-of-use rather than on a wide array of features. The novice EDK user can use this tool to quickly generate a hardware system that is correct by construction for the targeted board. However, the advanced EDK user can use this tool to quickly generate a starter system and then use other XPS features to further enhance the generated system.

- The user cannot design systems with multiple processors
- The user cannot specify or modify the address maps
- There is no re-entrant mode, the user have to specify the entire system in one sitting

- Only the hardware component of the system is generated. No software application is generated.
- BSB does not check resources for specific device. Please refer to Xilinx Databook for details.

Chapter 4

Import Peripheral Wizard

Overview

The Import Peripheral Wizard is a software tool that generates necessary core description files required by EDK for a user peripheral so that it becomes part of core repository. After it is imported, a user peripheral can be used the same way as any Xilinx cores.

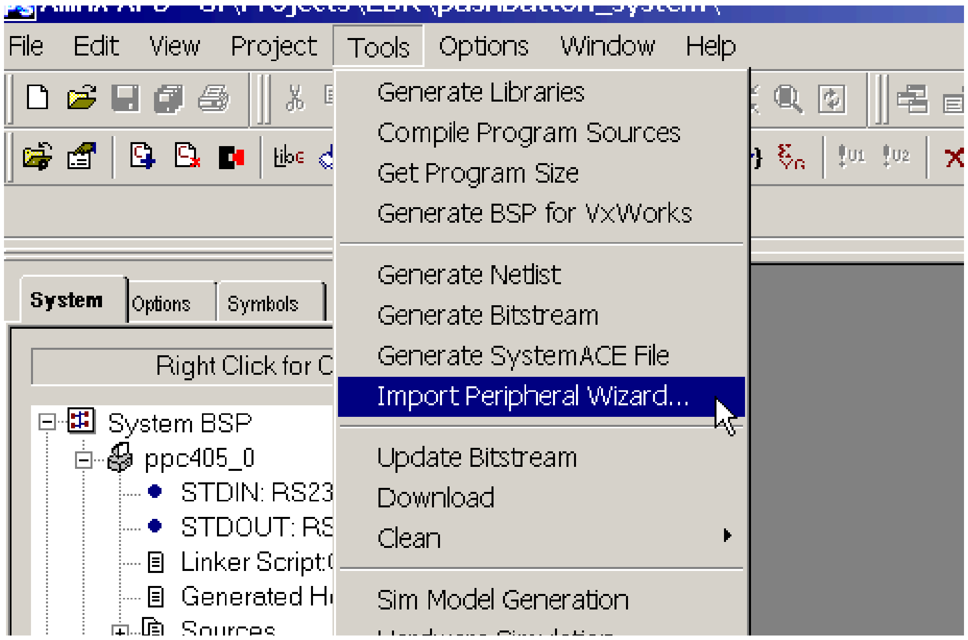
Before using this tool to import a peripheral, the user needs to insure that the peripheral complies with Xilinx implementation of IBM CoreConnect Bus Standard.

Import Peripheral Wizard Flow

The user can start Import Peripheral Wizard after creating a new project or opening an existing project in XPS.

Invoke Import Peripheral Wizard

With a project open, the user may start Import Peripheral Wizard at any point by selecting "Import Peripheral..." from Tools menu.



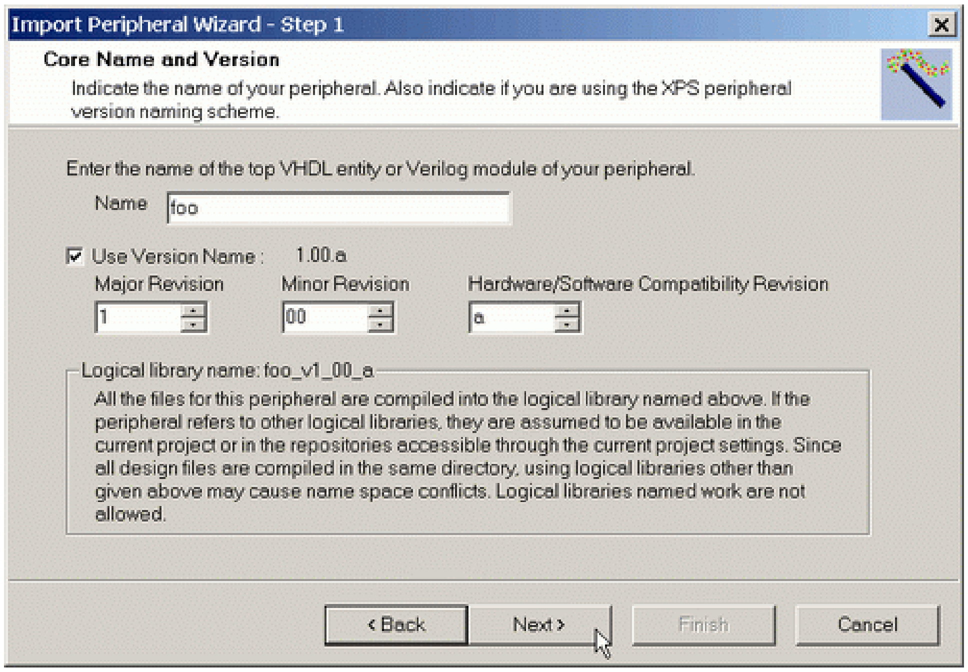
Define core name and version

First the user specifies the name of the user peripheral that has to be exactly the same as the top level VHDL entity or Verilog module name. Clicking on **Use Version Name** is an option and is not required. It provided for you to keep track of core versions. This will append the version to the core name and thus alter the logical library name of the core.

Note:

The PAO file that will be created later will reference this logical library name.

It is very important that all the elements of this peripheral are compiled into the indicated logical library or into some other logical library already available in the XPS project, or in any of the currently accessible repositories. This tool will actually process only the files that are compiled into the logical library indicated above. Other files are assumed to be available in the XPS project, or in any of the currently accessible repositories.



Indicate the name of your peripheral. Also indicate if you are using the XPS peripheral version naming scheme.

Enter the name of the top VHDL entity or Verilog module of your peripheral.

Name: foo

☑ Use Version Name : 1.00.a

Major Revision: 1
Minor Revision: 00
Hardware/Software Compatibility Revision: a

Logical library name: foo_v1_00_a

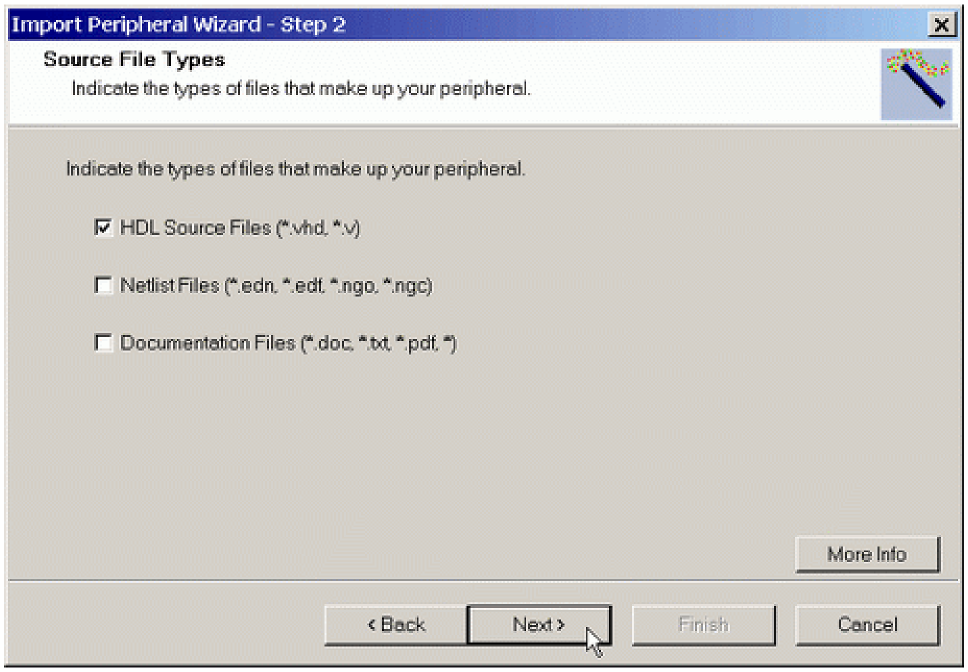
All the files for this peripheral are compiled into the logical library named above. If the peripheral refers to other logical libraries, they are assumed to be available in the current project or in the repositories accessible through the current project settings. Since all design files are compiled in the same directory, using logical libraries other than given above may cause name space conflicts. Logical libraries named work are not allowed.

Select source file types

The user then selects all applicable source file types including HDL source files (.vhd, .v), netlist files (.edn, .edf, .ngo, .ngc), and documentation files (.doc, .txt, .pdf). It is required that the top level be in HDL to allow the tool to collect necessary information. If user core only has netlists, the user can simply write a HDL wrapper before using the tool. In accordance with this rule, HDL source files is automatically selected if the user selects netlist files.

Note:

The top level HDL source file is expected to already conform to the Xilinx implementation of CoreConnect Bus conventions. Please review Chapter 1 and 2 on OPB/PLB usage of Processor IP User Guide found in the /doc directory with the install.


Import Peripheral Wizard - Step 2

Source File Types

Indicate the types of files that make up your peripheral.

Indicate the types of files that make up your peripheral.

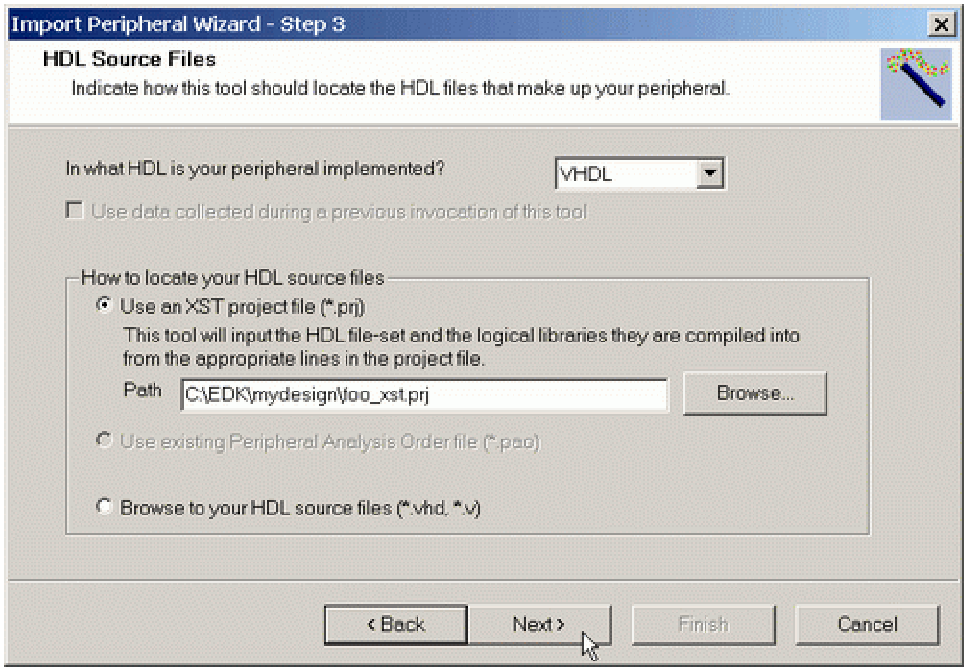
- [✓] HDL Source Files (*.vhd, *.v)
- [ ] Netlist Files (*.edn, *.edf, *.ngo, *.ngc)
- [ ] Documentation Files (*.doc, *.txt, *.pdf, *)

Collect source files

The user first selects the HDL language used to implement the peripheral. Both VHDL and Verilog are supported. Then the user informs the wizard how to locate the HDL source files. There are three different ways to collect all source files.

- The preferred way is to give an XST project file, as the file already has passed synthesis, and contains all appropriate HDL files as well as libraries. The wizard will extract necessary HDL file and library information automatically from the given project file and display them in the next panel.

  Read the XST manual under ISE installation to see the XST project syntax and how you can create one.
- The option of "Use existing Peripheral Analysis Order" is only enabled if existing core description files are detected. This can happen if the same core has been imported before, and the user is using it again to make some modifications.
- Directly browse to HDL source files. If this option is selected, the user needs to collect individual source files and libraries in the next panel.


Import Peripheral Wizard - Step 3

HDL Source Files

Indicate how this tool should locate the HDL files that make up your peripheral.

In what HDL is your peripheral implemented? VHDL

☐ Use data collected during a previous invocation of this tool

How to locate your HDL source files

● Use an XST project file (*.prj)

This tool will input the HDL file-set and the logical libraries they are compiled into from the appropriate lines in the project file.

Path C:\EDK\mydesign\foo_xst.prj

○ Use existing Peripheral Analysis Order file (*.pao)

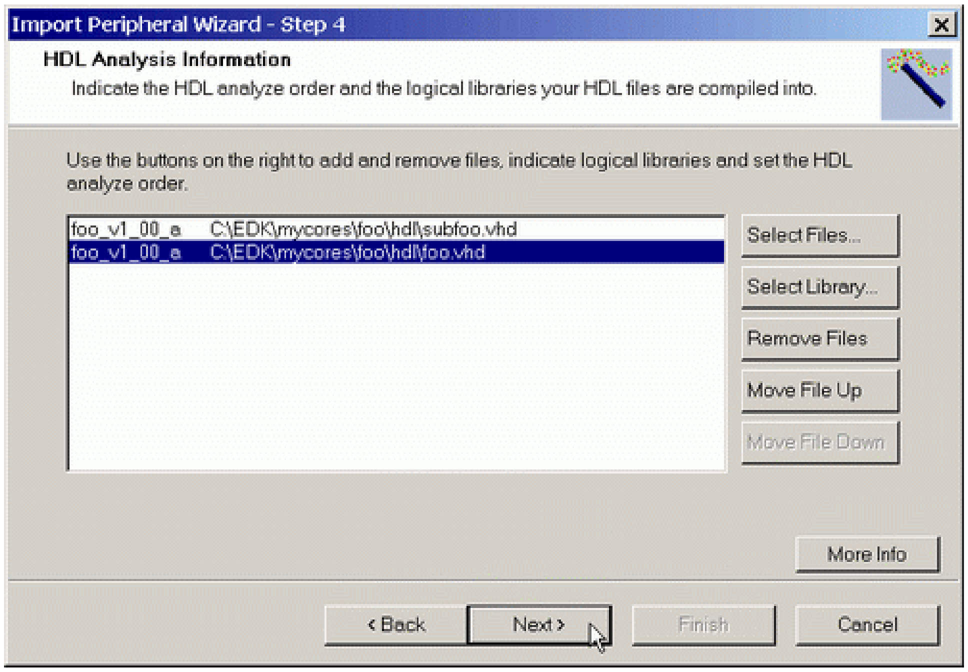
○ Browse to your HDL source files (*.vhd, *.v)

Define HDL analysis order

In this panel, the user is responsible for placing HDL files in the appropriate order that they need to be analyzed. Not providing all libraries or not placing them in the correct order may result in an error. If the XST project file flow is selected in the previous step and the 'nosort' keyword is present in the XST project file, all the buttons on the right will be disabled to not allow the user to change the analysis order as defined in the XST project file.

When the user clicks on the **Select Library** button, the libraries available in the repositories known to the current XPS project are displayed. When the user selects a library, the files available in the library are displayed. All files in the selected library are selected by default, but the user can deselect any files by unchecking the checkbox next to the file.

The newly selected files are displayed before the current highlighted line if there's one.


Import Peripheral Wizard - Step 4

HDL Analysis Information

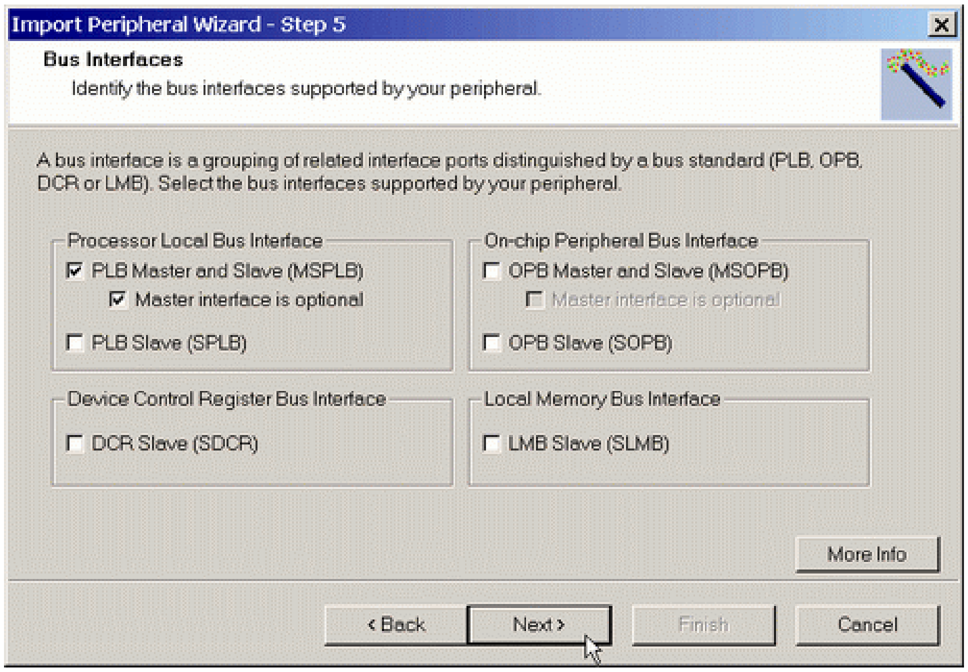
Indicate the HDL analyze order and the logical libraries your HDL files are compiled into.

Use the buttons on the right to add and remove files, indicate logical libraries and set the HDL analyze order.

```
foo_v1_00_a    C:\EDK\mycores\foo\hdl\subfoo.vhd
foo_v1_00_a    C:\EDK\mycores\foo\hdl\foo.vhd
```

Select bus interfaces

In this panel, the user selects bus interface(s) applicable to the user peripheral


Import Peripheral Wizard - Step 5

Bus Interfaces

Identify the bus interfaces supported by your peripheral.

A bus interface is a grouping of related interface ports distinguished by a bus standard (PLB, OPB, DCR or LMB). Select the bus interfaces supported by your peripheral.

Processor Local Bus Interface
- [✓] PLB Master and Slave (MSPLB)
  - [✓] Master interface is optional
- [ ] PLB Slave (SPLB)

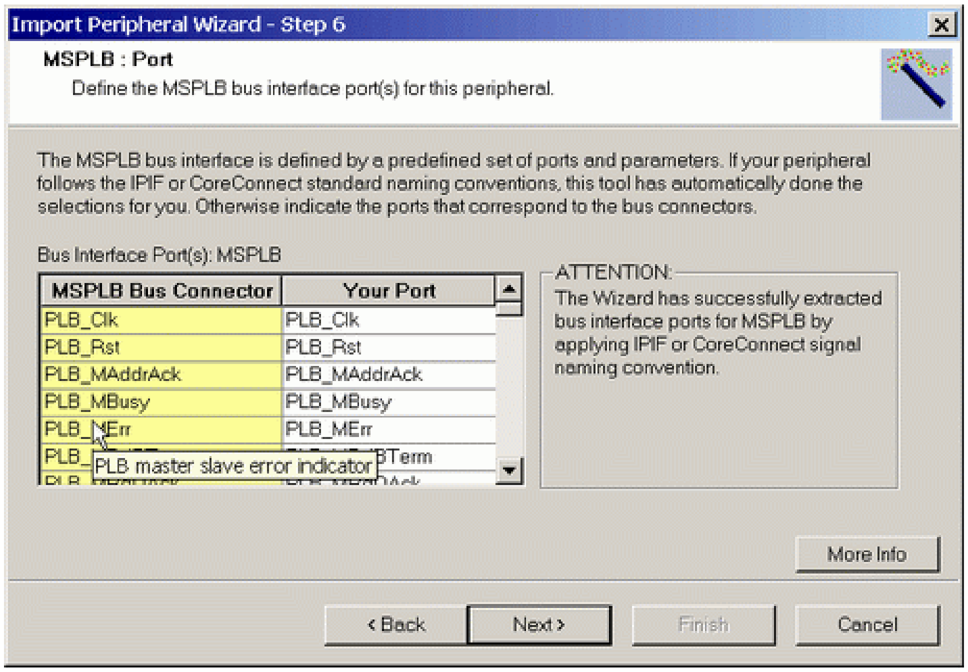
On-chip Peripheral Bus Interface
- [ ] OPB Master and Slave (MSOPB)
  - [ ] Master interface is optional
- [ ] OPB Slave (SOPB)

Device Control Register Bus Interface
- [ ] DCR Slave (SDCR)

Local Memory Bus Interface
- [ ] LMB Slave (SLMB)

Define bus interface ports

In this panel, the user defines the bus interface ports from your peripheral to the bus. The software will parse HDL source file and automatically match up ports if CoreConnect naming convention is followed. If a match is not found, the user will need to manually select the right port from a pull-down menu.



MSPLB : Port

Define the MSPLB bus interface port(s) for this peripheral.

The MSPLB bus interface is defined by a predefined set of ports and parameters. If your peripheral follows the IPIF or CoreConnect standard naming conventions, this tool has automatically done the selections for you. Otherwise indicate the ports that correspond to the bus connectors.

Bus Interface Port(s): MSPLB

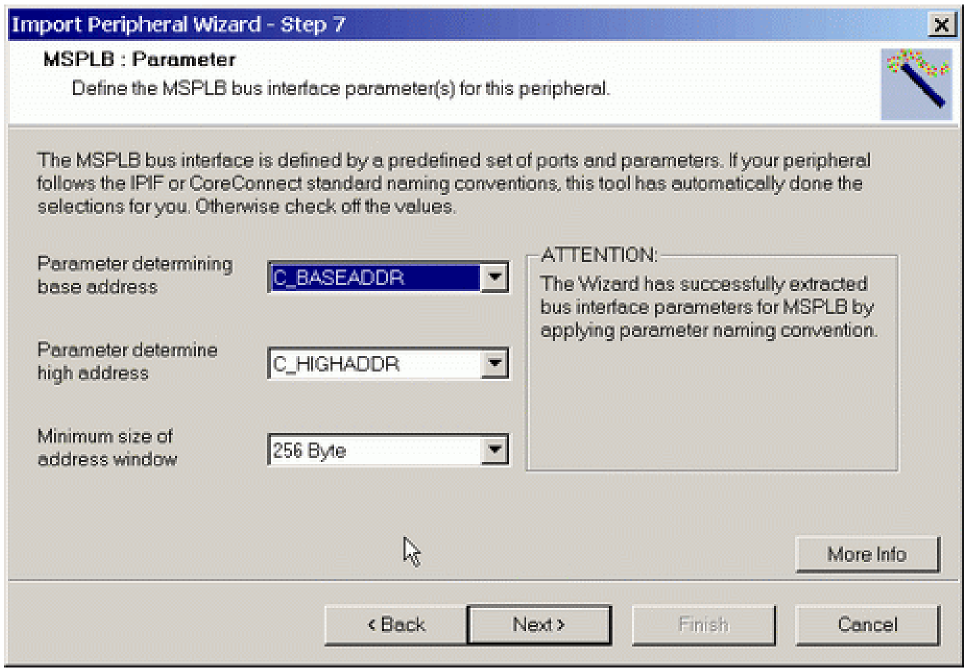
MSPLB Bus Connector	Your Port
PLB_Clk	PLB_Clk
PLB_Rst	PLB_Rst
PLB_MAddrAck	PLB_MAddrAck
PLB_MBusy	PLB_MBusy
PLB_MErr	PLB_MErr

PLB master slave error indicator

ATTENTION:
The Wizard has successfully extracted bus interface ports for MSPLB by applying IPIF or CoreConnect signal naming convention.

Define bus parameters

In this panel, the user defines the parameters, such as base and high addresses and minimum memory size, for each bus interface.



MSPLB : Parameter

Define the MSPLB bus interface parameter(s) for this peripheral.

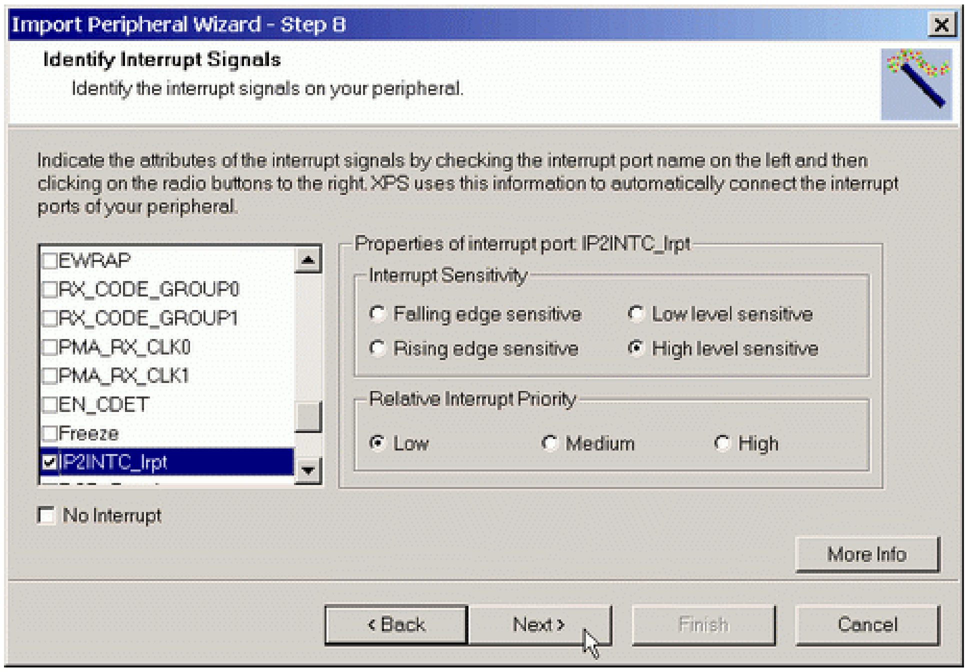
The MSPLB bus interface is defined by a predefined set of ports and parameters. If your peripheral follows the IPIF or CoreConnect standard naming conventions, this tool has automatically done the selections for you. Otherwise check off the values.

Parameter determining base address	C_BASEADDR
Parameter determine high address	C_HIGHADDR
Minimum size of address window	256 Byte

ATTENTION:
The Wizard has successfully extracted bus interface parameters for MSPLB by applying parameter naming convention.

Define interrupt signals

In this panel, the user defines interrupt signal(s) by first selecting the signal(s) and then setting sensitivity and priority level. If there is no interrupt signal in the system, check “No Interrupt” instead.



Identify Interrupt Signals

Identify the interrupt signals on your peripheral.

Indicate the attributes of the interrupt signals by checking the interrupt port name on the left and then clicking on the radio buttons to the right. XPS uses this information to automatically connect the interrupt ports of your peripheral.

- ☐ EWRAP
- ☐ RX_CODE_GROUP0
- ☐ RX_CODE_GROUP1
- ☐ PMA_RX_CLK0
- ☐ PMA_RX_CLK1
- ☐ EN_CDET
- ☐ Freeze
- ☑ IP2INTC_Irpt

Properties of interrupt port IP2INTC_Irpt

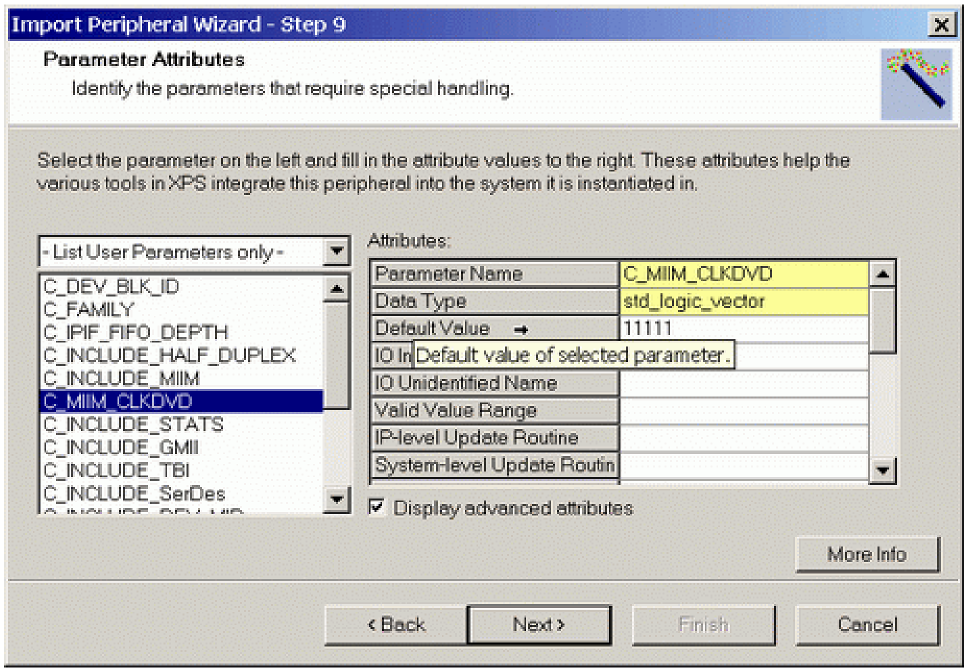
Interrupt Sensitivity: Falling edge sensitive, Low level sensitive, Rising edge sensitive, [✓ High level sensitive]

Relative Interrupt Priority: [✓ Low] Medium High

☐ No Interrupt

Define parameter attributes

In this panel, the user defines parameter attributes applicable to the core. Please refer to MPD chapter for detailed documentation on each attribute.


Import Peripheral Wizard - Step 9

Parameter Attributes

Identify the parameters that require special handling.

Select the parameter on the left and fill in the attribute values to the right. These attributes help the various tools in XPS integrate this peripheral into the system it is instantiated in.

- List User Parameters only -

C_DEV_BLK_ID
C_FAMILY
C_IPIF_FIFO_DEPTH
C_INCLUDE_HALF_DUPLEX
C_INCLUDE_MIIM
C_MIIM_CLKDVD
C_INCLUDE_STATS
C_INCLUDE_GMII
C_INCLUDE_TBI
C_INCLUDE_SerDes

Attributes:

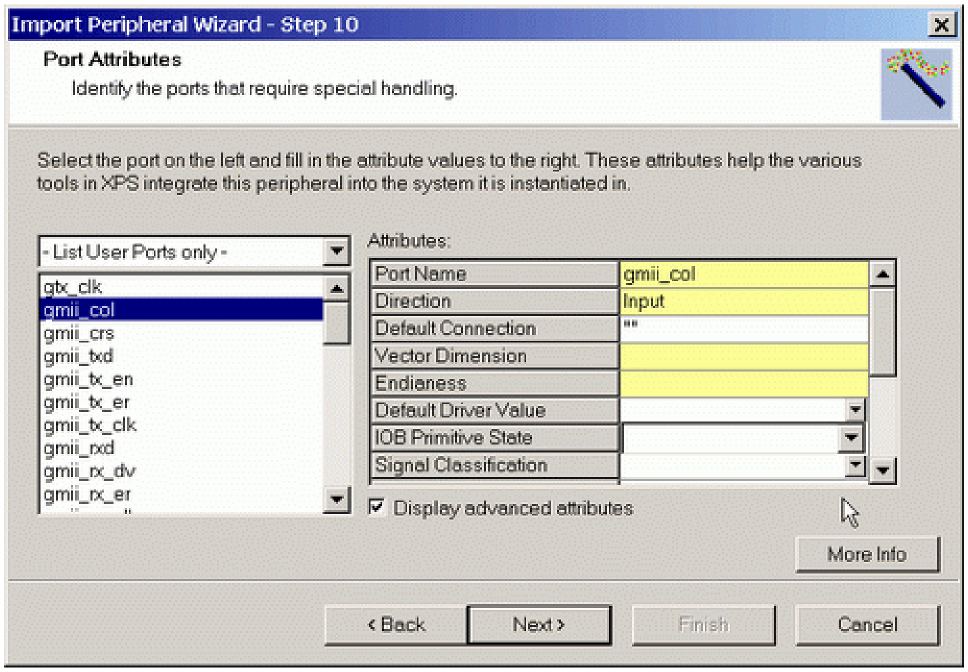
Parameter Name	C_MIIM_CLKDVD
Data Type	std_logic_vector
Default Value	11111
IO In	
IO Unidentified Name	
Valid Value Range	
IP-level Update Routine	
System-level Update Routin	

Default value of selected parameter.

☑ Display advanced attributes

Define port attributes

In this panel, the user defines port attributes applicable to the peripheral. Certain attributes are fixed as they are either defined in HDL file or specified in previous steps. Please refer to MPD chapter for detailed documentation on each attribute.



Import Peripheral Wizard - Step 10

Port Attributes

Identify the ports that require special handling.

Select the port on the left and fill in the attribute values to the right. These attributes help the various tools in XPS integrate this peripheral into the system it is instantiated in.

- List User Ports only -

gtx_clk
gmii_col
gmii_crs
gmii_txd
gmii_tx_en
gmii_tx_er
gmii_tx_clk
gmii_rxd
gmii_rx_dv
gmii_rx_er

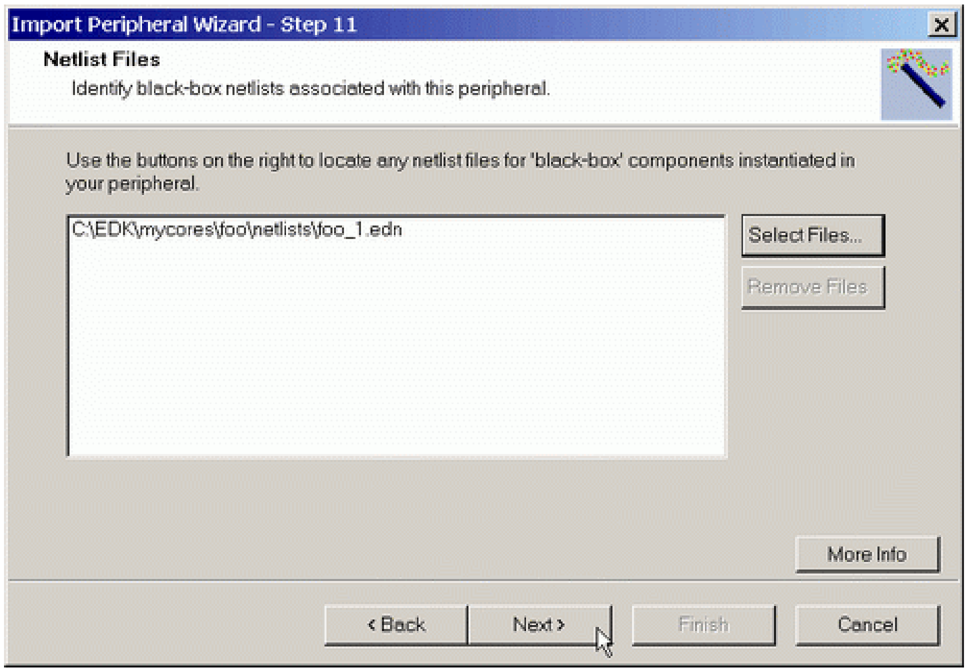
Attributes:

Port Name	gmii_col
Direction	Input
Default Connection	""
Vector Dimension	
Endianess	
Default Driver Value	
IOB Primitive State	
Signal Classification	

☑ Display advanced attributes

Collect netlist files

If netlist files are selected earlier, there would be corresponding panel to allow the user to browse and collect files.



Import Peripheral Wizard - Step 11

Netlist Files

Identify black-box netlists associated with this peripheral.

Use the buttons on the right to locate any netlist files for 'black-box' components instantiated in your peripheral.

C:\EDK\mycores\foo\netlists\foo_1.edn

Collect documentation files

If documentation files are selected earlier, there would be corresponding panel to allow the user to browse and collect files.

Import user peripheral

Upon completion, Import Peripheral Wizard copies all user source files and generates core description files under {project directory} / pcores directory:

- Under *data* subdirectory, both MPD and PAO files are created.
- All HDL source files are copied to *hdl/vhdl or hdl/verilog* subdirectory, .
- If the peripheral contains netlists, all netlist files are copied to *netlist* subdirectory and a BBD file will also be created under *data* subdirectory.
- If the peripheral has documentation files, all of them are copied to *doc* subdirectory.

If existing MPD and PAO files are detected, the user can choose to save them in a backup directory without overriding them on the last panel.

Note:

THE USER HAS TO CLOSE AND REOPEN THE CURRENT XPS PROJECT TO SEE THE IMPORTED PERIPHERAL.

Limitations.

- The wizard doesn't support master only bus interface because such use case is rare for a peripheral.
- The wizard doesn't allow parameterization of BBD file, as the default is sufficient for XPS / PlatGen to process the user peripheral.
- The wizard doesn't allow spaces in file path because of an XST limitation.

Platform Generator

Overview

The hardware component is defined by the Microprocessor Hardware Specification (MHS) file. An MHS file defines the configuration of the embedded processor system, and includes the following:

- Bus architecture
- Peripherals
- Connectivity of the system
- Interrupt request priorities
- Address space

Hardware generation is done with the Platform Generator (platgen) tool and an MHS file. This will construct the embedded processor system in the form of hardware netlists (HDL and implementation netlist files).

This chapter includes the following sections:

Tool Requirements

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

Tool Usage

Run PlatGen as follows:

```
platgen system.mhs
```

Tool Options

The following are the options supported in the current version:

-h (Help)

The **-h** option displays the usage menu and quits.

-v (Display version)

The -v option displays the version and quits.

-f <filename>

Read command line arguments and options from file.

-iobuf yes | no

IOB insertion at the top-level. The default is yes.

This allows the system to be included as a macro in a top-level design. Otherwise, the output from PlatGen is the top-level design.

-lang verilog | vhdl

HDL language output. The default is vhdl.

-log <logfile[.log]>

Specify log file. The default is platgen.log. Currently, not implemented.

-lp <library_path>

Add <library_path> to the list of IP search directories. A library is a collection of repository areas.

-od <output_dir>

Output directory path. The default is the current directory.

-p <partname>

Use specified part type to implement the design.

-st xst | none

Generate synthesis project files. The default is xst.

PlatGen produces a synthesis vendor specific project file.

-ti <instname>

Top-level instance name. When used with the **-iobuf no** option, the BMM and <system>_stub.[vhd | v] are populated with this value.

-tn <compname>

Top-level entity / module name.

Load Path

Refer to Figure 5-1 for a depiction of the peripheral directory structure.

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

- Current directory (where PlatGen was launched; not where the MHS resides)
- Set the EDK tool option **-lp** option

PlatGen uses a search priority mechanism to locate peripherals, as follows:

1. Search the pcores directory in the project directory
2. Search <library_path> / <Library Name> / pcores as specified by the **-lp** option
3. Search XILINX_EDK / hw / <Library Name> / pcores

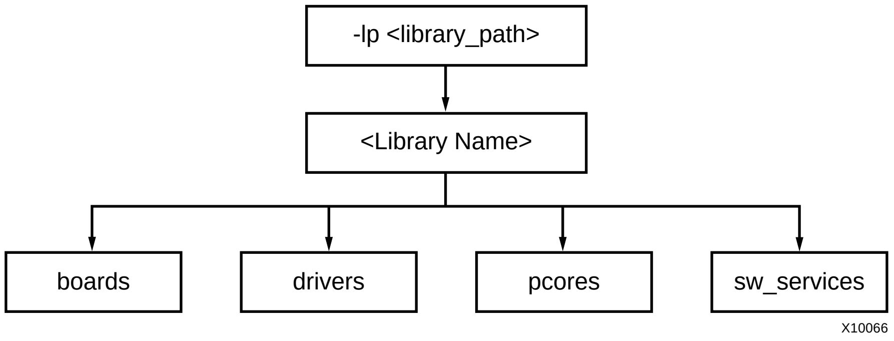


Figure 5-1: **Peripheral Directory Structure**

From the pcores directory, the peripheral name is the name of the root directory. From the root directory, the underlying directory structure is as follows:

```
data
hdl
netlist
simmodels
```

Output Files

PlatGen produces the following directories and files. From the project directory, this is the underlying directory structure:

```
hdl
implementation
synthesis
```

HDL Directory

The hdl directory contains the following:

`system.[vhd|v]`

This is the top level HDL file of the embedded processor system as defined in the MHS.

`<inst>_wrapper.[vhd|v]`

This is the HDL wrapper file for the of individual IP components defined in the MHS.

Implementation Directory

The implementation directory contains the following:

> `*peripheral*_wrapper.ngc`
>
> Implementation netlist file of the peripheral.

Synthesis Directory

The synthesis directory contains the following:

> `*system*.[prj|scr]`
>
> Synthesis project file.

About Memory Generation

PlatGen generates the necessary banks of memory and the initialization files for the BRAM Block (bram_block). The BRAM Block is coupled with a BRAM controller.

Current BRAM controllers include the following:

- DSOCM BRAM Controller (dsbram_if_cntlr) - PowerPC only
- ISOCM BRAM Controller (isbram_if_cntlr) - PowerPC only
- LMB BRAM Controller (lmb_bram_if_cntlr) - MicroBlaze only
- OPB BRAM Controller (opb_bram_if_cntlr)
- PLB BRAM Controller (plb_bram_if_cntlr)

The BRAM block (bram_block) and one of the BRAM controllers are tightly bound. Meaning that the associated options of the BRAM controller define the resulting BRAM block. Theses options are listed in every BRAM controller MPD file. For example, the OPB BRAM controller MPD defines the following:

```
OPTION NUM_WRITE_ENABLES = 4
OPTION ADDR_SLICE = 29
OPTION DWIDTH = 32
OPTION AWIDTH = 32
```

The definition of AWIDTH and DWIDTH is applied to C_AWIDTH and C_DWIDTH of the BRAM block, respectively. The port dimensions on ports A and B are symmetrical on the bram_block. Platgen overwrites all user-defined settings on the BRAM block to have uniform port widths.

You can only connect BRAM controllers of the same options values to the same BRAM block instance. For example, you can connect a OPB BRAM controller and LMB BRAM controller to the same BRAM block. However, you can not connect a OPB BRAM controller and a PLB BRAM controller to the same BRAM block instance. You can connect a LMB BRAM controller and a DSOCM BRAM controller to the same BRAM block instance.

The BRAM controller's MHS options, C_BASEADDR and C_HIGHADDR (see the Chapter 15, "Microprocessor Hardware Specification (MHS)," documentation for more information), define the different depth sizes of memory.

The MicroBlaze processor is a 32-bit machine, therefore, has data and instruction bus widths of 32-bit. Only predefined memory sizes are allowed. Otherwise, MUX stages have to be introduced to build bigger memories, thus slowing memory access to the memory banks. For Spartan-II, the maximum allowed memory size is 4 kBytes which uses 8 Select BlockRAM. For Spartan-IIE, the maximum allowed memory size is 8 kBytes which uses 16

Select BlockRAM. For Virtex / VirtexE, the maximum allowed memory size is 16 kBytes which uses 32 Select BlockRAM. For Virtex-II, it is 64 kBytes which also uses 32 Select BlockRAMs.

Table 5-1: **Predefined Memory Sizes**

Architecture	Memory Size (kBytes) 32-bit byte-write	Memory Size (kBytes) 64-bit byte-write
Spartan-II	2, 4	4,
Spartan-IIE	2, 4, 8, 16	4, 8, 16, 32
Spartan-3	8, 16, 32, 64	16, 32, 64, 128
Virtex	2, 4, 8, 16	4, 8, 16, 32
VirtexE	2, 4, 8, 16	4, 8, 16, 32
Virtex-II	8, 16, 32, 64	16, 32, 64, 128
Virtex-II PRO	8, 16, 32, 64	16, 32, 64, 128

Be sure to check your FPGA resources can adequately accommodate your executable image. For example, the smallest Spartan-II device, xc2s15, only 4 Select BlockRAMs are available for a maximum memory size of 2 kBytes. Whereas, the largest Spartan-II device, xc2s200, 14 Select BlockRAMs are available for a maximum memory size of 7 kBytes.

For example, for a memory size of 4 kBytes on a Virtex device, PlatGen uses 8 Select BlockRAMs.

BMM Policy

A BMM (BlockRAM Memory Map) file contains a syntactic description of how individual BlockRAMs constitute a contiguous logical data space. PlatGen has the following policy for writing a BMM file:

- If PORTA is connected and PORTB is not connected, then the BMM generated will be from PORTA point of reference.
- If PORTA is not connected and PORTB is connected, then the BMM generated will be from PORTB point of reference.
- If PORTA is connected and PORTB is connected, then the BMM generated will be from PORTA point of reference.

BMM Flow

The EDK tools Implementation Tools flow using Data2BRAM.

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

BitGen outputs <system>_bd.bmm that contains the physical location of BlockRAMs. The <system>_bd.bmm and <system>.bit files are input to Data2BRAM. Data2BRAM translates contiguous fragments of data into the proper initialization records for Virtex series BlockRAMs.

Reserved MHS Parameters

PlatGen automatically expands and populates certain reserved parameters. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved parameter names:

Table 5-2: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_FAMILY	FPGA Device Family
C_INSTANCE	Instance name of component
C_KIND_OF_EDGE	Vector of edge sensitive (rising / falling) of interrupt signals
C_KIND_OF_LVL	Vector of level sensitive (high / low) of interrupt signals
C_KIND_OF_INTR	Vector of interrupt signal sensitivity (edge / level)
C_NUM_INTR_INPUTS	Number of interrupt signals
C_MASK	LMB Decode Mask (deprecated)
C_NUM_MASTERS	Number of OPB masters (deprecated)
C_NUM_SLAVES	Number of OPB slaves (deprecated)
C_DCR_AWIDTH	DCR Address width
C_DCR_DWIDTH	DCR Data width
C_DCR_NUM_SLAVES	Number of DCR slaves
C_LMB_AWIDTH	LMB Address width
C_LMB_DWIDTH	LMB Data width
C_LMB_MASK	LMB Decode Mask
C_LMB_NUM_SLAVES	Number of LMB slaves
C_OPB_AWIDTH	OPB Address width
C_OPB_DWIDTH	OPB Data width
C_OPB_NUM_MASTERS	Number of OPB masters
C_OPB_NUM_SLAVES	Number of OPB slaves
C_PLB_AWIDTH	PLB Address width
C_PLB_DWIDTH	PLB Data width
C_PLB_MID_WIDTH	PLB master ID width
C_PLB_NUM_MASTERS	Number of PLB masters
C_PLB_NUM_SLAVES	Number of PLB slaves

Synthesis Netlist Cache

A system rebuild occurs if the following fundamental changes trigger a core rebuild:

- Instance name change
- Parameter value change

- Core version change
- Core is specified with the MPD “CORE_STATE=DEVELOPMENT” option

At least one of the above conditions is occurring to trigger a core rebuild.

Current Limitations

The current limitations of the PlatGen flow are:

- Vector slicing is not allowed.

Chapter 6

Simulation Model Generator

This chapter introduces the basics of HDL simulation and describes the Simulation Model Generator tool and COMPEDKLIB utility usage. It includes the following sections.

- "Overview"
- "Simulation Basics"
- "Simulation Basics"
- "Compiling EDK Simulation Libraries"
- "Simulation Models"
- "SimGen Syntax"
- "Output Files"
- "Memory Initialization"
- "Simulating Your Design"
- "Current Limitations"

Overview

The Simulation Model Generator (SimGen) creates and configures various VHDL and Verilog simulation models for a specified hardware. It takes a Microprocessor Hardware Specification (MHS) file as input that describes the hardware.

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 Microprocessor Hardware Specification (MHS) file. Please refer to Chapter 15, "Microprocessor Hardware Specification (MHS)" for more information.

Simulation Basics

This section introduces the basic facts and terminology of HDL simulation in EDK. There are three stages in the FPGA design process in which you conduct verification through simulation. Figure 6-1 shows these stages.

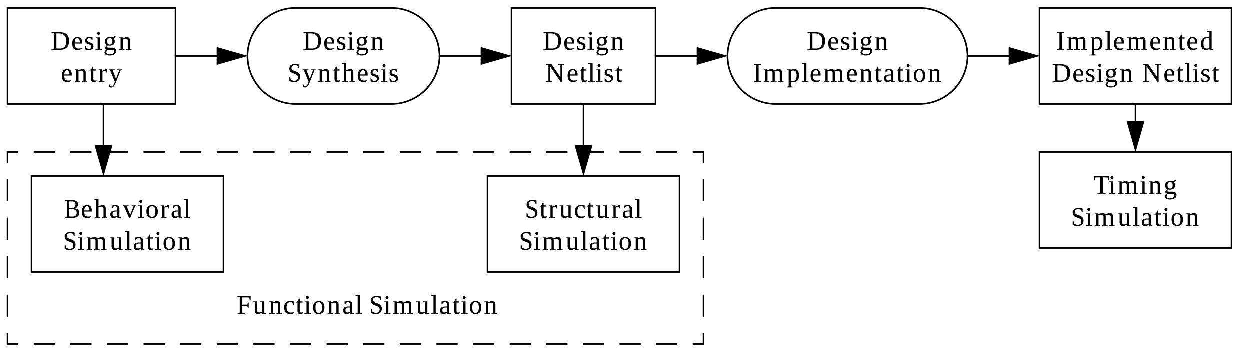


Figure 6-1: **FPGA design simulation stages**

Behavioral Simulation

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

Structural Simulation

After the behavioral simulation is error free, the HDL design is synthesized to gates. The post-synthesized structural simulation is a functional simulation with unit delay timing. The simulation can be used to identify initialization issues and to analyze don't care conditions. The post synthesis simulation generally uses the same testbench as functional simulation.

Timing Simulation

Structural timing simulation is a back-annotated timing simulation. Timing simulation is important in verifying the operation of your circuit after the worst case place and route delays are calculated for your design. The back annotation process produces a netlist of library components annotated in an SDF file with the appropriate block and net delays from the place and route process. The simulation will identify any race conditions and setup-and-hold violations based on the operating conditions for the specified functionality.

Simulation Libraries

The following libraries 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 Libraries

The following libraries are provided by Xilinx for simulation. These libraries can be compiled using COMPXLIB. Please refer to Chapter 6, "Verifying Your Design" in the "Synthesis and Verification Design Guide" in your ISE 6.1 distribution to learn more about compiling and using Xilinx simulation libraries.

UNISIM Library

This is a library of functional models used for behavioral and structural simulation. It contains default unit delays and 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 and structural simulation.

SIMPRIM Library

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

Structural and Timing simulation models generated by SimGen will instantiate SIMPRIM library components.

XilinxCoreLib Library

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

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

EDK Library

Used for behavioral simulation. It contains all the EDK IP components, precompiled for ModelSim SE and PE. EDK IP components library is provided for VHDL only.

The EDK library can be compiled with COMPEDKLIB. Please refer to the *Getting Started with EDK* document to find out how to compile these libraries.

Compiling EDK Simulation Libraries

Before starting behavioral simulation of your design, you must compile the EDK Simulation Libraries for the target simulator. For this purpose Xilinx® provides a tool called COMPEDKLIB.

COMPEDKLIB is a tool for compiling the EDK HDL based simulation libraries using the tools provided by the simulator vendor.

Usage

```
compedklib [ -h ] [ -o output-dir-name ] [ -lp repository-dir-name ]
      [ -X compxlib-output-dir-name ] [ -E compedklib-output-dir-name ]
```

This tool compiles the HDL in EDK pcore libraries for simulation using the simulators supported by the EDK. Currently, the only supported simulator is MTI PE/SE.

To print the COMPEDKLIB online help to your monitor screen, type the following at the command line:

```
compedklib -h
```

COMPEDKLIB Command Line Examples

Use Case I: Compiling HDL sources in the built-in repositories in the EDK

The most common use case is as follows:

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

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

Use Case II: Compiling HDL sources in your own repository

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

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

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

Other details

You can supply multiple '-X' and '-E' arguments. The order is important. If you have the same pcore in two places, the first one is used.

Some pcores are secure in that their source code is not available. In such cases, the repository contains the compiled models. These are copied out into `<compedklib-output-dir-name>`.

If your pcores are in your XPS project, you do not need to bother about Use Case 2. XPS / SIMGEN will create the scripts to compile them.

If you have the MODELSIM environment variable set, the modelsim.ini file that it points to gets modified when this tool is compiling the HDL sources for MTI SE / PE.

Presently only VHDL is supported.

The execution log is available in `compedklib.log`.

Simulation Models

This section describes how to generate each of the three FPGA simulation stages. For each stage, a different simulation model can be created by SimGen.

Behavioral Models

To create a behavioral simulation model, SimGen requires an MHS file as input. SimGen will create a set of hdl files that model the functionality of the design. Optionally, SimGen can generate a compile script for a specified vendor simulator. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any processor that may exist in the design. This data is obtained from an existing executable elf file.

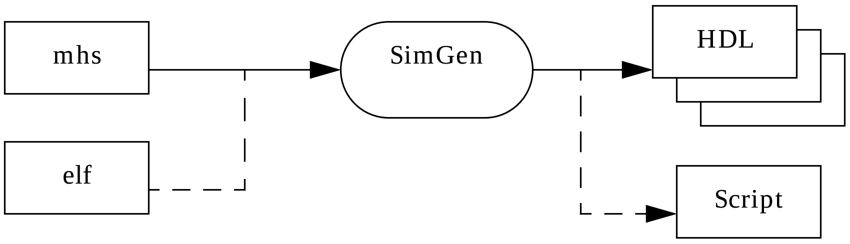


Figure 6-2: **Behavioral simulation model generation**

Structural Models

To create a structural simulation model, SimGen requires an MHS file as input and associated synthesized netlist files. From these netlist files SimGen will create 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. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any processor that may exist in the design. This data is obtained from an existing executable elf file.

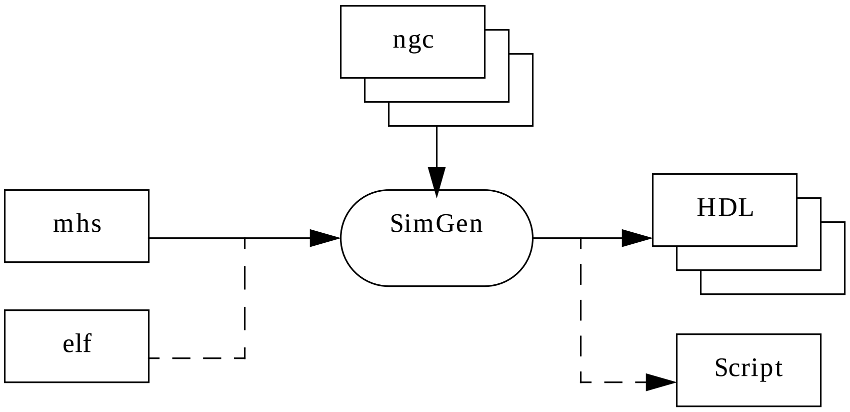


Figure 6-3: **Structural simulation model generation**

Note: The EDK design flow is modular. PlatGen will generate a set of netlist files that are used by SimGen to generate structural simulation models.

Timing Models

To create a timing simulation model, SimGen requires an MHS file as input and associated implemented netlist file. From this netlist file SimGen will create an hdl file that models the design and an SDF file with appropriate timing information for it. Optionally, SimGen can generate a compile script for a specified vendor simulator. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any processor that may exist in the design. This data is obtained from an existing executable elf file.

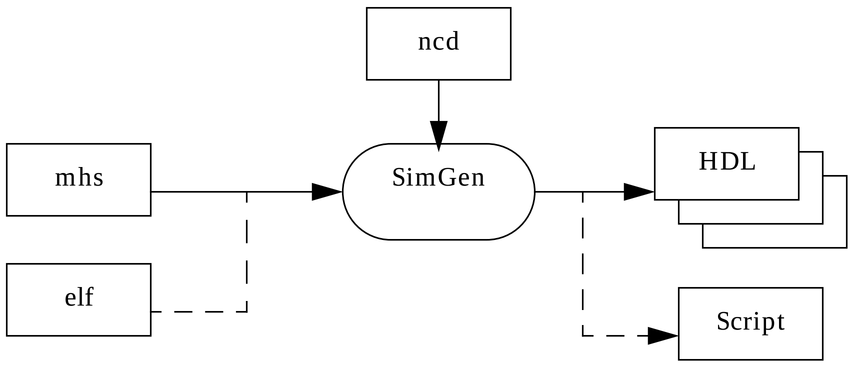


Figure 6-4: **Timing simulation model generation**

SimGen Syntax

At the prompt, execute SimGen with the MHS file and appropriate options as inputs.

For example,

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

Requirements

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

Options

The following options are supported in the current version:

Help

-h, -help

The **-h** option displays the usage menu and quits.

HDL Language

-lang vhdl | verilog

The **-lang** option specifies the HDL Language. This option will override the language specified in the MVS file.

Default: vhdl

Log output

-log <logfile[.log]>

The **-log** option specifies the log file.

Default: simgen.log

Library Directories

-lp <library_path>

The **-lp** option allows you to specify library directory paths. This option may be specified more than once for multiple library directories.

Simulation model type

-m beh | str | tim

The **-m** option allows you to select the type of simulation models to be used. The supported simulation model types are behavioral (beh), structural (str) and timing (tim).

Default: beh

Output Directory

-od <output_dir>

The **-pd** option specifies the project directory path. The default is the current directory.

Target part or family

-p <partname>

The **-p** option allows you to target a specific part or family. This option must be specified.

Processor Elf Files

-pe <proc_instance> <elf_file> {<elf_file>}

Specify a list of elf files to be associated with the processor with instance name as defined in the MHS.

Simulator

-s mti

The **-s** option specifies for which simulator to produce a compilation script file. The only supported simulator is Model Technology ModelSim (mti).

Source Directory

-sd <source_dir>

Source directory to search for netlist files.

Source Directory

-E <edklib_dir>

Path to EDK simulation libraries directory. This is the output directory of the compedklib tool.

Source Directory

-X <xlib_dir>

Path to Xilinx simulation libraries (unisim, simprim, XilinxCoreLib) directory. This is the output directory of the compxlib tool.

Version

-v

The **-v** option displays the version and quits.

Output Files

SimGen produces all simulation files in the simulation directory within the output directory, and inside a subdirectory for each of the simulation models.

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

After a successful simgen execution, the simulation directory contains the following files:

`*peripheral*_wrapper.[vhd|v]`

Modular simulation files for each component.

`*system_name*.[vhd|v]`

The top level HDL file of the design.

`*system_name*.sdf`

The Standard Delay Format file with the appropriate block and net delays from the place and route process used only for timing simulation.

`*system_name*.[do|f]`

Script to compile the hdl files and load the compiled simulation models in the simulator.

Memory Initialization

If a design contains banks of memory for a system, the corresponding memory simulation models can be initialized with data. With the -pe switch, a list of executable elf files to associate to a given processor instance can be specified.

The compiled executable files are generated with the appropriate 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.

VHDL

For vhdl simulation models, execute SimGen with the **-i** option to generate a VHDL file. This file will contain a configuration for the system with all initialization values. For example:

```
simgen system.mhs -pe mblaze executable.elf -l vhdl ...
```

This command generates the VHDL system configuration in the file *system_init.vhd*. This file is used along with your system to initialize memory. The bram blocks connected to the processor mblaze will contain the data in executable.elf.

Verilog

For verilog simulation models, execute SimGen with the **-i** option to generate a verilog file. This file will contain defparam constructs that initialize memory. For example:

```
simgen system.mhs -pe mblaze executable.elf -l verilog ...
```

This command generates the verilog memory initialization file *system_init.v*. This file is used along with your system to initialize memory. The bram blocks connected to the processor mblaze will contain the data in executable.elf.

Simulating Your Design

When simulating your design, there are some special considerations you need to keep in mind such as the global reset and tristate nets. Xilinx ISE Tools provide detailed information on how to simulate your VHDL or Verilog design. Please refer to Chapter 6, "Verifying Your Design" in the ISE "Synthesis and Verification Design Guide" for more information. A PDF version of this document can be found at

/doc/usenglish/books/docs/sim/sim.pdf

in your XILINX install area, or online at

http://www.xilinx.com/support/sw_manuals/xilinx6/index.htm

Current Limitations

SimGen does not support generation of mixed level simulation models.

Library Generator

Summary

This chapter describes the Library Generator utility needed for the generation of libraries and drivers for embedded soft processors. It also describes how the user can customize peripherals and associated drivers.

Overview

The Library Generator (libgen) is generally the first tool to run to configure libraries and device drivers. Libgen takes an MSS (Microprocessor Software Specification) file created by the user as input. The MSS file defines the drivers associated with peripherals, standard input/output devices, interrupt handler routines, and other related software features. Libgen configures libraries and drivers with this information. For more information on the MSS file format, please refer Chapter 19, "Microprocessor Software Specification (MSS)".

Note: The EDK offers a RevUp tool to convert any old MSS file format to a new MSS format. Please see Chapter 9, "Format Revision Tool" for more information.

Tool Usage

The Library Generator is run as follows:

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

Tool Options

The following options are supported in this version:

-h, -help (Help)

This option causes LibGen to display the usage menu and exit.

-v (Display version information)

This option displays the version number of LibGen.

-log *logfile[.log]*

This option specifies the log file. The default is libgen.log.

-p *family_name* (Architecture family)

This option defines the target architecture family. Use -h option to get a list of values for *Family_name*.

-od *output_dir* (Specify output directory)

This option 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 where libgen is invoked from is called ***USER_PROJECT*** for convenience in the documentation.

-sd *source_dir* (Specify source directory)

This option specifies the source directory *source_dir* for searching the input files (MHS). The default is the current working directory.

-lp *library_path* (Specify library path for user peripherals and drivers repositories)

This option specifies a library containing repositories of user peripherals, drivers, and libraries. LibGen looks for:

- drivers in the directory ***library_path*/<*sub_dir*>/drivers/**
- libraries in the directory ***library_path*/<*sub_dir*>/sw_services/**

Here ***<sub_dir>*** is a subdirectory under ***library_path***.

Please refer to the *Drivers* section of this document for more information on the search path for drivers.

-mhs *mhsfile.mhs* (Specify mhs file to be used)

This option specifies the mhsfile to be used for the libgen run. The following is the order used by libgen to find the name of an mhs file. The following is the order used by libgen to search to locate *mhsfile.mhs* for a run :

- Current working directory (***USER_PROJECT)***
- If no -mhs option is used, look in MSS file for parameter named HW_SPEC_FILE to get the mhsfilename.
- If no HW_SPEC_FILE parameter found in MSS file, use base name of mssfile (name without .mss extension) with .mhs extension as the mhsfilename.

-mode

Specifies the following modes for *all* processor instances in the MSS file.

`-mode executable`: This mode should be used if the user wants to generate a stand-alone executable program for all processor instances. The EXECUTABLE attribute in the MSS file is used in this mode. Note that in this mode, on-board debug support is not available. The MSS file should have the line

```
parameter EXECUTABLE = proc_inst_name/code/exec_file.elf
```
where the directory is relative to ***USER_PROJECT*** directory.

`-mode xmdstub`: (MicroBlaze only) This mode is used when user wants to use a debug stub for on-board debug. The xmdstub is created automatically for each processor instance in the MSS file by libgen as the file ***proc_inst_name*/code/xmdstub.elf**, relative to the ***OUTPUT_DIR*** directory

-xmdstub *proc_inst_name*

Note: Option valid for MicroBlaze only.

Specifies that the processor has its memory initialized with **xmdstub**s (debug stubs). Whereas the **-mode** option is a global option, applicable for all processors in the system, this option can be used to specify initialization modes for specific processor instances. When both **-mode** and **-xmdstub** options are used, the **-xmdstub** option takes precedence for that processor instance alone. Use multiple **-xmdstub** switches to set **xmdstub** mode for one or more processor instances.

-executable *proc_inst_name*

Similar in functionality for user executables as the **-xmdstub** option.

-lib

This option can be used to copy libraries and drivers but not compile them.

-stub

Creates the stub files (for MicroBlaze) and BRAM initialization script **bram_init.sh** . Using this option prevents the generation of libraries and drivers.

Load Path

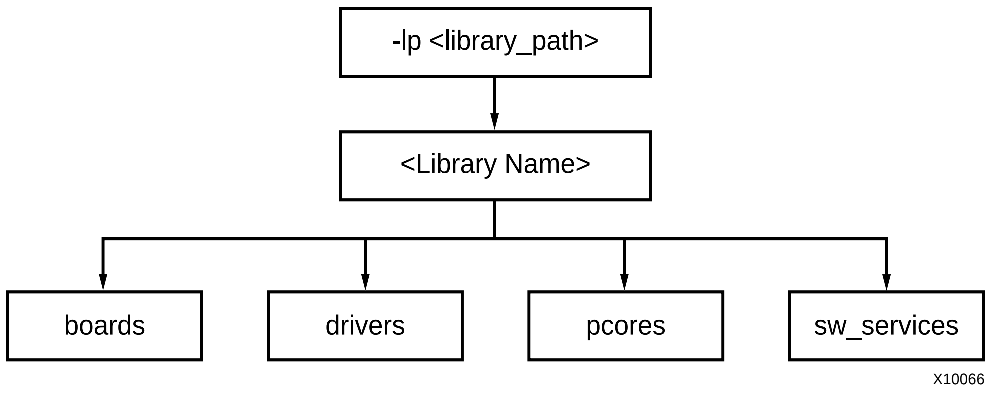


Figure 7-1: **Peripheral/Drivers directory structure**

Refer to Figure 7-1 and Figure 7-2 for a depiction of the drivers/libraries directory structure. On a UNIX system, system, the drivers/libraries reside in the following locations:

Drivers :

$XILINX_EDK/sw/<Library Name>/drivers

Libraries :

$XILINX_EDK / sw / <Library Name> / sw_services

On a PC, the drivers / libraries reside in the following location:

Drivers :

%XILINX_EDK% \ sw \ <Library Name> \ drivers

Libraries :

%XILINX_EDK% \ sw \ <Library Name> \ sw_services

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

- Current working directory where libgen was launched from.
- Set the EDK tool option **-lp** option. Libgen looks for drivers and libraries under each of the subdirectories of the path specified in -lp option.

Library Generator uses a search priority mechanism to locate drivers / libraries, as follows:

1. Searching the current working directory
   a. Drivers : Search for drivers inside *drivers* or *pcores* directory in the current working directory where libgen is invoked.
   b. Libraries : Search for libraries inside *sw_services* directory in the current working directory where libgen is invoked.
2. Searching the repositories under library path directory specified using **-lp** option
   a. Drivers : For drivers, search <library_path> / <Library Name> / drivers and <library_path> / <Library Name> / pcores (UNIX) or <library_path> \ <Library Name> \ drivers and <library_path> \ <Library Name> \ pcores (PC) as specified by the -lp option.
   b. Libraries :For Libraries, search <library_path> / <Library Name> / sw_services (UNIX) or <library_path> / <Library Name> \ sw_services (PC) as specified by the **-lp** option. Here <library_path> is the directory argument to **-lp** option and <Library Name> is a subdirectory under <library_path>.
3. Searching EDK install area.
   a. Drivers : Search $XILINX_EDK / sw / <Library Name> / drivers (UNIX) or %XILINX_EDK% \ sw \ <Library Name> \ drivers (PC)

b. Libraries : Search $XILINX_EDK/sw/<Library Name>/sw_services (UNIX) and %XILINX_EDK%\sw\<Library Name>\sw_services

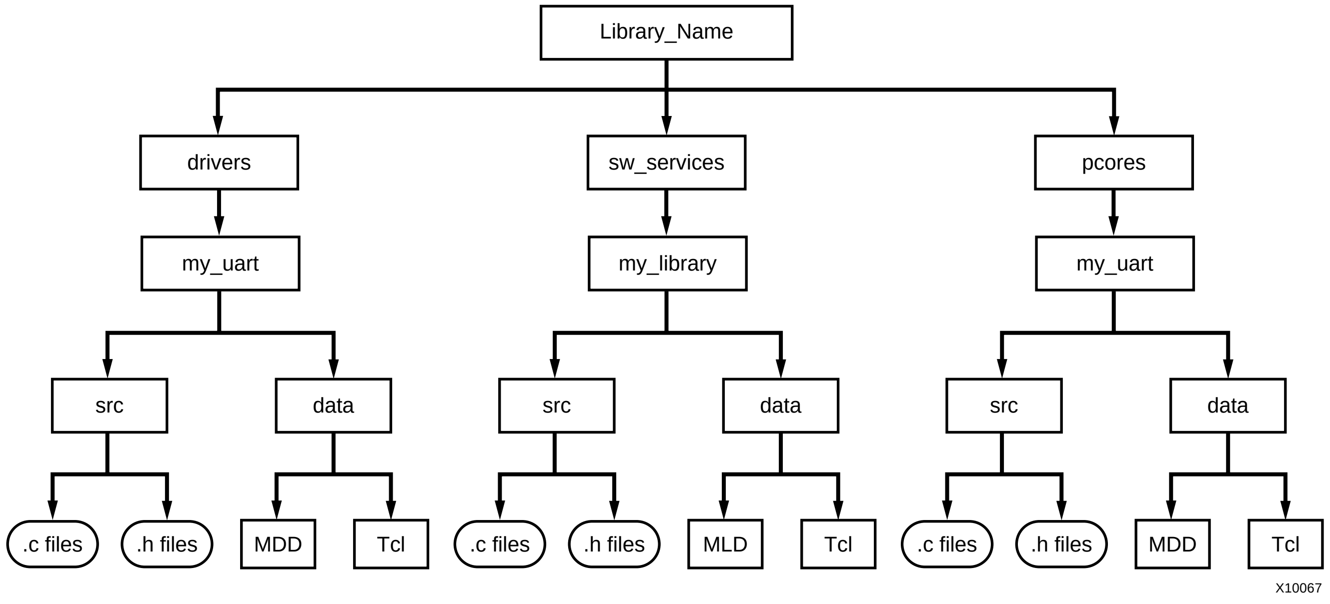


Figure 7-2: **Directory structure of Drivers and Libraries**

Output Files

Libgen generates directories and files in the ***USER_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.

include directory

The include directory contains C header files that are 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, libraries and user programs, and also function prototypes. The MDD file for each driver specifies the definitions that need to be customized for each peripheral that uses the driver. Please refer to Chapter 21, "Microprocessor Driver Definition (MDD)"for more information. The MLD file for each library specifies the definitions that need to be customized for each peripheral that uses the driver. Please refer to Chapter 20, "Microprocessor Library Definition (MLD)"for more information.

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. More information on the libraries can be found in Chapter 22, "Xilinx Microkernel (XMK)".

libsrc directory

The libsrc directory contains intermediate files and makefiles that are needed to compile the libraries and drivers. The directory contains peripheral specific driver files and library files that are copied from the EDK and user driver/library directories. Please refer the "Drivers" and "Libraries" section of this document for more information.

code directory

The code directory is used as a repository for EDK executables. Libgen creates xmdstub.elf (for MicroBlaze on-board debug) in this directory.

NOTE :

Libgen removes all the above directories everytime the tool is run. Users have to put in their sources/executables or any other files in a user created area.

Libraries and Drivers Generation

This section gives the basic philosophy of library and drivers generation.

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

- Build the directory structure as defined in the "Output Files" section of this document.
- Perform processor specific rule checks on the system. Currently LibGen checks if a debug peripheral is specified in the xmdstub mode for MicroBlaze. In future, checks will be expanded to include any processor specific DRC to be run.
- Copies the necessary source files for the drivers/libraries into the processor instance specific area : ***`OUTPUT_DIR/processor_instance_name/`*****`libsrc`**
- Copies the necessary BSP (Board Specific Package) files based on the Parameter OS defined in MSS for the processor
- Calls the design rule check (defined as an option in the MDD/MLD file) procedure for each of the drivers, libraries visible to the processor.
- Calls the **generate** Tcl procedure (if defined in the Tcl file associated with an MDD/MLD) for each of the drivers/libraries visible to the processor. This generates the necessary configuration files for each of the drivers/libraries in the include directory of the processor.
- Calls the **post_generate** Tcl procedure (if defined in the Tcl file associated with an MDD/MLD) for each of the drivers/libraries visible to the processor.
- Runs **make** (with targets **include** and **libs**) for the bsp's, drivers, libraries specific to the processor.
- Calls the **execs_generate** Tcl procedure (if defined in the Tcl file associated with an MDD) for each of the drivers/libraries visible to the processor.

MDD/MLD and Tcl

A Driver/Library has two data files associated with it. They are :

- Data Definition File (MDD/MLD) - This file defines the configurable parameters for the driver/library.
- Data Generation File (Tcl) : This file uses the parameters configured in the MSS file for a driver/library to generate data. Data generated includes but not limited to generation of header files, C files, running DRCs for the driver/library and generating executables. The Tcl file includes procedures that are called by Libgen at various stages of its execution. Various procedures in a Tcl file includes **DRC** (name of DRC given in the MDD/MLD file), **generate** (Libgen defined procedure) called after files are copied, **post_generate** (Libgen defined procedure) called after **generate** has been called on all drivers and libraries, **execs_generate** (Libgen defined procedure) called after the libraries and drivers have been generated.

Note that a driver/library need not have the data generation file (Tcl file).

For more information about the Tcl procedures and MDD/MLD related parameters refer to chapters Chapter 21, "Microprocessor Driver Definition (MDD)" and Chapter 20, "Microprocessor Library Definition (MLD)".

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, please refer Chapter 19, "Microprocessor Software Specification (MSS)"

Drivers

Most peripherals require software drivers. The EDK peripherals are shipped with associated drivers, libraries and BSPs. Please refer Chapter 28, "Device Drivers" for more information on driver functions.

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 LEVEL is also specified depending on the driver functionality required. The driver directory contains C source and header files for each level of drivers and a makefile for the driver.

A Driver has an MDD file and/or a Tcl file associated with it. The MDD file for the driver specifies all configurable parameters for the drivers. This is the data definition file. Each MDD file has a corresponding Tcl file associated with it. This Tcl file generates data that includes generation of header files, generation of C files, running DRCs for the driver and generating executables. Please refer Chapter 21, "Microprocessor Driver Definition (MDD)" and Chapter 19, "Microprocessor Software Specification (MSS)" for more information.

Users can write their own drivers. These drivers must be in a specific directory under ***`USER_PROJECT`*****`/drivers`** or ***`library_name`*****`/drivers`** as shown in Figure 7-1. The DRIVER_NAME attribute allows the user to specify any name for their drivers, which is also the name of the driver directory. The source files and makefile for the driver must be in the **src/** subdirectory under the *driver_name* directory. The makefile should have the targets "include" and "libs". Each driver must also contain an MDD file and a Tcl file in the **data/** subdirectory. Please refer to the existing EDK drivers to get an understanding of the structure of the drivers. Please referto Chapter 21, "Microprocessor Driver Definition (MDD)" for details on how to write an MDD and its corresponding Tcl file.

Libraries

The MSS file now 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. The library directory contains C source and header files and a makefile for the library.

The MLD file for each driver specifies all configurable options for the drivers. Each MLD file has a corresponding Tcl file associated with it. Please refer Chapter 20, "Microprocessor Library Definition (MLD)" and Chapter 19, "Microprocessor Software Specification (MSS)" for more information.

Users can write their own libraries. These libraries must be in a specific directory under ***`USER_PROJECT/`***`sw_services` or ***`library_name/`***`sw_services` as shown in Figure 7-1. The LIBRARY_NAME attribute allows the user to specify any name for their libraries, which is also the name of the library directory. The source files and makefile for the library must be in the **src/** subdirectory under the *library_name* directory. The makefile should have the targets "include" and "libs". Each library must also contain an MLD file and a Tcl file in the **data/** subdirectory. Please refer to the existing EDK libraries to get an understanding of the structure of the libraries. Please referChapter 20, "Microprocessor Library Definition (MLD)" for details on how to write an MLD and its corresponding Tcl file.

Interrupts and Interrupt Controller

An interrupt controller peripheral must be instantiated if the MHS file has multiple interrupt ports connected. When **Level 0** interrupt controller driver is used, libgen statically configures interrupts and interrupt handlers through the Tcl file for interrupt controller. When the **Level 1** driver are used, the user is responsible for registering interrupt handlers and enabling interrupts for the peripherals in the user code.

Level 0 interrupt controller driver customization

In the MSS file, the INT_HANDLER parameter allows an interrupt handler routine to be associated with the interrupt signal. Interrupt Controller's Tcl file uses this parameter to configure the interrupt controller handler to call the appropriate peripheral handlers on an interrupt. The functionality of these handler routines is left to the user to implement. If the INT_HANDLER parameter is not specified, a default dummy handler routine for the peripheral is used.

For MicroBlaze, if there is only one interrupt driven peripheral, an interrupt controller need not be used. However, the peripheral should still have an interrupt handler routine specified. Otherwise a default one is used.

When the processor to which the interrupt controller is connected is MicroBlaze, and the compiler used to compile drivers is **mb-gcc**, the Tcl file associated with the MicroBlaze driver MDD, designates the interrupt controller handler as the main interrupt handler. For the PowerPC processor, the user is responsible for setting up the exception table. Please refer Chapter 31, "Interrupt Management" for more information.

Debug Peripherals (MicroBlaze Specific)

These are peripherals that are specifically used to download xmdstub. The attribute DEBUG_PERIPHERAL is used for denoting the debug peripheral instance. Libgen uses this attribute in xmdstub mode.

STDIN and STDOUT Peripherals

Peripherals that handle I/O need drivers to access data. Two files **`inbyte.c`** and **`outbyte.c`** are automatically generated with calls to the driver I/O functions for STDIN and STDOUT peripherals. The driver I/O functions are specified in the MDD as the parameters INBYTE and OUTBYTE. These inbyte and outbyte functions are used by C library functions like *scanf* and *printf*. The peripheral instance should be specified as STDIN or STDOUT in the MSS file.Note that the inbyte and outbyte functions are generated only when the STDIN and STDOUT peripheral uses a level 0 driver. A level 1 driver for such peripherals would mean that the user is providing these functions. Please refer to Chapter 21, "Microprocessor Driver Definition (MDD)" for more information.

Chapter 8

Platform Specification Utility

Summary

This document describes the various features and the usage of the Platform Specification Utility (PsfUtil) tool that enables automatic generation of Microprocessor Peripheral Description files (MPD) required to create an IP core compliant with the Embedded Development Kit (EDK). Many of these features may be used with the help of wizards in the Xilinx Platform Studio (XPS) GUI tool.

The following topics are covered in this document:

- Tool Options
- Overview of the automatic MPD creation process.
- Detailed use case scenarios for generating MPDs
- About specification of VHDL attributes to enable generation of complete MPDs.
- DRC checks performed by the tool to enable delivery of verified MPDs
- Verilog support
- VHDL Peripherals Definition Section detailing signal naming conventions and PSF attribute syntax to be specified in VHDL for automatic MPD generation

Tool Options

-h

Display Usage

-v

Display version

-hdl2mpd <hdlfile>

Generate MPD from VHDL / Ver src / prj file.

Sub-options:

-lang <ver | vhdl | pao>

Specify language

-top <design>

Specify top level entity / module name

{-bus <opb | plb | dcr | lmb> <m | s | ms>}

Specify one or more Bus Interfaces of the core

{-tbus <transparent_bus_name> bram_port}

Specify one or more Transparent Bus Interfaces of the core

-o <outfile>

Specify output filename, Default : stdout

-pao2mpd <paofile>

Generate MPD from Peripheral Analyze Order (PAO) file.

Sub-options:

-lang <ver | vhdl | pao>

Specify language

-top <design>

Specify top level entity / module name

{-bus <opb | plb | dcr | lmb> <m | s | ms>}

Specify one or more Bus Interfaces of the core

{-tbus <transparent_bus_name> bram_port}

Specify one or more Transparent Bus Interfaces of the core

-o <outfile>

Specify output filename, Default : stdout

Overview of the MPD Creation Process

PsfUtility may be used automatically create MPD specifications from the VHDL specification of the core. The steps involved to create a core and deliver it through EDK are

- Code the IP in VHDL or Verilog using strict naming conventions for all Bus signals, Clock signals, Reset signals and Interrupt signals. These naming conventions are described in detail in VHDL IP Peripheral Guide. ***Following these naming conventions will enable PsfUtility create a correct and complete MPD.***
- In the top-level entity declaration, add additional attributes to specify special attributes on the entity, parameters and ports. These attributes would be translated by PsfUtility into appropriate MPD syntax. ***Not providing these additional attributes might sometimes result in the generation of an MPD that is syntactically correct but does not achieve the desired implementation.*** More information on specification of attributes and most commonly specified attributes is described in the Section "About Specification of VHDL attributes".
- Create an XST project file or a Peripheral Analyze Order (PAO) file that lists all the HDL sources required to implement the IP. Invoke PsfUtility by providing the XST project file or the PAO file with additional options. More information on invoking PsfUtility with different options is provided in the Section "Detailed Use Models for Automatic MPD Creation".

Detailed Use Models for Automatic MPD Creation

PsfUtility may be invoked in a variety of ways depending on the bus standard and type of bus interfaces of the peripheral and the number of bus interfaces a peripheral contains. Bus standards and types may be one of

- OPB SLAVE
- OPB MASTER
- OPB MASTER_SLAVE
- PLB SLAVE
- PLB MASTER
- PLB MASTER_SLAVE
- DCR SLAVE
- LMB SLAVE
- TRANSPARENT BUS (special case)

Peripherals with a Single Bus Interface

Majority of processor peripherals fall into this category. This is also the simplest usage model for PsfUtility. For most peripherals, complete MPD specifications can be obtained without specification of any additional attributes in the source code.

Signal Naming Conventions

The signal names must follow conventions as specified in the VHDL Peripheral Description Guide. Since there is only one bus interface, no *bus identifier* needs to be specified for the bus signals.

Invoking 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 OPB SLAVE peripheral, say uart, the command would be

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

Peripherals with Multiple Bus Interfaces

Some peripherals may have multiple bus interfaces associated with it. These interfaces may be Exclusive bus interfaces or Non-exclusive bus interfaces or a combination of both. All bus interfaces of the peripheral that can be connected to the peripheral at the same time are exclusive interfaces. For example, an OPB Slave bus interface and a DCR Slave bus interface are exclusive bus interfaces on a peripheral as they can both be connected at the same time. ***Peripherals with exclusive bus interfaces CAN NOT have any ports that can be connected to more than one of the exclusive interfaces.***

Non-exclusive bus interfaces are those interfaces that cannot be connected at the same time. ***Peripherals with non-exclusive bus interfaces WILL HAVE ports that can be connected to more than one of the non-exclusive interfaces. Further, non-exclusive interfaces WOULD have the same bus interface standard.*** For example, an OPB Slave interface and a OPB Master Slave interface are non-exclusive if they are connected to the same slave ports of the peripheral.

Non-Exclusive Bus Interfaces

Signal Naming Conventions

The signal names must follow conventions as specified in the VHDL Peripheral Description Guide. For non-exclusive bus interfaces, *bus identifiers* need not be specified for the bus signals.

Invoking PsfUtility with buses specified in command line

Buses can be specified on the command line when the bus signals are not prefixed with bus identifiers. The command line for invoking PsfUtil 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 a peripheral with a PLB slave interface and a PLB Master Slave interface, say gemac, the command would be

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

Invoking PsfUtility with buses specified as attributes

When the bus signals of the peripheral are prefixed with bus identifiers, a special BUSID attribute must be specified as defined in the VHDL Peripheral Definition document. PsfUtility may then be invoked without specifying the buses in the command line.

Exclusive Bus Interfaces

Signal Naming Conventions

The signal names must follow conventions as specified in the VHDL Peripheral Description Guide. B*us identifiers* need to be specified only when the peripheral has more than one bus interface of the same bus standard and type.

Invoking PsfUtility with buses specified in command line

Buses can be specified on the command line when the bus signals are not prefixed with bus identifiers. The command line for invoking PsfUtil 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 a peripheral with a PLB slave interface and a DCR Slave interface, the command would be

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

Invoking PsfUtility with buses specified as attributes

When the bus signals of the peripheral are prefixed with bus identifiers, a special BUSID attribute must be specified as defined in the VHDL Peripheral Definition document

Peripherals with TRANSPARENT Bus Interfaces

Some peripherals like bram controllers might have transparent bus interfaces (BUS_STD=TRANSPARENT, BUS_TYPE = UNDEF).

BRAM PORTS

To add a transparent BRAM bus interface to your core, invoke psfutil with an additional -tbus option

```
psfutil -hdl2mpd bram_ctlr.prj -lang vhdl -top bram_ctlr -bus opb s -tbus PORTA bram_port
```

Note that the BRAM ports should follow signal naming conventions as specified in the VHDL Peripheral Definition document.

About Specification of VHDL Attributes

The MPD format of EDK consists of additional sub-properties that are required for successful Platform Generation. Many of these sub-properties are automatically inferred by PsfUtility from the HDL specification (provided the HDL followed the naming conventions as specified in the VHDL Peripheral Definition Section of this document).

Global IP Core Options

All core options have default values generated by PsfUtility. These may be overridden by specifying attributes in the source VHDL.

IMP_NETLIST

When not specified as an attribute, the IMP_NETLIST core option is automatically created by PsfUtility with a value “FALSE”.

IPTYPE

When not specified as an attribute, the IPTYPE core option is automatically created by PsfUtility with a value “PERIPHERAL”.

IP_GROUP (User MUST specify)

When not specified as an attribute, the IP_GROUP core option is automatically created by PsfUtility with a value “USER”. The allowed values are “LOGICORE”, “INFRASTRUCTURE”, “REFERENCE”, “ALLIANCE” and “USER”.

HDL

This value is automatically created by PsfUtility and set to the language of the source

SPECIAL (User MUST specify if required)

Specify any SPECIAL attributes on the core.

ALERT (User MUST specify if required)

Specify any ALERT statements that need to be printed when the software tools process the cores. The ALERT statements is a string that can have “\n” characters to specify newline.

Properties on Ports

Port Value

When signal naming conventions are followed, PsfUtility automatically connects the bus signals to the appropriate bus connector.

DIR

This value is automatically generated by PsfUtility.

VEC

This value is automatically generated by PsfUtility.

BUSIF

When signal naming conventions are followed, PsfUtility automatically associates a BUS with a port.

For transparent buses however, the BUSIF attribute **MUST BE SPECIFIED** for the port.

SIGIS (User MUST Specify)

All Clock ports that must be driven by a clock buffer must have the SIGIS attribute on the clock ports set to CLK.

All Interrupt ports that must have a SIGIS attribute on the interrupt ports set to INTR_LEVEL_HIGH, INTR_LEVEL_LOW, INTR_EDGE_RISING or INTR_EDGE_FALLING based on whether the Interrupt is a Level High, Level Low, Rising Edge or Falling Edge triggered interrupt.

IOB_STATE (User MUST Specify)

The IOB_STATE attribute must be specified if the port must be driven by an IO buffer or register. Valid values are REG, BUF, INFER.

THREE_STATE

All signals that have a signame_I, signame_O and signame_T names specified in the HDL are automatically inferred as tristate signals by PsfUtility. Note that in order to propagate other attributes (like say IOB_STATE on signame), these attributes must be specified on the "_I" signal. The ENABLE=MULTI is automatically inferred based on the size of the signame_T signal.

ENDIAN (User MUST specify)

For all signals that are little endian, but cannot be automatically inferred as little endian, the user must specify the endian attribute on ports. When the range of ports cannot be resolved (both left and right range are the same, or the ranges are based on parameters), PsfUtility cannot resolve the Endianess automatically. For these kinds of ports, the endianess must be specified as LITTLE if the port is little endian.

Properties on Parameters

MIN_SIZE (for Address parameter - User MUST specify)

Specifies the minimum size in words of the peripheral address space

ADDRESS (for address parameter)

ADDRESS can take the values BASE or HIGH or NONE. Specifies whether parameter is base or high address or not an address at all. All parameters ending with _BASEADDR will be assigned ADDRESS=BASE. All parameters ending with _HIGHADDR will be assigned ADDRESS=HIGH. If it a parameter ends with _BASEADDR or _HIGHADDR but is not an address of the core, user MUST specify attribute indicating ADDRESS = NONE (the suggested method is to not have non core address parameters ending with _BASEADDR or _HIGHADDR).

BUS (for address parameter)

The address parameter MUST follow naming conventions for the BUS to be automatically generated by PsfUtility. The BUS attribute indicates which bus interfaces the address corresponds to. Please refer to the VHDL Peripheral definition section for details on address parameter naming conventions.

DRC Checks in PsfUtility

The following DRC errors are reported by PsfUtility to enable generation of correct and complete MPDs from HDL sources. The DRC checks are listed in the order that the checks are performed.

HDL Source Errors

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

Attribute Specification Errors

All PSF specific attributes are defined in the VHDL Peripheral Definition Section for valid values. Wrongly specified values are flagged as errors.

Bus Interface Checks

Given the list of bus interface of the cores, PsfUtility verifies the following

- Check and report any missing Bus Signals for every specified bus interface
- Check and report any repeated Bus Signals for every specified bus interface

PsfUtility will not generate an MPD unless all bus interface checks are completed.

Verilog Language Support

PsfUtility supports Verilog language as well. Currently, there exists no means for specifying attributes of ports / parameters in Verilog.

VHDL Peripheral Definitions

The top-level VHDL source file for an IP core defines the interface of the design. The VHDL source file has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters (generics) and default values
- Any VHDL source parameter is overwritten by the equivalent MHS assignment

Individual peripheral documentation contains information on all source file options.

VHDL Syntax

VHDL file syntax is case insensitive.

The VHDL file is supplied by the IP provider and provides peripheral information. This file lists ports and default connectivity to the bus interface. Parameters that you set in this file are used to automatically generate the MPD file for the platform generation tools.

Comments

The standard VHDL comment characters are used to separate comments from VHDL code. Double dashes, "--", are the VHDL comment separator. No IP core definition information will be included in the comments.

Format

Standard VHDL syntax is used to specify the peripheral ports and generics. Additional information required for automated system generation is added as attributes to the core's top-level entity, ports, and generics.

Bus Interface Naming Conventions

A bus interface is a grouping of interface signals which are related. For the automation tools to function properly, certain conventions must be adhered to in the naming of the signals and parameters associated with a bus interface. When the signal naming conventions are followed, the following interface types will be automatically recognized and the MPD file will contain the BUS_INSTANCE label shown in Table 8-1.

Table 8-1: **Recognized Bus Interfaces**

Description	Bus label in MPD
Slave DCR interface	SDCR
Slave LMB interface	SLMB
Master OPB interface	MOPB
Master / slave OPB interface	MSOPB
Slave OPB interface	SOPB
Master PLB interface	MPLB
Master / slave PLB interface	MSPLB
Slave PLB interface	SPLB

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

Naming Conventions for VHDL Generics

A key concept for cores with more than one bus interface port is the use of a *bus identifier*, which is attached to all signals grouped together in a port as well as the generics that are associated with the bus interface port. The bus identifier is discussed below.

Generic names must be VHDL compliant. As with any language, VHDL has certain naming rules and conventions that you must follow. Additional conventions for IP cores are:

- The generic must start with "C_".
- If more than one instance of a particular bus interface type is used on a core, a bus identifier, <*BI*>, must be used in the signal identifier and a corresponding BUSID attribute must be defined for the entity. If a bus identifier is used for the signals associated with a port, then the generics associated with that port may also optionally use the <*BI*>. If no <*BI*> string is used in the name, then 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.
- For cores 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 will not typically be included. If the bus identifier is used, a corresponding BUSID attribute must be used on the entity as well.
- All generics that specify a base address must end with _BASEADDR, and all generic that specify a high address must end with _HIGHADDR. Further, to tie these addresses with buses, these must also follow the conventions for parameters as listed above. For peripherals with more than one type of bus interface, the parameters must have the bus standard type specified in the name. For example, an address on the PLB bus must be specified as C_PLB_BASEADDR and C_PLB_HIGHADDR.

The Platform Generator automatically expands and populates certain reserved generics. In order for this to work correctly, a bus tag must be associated with these parameters. In order to have PsfUtility automatically infer this information, all the above specified conventions must be followed for all reserved generics as well. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved generic names:

Figure 8-1: **Automatically Expanded Reserved Generics**

Parameter	Description
C_BUS_CONFIG	Bus Configuration of MicroBlaze
C_FAMILY	FPGA Device Family
C_INSTANCE	Instance name of component
C_KIND_OF_EDGE	Vector of edge sensitive (rising / falling) of interrupt signals
C_KIND_OF_LVL	Vector of level sensitive (high / low) of interrupt signals
C_KIND_OF_INTR	Vector of interrupt signal sensitivity (edge / level)
C_NUM_INTR_INPUTS	Number of interrupt signals
C_<*BI*>OPB_NUM_MASTERS	Number of OPB masters
C_<*BI*>OPB_NUM_SLAVES	Number of OPB slaves
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*>LMB_AWIDTH	LMB Address width

Figure 8-1: **Automatically Expanded Reserved Generics**

Parameter	Description
C_<*BI*>LMB_DWIDTH	LMB Data width
C_<*BI*>LMB_NUM_SLAVES	Number of LMB slaves
C_<*BI*>OPB_AWIDTH	OPB Address width
C_<*BI*>OPB_DWIDTH	OPB Data width
C_<*BI*>OPB_NUM_MASTERS	Number of OPB masters
C_<*BI*>OPB_NUM_SLAVES	Number of OPB slaves
C_<*BI*>PLB_AWIDTH	PLB Address width
C_<*BI*>PLB_DWIDTH	PLB Data width
C_<*BI*>PLB_MID_WIDTH	PLB master ID width
C_<*BI*>PLB_NUM_MASTERS	Number of PLB masters
C_<*BI*>PLB_NUM_SLAVES	Number of PLB slaves

Reserved Parameters

C_BUS_CONFIG

The C_BUS_CONFIG parameter defines the bus configuration of the MicroBlaze processor. This parameter is automatically populated by Platform Generator.

C_FAMILY

The C_FAMILY parameter defines the FPGA device family. This parameter is automatically populated by Platform Generator.

C_INSTANCE

The C_INSTANCE parameter defines the instance name of the component. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_MASTERS

The C_OPB_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_SLAVES

The C_OPB_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_DCR_AWIDTH

The C_DCR_AWIDTH parameter defines the DCR address width. This parameter is automatically populated by Platform Generator.

C_DCR_DWIDTH

The C_DCR_DWIDTH parameter defines the DCR data width. This parameter is automatically populated by Platform Generator.

C_DCR_NUM_SLAVES

The C_DCR_NUM_SLAVES parameter defines the number of DCR slaves on the bus. This parameter is automatically populated by Platform Generator.

C_LMB_AWIDTH

The C_LMB_AWIDTH parameter defines the LMB address width. This parameter is automatically populated by Platform Generator.

C_LMB_DWIDTH

The C_LMB_DWIDTH parameter defines the LMB data width. This parameter is automatically populated by Platform Generator.

C_LMB_NUM_SLAVES

The C_LMB_NUM_SLAVES parameter defines the number of LMB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_AWIDTH

The C_OPB_AWIDTH parameter defines the OPB address width. This parameter is automatically populated by Platform Generator.

C_OPB_DWIDTH

The C_OPB_DWIDTH parameter defines the OPB data width. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_MASTERS

The C_OPB_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_SLAVES

The C_OPB_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_PLB_AWIDTH

The C_PLB_AWIDTH parameter defines the PLB address width. This parameter is automatically populated by Platform Generator.

C_PLB_DWIDTH

The C_PLB_DWIDTH parameter defines the PLB data width. This parameter is automatically populated by Platform Generator.

C_PLB_MID_WIDTH

The C_PLB_MID_WIDTH parameter defines the PLB master ID width. This is set to log2(S). This parameter is automatically populated by Platform Generator.

C_PLB_NUM_MASTERS

The C_PLB_NUM_MASTERS parameter defines the number of PLB masters on the bus. This parameter is automatically populated by Platform Generator.

C_PLB_NUM_SLAVES

The C_PLB_NUM_SLAVES parameter defines the number of PLB slaves on the bus. This parameter is automatically populated by Platform Generator.

Signal Naming Conventions

This section provides naming conventions for bus interface signal names. These conventions are flexible to accommodate embedded processor systems that have more than one bus interface and more than one bus interface port per component. A key concept for cores with more than one bus interface port is the use of a *bus identifier*, which is attached to all signals grouped together in a port as well as the parameters that are associated with the bus interface port. The bus identifier is discussed below.

The names must be VHDL compliant. As with any language, VHDL has certain naming rules and conventions that you must follow. Additional conventions for IP cores 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 type of bus interface.
- If more than one instance of a particular bus interface type is used on a core, a bus identifier, <*BI*>, must be used 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. The <*BI*> must be included in the port's signal identifiers in the following cases:
  - The core has more than one slave PLB port.
  - The core has more than one master PLB port.
  - The core has more than one slave LMB port.
  - The core has more than one slave DCR port.
  - The core has more than one master DCR port.
  - The core has more than one OPB port of any type (master, slave, or master / slave).
  - The core has more than one port of any type and the choice of <*Mn*> or <*Sln*> causes ambiguity in the signal names. For example, a core with both a master OPB port and master PLB port and the same <*Mn*> string for both ports would require a <*BI*> string to differentiate the ports since the address bus signal would be ambiguous without <*BI*>.

For cores 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 will not typically be 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).

LMB - Clock and Reset

```
LMB_Clk
LMB_Rst
```

OPB - Clock and Reset

```
OPB_Clk
OPB_Rst
```

PLB - Clock and Reset

```
PLB_Clk
PLB_Rst
```

Slave DCR Ports

Slave DCR ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, "DCR" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nDCR*> is a meaningful name or acronym for the slave input. The last three characters of <*nDCR*> must contain the string, "DCR" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single slave DCR port, and required for peripherals with multiple slave DCR ports. <*BI*> must not contain the string, "DCR" (upper or lower case or mixed case). For peripherals with multiple slave DCR ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

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 LMB Ports

Slave LMB ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, "LMB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nLMB*> is a meaningful name or acronym for the slave input. The last three characters of <*nLMB*> must contain the string, "LMB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single slave LMB port, and required for peripherals with multiple slave LMB ports. <*BI*> must not contain the string, "LMB" (upper or lower case or mixed case). For peripherals with multiple slave LMB ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

LMB Slave Outputs

For interconnection to the LMB, all 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, all 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 OPB Ports

The signal list shown below applies to master OPB ports that are independent of slave OPB ports. For the signal list for cores that use a combined master / slave bus interface, see XXX.

Master OPB ports must follow these naming conventions:

- <*Mn*> is a meaningful name or acronym for the master output. <*Mn*> must not contain the string, "OPB" (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.
- <*nOBP*> is a meaningful name or acronym for the master input. The last three characters of <*nOPB*> must contain the string, "OPB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single OPB port (of any type), and required for peripherals with multiple OPB ports (of any type or mix of types). <*BI*> must not contain the string, "OPB" (upper or lower case or mixed case). For peripherals with multiple OPB ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Mn*> is optional.

OPB Master Outputs

For interconnection to the OPB, all masters must provide the following outputs:

```
<BI><Mn>_ABus     : out std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><Mn>_BE       : out std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><Mn>_busLock  : out std_logic;
<BI><Mn>_DBus     : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Mn>_request  : out std_logic;
<BI><Mn>_RNW      : out std_logic;
<BI><Mn>_select   : out std_logic;
<BI><Mn>_seqAddr  : out std_logic;
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

OPB Master Inputs

For interconnection to the OPB, all masters must provide the following inputs:

```
<BI><nOPB>_Clk     : in  std_logic;
<BI><nOPB>_DBus    : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_errAck  : in  std_logic;
<BI><nOPB>_MGrant  : in  std_logic;
<BI><nOPB>_retry   : in  std_logic;
<BI><nOPB>_Rst     : in  std_logic;
<BI><nOPB>_timeout : in  std_logic;
<BI><nOPB>_xferAck : in  std_logic;
```

Examples:

```
IOPB_DBus     : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
OPB_DBus      : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
Bus1_OPB_DBus : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Slave OPB Ports

The signal list shown below applies to master OPB ports that are independent of slave OPB ports. For the signal list for cores that use a combined master / slave bus interface, see XXX.

Slave OPB ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, "OPB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nOPB*> is a meaningful name or acronym for the slave input. The last three characters of <*nOPB*> must contain the string, "OPB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single OPB port, and required for peripherals with multiple OPB ports (of any type). <*BI*> must not contain the string, "OPB" (upper or lower case or mixed case). For peripherals with multiple OPB ports (of any type or mix of types), the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

OPB Slave Outputs

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

```
<BI><Sln>_DBus    : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Sln>_errAck  : out std_logic;
<BI><Sln>_retry   : out std_logic;
<BI><Sln>_toutSup : out std_logic;
<BI><Sln>_xferAck : out std_logic;
```

Examples:

```
Tmr_xferAck   : out std_logic;
Uart_xferAck  : out std_logic;
Intc_xferAck  : out std_logic;
```

OPB Slave Inputs

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

```
<BI><nOPB>_ABus    : in  std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><nOPB>_BE      : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><nOPB>_Clk     : in  std_logic;
<BI><nOPB>_DBus    : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_Rst     : in  std_logic;
<BI><nOPB>_RNW     : in  std_logic;
<BI><nOPB>_select  : in  std_logic;
<BI><nOPB>_seqAddr : in  std_logic;
```

Examples:

```
OPB_DBus       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
IOPB_DBus      : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
Bus1_OPB_DBus  : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Master/Slave OPB Ports

The signal list shown below applies to master / slave type OPB ports that attach to the same OPB bus and share the input and output data buses. This type of bus interface is typically used when a peripheral has both master and slave functionality (typical when DMA is included with the peripheral) and it is advantageous for the master and slave to share the input and output data buses.

Master / Slave OPB ports must follow these naming conventions:

- <*Mn*> is a meaningful name or acronym for the master output. <*Mn*> must not contain the string, “OPB” (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.
- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, “OPB” (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nOPB*> is a meaningful name or acronym for the slave input. The last three characters of <*nOPB*> must contain the string, “OPB” (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single OPB port, and required for peripherals with multiple OPB ports (of any type). <*BI*> must not contain the string, “OPB” (upper or lower case or mixed case). For peripherals with multiple OPB ports (of any type or mix of types), the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> and <*Mn*> are optional.

OPB Master/Slave Outputs

For interconnection to the OPB, all master / slaves must provide the following outputs:

```
<BI><Mn>_ABus     : out std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><Mn>_BE       : out std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><Mn>_busLock  : out std_logic;
<BI><Mn>_request  : out std_logic;
<BI><Mn>_RNW      : out std_logic;
<BI><Mn>_select   : out std_logic;
<BI><Mn>_seqAddr  : out std_logic;
<BI><Sln>_DBus    : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Sln>_errAck  : out std_logic;
<BI><Sln>_retry   : out std_logic;
<BI><Sln>_toutSup : out std_logic;
<BI><Sln>_xferAck : out std_logic;
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

OPB Master/Slave Inputs

For interconnection to the OPB, all master / slaves must provide the following inputs:

```
<BI><nOPB>_ABus    : in  std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><nOPB>_BE      : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><nOPB>_Clk     : in  std_logic;
<BI><nOPB>_DBus    : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_errAck  : in  std_logic;
<BI><nOPB>_MGrant  : in  std_logic;
<BI><nOPB>_retry   : in  std_logic;
<BI><nOPB>_RNW     : in  std_logic;
<BI><nOPB>_Rst     : in  std_logic;
<BI><nOPB>_select  : in  std_logic;
<BI><nOPB>_seqAddr : in  std_logic;
<BI><nOPB>_timeout : in  std_logic;
<BI><nOPB>_xferAck : in  std_logic;
```

Examples:

```
IOPB_DBus      : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
OPB_DBus       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
Bus1_OPB_DBus  : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Master PLB Ports

Master PLB ports must follow these naming conventions:

- <*Mn*> is a meaningful name or acronym for the master output. <*Mn*> must not contain the string, "PLB" (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.
- <*nPLB*> is a meaningful name or acronym for the master input. The last three characters of <*nOPB*> must contain the string, "PLB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single master PLB port, and required for peripherals with multiple master PLB ports. <*BI*> must not contain the string, "PLB" (upper or lower case or mixed case). For peripherals with multiple master PLB ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Mn*> is optional.

PLB Master Outputs

For interconnection to the PLB, all masters must provide the following outputs:

```
<BI><Mn>_ABus      : out std_logic_vector(0 to C_<BI>PLB_AWIDTH-1);
<BI><Mn>_BE        : out std_logic_vector(0 to C_<BI>PLB_DWIDTH/8-1);
<BI><Mn>_RNW       : out std_logic;
<BI><Mn>_abort     : out std_logic;
<BI><Mn>_busLock   : out std_logic;
<BI><Mn>_compress  : out std_logic;
<BI><Mn>_guarded   : out std_logic;
<BI><Mn>_lockErr   : out std_logic;
<BI><Mn>_MSize     : out std_logic;
<BI><Mn>_ordered   : out std_logic;
<BI><Mn>_priority  : out std_logic_vector(0 to 1);
<BI><Mn>_rdBurst   : out std_logic;
<BI><Mn>_request   : out std_logic;
<BI><Mn>_size      : out std_logic_vector(0 to 3);
<BI><Mn>_type      : out std_logic_vector(0 to 2);
<BI><Mn>_wrBurst   : out std_logic;
<BI><Mn>_wrDBus    : out std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

PLB Master Inputs

For interconnection to the PLB, all masters must provide the following inputs:

```
<BI><nPLB>_Clk          : in  std_logic;
<BI><nPLB>_Rst          : in  std_logic;
<BI><nPLB>_AddrAck      : in  std_logic;
<BI><nPLB>_Busy         : in  std_logic;
<BI><nPLB>_Err          : in  std_logic;
<BI><nPLB>_RdBTerm      : in  std_logic;
```

```
<BI><nPLB>_RdDAck      : in  std_logic;
<BI><nPLB>_RdDBus      : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><nPLB>_RdWdAddr    : in  std_logic_vector(0 to 3);
<BI><nPLB>_Rearbitrate : in  std_logic;
<BI><nPLB>_SSize       : in  std_logic_vector(0 to 1);
<BI><nPLB>_WrBTerm     : in  std_logic;
<BI><nPLB>_WrDAck      : in  std_logic;
```

Examples:

```
IPLB_MBusy     : in  std_logic;
Bus1_PLB_MBusy : in  std_logic;
```

Slave PLB Ports

Slave PLB ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, “PLB” (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nPLB*> is a meaningful name or acronym for the slave input. The last three characters of <*nPLB*> must contain the string, “PLB” (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single slave PLB port, and required for peripherals with multiple slave PLB ports. <*BI*> must not contain the string, “PLB” (upper or lower case or mixed case). For peripherals with multiple PLB ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

PLB Slave Outputs

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

```
<BI><Sln>_addrAck     : out std_logic;
<BI><Sln>_MErr        : out std_logic_vector(0 to C_<BI>PLB_NUM_MASTERS-1);
<BI><Sln>_MBusy       : out std_logic_vector(0 to C_<BI>PLB_NUM_MASTERS-1);
<BI><Sln>_rdBTerm     : out std_logic;
<BI><Sln>_rdComp      : out std_logic;
<BI><Sln>_rdDAck      : out std_logic;
<BI><Sln>_rdDBus      : out std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><Sln>_rdWdAddr    : out std_logic_vector(0 to 3);
<BI><Sln>_rearbitrate : out std_logic;
<BI><Sln>_SSize       : out std_logic(0 to 1);
<BI><Sln>_wait        : out std_logic;
<BI><Sln>_wrBTerm     : out std_logic;
<BI><Sln>_wrComp      : out std_logic;
<BI><Sln>_wrDAck      : out std_logic;
```

Examples:

```
Tmr_addrAck  : out std_logic;
Uart_addrAck : out std_logic;
Intc_addrAck : out std_logic;
```

PLB Slave Inputs

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

```
<BI><nPLB>_Clk      : in  std_logic;
```

```
<BI><nPLB>_Rst      : in  std_logic;
<BI><nPLB>_ABus     : in  std_logic_vector(0 to C_<BI>PLB_AWIDTH-1);
<BI><nPLB>_BE       : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH/8-1);
<BI><nPLB>_PAValid  : in  std_logic;
<BI><nPLB>_RNW      : in  std_logic;
<BI><nPLB>_abort    : in  std_logic;
<BI><nPLB>_busLock  : in  std_logic;
<BI><nPLB>_compress : in  std_logic;
<BI><nPLB>_guarded  : in  std_logic;
<BI><nPLB>_lockErr  : in  std_logic;
<BI><nPLB>_masterID : in  std_logic_vector(0 to C_<BI>PLB_MID_WIDTH-1);
<BI><nPLB>_MSize    : in  std_logic_vector(0 to 1);
<BI><nPLB>_ordered  : in  std_logic;
<BI><nPLB>_pendPri  : in  std_logic_vector(0 to 1);
<BI><nPLB>_pendReq  : in  std_logic;
<BI><nPLB>_reqpri   : in  std_logic_vector(0 to 1);
<BI><nPLB>_size     : in  std_logic_vector(0 to 3);
<BI><nPLB>_type     : in  std_logic_vector(0 to 2);
<BI><nPLB>_rdPrim   : in  std_logic;
<BI><nPLB>_SAValid  : in  std_logic;
<BI><nPLB>_wrPrim   : in  std_logic;
<BI><nPLB>_wrBurst  : in  std_logic;
<BI><nPLB>_wrDBus   : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><nPLB>_rdBurst  : in  std_logic;
```

Examples:

```
PLB_size  : in  std_logic_vector(0 to 3);
IPLB_size : in  std_logic_vector(0 to 3);
DPLB_size : in  std_logic_vector(0 to 3);
```

Entity-level VHDL Attributes for Automation Support

Table 8-2: **Entity-level VHDL Attributes**

Attribute	Type	Values	Default	PsfUtil Automation	Definition
ADDR_SLICE	*integer*	-	X	-	Address slice of BRAM controller
AWIDTH	*integer*	-	X	-	Address width of BRAM controller
ALERT	*string*	-	X	-	Alert message
CORE_STATE	*string*	ACTIVE DEPRECATED OBSOLETE DEVELOPMENT	ACTIVE	-	Core state
BUSID	*string*	-	X	-	Bus Identifier string for cores using the optional <BI> as part of the bus signal names
DWIDTH	*integer*	-	X	-	Data width of BRAM controller

Table 8-2: **Entity-level VHDL Attributes**

Attribute	Type	Values	Default	PsfUtil Automation	Definition
HDL	*string*	BOTH VERILOG VHDL	VHDL	Input Language of source	HDL design availability.
IMP_NETLIST	*string*	TRUE FALSE	FALSE	TRUE	Synthesize HDL to a hardware implementation netlist
IPTYPE	*string*	BRIDGE BUS BUS_ARBITER IP PERIPHERAL PROCESSOR	IP	PERIPHERAL	Type of component
IP_GROUP	string	LOGICORE INFRASTRUCTURE REFERENCE ALLIANCE USER	USER	USER	Defines the logical grouping to which IP belongs.
NUM_WRITE_ENABLES	*integer*	-	X	-	Number of write enables of BRAM controller
SPECIAL	*string*	BRAM BRAM_CNTLR	X	-	Special class of components that require special handling
STYLE	*string*	BLACKBOX MIX HDL	HDL	-	Design style
TOP	*string*	-	X	-	Top-level name

Entity-level attributes are included in the entity declaration section, as shown in the example below:

```
entity OPB_Core is
  generic (
    C_BASEADDR           : std_logic_vector := X"2000_0000";
    C_HIGHADDR           : std_logic_vector := X"2000_FFFF"
    );
  port (
    -- OPB signals
    SOPB_Clk           : in  std_logic;
    SOPB_Rst           : in  std_logic;
    SOPB_ABus          : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    ...
```

```
        M_ABus              : out std_logic_vector(0 to C_OPB_AWIDTH-1);
        M_BE                : out std_logic_vector(0 to C_OPB_DWIDTH/8-1);
        M_busLock           : out std_logic;
        M_DBus              : out std_logic_vector(0 to C_OPB_DWIDTH-1);
        ...
        );

    attribute BUSID : string;
    attribute IMP_NETLIST : string;

    attribute BUSID       of OPB_Core:entity is "S:OPB_SLAVE,M:OPB_MASTER";
    attribute IMP_NETLIST of OPB_Core:entity is "TRUE";

    end entity OPB_Core;
```

ADDR_SLICE Attribute

The address slice position supported by the BRAM controller is specified by the ADDR_SLICE attribute.

Format

```
attribute ADDR_SLICE : integer;
attribute ADDR_SLICE of Peripheral:entity is 29;
```

Used only by components of SPECIAL=BRAM_CNTLR.

AWIDTH Attribute

The address width supported by the BRAM controller is specified by the AWIDTH attribute.

Format

```
attribute AWIDTH : integer;
attribute AWIDTH of Peripheral:entity is 32;
```

Used only by components of SPECIAL=BRAM_CNTLR.

ALERT Attribute

A message alert for the IP core is specified with the ALERT attribute. The character \n may be used as a newline character within the ALERT string.

Format

```
attribute ALERT : string;
attribute ALERT of Peripheral:entity is "This belongs to Xilinx.";
```

BUSID Attribute

The BUSID attribute is used to define all of the Bus Identifiers that are used in the signal list and generic list. Any bus that uses the <BI> field in the naming of its signals must have a corresponding BUSID attribute so that the signal names can be parsed correctly. The format of the BUSID string is:

"<BI$_1$>:<interface_type>[:<interface_type>][,<BI$_2$>:<interface_type>[:<interface_type>]"

where as many Bus Identifiers as required can be defined, each with multiple interface types (for signals that are shared between more than one interface). <interface_type> is one of:

DCR_SLAVE, LMB_SLAVE, OPB_SLAVE, PLB_SLAVE, OPB_MASTER, PLB_MASTER, or OPB_MASTER_SLAVE

Format

Examples:

```
attribute BUSID : string;
attribute BUSID of Peripheral:entity is "M:OPB_SLAVE";

attribute BUSID : string;
attribute BUSID of Peripheral:entity is "S:OPB_SLAVE:OPB_MASTER_SLAVE";

attribute BUSID : string;
attribute BUSID of Peripheral:entity is
"S:OPB_SLAVE:OPB_MASTER_SLAVE,M:OPB_MASTER";
```

The BUSID attribute *must* be used when a bus that is used by the core uses the optional <BI> field in the names associated with the bus. For example, the following signals are used to define a slave OPB connection:

```
<BI><nOPB>_ABus    : in  std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><nOPB>_BE      : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><nOPB>_Clk     : in  std_logic;
<BI><nOPB>_DBus    : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_Rst     : in  std_logic;
<BI><nOPB>_RNW     : in  std_logic;
<BI><nOPB>_select  : in  std_logic;
<BI><nOPB>_seqAddr : in  std_logic;

<BI><Sln>_DBus    : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Sln>_errAck  : out std_logic;
<BI><Sln>_retry   : out std_logic;
<BI><Sln>_toutSup : out std_logic;
<BI><Sln>_xferAck : out std_logic;
```

The <BI> field is optional if the core has only one OPB connection, but required if more than one OPB is present on the core. For example, if a core has two OPB ports named A and B, then the OPB signal list and BUSID would look like the following:

```
...
AOPB_ABus    : in  std_logic_vector(0 to C_AOPB_AWIDTH-1);
AOPB_BE      : in  std_logic_vector(0 to C_AOPB_DWIDTH/8-1);
AOPB_Clk     : in  std_logic;
AOPB_DBus    : in  std_logic_vector(0 to C_AOPB_DWIDTH-1);
AOPB_Rst     : in  std_logic;
AOPB_RNW     : in  std_logic;
AOPB_select  : in  std_logic;
AOPB_seqAddr : in  std_logic;

ASlave_DBus    : out std_logic_vector(0 to C_AOPB_DWIDTH-1);
ASlave_errAck  : out std_logic;
ASlave_retry   : out std_logic;
ASlave_toutSup : out std_logic;
ASlave_xferAck : out std_logic;
```

```
BOPB_ABus    : in  std_logic_vector(0 to C_BOPB_AWIDTH-1);
BOPB_BE      : in  std_logic_vector(0 to C_BOPB_DWIDTH/8-1);
BOPB_Clk     : in  std_logic;
BOPB_DBus    : in  std_logic_vector(0 to C_BOPB_DWIDTH-1);
BOPB_Rst     : in  std_logic;
BOPB_RNW     : in  std_logic;
BOPB_select  : in  std_logic;
BOPB_seqAddr : in  std_logic;

BSlave_DBus    : out std_logic_vector(0 to C_BOPB_DWIDTH-1);
BSlave_errAck  : out std_logic;
BSlave_retry   : out std_logic;
BSlave_toutSup : out std_logic;
BSlave_xferAck : out std_logic;
...
attribute BUSID : string;
attribute BUSID of Peripheral:entity is "A:OPB_SLAVE,B:OPB_SLAVE";
```

CORE_STATE Attribute

The state of the IP core is specified with the CORE_STATE attribute.

Format

```
attribute CORE_STATE : string;
attribute CORE_STATE of Peripheral:entity is "ACTIVE";
```

The following values are valid:

- ACTIVE - Core is active (full uninhibited use) by EDK. This is the default setting.
- DEPRECATED - Core is deprecated. EDK tools allow use of core, but issues a warning that the core is deprecated.
- OBSOLETE - Core is obsolete. EDK tools issue an error that this core is no longer valid.
- DEVELOPMENT - Core is in development and will be synthesized each time the platform generation tools are run (no cacheing of synthesis results).

DWIDTH Attribute

The data width supported by the BRAM controller is specified by the DWIDTH attribute.

Format

```
attribute DWIDTH : integer;
attribute DWIDTH of Peripheral:entity is 32;
```

Used only by components of SPECIAL=BRAM_CNTLR.

HDL Attribute

The HDL attribute lists the HDL availability. The design is either completely written in VHDL, or completely written in Verilog. The BOTH value signifies that design is available in VHDL or Verilog format. PsfUtility automatically inserts this attribute for a single language.

Format

```
attribute HDL : string;
attribute HDL of Peripheral:entity is "VHDL";
```

IMP_NETLIST Attribute

In hierarchal mode, this attribute directs the Platform Generator to write an implementation netlist file for the peripheral. In flatten mode, the IMP_NETLIST attribute is ignored since the entire system is synthesized. PsfUtility automatically inserts this attribute with a value set to TRUE if not otherwise specified.

Format

```
attribute IMP_NETLIST : string;
attribute IMP_NETLIST of Peripheral:entity is TRUE;
```

IPTYPE Attribute

The IPTYPE attribute lists defines the type of the component. PsfUtility automatically sets the value to PERIPHERAL if not otherwise specified.

Format

```
attribute IPTYPE : string;
attribute IPTYPE of Peripheral:entity is "PERIPHERAL";
```

The IPTYPE attribute can have the following values:

- BRIDGE - bridge component
- BUS - bus component
- BUS_ARBITER - combined bus and arbiter component
- IP - component that is detached from a bus
- PERIPHERAL - component that is attached to a bus
- PROCESSOR - processor component (MicroBlaze or PPC405)

IP_GROUP Attribute

The IP_GROUP attribute lists defines the logical grouping to which an IP belongs. PsfUtility automatically sets the value to USER if not otherwise specified.

Format

```
attribute IP_GROUP : string;
attribute IP_GROUP of Peripheral:entity is "LOGICORE";
```

The IP_GROUP attribute can have the following values:

- LOGICORE
- INFRASTRUCTURE
- REFERENCE
- ALLIANCE
- USER

- PROCESSOR - processor component (MicroBlaze or PPC405)

NUM_WRITE_ENABLES Attribute

The number of write enables supported by the BRAM controller is specified by the NUM_WRITE_ENABLES attribute.

Format

```
attribute NUM_WRITE_ENABLES : integer;
attribute NUM_WRITE_ENABLES of Peripheral:entity is 8;
```

For a byte-write 32-bit data memory, the NUM_WRITE_ENABLES = 4. For a byte-write 64-bit data memory, the NUM_WRITE_ENABLES = 8.

Used only by components of SPECIAL=BRAM_CNTLR.

SPECIAL Attribute

The SPECIAL attribute defines a class of components that require special handling.

Format

```
attribute SPECIAL : string;
attribute SPECIAL of Peripheral:entity is "BRAM_CNTLR";
```

This attribute is reserved for internal use only.

STYLE Attribute

The STYLE attribute defines the design composition of the peripheral.

If you have a mix of optimized hardware netlists and HDL files, you must specify the MIX value for the STYLE attribute. In this case, the PAO and BBD files are read by the Platform Generator.

If you have only HDL files, you must specify the HDL value for the STYLE attribute. In this case, only the PAO file is read by the Platform Generator.

Format

```
attribute STYLE : string;
attribute STYLE of Peripheral:entity is value;
```

Where *value* is BLACKBOX, MIX, or HDL. The default value is HDL.

Generic-level VHDL Attributes for Automation Support

Table 8-3: **Generic-level VHDL Attributes**

Attribute	Type	Values	Default	PsfUtil Automation	Definition
MIN_SIZE	*string*	2^n in hexadecimal notation	0	-	Minimum size address window; format is a string representing a C-style hexadecimal number.
RESERVED	*string*	TRUE FALSE	FALSE	-	Indicates that the generic is reserved for core use only and is not modifiable. The generic should not be included in the MPD file.
ADDRESS	*string*	BASE HIGH NONE	-	Automatic inference if generic naming conventions are followed	Indicates that the parameters reprents an address range, and specifies the type.
BUS	*string*	-	-	Automatic inference if generic naming conventions are followed	Indicates that parameters is a bus specific parameter
BRIDGE_TO	*string*	bus interface name	-	-	Associated with address parameters of bridge type cores. Specifies which bus interface the address bridges to.

MIN_SIZE Attribute

The minimum size address window of an address is specified by attaching the MIN_SIZE attribute to the corresponding C_BASEADDR generic. Note that in the attribute specification the type of C_BASEADDR is “constant”.

Format

```
entity Peripheral is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000"
    );
  port ( ... );
  attribute MIN_SIZE : string;
  attribute MIN_SIZE of C_BASEADDR:constant is "0x100";
end entity Peripheral;
```

ADDRESS Attribute

The address type of an address parameter is specified by attaching a ADDRESS attribute to the _BASEADDR or _HIGHADDR generic. The default ADDRESS attribute for all signals that end with _BASEADDR is BASE, and the default value for all signals that end with _HIGHADDR is HIGH. This is automatically inserted by PsfUtility.

Format

```
entity Peripheral is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000"
    );
  port ( ... );
  attribute ADDRESS : string;
  attribute ADDRESS of C_BASEADDR:constant is "BASE";
  attribute ADDRESS of C_HIGHADDR:constant is "HIGH";
end entity Peripheral;
```

Signal-level VHDL Attributes for Automation Support

Table 8-4: **Signal-level VHDL Attributes**

Attribute	Type	Values	Default	PsfUtil Automation	Definition
THREE_STATE	*string*	TRUE FALSE	X	Automatic inference based on signal naming conventions	3-state expansion (equivalent to the 3STATE parameter in MPD file)
IOB_STATE	*string*	BUF INFER REG	INFER	-	Identifies ports that instantiate or infer IOB primitives
SIGIS	*string*	CLK INTR_LEVEL_LOW INTR_LEVEL_HIGH INTR_EDGE_RISING INTR_EDGE_FALLING RST	X	-	Signal classification
ENDIAN	*string*	BIG, LITTLE	BIG	Semi-automatic inference when ranges can be resolved at compile time	Specifies endianess of signals.
INITIALVALL	*string*	VCC, GND	GND	-	defines initial value of signal if unconnected
BUSIF	*string*	-	-	Automatic inference based on signal naming conventions	specifies bus interfaces associated to signal.
SIGVAL	*string*	-	-	Automatic inference for bus interface signals based on signal naming conventions	specifies default signal connector names to connect to signal

THREE_STATE Attribute

The THREE_STATE attribute enables / disables tri-state IOB buffer insertion. When this attribute is not specified, PsfUtility automatically generates a THREE_STATE attribute set to true for all signals in the HDL that end with _I, _O and _T.

Format

```
entity Peripheral is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000"
    );
  port (
    PAR_I : in  std_logic;
    PAR_O : out std_logic;
    PAR_T : out std_logic;
    );
  attribute THREE_STATE : string;
  attribute THREE_STATE of PAR_I:signal is "FALSE";
  attribute THREE_STATE of PAR_O:signal is "FALSE";
  attribute THREE_STATE of PAR_T:signal is "FALSE";
end entity Peripheral;
```

IOB_STATE Attribute

The IOB_STATE attribute identifies ports that instantiate or infer IOB primitives.

Format

```
entity Peripheral is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000"
    );
  port (
    DDR_Addr : out std_logic
    );
  attribute IOB_STATE : string;
  attribute IOB_State of DDR_Addr:signal is "BUF";
end entity Peripheral;
```

The values are BUF, INFER, or REG. The default is INFER.

When a port has an IOB register (IOB_STATE=REG) or requires an IOB primitive (IOB_STATE=INFER), Platgen instantiates an IOB buffer. When a port has an IOB buffer (IOB_STATE=BUF), Platgen does not instantiate an IOB primitive.

SIGIS Attribute

The class of a signal is specified by the SIGIS option.

Format

```
entity Peripheral is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000"
    );
  port (
    Interrupt_sig : out std_logic
    );
  attribute SIGIS : string;
  attribute SIGIS of Interrupt_sig:signal is "INTR_LEVEL_HIGH";
end entity Peripheral;
```

Where the SIGIS value can have the following values

- CLK : indicating it is a Clock signal
- INTR_LEVEL_HIGH : indicating it is an Interrupt signal with Level High Sensitivity
- INTR_LEVEL_LOW: indicating it is an Interrupt signal with Level Low Sensitivity
- INTR_EDGE_RISING: indicating it is an Intr signal sensitive on rising edge
- INTR_EDGE_FALLING: indicating it is an Intr signal sensitive on falling edge
- RST: indicating it is a Reset signal

INITIALVAL Attribute

This specifies the Initial Value on a signal if it is unconnected.

Format

```
entity Peripheral is
   port (
       M_DBus : in std_logic
    );
    attribute INITALVAL : string;
    attribute INTIALVAL of sig:signal is "VCC";
end entity Peripheral;
```

BUSIF Attribute

This specifies the bus interface attribute associated with a port. PsfUtility automatically infers the association for all bus ports provided signal naming convention was followed.

Format

```
entity Peripheral is
   port (
       mysig : in std_logic
    );
    attribute BUSIF : string;
    attribute BUSIF of mysig:signal is "SOPB";
end entity Peripheral;
```

SIGVAL Attribute

This specifies the Connector Name for a signal. PsfUtility automatically infers all connector names for bus signals.

Format

```
entity Peripheral is
   port (
       mysig : in std_logic
    );
    attribute SIGVAL : string;
    attribute SIGVAL of mysig:signal is "conn_sig";
end entity Peripheral;
```

Chapter 9

Format Revision Tool

Overview

The Format Revision Tool (RevUp) updates an existing EDK 3.1 or 3.2 project to a format for EDK 6.1. Note that if you open an old project with XPS, then it will automatically revup the project to the new format. A project revup will also automatically cause revup of all the hardware repository data files (MPD, BBD, and PAO) referred to by that project and that of the local **myip** and **pcores** directories. RevUp can optionally update just the hardware repository data files . The upgrade is a format update and not an IP upgrade. Note that there is no update required for software repository (MDD, MLD) files.

Current PSF version is 2.1.0. Previous supported versions include 2.0.0 for MPD, BBD, and PAO files and version 2.1.0 for MDD, and MLD files.

EDK tools are always running with the latest formats. Only RevUp needs to maintain compatibility with older versions.

This chapter includes the following sections:

"Tool Requirements"

"Tool Usage"

"Tool Options"

"Current Limitations"

Tool Requirements

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

Tool Usage

Run RevUp as follows:

```
revup <system>.xmp
revup -rd <repository_dir>
```

Tool Options

The following are the options supported in the current version:

-h (Help)

The **-h** option displays the usage menu and quits.

-rd (repository directory)

The **-rd** option allows you to specify the repository directory which needs revup. If this option is specified, then you can not revup an old EDK project (XMP) at the same time.

Current Limitations

The current limitations of the RevUp flow are:

- If you have any IP in **myip** directory, even though revup will update the format of data files, you must manually move those IP to **pcores** directory. EDK 6.1 tools do not search for IPs in **myip** directory.
- If you have a EDK 3.1 project, the software repository revup does not happen automatically. If you your own MDD files, you must manually update them to 2.1.0 format. A manual update of MDD files was required even when reving up from EDK 3.1 to EDK 3.2.

Bitstream Initializer

Summary

This chapter describes the Bitstream Initializer (BitInit) utility.

Overview

The Bitstream Initializer tool initializes the instruction memory of processors on the FPGA. The instruction memory of processors are stored in BlockRAMs in the FPGA. This utility reads an MHS file, and invokes the data2mem utility provided in ISE to initialize the FPGA BlockRAMs.

Tool Usage

The BitInit tool is invoked as follows:

```
bitinit <mhsfile> [options]
```

Note: Please specify <mhsfile> before specifying other tool options.

Tool Options

The following options are supported in the current version of BitInit:

-h (Display Help)

The **-h** option displays the usage menu and quits.

-v (Display Version)

The **-v** option displays the version and quits.

-bm (Input BMM file)

The **-bm** option 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

-bt (Bitstream file)

The **-bt** option specifies the input bitstream file that does not have it's memory initialized.

Default: implementation/<sysname>.bit

-o (Output Bitstream file)

The **-o** option specifies the name of the output file to generate the bitstream with initialized memory.

Default: implementation / download.bit

-pe (Specify the Processor Instance name and list of elf files)

The **-pe** option specifies the name of the processor instance in the MHS and it's associate list of ELF files that form it's instruction memory. This option may be repeated several times based on the number of processor instances in the design.

-lp (Libraries path)

The **-lp** option specifies the path to repository libraries. This option may be repeated to specify multiple libraries.

-log (Log file name)

The **-log** option specifies the name of log file to capture the log.

Default: bitinit.log

-quiet

Runs the tool in quiet mode.

Note: The tool also produces a file named "data2mem.dmr" that is the log file generated during invokation of the data2mem utility.

Chapter 11

GNU Compiler Tools

Scope

This chapter describes the various options supported by MicroBlaze and Power PC GNU tools. The MicroBlaze GNU tools include **mb-gcc** compiler, **mb-as** assembler and **mb-ld** loader/linker. The Power PC tools include **powerpc-eabi-gcc** compiler, **powerpc-eabi-as** assembler and the **powerpc-eabi-ld** linker. The EDK GNU tools also support C++.

In this chapter, only those options are discussed, which have been added or enhanced for Embedded Development Kit (EDK).

GNU Compiler Framework

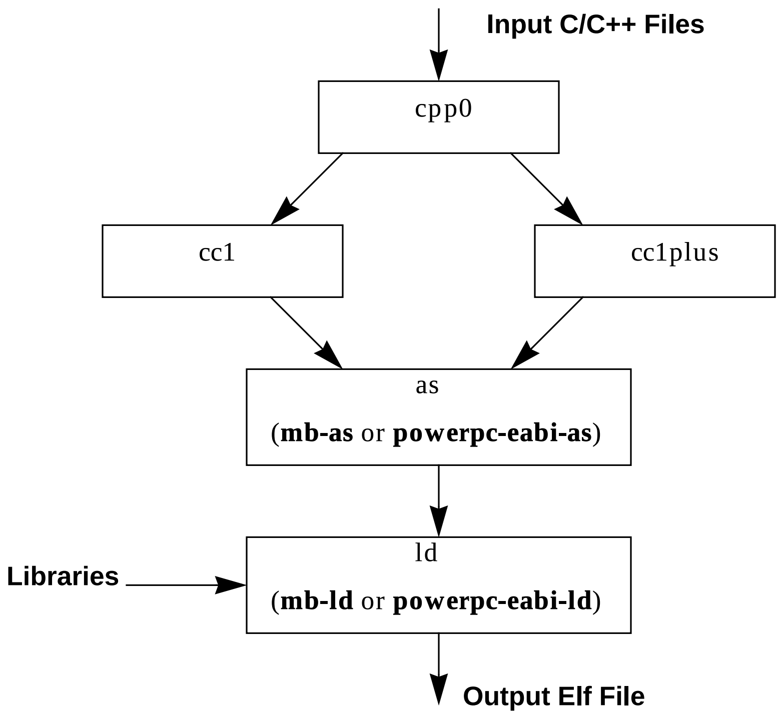


Figure 11-1: **GNU Tool Flow**

This section discusses the common features of both the MicroBlaze as well as PowerPC compiler. Figure 11-1shows the GNU tool flow. The GNU compiler is named **mb-gcc** for

MicroBlaze and **powerpc-eabi-gcc** for **Power PC**. The GNU compiler is a wrapper which in turn calls four different executables:

1. 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.
2. Machine and Language specific Compiler (**cc1)**
   - The compiler works on the pre-processed code, which is the output of the first stage.

   a. C Compiler (**cc1**)
   - The compiler is responsible for most of the optimizations done on the input C code and generates an assembly code.

   b. C++ Compiler (**cc1plus**)
   - The compiler is responsible for most of the optimizations done on the input C++ code and generates an assembly code.
3. Assembler (**mb-as [***For MicroBlaze***]** and **powerpc-eabi-as [***for PowerPC***]**)
   - 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.
   - The assembler creates an object file, which is passed on to the linker
4. Linker (**mb-ld [***For MicroBlaze***]** and **powerpc-eabi-ld [***for PowerPC***]**)
   - The linker 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.

Options for all these executables in discussed in this chapter.

Note: Any reference to gcc in this chapter indicates reference to both MicroBlaze compiler (**mb-gcc**) as well as PowerPC compiler (**powerpc-eabi-gcc**)

Compiler Usage and Options

Usage

GNU Compiler usage is as follows

```
Compiler_Name [options] files...
```

Where `Compiler_Name` is **powerpc-eabi-gcc** or **mb-gcc**

Quick Reference

Table 11-1 briefly describes the commonly used compiler options. These options are common to both the compilers, i.e MicroBlaze and PowerPC. Please note that the compiler options are case sensitive.

Table 11-1: **Commonly Used Compiler Options**

Options	Explanation
-E	Preprocess only; Do not compile, assemble and link. The preprocessed output is displayed on the standard out device
-S	Compile only; Do not assemble and link (Generates **.s** file)
-c	Compile and Assemble only; Do not link (Generates **.o** file)
-g	Add debugging information, which is used by GNU debugger (**mb-gdb** or **powerpc-eabi-gdb**)
-gstabs	Add debugging information to the compiled assembly file. Pass this option directly to the GNU assembler or through the **-Wa** option to the Compiler
-Wa,*option*	Pass comma-separated *options* to the assembler
-Wp,*option*	Pass comma-separated *options* to the preprocessor
-Wl,*option*	Pass comma-separated *options* to the linker
-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*[a] for undefined symbols.
-v	(Verbose). Display the programs invoked by the compiler
-o *filename*	Place the output in the *filename*
-save-temps	Store the intermediate files, i.e files produced at the end of each pass,
--help	Display a short listing of options.
-O *n*	Specify Optimization level *n = 0,1,2,3*

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

Compiler Options

Some of the compiler options are discussed in details in this section

-g

This option adds 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 as well as the assembly level. This option adds debugging information only when the input is a C/C++ source file.

-gstabs

Use this option for adding debugging symbols to assembly(.S) files. This is a assembler option and should be 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,** .

-O*n*

The GNU compiler provides optimizations at different levels. These optimization levels are applied only to the C and C++ source files.

Table 11-2: **Optimizations for different values of n**

n	Optimization
0	No Optimization
1	Medium Optimization
2	Full optimization
3	full optimization, and also attempt automatic inlining of small subprograms.
S	Optimize for speed

Note: **Optimization levels 1 and above will cause code re-arrangement. While debugging your code, use of no optimization level is advocated. 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 finding out the default options for each tool.

-save-temps

The GNU compiler provides a mechanism to save all 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 default output of the entire compilation is stored as *a.out*.

-o *Filename*

The default output of the compilation process is stored in an elf file name *a.out*. The default name can be changed using the -*o output_file_name*. The output file is created in elf format.

-Wp,*option*

-Wa,*option*

-Wl,*option*

As described earlier in this chapter, the compiler (**mb-gcc** or **powerpc-eabi-gcc**) is a wrapper around other executables such as the preprocessor, compiler (cc1), assembler and the linker. These components of the compiler can be executed through the top level compiler or individually.

There are certain options which are required by tool, but might not be necessary for the top level compiler. These command can be issues using the options as indicated in Table 11-3

Table 11-3: **Tool specific options passed to the top level gcc compiler**

Option	Tool
-Wp,*option*	Preprocessor
-Wa,*option*	Assembler
-Wl,*option*	Linker

--help

Use this option with any GNU compiler to get more information about the available options or consult the GCC manual available online at http://www.gnu.org/manual/manual.html

Library Search Options

-l *libraryname*

The compiler, by default, searches only the standard libraries such as libc, libm and libxil. The users can create their own libraries containing some commonly used functions. The users can indicate to the compiler, the name of the library, where the compiler can find the definition of these functions. The compiler prefixes the word **"lib"** to the *libraryname* provided by the user.

The compiler is sensitive to the order in which the various options are provided, especially the **-l** command line switch. This switch should be provided only after all the sources in the command line.

For example, if a user creates his own library called **libproject.a.**, he/she can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -lproject
```

Caution! If the library flag **-llibrary name** is given before the source files, the compiler will not be able to find the functions called from any of the sources. The compiler search is only done in one direction and does not keep a list of libraries available.

-L *Lib Directory*

This option indicates to the compiler, the directories to search for the libraries. The compiler has a default library search path, where it looks for the standard library. By providing **-L** option, the user can include some additional directories in the compiler search path.

Header Files Search Option

-I *Directory Name*

The option **-I**, indicates to the compiler to search for header files in the directory *Directory Name* before searching the header files in the standard path.

Linker Options

-defsym _STACK_SIZE=*value*

The total memory allocated for the stack and the heap can be modified by using the above linker option. The variable STACK_SIZE is the total space allocated for heap as well as the stack. The variable STACK_SIZE is given the default value of 100 words (i.e 400 bytes). If any user program is expected to need more than 400 bytes for stack and heap together, it is recommended that the user should increase the value of STACK_SIZE using the above option. This option expects value 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 will try to write in other forbidden section of the code, leading to wrong execution of the code.

Note: For MicroBlaze systems, minimum stack size of 16 bytes (0x0010) is required for programs linked with the C runtime routines (crt0.o and crt1.o).

Linker Scripts

The linker utility makes use of the linker scripts to divide the user's program on different blocks of memories. To provide a linker script on the gcc command line, use the following command line option:

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

If the linker is executed on its own, the linker script could be included as follows:

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

For more information about usage of linker scripts, please refer to Chapter 30, "Address Management"

Search Paths

The compilers (**mb-gcc** and **powerpc-eabi-gcc**) search certain paths for libraries and header files.

On Solaris

Libraries are searched in the following order:

1. Directories passed to the compiler with the **`-L dir name`** option.
2. Directories passed to the compiler with the **`-B dir name`** option.
3. **`${XILINX_EDK}`**`/gnu/`*`processor`*[1]`/sol/microblaze/lib`
4. **`${XILINX_EDK}`**`/lib/`*`processor`*

Header files are searched in the following order:

1. Directories passed to the compiler with the **-I dir name** option.$
2. **`${XILINX_EDK}`**`/gnu/`*`processor`*`/sol/`*`processor`*`/include`

1. Processor indicates **powerpc-eabi** for PowerPC and **microblaze** for MicroBlaze

Initialization files are searched in the following order[1]:

1. Directories passed to the compiler with the **-B dir name** option.
2. `**${XILINX_EDK}**/gnu/*processor*/sol/*processor*/lib`

On Windows Xygwin Shell

The GNU compilers (**mb-gcc** and **powerpc-eabi-gcc)** search certain paths for libraries and header files.

Libraries are searched in the following order:

1. Directories passed to the compiler with the **`-L dir name`** option.
2. Directories passed to the compiler with the **`-B dir name`** option.
3. `**%XILINX_EDK%**/gnu/*processor*/nt/*processor*/lib`
4. `**%XILINX_EDK%**/lib/*processor*`

Header files are searched in the following order:

1. Directories passed to the compiler with the **-I dir name** option.$
2. `**%XILINX_EDK%**/gnu/*processor*/nt/*processor*/include`

Initialization files are searched in the following order:

1. Directories passed to the compiler with the **-B dir name** option.
2. `%XILINX_EDK%/gnu/*processor*/nt/*processor*/lib`

File Extensions

The GNU compiler can determine the type of your file depending on the extension.Table 11-4 illustrates the valid extension and the corresponding file type.The gcc wrapper will call the appropriate lower level tools by recognizing these file types.

Table 11-4: **File Extensions**

Extension	File type
.c	C File
.C	C++ File
.cxx	C++ File
.cpp	C++ File
.c++	C++ File
.cc	C++ File

1. Initialization files such as crt0.o are searched by the compiler only for **mb-gcc.** For **powerpc-eabi-gcc**, the C runtime library is a part of the library and is picked up by default from the library *libxil.a*

Table 11-4: **File Extensions**

Extension	File type
.S	Assembly File, but might have preprocessor directives
.s	Assembly File with no preprocessor directives

Libraries

Both the compiler (**powerpc-eabi-gcc** and **mb-gcc**) use certain libraries. The following libraries are needed for all the program.

Table 11-5: **Libraries used by the compilers**

Library	Particular
libxil.a	Contain drivers, software services (such as XilNet & XilMFS) and initialization files developed for the EDK tools
libc.a	Standard C libraries, including functions like **strcmp, strlen** etc
libm.a	Math Library, containing functions like **cos, sine** etc

All the libraries are linked in automatically by both the compiler. The search path for these libraries might have to be given to the compiler, if the standard libraries are overridden. The libxil.a is modified by the Library Generator tool to add driver and library routines.

Compiler Interface

Input Files

The compiler (mb-gcc and the powerpc-eabi-gcc) take one or more of the following files are input

- C source files.
- C++ source files.
- Assembly Files.
- Object Files.
- Linker scripts (These are optional and if not specified, the default linker script embedded in the linker (mb-ld or powerpc-eabi-ld) will be used.

The default extensions for each of these types is detailed in Table 11-4. In addition to the files mentioned above, the compiler implicitly refers to the following files.

- Libraries (libc.a, libm.a and libxil.a). The default location for these files is the EDK installation directory.

Output Files

The compiler generates the following files as output

- An elf file (The default output file name is **a.out** on Solaris and **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)

MicroBlaze GNU Compiler

The MicroBlaze GNU compiler is an enhancement over the standard GNU tools and hence provides some additional options, which are specific to the MicroBlaze system.These options are available only in the MicroBlaze GNU compiler.

Quick Reference

Table 11-6: **MicroBlaze Specific Options**

Options	Explanation
-xl-mode-executable	Default mode for compilation.
-xl-mode-xmdstub	Software intrusive debugging on the board. Should be used only with xmdstub downloaded on to MicroBlaze
-xl-mode-xilkernel	If you use the xilkernel module, all the programs should be compiled with this option.
-mxl-gp-opt	Use the small data area anchors. Optimization for performance and size.
-mxl-soft-mul	Use the software routine for all multiply operations. This option should be used for devices without the hardware multiplier. This is the default option in mb-gcc
-mno-xl-soft-mul	Do not use software multiplier. Compiler generates “mul” instructions.
-mxl-soft-div	Use the software routine for all divide operations.This is the default option.
-mxl-no-soft-div	Use the hardware divide available in the MicroBlaze
-mxl-stack-check	Generates code for checking stack overflow.
-mxl-barrel-shift	Use barrel shifter. Use this option when a barrel shifter is present in the device

MicroBlaze Compiler

The mb-gcc compiler for Xilinx’s MicroBlaze soft processor introduces some 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.

-mxl-soft-mul

In some devices, a hardware multiplier is not present. In such cases, the user has the option to either build the multiplier in hardware or use the software multiplier library routine provided. MicroBlaze compiler mb-gcc assumes that the target device does not have a hardware multiplier and hence every multiply operation is replaced by a call to **mulsi3_proc** defined in library **libc.a**. Appropriate arguments are set before calling this routine.

-mno-xl-soft-mul

Certain devices such as Virtex II have a hardware multiplier integrated on the device. Hence the compiler can safely generate the **mul** or **muli** instruction. Using a hardware multiplier gives better performance, but can be done only on devices with hardware multiplier such as Virtex II.

-mxl-soft-div

The MicroBlaze processor does not come with a hardware divide unit. The users would need the software routine in the libraries for the divide operation. This option is turned on by default in mb-gcc.

-mno-xl-soft-div

In MicroBlaze version 2.00 and beyond, the user can instantiate a hardware divide unit in MicroBlaze. If such a unit is present, this option should be provided to mb-gcc compiler. Refer to the MicroBlaze Reference Guide for more details about the usage of hardware divide option in the MicroBlaze.

-mxl-stack-check

This option lets users check if the stack overflows during the execution of the program. 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.

The standard stack overflow handler can be overridden by providing the function **`_stack_overflow_exit`** in the source code, which acts as the stack overflow handler.

-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 is to assume that no barrel shifter is present and hence the compiler will use add and multiply operations to shift the operands. Barrel shift can increase the speed significantly, especially while doing floating point operations.Refer to the MicroBlaze Reference Guide for more details about the usage of the barrel shifter option in the MicroBlaze.

-mxl-gp-opt

If the memory location requires more than 32K, the load / store operation requires two instructions. MicroBlaze ABI offers two global small data areas, which can contain up to 64K bytes of data each. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value. Hence needing only one instruction for load / store to the small data area.This optimization can be turned ON with the -mxl-gp-opt command line parameter. Variables of size lesser than a certain threshold value are stored in these areas. The value of the pointers is determined during linking.

-xl-mode-executable

This is the default mode used for compiling programs with mb-gcc. The final executable created starts from address location 0x0 and links in crt0.o. This option need not be provided on the command line for mb-gcc.

-xl-mode-xmdstub

The mb-gcc compiler links certain initialization files along with the program being compiled. If the program is being compiled to work along with xmd, **crt1.o** initialization file is used, which returns the control of the program to the xmdstub after the execution of the user code is done. In other cases, **crt0.o** is linked to the output program, which jumps to halt after the execution of the program. Hence the option -xl-mode-xmdstub helps the compiler in deciding which initialization file is to be linked with the current program.

The code start address is set to **0x400** for programs compiled for a system with xmd. This ensures that the compiled program starts after the xmdstub. If you intend to modify the default xmdstub, leading to increase in the size of the xmdstub, you should take care to change the start address of the text section. This option is described in the *Linker Loader Options* section. For more details on the address management in MicroBlaze, refer to Chapter 30, “Address Management”

-xl-mode-xmdstub is allowed only in hardware debugging mode and with xmdstub loaded in the memory. For software debugging (even with xmdstub), do not use this option. For more details on debugging with xmd, please refer to Chapter 13, “Xilinx Microprocessor Debugger”

-xl-mode-xilkernel

A kernel consisting of few key RTOS features is provided with the EDK tools. All the program compiled to work with the kernel should have the above option.Refer to the Chapter , “LibXil Kernel” for more information regarding the various option provided by the Xilinx MicroKernel.

Caution! mb-gcc will signal fatal error, if more than one mode of execution is supplied on the command line.

MicroBlaze Assembler

The mb-as assembler for Xilinx’s 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 they are required - 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 Reference Guide.

The mb-as assembler requires all Type B MicroBlaze instructions (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 will compute it, and will include 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-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. The `relax` option of the linker should be used to remove any imm instructions that are found to be unnecessary.

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 is used to add the constant 200,000 to the contents of register r3, and store the result in register r4:

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

The mb-assembler will recognize that this operand needs an imm instruction, and insert one automatically.

In addition to the standard MicroBlaze instruction set, the mb-as assembler also supports some pseudo-opcodes to ease the task of assembly programming. The supported pseudo-ops are listed in Table 11-7.

Table 11-7: **Pseudo-Opcodes supported by the Gnu Assembler**

Pseudo Opcodes	Explanation
nop	No operation. Replaced by instruction: **or** R0, R0, R0
la Rd, Ra, Imm	Replaced by instruction: **addi** Rd, Ra, imm; = Rd = Ra + Imm;
not Rd, Ra	Replace by instruction: **xori** Rd, Ra, -1
neg Rd, Ra	Replace by instruction: **rsub** Rd, Ra, R0
sub Rd, Ra, Rb	Replace by instruction: **rsub** Rd, Rb, Ra

MicroBlaze Linker

The mb-ld linker for Xilinx's MicroBlaze soft processor introduces some new options in addition to those supported by the gnu compiler tools. The new 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 0x0. This can be overridden by using the above options. If this is supplied to **mb-gcc**, the text section of the output code will now start from the given *value*. When the compiler is invoked with **-xl-mode-xmdstub**, the user program starts at 0x400 by default.

The user does not have to use **-defsym _TEXT_START_ADDR**, if they wish to use the default start address set by the compiler.

This is a linker option and should be used when the user is invoking the linker separately. If the linker is being invoked as a part of the **mb-gcc** flow, the user has to use the following option

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

-relax

This is a linker option, used to remove all the unwanted **imm** instructions generated by the assembler. The assembler generates **imm** instruction for every instruction where the value of the immediate can not be calculated during the assembler phase. Most of these instructions won't need an **imm** instruction. These are removed by the linker when the **-relax** command line option is provided to the linker.

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 to be 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, you should have use this option.

For more details on this option, please refer to the GNU manuals online at http://www.gnu.org/manual/manual.html

Initialization Files

The final executable needs certain registers such as the small data area anchors (R2 and R13) and the stack pointer (R1) to be initialized. These initialization files are distributed with the Embedded Development Kit. In addition to the precompiled object files, source files are also distributed in order to help user make their own changes as per their requirements. Initialization can be done using one of the three C runtime routines:

crt0.o

This initialization file is to be used for programs which are to be executed standalone, i.e without the use of any bootloader or debugging stub (such as **xmdstub**).

crt1.o

This file is located in the same directory and should be used when the **xmd** debugger is to be present in the system.

crt4.o

When the kernel module is used in a particular MicroBlaze system, crt4.o is picked up by the compiler.

The source for initialization files is available in the

<*XILINX_EDK*>/gnu/microblaze/<*platform*>/microblaze/lib/src directory,

- <*XILINX_EDK*> : Installation area
- <*platform*> = **sol** (Solaris) or **nt** (on Windows)

These files can be changed as per the requirements of the project. These changed files have to be then assembled to generate an object file (.o format). 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.`**

According to the C standard specification, all global and static variables need to be initialized to 0. This is a common functionality required by all the crt's above. Hence another routine **_crtinit** is defined in **crtinit.o** file. This file is part of the **libc.a** library.

The **_crtinit** routine will initialize memory in the **bss** section of the program, defined by the default linker script. If you intend to provide your own linker script, you will need to compile a new **_crtinit** routine. The default **crtinit.S** file is provided in assembly source format as a part of the Embedded Development Kit.

Command Line Arguments

MicroBlaze programs can not take in command line arguments. The command line arguments argc and argv are initialized to 0 by the C runtime routines.

Interrupt Handlers

Interrupt handlers need to be compiled in a different manner as compared to the normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers have to save the volatile registers which are being used. Interrupt handler should also store the value of the machine status register (RMSR), when an interrupt occurs.

_interrupt_handler attribute

In order 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: Attribute for interrupt handler is to be given only in the prototype and not the definition.

Interrupt handlers might also call other functions, which might use volatile registers. In order to maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function(1).

Interrupt handlers can also be defined in the MicroBlaze Hardware Specification (MHS) and the MicroBlaze Software Specification (MSS) file. These definitions would automatically add the attributes to the interrupt handler functions. For more information please refer MicroBlaze Interrupt Management document.

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 attributes save all the volatiles for non-leaf functions and only the used volatiles in case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

The attributes with their functions are tabulated in Table 11-8.

1. Functions which have calls to other sub-routines are called non-leaf functions.

Table 11-8: **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. **`rtid`** is used for returning 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 interrupt_handler, but it used **`rtsd`** to return to the interrupted function, instead of **`rtid`**.

Power PC GNU Compiler

Compiler Options

The Power PC GNU compiler **(powerpc-eabi-gcc)** is built using the GNU gcc version 2.95.3-4. No enhancements have been done to the compiler. The PowerPC compiler does not support any special options. All the listed common options are supported by the powerpc-eabi compiler.

Linker Options

-defsym _START_ADDR=*value*

By default, the text section of the output code starts with the base address 0xffff0000, since this is the start address indicated in the default linker script. This can be overridden by

- using the above option OR
- providing a linker script, which lists the value for start address

The user does not have to use **-defsym _START_ADDR**, if they wish to use the default start address set by the compiler.

This is a linker option and should be used when the user is invoking the linker separately. If the linker is being invoked as a part of the **powerpc-eabi-gcc** flow, the user has to use the following option

`-Wl,-defsym -Wl,_START_ADDR=``value`

Initialization Files

The compiler looks for certain initialization files (such as **boot.o, crt0.o**). These files are compiled along with the drivers and archived in libxil.a library. This library is generated using LibGen by compiling the distributed sources in the Board Support Package (BSP). For more information about libgen, please refer to the , "Library Generator"chapter.

Chapter 12

GNU Debugger

Summary

This chapter describes the general usage of the Xilinx GNU debugger for MicroBlaze and PowerPC.

Overview

GDB is a powerful yet flexible tool which provides a unified interface for debugging / verifying MicroBlaze and PowerPC systems during various development phases.

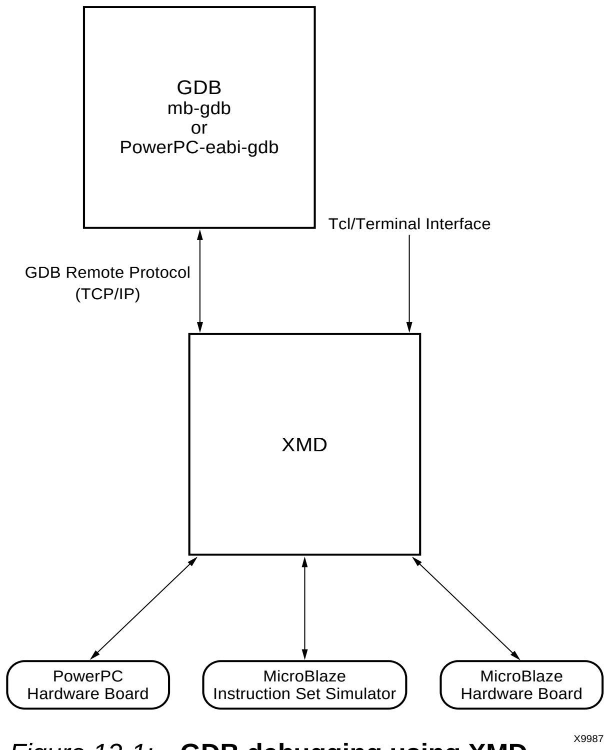


Figure 12-1: **GDB debugging using XMD**

Tool Usage

MicroBlaze GDB usage:

```
mb-gdb [options] [executable-file]
```

PowerPC GDB usage:

```
powerpc-eabi-gdb [options] [executable-file]
```

Tool Options

The most common options in the MicroBlaze GNU debugger are:

--command=FILE

Execute GDB commands from FILE. Used for debugging in batch / script mode.

--batch

Exit after processing options. Used for debugging in batch / script mode.

--nw

Do not use a GUI interface.

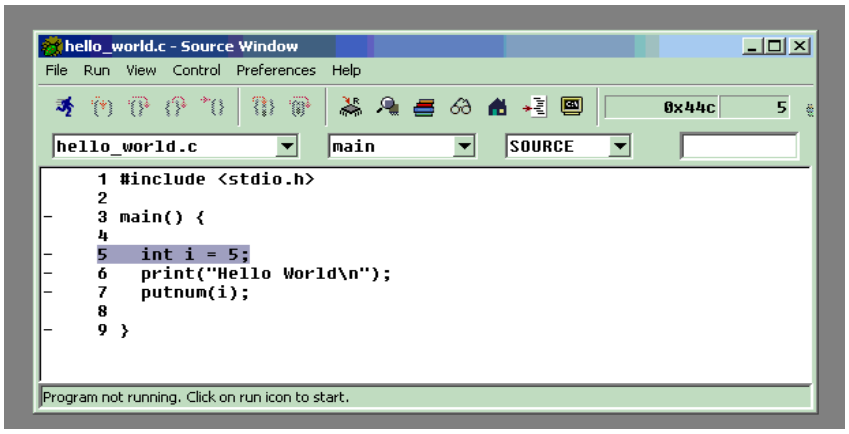
-w

Use a GUI interface. (Default)

MicroBlaze GDB Targets

Currently, there are three possible targets that are supported by the MicroBlaze GNU Debugger and XMD tools - a built-in simulator target and two remote targets (XMD):

```
xilinx > mb-gdb hello_world.elf
```



```
#include <stdio.h>

main() {

  int i = 5;
  print("Hello World\n");
  putnum(i);

}
```

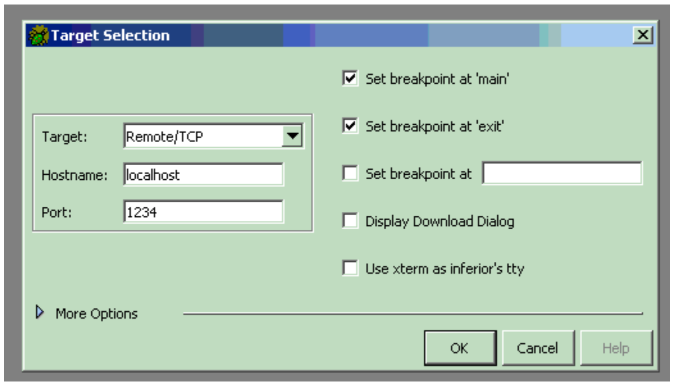
Program not running. Click on run icon to start.

From the **Run** pull-down menu, select **Connect to target** in the mb-gdb window. In the Target Selection dialog, you can choose between the **`Simulator`** (built-in) and **`Remote/TCP`** (for XMD) targets.

In the target selection dialog, choose:

- Target: Remote/TCP
- Hostname: localhost
- Port: 1234

Click **OK** and mb-gdb attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD was started.



Set breakpoint at 'main' [✓]
Set breakpoint at 'exit' [✓]
Set breakpoint at [ ]
Display Download Dialog [ ]
Use xterm as inferior's tty [ ]

Target:	Remote/TCP
Hostname:	localhost
Port:	1234

At this point, **`mb-gdb`** is connected to XMD and controls the debugging. The simple but powerful GUI can be used to debug the program, read and write memory and registers.

GDB Built-in Simulator

The MicroBlaze debugger provides an instruction set simulator, which can be used to debug programs that do not access any peripherals. This simulator makes certain assumption about the executable being debugged:

- The size of the application being debugged determines the maximum memory location which can be accessed by the simulator.
- The simulator assumes that the accesses are made only to the fast local memory (LMB).

When using the command `info target`, the number of cycles reported by the simulator are under the assumptions that memory access are done only into local memory (LMB). Any access to the peripherals results in the simulator indicating an error. This target does not require **`xmd`** to be started up. This target should be used for basic verification of functional correctness of programs which do not access any peripherals or OPB or external memory.

Remote

Remote debugging is done through XMD. The XMD server program can be started on a host computer with the Simulator target or with the Hardware target transparent to mb-gdb. Both the Cycle-Accurate Instruction Set Simulator and the Hardware interface provide powerful debugging tools for verifying a complete MicroBlaze system. **`mb-gdb`** connects to **`xmd`** using the GDB Remote Protocol over TCP/IP socket connection.

Simulator Target

The XMD simulator is a Cycle-Accurate Instruction Set Simulator of the MicroBlaze system which presents the simulated MicroBlaze system state to GDB.

Hardware Target

With the hardware target, XMD communicates with 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 the XMD Chapter.

Note

1. The simulators provide a non-intrusive method of debugging a program. Debugging using the hardware target is intrusive because it needs an xmdstub to be running on the board.
2. If the program has any I/O functions like print() or putnum(), that write output onto the UART or JTAG Uart, it will be printed on the console/terminal where the xmd server was started. (Refer to the MicroBlaze Libraries documentation for libraries and I/O functions information).

Compiling for Debugging on MicroBlaze targets

In order to debug a program, you need to generate debugging information when you compile it. 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 Xilinx's 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. mb-gcc adds appropriate information to the executable file, which helps in debugging the code. mb-gdb provides debugging at source, assembly and mixed (both source and assembly) together. While initially verifying the functional correctness of a C program, it is also advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may make debugging difficult to follow. For debugging with **`xmd`** in hardware mode, the **`mb-gcc`** option **`-xl-mode-xmdstub`** must be specified. Refer to the XMD documentation for more information about compiling for specific targets.

PowerPC Targets

GUI mode

Hardware debugging for the PowerPC405 on Virtex-II Pro is supported by **powerpc-eabi-gdb** and **xmd** through the GDB Remote TCP protocol. To connect to a hardware PowerPC target, first start **xmd** and connect to the board using the **ppcconnect** command as described in the XMD chapter. Next, select **Run->Connect to target** from GDB and in the GDB target selection dialog, choose:

- Target: Remote/TCP
- Hostname: localhost
- Port: 1234

Click **OK** and **powerpc-eabi-gdb** attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD was started.

Console mode

To start powerpc-eabi-gdb in the console mode type :

```
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
```

For the consolse 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, click on **`Help->Help Topics`** in the GUI mode

or type "**`help`**" in the console mode.

In the GUI mode, to open a console window, click on **`View->Console`**

For a comprehensive online documentation on using GDB, refer to **http://www.gnu.org/manual/gdb/**

For information about the mb-gdb Insight GUI, refer to the Red Hat Insight webpage **http://sources.redhat.com/insight**

Table 12-1 briefly describes the commonly used mb-gdb console commands. The

Table 12-1: **Commonly Used GDB Console Commands**

Command	Description
load [program]	load the program into the target
b main	Set a breakpoint in function main
r	Run the program (for the built-in simulator only)
c	Continue after a breakpoint, or Run the program (for the xmd simulator only)
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 *xyz* data

equivalent GUI versions can be easily identified in the mb-gdb GUI window icons. Some of the commands like info target, monitor info, may be available only in the console mode.

Xilinx Microprocessor Debugger

Overview

The Xilinx Microprocessor Debugger (XMD) is a tool that facilitates a unified GDB interface as well as a Tcl (Tool Command Language) interface for debugging programs and verifying systems using the PowerPC (Virtex-II Pro) or MicroBlaze microprocessors. It supports debugging user programs on different targets such as

- PowerPC system on a hardware board
- Cycle-accurate PowerPC instruction set simulator
- Cycle-accurate MicroBlaze instruction set simulator
- MicroBlaze connected to **`opb_mdm`** (hardware debug core) on a board
- MicroBlaze system running **`xmdstub`** (ROM monitor) on a hardware board

XMD is used along with PowerPC and MicroBlaze GDB (**`powerpc-eabi-gdb & mb-gdb`**) for debugging. **`powerpc-eabi-gdb`** and **`mb-gdb`** communicate with **`xmd`** using the GDB Remote TCP protocol and control the corresponding targets. In either case, GDB can connect to **`xmd`** running on the same computer or on a remote computer on the Internet.

The xmd Tcl interface can be used for command line control and debugging of the target as well as for running complex verification test scripts to test complete system.

.

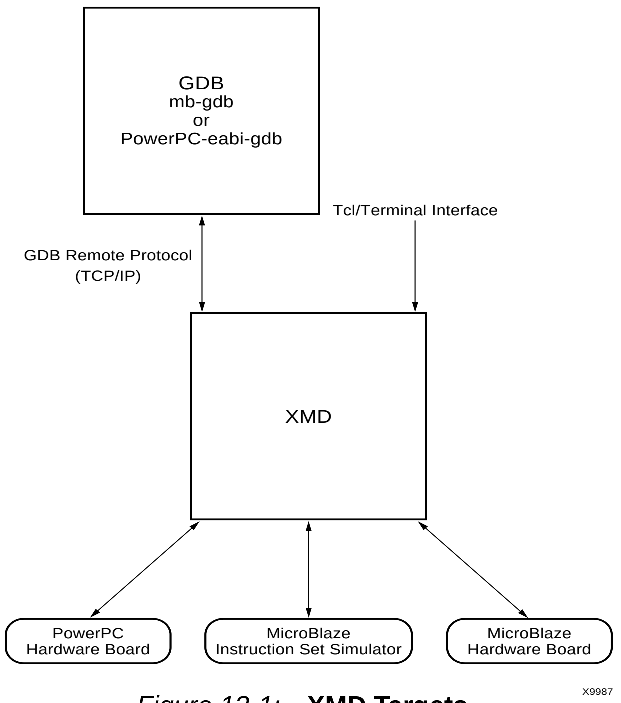


Figure 13-1: **XMD Targets**

XMD usage

To start the XMD engine, simply execute xmd from a shell as follows.

```
> xmd [xmd Tcl script]
```

If an xmd script is specified, xmd will execute the script and quit. Otherwise, xmd will be started in an interactive mode. In this case, if there is a file names **xmd.ini** in the current directory, **xmd** will source the xmd.ini as if it is a Tcl script file, before presenting the XMD% prompt From the xmd Tcl prompt, **xmd** can be connected to the desired target using the commands described in the following sections. After connecting to a target, commands described in Table 13-1 can be used.

PowerPC Target

xmd can connect to one or more hardware PowerPC targets over a JTAG connection to a board containing a Virtex-II Pro device.

PowerPC Target options

Use the **ppcconnect** command to connect to the PowerPC target and start a remote GDB server. Once **xmd** is connected to the PowerPC target, **powerpc-eabi-gdb** can connect to the processor target through xmd and debugging can proceed. Refer to the GDB documentation in the est_guide for more information about connecting GDB to xmd using GDB's Remote TCP target. When no option is specified, **xmd** will detect the JTAG cable, chain and the PowerPC processors automatically. Users can override it using the following options.

ppcconnect [-cable <**JTAG cable options**>] [-configdevice <**JTAG chain options**>] [-debugdevice <**PPC405 options**>]

JTAG cable options

- **type** <cable type>

  Valid cable types are: **xilinx_parallel3, xilinx_parallel4, xilinx_svffile**

  In the case of xilinx_svffile, the JTAG commands are written into a file specified by the **fname** option

- **port** <parallel port name>

  Valid arguments for port are **lpt1, lpt2**

- **fname** <filename>

  Filename for creating the SVF file

JTAG chain options

- **partname** <devicename>

  Name of the device

- **devicenr** <device position>

  Position of the device in the JTAG chain

- **irlength** <length of the JTAG Instruction Register>

  Length of the IR register of the device. This information can be found in the device BSDL file.

- **idcode** <device idcode>

  JTAG Idcode of the device

PPC405 options

- **devicenr** <PowerPC device position>

  Position of the VirtexIIPro device containing the PowerPC, in the JTAG chain

- **cpunr** <CPU Number>

  ID of the specific PowerPC to be debugged in a VirtexIIPro containing multiple PowerPC processors

The following options allow users to map special PowerPC features like ISOCM, Caches, TLB, DCR registers, etc. to unused memory addresses and then from the debugger access it as memory addresses. This is helpful for reading and writing to these registers / memory from GDB or XMD. ***Note*** that, these options **do not** create any real memory mapping in hardware.

- **icachestartadr** <I-Cache start address>

  Start address for reading or writing the instruction cache contents

- **dcachestartadr** <D-Cache start address>

  Start address for reading or writing the data cache contents

- `itagstartadr` <I-Cache start address>

  Start address for reading or writing the instruction cache tags
- `dtagstartadr` <D-Cache start address>

  Start address for reading or writing the data cache tags
- `isocmstartadr` <ISOCM start address>

  Start address for the ISOCM
- `isocmsize` <ISOCM size>

  Size of the ISBRAM memory connected to the ISOCM interface
- `isocmdcrstartadr` <ISOCM DCR address>

  DCR address corresponding to the ISOCM interface specified using the TIEISOCMDCRADDR signals on PowerPC
- `tlbstartadr` <TLB start address>

  Start address for reading and writing the Translation Look-aside Buffer
- `dcrstartadr` <DCR start address>

  Start address for reading and writing the Device Control Registers. Using this option, the entire DCR address space (2^{10} addresses) can be mapped to addresses starting from <dcrstartadr> for debugging purposes from XMD and GDB

PowerPC Target Requirements

There are two possible methods for `xmd` to connect to the PowerPC 405 processors over a JTAG connections. The requirements for each of these methods are described below.

1. **Debug connection using the JTAG port of the Virtex-II Pro 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 Figure 13-2. Please refer to the **"Virtex-II Pro PPC405 JTAG Debug Port"** section in the PowerPC 405 Processor Block Reference Guide for more information about this debug setup. NOTE that there is a core named `jtagppc_cntlr` in EDK that helps in setting up this connection.
2. **Debug connection using user IO pins connected to the JTAG port of the PowerPC**

   If the JTAG ports of the PowerPC processors are brought out of the FPGA using user IO pins, xmd can directly connect to the PowerPC for debugging. Please refer to the **"Virtex-II Pro PPC405 JTAG Debug Port"** section in the PowerPC 405 Processor Block Reference Guide for more information about this debug setup.

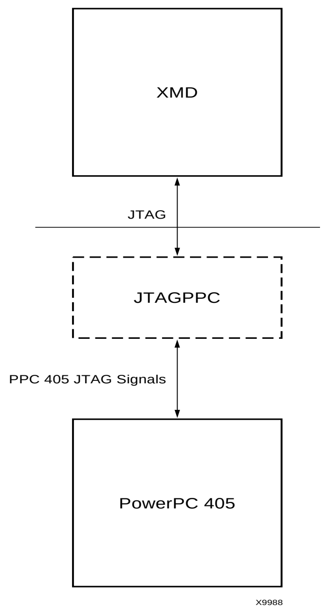


Figure 13-2: **PowerPC Target**

Example debug session with a PowerPC target

This example demonstrates a simple debug session with a PowerPC target. Basic **xmd**-based commands are used after connecting to the PowerPC target using the “ppcconnect” command. At the end of the session, GDB (powerpc-eabi-gdb) is connected to **xmd** using the GDB remote target. Refer to the GDB section of the est_guide for more information about about connecting GDB to **xmd**.

```
XMD% ppcconnect

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
assumption: selected device 2 for debugging.

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffef20
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
    ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD% 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
  dbsr: 00000300   dbcr0: 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: ffffe204    iac4: ffffe204    sler: 00000000   sprg4: ffffe204
 sprg5: ffffe204   sprg6: ffffe204   sprg7: ffffe204    su0r: 00000000
usprg0: ffffe204
XMD% rst
Sending System Reset
Target reset successfully
XMD% rwr 0 0xAAAAAAAA
XMD% rwr 1 0x0
XMD% rwr 2 0x0
XMD% rrd
    r0: aaaaaaaa      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: fffffffc     msr: 00000000
XMD% mrd 0xFFFFFFFC
FFFFFFFC:    4BFFFC74
XMD% stp
fffffc70:
XMD% stp
fffffc74:
XMD% mrd 0xFFFFC000 5
FFFFC000:    00000000
FFFFC004:    00000000
FFFFC008:    00000000
FFFFC00C:    00000000
FFFFC010:    00000000
XMD% mwr 0xFFFFC004  0xabcd1234 2
XMD% mwr 0xFFFFC010  0xa5a50000
XMD% mrd 0xFFFFC000 5
FFFFC000:    00000000
FFFFC004:    ABCD1234
FFFFC008:    ABCD1234
FFFFC00C:    00000000
FFFFC010:    A5A50000
XMD%
```

```
XMD: Accepted a new GDB connection from nnn.nnn.n.nn on port nnnn
XMD%
XMD: Closed connection
XMD%
```

Example debug session with program running in ISOCM memory and accessing DCR registers

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% ppcconnect -debugdevice \
isocmstartadr 0xFFFFE000 isocmsize 8192 isocmdcrstartadr 0x15 \
dcrstartadr 0xab000000

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
assumption: selected device 2 for debugging.

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffe218
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
   ISOCM           :  Start Address - 0xffffe000
    TLB             :  Disabled
   DCR            :  Start Address - 0xab000000

Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD% stp
ffffe21c:
XMD% stp
ffffe220:
XMD% bps 0xFFFFE218
Setting breakpoint at 0xffffe218
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
8
Processor stopped at PC: 0xffffe218
XMD% bpl
HW BP: BP_ID 0 : addr = 0xffffe218 <--- Automatic Hardware Breakpoint
                                     for ISOCM
XMD% mrd 0xFFFFE218
Warning: Attempted to read location: 0xffffe218. Reading ISOCM memory
not supported in V2Pro
Cannot read from target
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 debug session for special JTAG chain setup (Non-Xilinx devices)

This example demonstrates the use of -configdevice option to specify the JTAG chain on the board, in case **xmd** is unable to auto detect the JTAG chain. The auto detect in **xmd** might 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 BSDL files provided by device vendors. For these "Unknown" devices, IRLength is the only critical information needed and the other fields like partname and idcode are optional.

Following is a description of the options use in the example below,

- 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
  - only the IRLength of the 2nd device is specified. Partname is inferred from the idcode since **xmd** knows about the XC2VP4 device
- The debugdevice option explicitly specifies to **xmd** that the FPGA device of interest is the 2nd device in the JTAG chain. In the Virtex-II Pro, it is also explicitly specified that the connection is for the 1st PowerPC processor (if there are more than one)

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% ppcconnect -cable type xilinx_parallel port LPT1 \
> -configdevice devicenr 1 partname PROM irlength 8 idcode 0x05026093 \
> -configdevice devicenr 2 irlength 10 \
> -debugdevice devicenr 2 cpunr 1

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        PROM_XC18V04
 2       0123e093          10        XC2VP4

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffee18
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
    ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
```

```
XMD%
```

PowerPC Simulator target

xmd can connect to one or more PowerPC simulator (ISS) targets through socket connection. Use the **ppcconnect sim** command to start the PowerPC ISS on localhost , connect to it and start a remote GDB server. **ppcconnect sim** can also connect to PowerPC ISS running on localhost or other machine. Once **`xmd`** is connected to the PowerPC target, **`powerpc-eabi-gdb`** can connect to the target through xmd and debugging can proceed.

Running PowerPC ISS

On ppcconnect sim command xmd starts the ISS with default configuration. The ISS executable can be found in ${XILINX_EDK}/third_party/bin/<platform>/ directory. The configuration file is ${XILINX_EDK}/third_party/data/iss405.icf. User can run ISS with different configuration option and xmd can connect to the ISS target. Refer "ISS User Guide" document from IBM for more details. The following are the default configuration for ISS.

- Two local memory banks: Mem0 start address = 0x0, length = 0x80000 and speed = 0. Mem1 start address = 0xfff80000, length = 0x80000 and speed = 0.
- Connect to Debugger (xmd)
- Debugger Port at 6470
- Data Cache size of 8k
- Instruction Cache size of 16k
- Non-Deterministic Multiply cycles
- Processor Clock Period and Timer Clock Period of 5ns (200 Mhz)

PowerPC Simulator target options

When no option is specified to **ppcconnect sim,** xmd starts the ISS with default configuration and connects to ISS. Optionally an user can specify IP address, to connect to host running ISS.

ppcconnect sim [IP address]

Program Trace options

- **traceopen** <trace file>

  Open a trace file for storing the trace output.
- **tracestart**

  Start tracing or collecting trace information
- **tracestop**

  Stop tracing.
- **traceclose**

  Close the trace file.
- **stats** <trace fie>

  Collect Program Statical information from the trace file.

Example debug session for PowerPC ISS target.

```
XMD% ppcconnect sim
Instruction Set Simulator (ISS)
PPC405, PPC440
Version 1.5 (1.69)
(c) 1998, 2002 IBM Corporation
Waiting to connect to controlling interface (port=6470,
protocol=tcp)....
[XMD] Connected to PowerPC Sim
Controling 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% traceopen trace.out
XMD% tracestart
XMD% con
Processor started. Type "stop" to stop processor

RUNNING>

XMD% tracestop
XMD% traceclose
XMD% stats trace.out
Program Stats ::

       Instructions :  197491
              Loads :   20296
             Stores :   19273
    Multiplications :    3124
           Branches :   27262
     Branches taken :   20985
            Returns :  2070
```

MicroBlaze MDM Target

xmd can connect through JTAG to one or more MicroBlaze processors using the **opb_mdm** peripheral. Use the command “**mbconnect mdm**“ in order 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 like xmdstub.

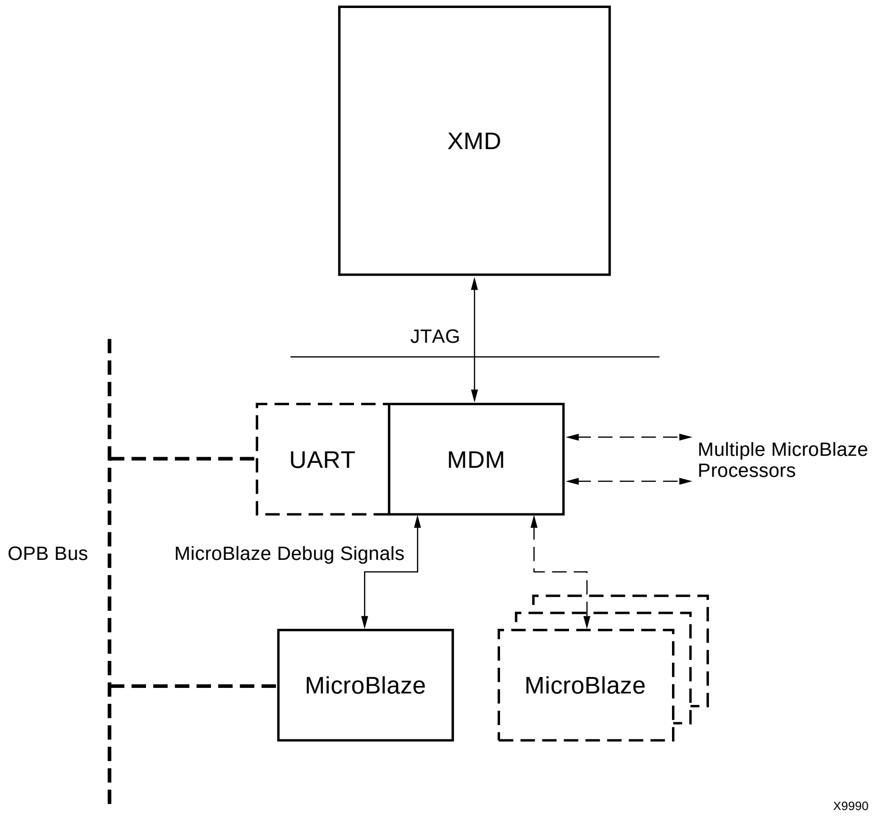


Figure 13-3: **MicroBlaze MDM Target**

MDM Target options

When no option is specified to the **`mbconnect mdm`**, xmd will automatically detect the JTAG cable, chain and the FPGA device containing the MicroBlaze-MDM system. If xmd is unable to detect the JTAG chain or the FPGA device automatically, users can explicitly specify them, using the following options.

`mbconnect` mdm [-sim] [-cable <**`JTAG cable options`**>] [-configdevice <**`JTAG chain options`**>] [-debugdevice <**`MicroBlaze options`**>]

JTAG cable options

- **`type`** <cable type>

  Valid cable types are: **`xilinx_parallel3, xilinx_parallel4, xilinx_svffile`**

  In the case of xilinx_svffile, the JTAG commands are written into a file specified by the **`fname`** option

- **`port`** <parallel port name>

  Valid arguments for port are **`lpt1, lpt2`**

- **`fname`** <filename>

  Filename for creating the SVF file

JTAG chain options

- **`partname`** <devicename>

  Name of the device
- **`devicenr`** <device position>

  Position of the device in the JTAG chain
- **`irlength`** <length of the JTAG Instruction Register>

  Length of the IR register of the device. This information can be found in the device BSDL file.
- **`idcode`** <device idcode>

  JTAG Idcode of the device

MicroBlaze options

- **`devicenr`** <FPGA device position>

  Position of the FPGA device containing the MicroBlaze, in the JTAG chain

MDM Target requirements

1. In order to use the hardware debug features on MicroBlaze like hardware breakpoints, hardware debug control functions like stopping, stepping, etc, MicroBlaze's hardware debug port must be connected to the MicroBlaze Debug Module, the **`opb_mdm`** core. The following MHS snippet demonstrates the debug port connection needed between the MDM and MicroBlaze.

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 8
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
 PORT DBG_CAPTURE = DBG_CAPTURE_s
 PORT DBG_CLK = DBG_CLK_s
 PORT DBG_REG_EN = DBG_REG_EN_s
 PORT DBG_TDI = DBG_TDI_s
 PORT DBG_TDO = DBG_TDO_s
 PORT DBG_UPDATE = DBG_UPDATE_s
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0x0000c000
 PARAMETER C_HIGHADDR = 0x0000c0ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT DBG_CAPTURE_0 = DBG_CAPTURE_s
 PORT DBG_CLK_0 = DBG_CLK_s
 PORT DBG_REG_EN_0 = DBG_REG_EN_s
 PORT DBG_TDI_0 = DBG_TDI_s
 PORT DBG_TDO_0 = DBG_TDO_s
 PORT DBG_UPDATE_0 = DBG_UPDATE_s
END
```

2. In order to use the UART functionality in the MDM target, users have to set the C_USE_UART parameter while instantiating the **opb_mdm** in a system. In order to print program STDOUT onto the xmd console, C_UART_WIDTH should be set as 8. UART input can also be provided from the host to the program running on MicroBlaze by using the **"xuart w <byte>"** command.

Note: Unlike the MicroBlaze stub target, programs can be compiled in executable mode and need NOT be compiled in xmdstub mode while using the MDM target. Consequently, users need NOT specify a DEBUG PERIPHERAL for compiling the xmdstub

Example debug session with 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 "mbconnect" command. At the end of the session, GDB (mb-gdb) is connected to **xmd** using the GDB remote target. Refer to the GDB section of the est_guide for more information about about connecting GDB to **xmd**.

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% mbconnect mdm

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1
MicroBlaze Configuration :
Version - 2.00.a
No of PC HW Breakpoints : 8
No of Read Addr/Data Watchpoints : 1
No of Write Addr/Data Watchpoints : 1
Instruction Cache Support : off
Data Cache Support : off

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" 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
Processor started. Type "stop" to stop processor
RUNNING> stop <--- From this “RUNNING>” prompt, the debugging commands
“stop”, “xuart”, “xrreg 0 32” and all basic Tcl commands can be
executed.
XMD%
Processor stopped at PC: 0x0000010c
XXMD% con
Processor started. Type "stop" to stop processor
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x000000f4 <--- With the MDM, the current PC of MicroBlaze can be
                    read while the program is running
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000110 <--- Note: the PC is constantly changing, as the
                     program is running
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000118 <--- Note: “format” is a basic Tcl command like printf
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000118
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
Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x00000100
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x0000011c
```

Example debug session with 2 MicroBlaze processors and using the JTAG-based UART in MDM

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% mbconnect mdm

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 2
MicroBlaze Configuration :
Version - 2.00.a
No of PC HW Breakpoint : 8
No of Read Addr/Data Watchpoints : 1
No of Write Addr/Data Watchpoints : 1
Instruction Cache Support : off
Data Cache Support : off

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

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
Connected to MicroBlaze "mdm" target. id = 1
Starting GDB server for "mdm" target (id = 0) at TCP port no 1235
<--- Note: Two GDB servers are started at different TCP ports for
parallel debugging from GDB -->
XMD% targets
List of connected targets
```

```
Target ID          Target Type
------------------------------
0                  MicroBlaze MDM-based (hw) Target
1                 MicroBlaze MDM-based (hw) Target *
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000000     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
   pc: 00000188    msr: 00000000
XMD% targets 0
Setting current target to target id 0
List of connected targets

Target ID          Target Type
------------------------------
0                  MicroBlaze MDM-based (hw) Target *
1                  MicroBlaze MDM-based (hw) Target
XMD% rrd
    r0: 00000000      r8: 00000000     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: 02020202     r13: 00000190     r21: 00000000     r29: 00000000
    r6: 0000c200     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 0000006f     r15: 0000005c     r23: 00000000     r31: 00000000
   pc: 000000f8    msr: 00000000
XMD% mrd 0xC000 4    <--- Reading the MDM UART's registers from
                         MicroBlaze's point of view
    C000:   00000000
    C004:   00000000
    C008:   00000004 <--- Note: Status reg is 4, i.e UART is empty
    C00C:   00000000
XMD% xuart w 0x42 <--- Write a character onto the MDM UART from the host
XMD% mrd 0xC008  <--- Read the MDM UART status reg using MicroBlaze
    C008:   00000005 <--- Status is "valid data present"
XMD% mrd 0xC000 <--- Read the UART data i.e consume the char
    C000:   00000042
XMD% mrd 0xC008
    C008:   00000004 <--- Status is again "empty"
XMD% scan "Hello" "%c%c%c%c%c" ch1 ch2 ch3 ch4 ch5
5
XMD% xuart w $ch1
XMD% xuart w $ch2
XMD% xuart w $ch3
XMD% xuart w $ch4
XMD% xuart w $ch5
XMD% dow uart_test.elf
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> Hello
```

Example debug session with Read Address breakpoints

In this debug session, there is a program running on the board that is polling and waiting on MDM UART input - UART is at Baseaddress 0xC000. The program loops around waiting for the data valid bit to be set in the status register 0xC008. Using a read address breakpoint, MicroBlaze is stopped as soon as there is load from address 0xC000. The main commands to note are **“xbreakpoint <target id> <addr> <> <hw bp id>”**. As can be seen in the MicroBlaze configuration below, there are 4 PC hw breakpoints, 1 Read Addr / Data breakpoint (catchpoint) and 1 Write Addr / Data breakpoint (catchpoint).

Note: The number of PC hardware breakpoints, setting and clearing the breakpoints are automatically managed by xmd when the “bps <addr> <hw>” and “bpr <addr>” commands are used. For address breakpoint (catchpoints), currently users have to explicitly set the breakpoint using the breakpoint ID and “xbreakpoint” command. The hardware breakpoint IDs for MicroBlaze are as follows :

- PC hardware breakpoint IDs - 0 to (No of PC BRK -1)
  - For the example below, PC breakpoints are 0-3
- Read Addr / Data breakpoint IDs - Max PC BRK to Max PC BRK + (Read BRK *2)
  - For the example below, Read Addr Breakpoint is 4 and Read Data Breakpoint is 5.
  - The Addr and Data breakpoints for Read or Write always co-exist. If the Addr or Data part of the breakpoint has to be “Dont-Cares”, you can currently set it in XMD by setting the breakpoint to be at addr “0xFFFFFFFF”.
- Write Addr / Data breakpoint IDs - Max RD Addr / Data BRK + (Write BRK * 2)
  - For the example below, Write Addr Breakpoint is 6 and Write Data Breakpoint is 7
  - The Addr and Data breakpoints for Read or Write always co-exist. If the Addr or Data part of the breakpoint has to be “Dont-Cares”, you can currently set it in XMD by setting the breakpoint to be at addr “0xFFFFFFFF”.

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% mbconnect mdm

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1
MicroBlaze Configuration :
Version - 2.00.a
No of PC HW Breakpoints : 4
No of Read Addr/Data Watchpoints : 1
No of Write Addr/Data Watchpoints : 1
Instruction Cache Support : off
Data Cache Support : off

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD% mrd 0xC000 4
```

```
    C000:   00000000
    C004:   00000000
    C008:   00000004
    C00C:   00000000
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000000     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000042     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
    pc: 00000190     msr: 00000000
XMD% dis 0x188 5
     188:   E8650008  lwi     r3, r5, 8
     18C:   A4630001  andi    r3, r3, 1
     190:   BC03FFF8  beqi    r3, -8
     194:   C8602800  lw      r3, r0, r5
     198:   B60F0008  rtsd    r15, 8
XMD% xbreak 0 0xC000 hw 4
Setting breakpoint at 0x0000c000
XMD% xbreak 0 0xFFFFFFFF hw 5
Setting breakpoint at 0xffffffff
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> xuart w 0x42

RUNNING>
Processor stopped at PC: 0x00000198
XMD% dis 0x194
     194:   C8602800  lw      r3, r0, r5
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000042     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
    pc: 00000198     msr: 00000000
XMD%
```

Example debug session for special JTAG chain setup (Non-Xilinx devices)

This example demonstrates the use of -configdevice option to specify the JTAG chain on the board, in case **xmd** is unable to autodetect the JTAG chain. The autodetect in **xmd** might 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 BSDL files provided by device vendors. For these “Unknown” devices, IRLength is the only critical information needed and the other fields like partname and idcode are optional.

Following is a description of the options use in the example below,

- Xilinx Parallel cable (III or IV) connection is done over the LPT1 parallel port.
- The two devices in the JTAG chain are explicitly specified

   - only the IRLength of the PROM is specified. Partname is inferred from the idcode since **xmd** knows about the XC18V04 PROM device
   - the IRLength, partname and idcode of the 2nd device is specified.
- The debugdevice option explicitly specifies to **xmd** that the FPGA device of interest is the 2nd device in the JTAG chain.

```
$ xmd
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 6.1.1 Build EDK_G.13
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.
XMD% mbconnect mdm \
> -configdevice devicenr 1 irlength 8 \
> -configdevice devicenr 2 irlength 10 idcode 0x0123e093 partname V2P4 \
> -debugdevice devicenr 2

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        V2P4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1
MicroBlaze Configuration :
Version - 2.00.a
No of PC HW Breakpoints : 8
No of Read Addr/Data Watchpoints : 1
No of Write Addr/Data Watchpoints : 1
Instruction Cache Support : off
Data Cache Support : off

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD%
```

MicroBlaze Stub target

Table 13-1: **XMD commands**[a]

command [options]	Description
rrd	Register Read
srr	Read special registers (PowerPC target only)
rwr *reg_num word*	Register Write
mrd *address* [*num_words*]	Memory Read
mrd_var *variable [filename]*	Read Memory corresponding to global variable in the ELF file "filename" or in a previously downloaded ELF file
mwr *address word*	Memory Write
dis [*address*] [*num_words*]	Disassemble
con [*address*]	Continue from current PC or "address". While a program is running, the target can be stopped by pressing the 'b' or 's' keys.
stp [*number*]	Step one or "number" instructions
rst	Reset target
bps *address*	Set Breakpoint at "address"
bps_func *function [filename*]	Set Breakpoint at start of function in the ELF file "filename" or in a previously downloaded ELF file
bpr *address*	Remote Breakpoint from "address"
bpr_func *function [filename*]	Remove Breakpoint at start of function in the ELF file "filename" or in a previously downloaded ELF file
bpl	List Breakpoints
dow [-data] *filename* [*addr*]	Download the given ELF or data file (with -data option) onto the current target's memory. If no address is provided along with ELF file, the download address is determined from the ELF file by reading its headers. If an address is provided with the ELF file (only for Microblaze targets), it is treated as PIC code (Position Independent Code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics. Note that NO Bounds checking is done by xmd, except preventing writes into xmdstub area (address 0x0 to 0x400) for the MicroBlaze Stub target.
stats	Display execution statistics for the MicroBlaze simulator target
disconnect *target id*	Disconnect from the current Processor target, close the corresponding GDB server and revert to the previous Processor target if any.
targets <target id>	List information about all current targets or change the current target
help	List all commands

a.

xmdterm.tcl script in the installation directory provides commands for doing assembly level debugging using the low level xmd commands. **xmdterm.tcl** is automatically loaded by xmd on startup. Powerful verification scripts can be written in Tcl based on the xmdterm script. User scripts with helper commands can be loaded into xmd by using the Tcl command "*source script.tcl*". Refer to the Tcl documentation at the Tcl Developer site for more information on writing Tcl scripts and custom commands.

Connect to a MicroBlaze target using the xmdstub (a ROM monitor running on the target) as well as start a GDB server for the target.

MicroBlaze Stub Target Options

When no option is specified to the **mbconnect stub**, xmd will automatically detect the JTAG cable, chain and the FPGA device containing the MicroBlaze system, and connect to the xmdstub on the device. If xmd is unable to detect the JTAG chain or the FPGA device automatically, users can explicitly specify them, using the following options.

mbconnect stub [-comm <**serial | jtag**>] [-port <**serial port**>] [-baud <**baudrate**>] [-cable <**JTAG cable options**>] [-configdevice <**JTAG chain options**>] [-debugdevice <**MicroBlaze options**>] [-timeout <**connection timeout**>]

Stub Communication options

- **-comm** <serial | jtag>

  Method of communicating to the xmdstub target - Serial port or JTAG connection

- **-timeout** <timeout in secs>

  Timeout period while waiting for reply from xmdstub for xmd commands

Serial Port options

- **-port** <serial port>

  Specify the serial port to which the remote hardware is connected, when xmd communication is over the serial cable. The default serial port is */dev/ttya* on Solaris and *Com1* on Windows

- **-baud** <serial port baud rate>

  Specify the serial port baud rate in bps. The default value is *19200* bps.

JTAG cable options

- **type** <cable type>

  Valid cable types are: **xilinx_parallel3, xilinx_parallel4, xilinx_svffile**

  In the case of xilinx_svffile, the JTAG commands are written into a file specified by the **fname** option

- **port** <parallel port name>

  Valid arguments for port are **lpt1, lpt2**

- **fname** <filename>

Filename for creating the SVF file

JTAG chain options

- **`partname`** <devicename>

  Name of the device

- **`devicenr`** <device position>

  Position of the device in the JTAG chain

- **`irlength`** <length of the JTAG Instruction Register>

  Length of the IR register of the device. This information can be found in the device BSDL file.

- **`idcode`** <device idcode>

  JTAG Idcode of the device

MicroBlaze options

- **`devicenr`** <FPGA device position>

  Position of the FPGA device containing the MicroBlaze, in the JTAG chain

With a hardware target, user programs can be downloaded from **`mb-gdb`** directly onto a remote hardware board and be executed with support of the xmd stub running on the board. A sample session of XMD with a hardware stub target is shown below.

```
XMD% mbconnect stub
```

Now XMD connects to the hardware target and waits for a connection from **`mb-gdb`**. Refer to the GNU Debugger chapter to see how to start **`mb-gdb`**, make a remote connection from **`mb-gdb`** to **`xmd`**, download a program onto the target and debug the program.

To debug a program by downloading on the remote hardware board, the program must be compiled with **-g -xl-mode-xmdstub** options to mb-gcc**.**

Note: User Program outputs. If the program has any I/O functions like print() or putnum(), that write output onto the UART or JTAG Uart, it will be printed on the console/terminal where the **xmd** was started. (Refer to the MicroBlaze Libraries chapter for libraries and I/O functions information).

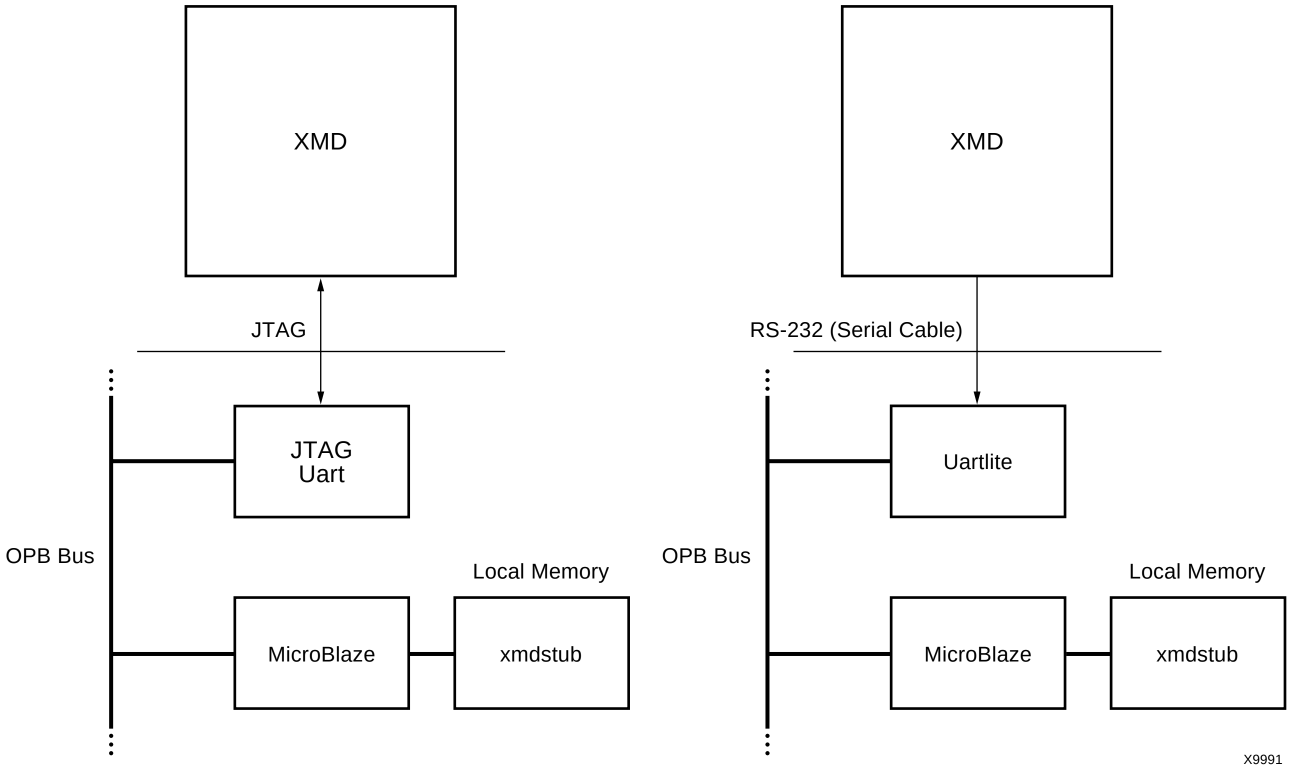


Figure 13-4: **MicroBlaze stub Target with JTAG UART and Uartlite**

Stub Target Requirements

To debug programs on the hardware board using XMD, the following requirements have to be met.

- **xmd** uses a JTAG or serial connection to communicate with **xmdstub** on the board. Hence a JTAG Uart or a Uart designated as DEBUG_PERIPHERAL in the mss file is needed on the target MicroBlaze system.

  Platform Generator can create a system that includes a JTAG Uart or a Uart, if specified in the system's mhs file. For more information on creating a system with a Uart or a JTAG Uart, refer to the MicroBlaze Hardware Specification Format chapter. The cables supported with the xmdstub mode are : Xilinx Parallel Cable III and Parallel Cable IV.

- **xmdstub** on the board uses the JTAG Uart or Uart to communicate with the host computer. Hence, it must be configured to use the JTAG Uart or Uart in the MicroBlaze system.

  Library Generator can configure the **xmdstub** to use the DEBUG_PERIPHERAL in the system. **libgen** will generate a **xmdstub** configured for the DEBUG_PERIPHERAL and put it in **code/xmdstub.elf** as specified by the

XMDSTUB attribute in the mss file. For more information, refer to the Library Generator chapter.

- **`xmdstub`** executable must be included in the MicroBlaze local memory at system startup. To have the **`xmdstub`** included in the MicroBlaze local memory, the `xmdstub.elf` file should be specified in the user's mss file as follows:

  ```
  PARAMETER XMDSTUB=code/xmdstub.elf
  ```

  Data2BRAM can populate the MicroBlaze memory with **`xmdstub`**. libgen generates a Data2BRAM script file that can be used to populate the BRAM contents of a bitstream containing a MicroBlaze system. It uses the executable specified in the DEFAULT_INIT.

- Any user program that has to be downloaded on the board for debugging should have a program start address higher than 0x400 and the program should 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 **`-xl-mode-xmdstub`**. For source level debugging, programs should also be compiled with **`-g`** option. While initially verifying the functional correctness of a C program, it is advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may 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 XMD. A sample session of XMD and GDB is shown below.

```
XMD% mbconnect sim
Connected to MicroBlaze “sim” target. id = 0
Starting Remote GDB server for “sim” target (id = 0) at TCP port no 1234
XMD%
```

Now XMD is running with the simulator target and waiting for a connection from mb-gdb. The xmd Tcl prompt can also be used simultaneously for executing xmd commands.

Refer to the MicroBlaze GNU Debugger document to see how to start **`mb-gdb`**, make a remote connection from **`mb-gdb`** to **`xmd`**, download a program onto the target and debug the program. With **`xmd`** and **`mb-gdb`**, the debugging user interface is uniform with simulation or hardware targets.

MicroBlaze Simulation Target Options

`mbconnect` sim [-memsize <**`size`**>]

- **`memsize`** <size>

  Size of the memory allocated in the simulator. Programs can access the memory range from **`0`** to **`size-1`**

Simulation Statistics

While **`mb-gdb`** is connected to XMD with the simulator target, the statistics of the cycle-accurate simulator can be viewed from **`xmd`** as follows:

- In the **`xmd`** prompt type **`stats`**

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, the following requirements have to be met.

- Programs should be compiled for debugging and should be linked with the startup code in crt0.o

  `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. (Explicit option: **`-xl-mode-executable`**)

- Programs can have a maximum size of 64Kbytes only.
- Currently, XMD with simulator target does not support the simulation of OPB peripherals.

XMD Tcl commands

In the Tcl interface mode, xmd starts a Tcl shell augmented with xmd commands. All xmd Tcl commands start with 'x' and can be listed from xmd by typing "x?". It is recommended to use the Tcl wrappers for these internal commands as described in Figure 13-1. The Tcl wrappers would pretty print the output of most of these commands and also provide more options. While the Tcl wrappers will be backwards compatible, these x<name> commands may be deprecated in a future EDK release.

- xrmem *target addr* [*num*]

  Read num bytes or 1 byte from memory address <addr>

- xwmem *target addr value*

  Write a 8-bit byte *value* at the specified memory *addr.*

- xrreg *target* [*reg*]

  Read all registers or only register number **`reg.`**

- xwreg *target reg value*

  Write a 32-bit *value* into register number *reg*

- xdownload *target* [-data] *filename* [*addr*]

  Download the given ELF or data file (with -data option) onto the current target's memory. If no address is provided along with ELF file, the download address is determined from the ELF file by reading its headers. If an address is provided with the ELF file, it is treated as PIC code (Position Independent Code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics. Note that NO Bounds checking is done by xmd, except preventing writes into xmdstub area (address 0x0 to 0x400).

- xcontinue *target* [*addr*]

  Continue execution from the current PC or from the optional address argument.

- xcycle_step *target [cycles]*

  Cycle step through one clock cycle of PowerPC ISS. If *cycles* is specified, then step "*cycles*" number of clock cycles. **Note:** This command is only for PowerPC ISS target.

- xstep *target*

  Single step one MicroBlaze instruction. If the PC is at an IMM instruction the next instruction is executed as well. During a single step, interrupts are disabled by keeping the BIP flag set. Use xcontinue with breakpoints to enable interrupts while debugging.

- xreset target [reset type]

  Reset target. Optionally provide target specific reset types like signals mentioned in , "XMD MicroBlaze Hardware target signals".

- xbreakpoint *target addr <sw | hw> [<Hardware Breakpoint ID>]*

  Set a breakpoint at the given address. Note - Breakpoints on instructions immediately following **`imm`** instruction can lead to undefined results for xmdstub target. The Hardware Breakpoint ID is valid only for the MicroBlaze MDM target, where this is used to set a specific breakpoint.

- xremove *target addr [<Hardware Breakpoint ID>]*

  Remove breakpoint at given address.

- xlist *target*

  List all the breakpoint addresses.

- xdisassemble *inst*

  Disassemble and display one 32-bit instruction.

- xsignal *target signal*

  Send a signal to a hardware target. This is only supported by the JTAG UART when the debug signals for Processor Break, Reset and System reset are connected to MicroBlaze and the OPB bus. Platform Generator automatically connects these signals by default of the implicit name matching in the respective MPD files. Supported signals are listed in Table 13-2.

Table 13-2: **XMD MicroBlaze Hardware target signals**

Signal Name (value)	Description
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 addr 0x18
Non-maskable Break (0x10)	Similar to the Break signal but works even while the BIP flag is already set. Refer the MicroBlaze ISA documentation for more information about the BIP flag.
System Reset (0x40)	Resets the entire system by sending an OPB Rst using the JTAG UART Debug_SYS_Rst signal.
Processor Reset (0x80)	Resets MicroBlaze using the JTAG UART Debug_Rst signal.

- xstats *target* [*options*]

  Display the simulation statistics for the current session.'reset' option can be provided to reset the simulation statistics.

- xtargets [*target*]

  Print the target ID and target type of all current targets or a specific target.

- xtraceopen *target [filename]*

  Open a trace file to collect trace information. If *filename* is not specified, *isstrace.out* is used as the default filename.**Note:** This command is only for PowerPC ISS target.

- xtracestart *target*

Start collecting trace information. Trace file should be opened before trace start.**Note:** This command is only for PowerPC ISS target.

- xtracestop *target*

  Stop collecting trace information. **Note:** This command is only for PowerPC ISS target.

- xtraceclose *target*

  Close the trace file. **Note:** This command is only for PowerPC ISS target.

- xuart *<r|w|s> [<data>]*

  Perform one of 3 UART operations on the MDM's UART if it is enabled. This command is valid only for the MDM target.

  xuart <r> **-** Read byte from the MDM UART

  xuart <w> <data> **-** Write byte onto the MDM UART

  xuart <s> **-** Read the status of MDM Uart

Chapter 14

Platform Specification Format (PSF)

Overview

The Platfom Specification Format (PSF) defines the compatible set of infrastructure files for a EDK tool release. The infrastructure files are BBD, MDD, MHS, MPD, MSS, MVS, and PAO files.

This chapter includes the following sections:

Files

BBD - Black Box Definition

The Black Box Definition (BBD) file manages the file locations of optimized hardware netlists for the black-box sections of your peripheral design.

Please see Chapter 18, "Black-Box Definition (BBD)," for more information.

MDD - Microprocessor Driver Definition

An MDD file contains directives for customizing software drivers.

Please see Chapter 19, "Microprocessor Driver Definition (MDD)," for more information.

MHS - Microprocessor Hardware Specification

The Microprocessor Hardware Specification (MHS) file defines the hardware component. An MHS file is supplied by the user as an input to the Platform Generator (PlatGen) tool.

Please see Chapter 15, "Microprocessor Hardware Specification (MHS)," for more information.

MPD - Microprocessor Peripheral Definition

The Microprocessor Peripheral Definition (MPD) file defines the interface of the peripheral.

Please see Chapter 16, “Microprocessor Peripheral Description (MPD),” for more information.

MSS - Microprocessor Software Specification

An MSS file is supplied by the user as an input to the Library Generator (LibGen). The MSS file contains directives for customizing libraries, drivers and file systems.

Please see Chapter 19, “Microprocessor Software Specification (MSS),” for more information.

MVS - Microprocessor Verification Specification

An MVS file is supplied by the user as an input to the Simulation Model Generator (SimGen) tool. The MVS file contains directives for customizing a simulation model for a defined system.

Please see Chapter 17, “Microprocessor Verification Specification (MVS),” for more information.

PAO - Peripheral Analyze Order

A PAO (Peripheral Analyze Order) file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation.

Please see Chapter 17, “Peripheral Analyze Order (PAO),” for more information.

Version Scheme

Form of the version level is X.Y.Z

- X - major revision
- Y - minor revision
- Z - patch level

Version Setting for MHS, MSS, and MVS

In the body of the MHS, MSS, and MVS file, add the following statement:

Format

```
PARAMETER VERSION = 2.0.0
```

The version is specified as a literal of the form 2.0.0.

Version Setting for BBD, MPD, and PAO

The version level is concatenated to the basename of the data files. The literal form of the version level is vX_Y_Z.

Format

- <*ipname*>_vX_Y_Z.mpd
- <*ipname*>_vX_Y_Z.bbd

- <*ipname*>_vX_Y_Z.pao
- <*ipname*>_vX_Y_Z.mdd

Load Path

Refer to Figure 14-1 for a depiction of the peripheral directory structure.

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

- Current directory
- Set the EDK tool option **-lp** option

EDK tools use a search priority mechanism to locate peripherals, as follows:

1. Search the pcores directory in the project directory
2. Search <library_path>/<Library Name>/pcores as specified by the **-lp** option

Search XILINX_EDK/hw/<Library Name>/pcores

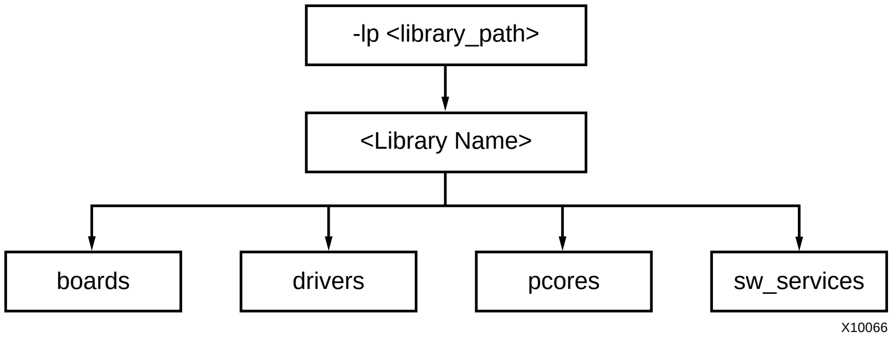


Figure 14-1: **Peripheral Directory Structure**

Using Versions

You can create multiple versions of your peripheral. The version is specified as a literal of the form 1.00.a. The version is always specified in lower-case.

At the MHS level, use the HW_VER parameter to set the hardware version. The Platform Generator concatenates a "_v" and translates periods to underscores. The peripheral name and HW_VER are joined together to form a name for a search level in the load path. For example, if your peripheral is version 1.00.a, then the MPD, BBD, and PAO files are found in the following location:

<repository_dir>/pcores/<*peripheral*>_v1_00_a/data (UNIX)

<repository_dir>\pcores\<*peripheral*>_v1_00_a\data (PC)

Creating User IP

To build your own refernce depends on the characteristics of your design.

Is Your IP Pure HDL?

Read about MPD and PAO files. The MPD keyword IPTYPE has the value HDL.

Is Your IP Only A Black-Box Netlist?

Read about MPD and BBD files. The MPD keyword IPTYPE has the value BLACKBOX.

Is Your IP A Mixture Of Black-Box Netlists And VHDL or Verilog?

Read about MPD, BBD, and PAO files. The MPD keyword IPTYPE has the value MIX.

Chapter 15

Microprocessor Hardware Specification (MHS)

Overview

The Microprocessor Hardware Specification (MHS) file defines the hardware component. An MHS file is supplied by the user as an input to the Platform Generator (PlatGen) tool. An MHS file defines the configuration of the embedded processor system, and includes the following:

- Bus architecture
- Peripherals
- Processor
- Connectivity of the system
- Interrupt request priorities
- Address space

This chapter includes the following sections:

MHS Syntax

MHS file syntax is case insensitive. Current version is 2.1.0.

MHS parameter / component / instance / signal name must be HDL (VHDL, Verilog) compliant. VHDL and Verilog have certain naming rules and conventions that must be followed.

Due to this translation, MHS is inherently violating syntax rules in either VHDL or Verilog in the downstream HDL compliant synthesis / simulation tools.

For example, it is illegal in VHDL to use an instance name that already exists as a component name.

```
microblaze : microblaze
port map ( <snip> );
```

However, Verilog allows such a declaration:

```
microblaze microblaze ( <snip> );
```

It is also illegal in VHDL to declare an object (parameter / component / instance / signal) name that already exists as a name of another object. For example, it is illegal to declare in VHDL a signal name, MYTESTNAME, and also declare an instance name of MYTESTNAME.

```
signal MYTESTNAME : std_logic;
MYTESTNAME : microblaze
port map ( <snip> );
```

However, this is legal in Verilog.

It's the user's responsibility to reconize their output format and comply with the rules of the HDL language.

Comments

You can insert comments in the MPD file without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments can continue to the end of the line
- Comments can be anywhere on the line

Format

Use the following format at the beginning of a component definition:

```
BEGIN peripheral_name
```

The BEGIN keyword signifies the beginning of a new peripheral.

Use the following format for assignment commands:

```
command name = value
```

Use the following format to end a peripheral definition:

```
END
```

Assignment Commands

There are three assignment commands:

1. BUS_INTERFACE
2. PARAMETER
3. PORT

MHS Example

The following is an example MHS file:

```
# Parameters
PARAMETER VERSION = 2.1.0

# Global Ports

# Assign power signals
PORT vcc_out = net_vcc, DIR=OUTPUT
PORT gnd_out = net_gnd, DIR=OUT
PORT gnd_out6 = net_gnd, DIR=OUTPUT, VEC=[0:5]

PORT intr1 = intr_1, DIR=IN, SENSITIVITY=EDGE_RISING, SIGIS=INTERRUPT
PORT intr2 = intr2, DIR=INPUT, SENSITIVITY=LEVEL_HIGH, SIGIS=INTERRUPT

# Assign constant signals
PORT const1 = 0b1010, DIR=OUTPUT, VEC=[0:3]
PORT const2 = 0xC, DIR=OUTPUT, VEC=[0:3]

PORT sys_rst = sys_rst, DIR=IN
PORT sys_clk = sys_clk, DIR=IN, SIGIS=CLK
PORT gpio_io = gpio_io, DIR=INOUT, VEC=[0:31]

# Sub Components

######################################################################
BEGIN lmb_v10
PARAMETER INSTANCE = ilmb_v10
PARAMETER HW_VER = 1.00.a
PORT LMB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
######################################################################
BEGIN lmb_v10
PARAMETER INSTANCE = dlmb_v10
PARAMETER HW_VER = 1.00.a
PORT LMB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
######################################################################
BEGIN opb_v20
PARAMETER INSTANCE = myopb_bus
PARAMETER HW_VER = 1.10.b
PARAMETER C_PROC_INTRFCE = 0
PORT OPB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
######################################################################
BEGIN opb_gpio
PARAMETER INSTANCE = mygpio
PARAMETER HW_VER = 1.00.a
PARAMETER C_GPIO_WIDTH = 32
PARAMETER C_ALL_INPUTS = 0
PARAMETER C_BASEADDR = 0xffff0100
PARAMETER C_HIGHADDR = 0xffff01ff
PORT GPIO_IO = gpio_io
PORT OPB_Clk = sys_clk
BUS_INTERFACE SOPB = myopb_bus
END
######################################################################
BEGIN bram_block
```

```
PARAMETER INSTANCE = bram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = ilmb1_porta
BUS_INTERFACE PORTB = dlmb1_portb
END
######################################################################
BEGIN lmb_bram_if_cntlr
PARAMETER INSTANCE = my_ilmb_cntlr1
PARAMETER HW_VER = 1.00.b
PARAMETER C_BASEADDR = 0x00000000
PARAMETER C_HIGHADDR = 0x00000fff
BUS_INTERFACE SLMB = ilmb_v10
BUS_INTERFACE BRAM_PORT = ilmb1_porta
END
######################################################################
BEGIN lmb_bram_if_cntlr
PARAMETER INSTANCE = my_dlmb_cntlr1
PARAMETER HW_VER = 1.00.b
PARAMETER C_BASEADDR = 0x00000000
PARAMETER C_HIGHADDR = 0x00000fff
BUS_INTERFACE SLMB = dlmb_v10
BUS_INTERFACE BRAM_PORT = dlmb1_portb
END
######################################################################
BEGIN microblaze
PARAMETER INSTANCE = mblaze
PARAMETER HW_VER = 2.00.a
PORT Clk = sys_clk
BUS_INTERFACE DLMB = dlmb_v10
BUS_INTERFACE ILMB = ilmb_v10
BUS_INTERFACE DOPB = myopb_bus
PORT Interrupt = mblaze_intr
END
######################################################################
# Priorities are numbered 1 to N, where 1 is the highest priority
BEGIN opb_intc
PARAMETER INSTANCE = opb_intc_1
PARAMETER HW_VER = 1.00.c
PARAMETER C_HIGHADDR = 0xC800001F
PARAMETER C_BASEADDR = 0xC8000000
PARAMETER C_HAS_IPR = 1 # Interrupt Pending Register present
PARAMETER C_HAS_SIE = 0 # Set Interrupt Enable bits not present
PARAMETER C_HAS_CIE = 0 # Clear Interrupt Enable bits not present
PARAMETER C_HAS_IVR = 0 # Interrupt Vector Register not present
BUS_INTERFACE SOPB = myopb_bus
PORT Intr = intr2 & intr_1
PORT Irq = mblaze_intr
PORT OPB_Clk = sys_clk
END
```

Bus Interface

A bus interface is a grouping of interconnecting signals which are related.

Several components often have many of the same ports, requiring redundant port declaration for each component. Every component connected to a OPB bus, for example, must have the same ports defined and connected together.

A bus interface provides a high level of abstraction for component connectivity of a common interface. Components can use a bus interface the same as if it were a single port. In its simplest form, a bus interface can be considered a bundle of signals.

The following list are recommendations for bus labels:

Table 15-1: **Bus Labels**

Bus Name	Description
SDCR	Slave DCR interface
SLMB	Slave LMB interface
MOPB	Master OPB interface
MSOPB	Master-slave OPB interface
SOPB	Slave OPB interface
MPLB	Master PLB interface
MSPLB	Master-slave PLB interface
SPLB	Slave PLB interface

For components that have more than one bus interface, please look at the MPD file for a definition of listed bus interface labels. For example, the data-side OPB and instruction-side OPB are named DOPB and IOPB, respectively.

A bus interface is assigned by name to an instance of the bus in your system.

Example

For example, the OPB bus instance is named “myopb”, and a connection to the OPB slave interface of the OPB Uart Lite is made with the bus_interface command.

```
BEGIN opb_uartlite
PARAMETER HW_VER = 1.00.b
PARAMETER INSTANCE = myuartlite
PARAMETER C_HIGHADDR = 0xFFFF80FF
PARAMETER C_BASEADDR = 0xFFFF8000
BUS_INTERFACE SOPB = myopb
PORT OPB_Clk = sys_clk
PORT RX = rx1
PORT TX = tx1
PORT Interrupt = uart_intr
END
```

Global Parameter

A global parameter is defined outside of a BEGIN-END block.

A global parameter can have the following keywords:

Table 15-2: **Global Parameter keywords**

Keyword	Values	Default	Definition
VERSION	2.1.0	No Default	MHS version

VERSION

Use the VERSION keyword to set the MHS version.

Format

```
PARAMETER VERSION = 2.1.0
```

The version is specified as a literal of the form 2.1.0.

Local Parameter

A local parameter is defined between a BEGIN-END block.

A local parameter can have the following keywords:

Table 15-3: **Local Parameter Keywords**

Keyword	Values	Default	Definition
HW_VER	1.00.a	No Default	Hardware version
INSTANCE		No Default	User-defined instance name Must be lower-case

HW_VER

Use the HW_VER keyword to set the hardware version.

Format

```
PARAMETER HW_VER = 1.00.a
```

The version is specified as a literal of the form 1.00.a.

INSTANCE

Use the INSTANCE keyword to set the instance name of peripheral. This keyword is mandatory, and the instance name must be specified in lower-case.

Format

```
PARAMETER INSTANCE = my_uart0
```

Local Bus Interface

A local bus interface between a BEGIN-END block can have the following keywords:

Table 15-4: **Local Bus Interface Keywords**

Keyword	Values	Default	Definition
POSITION	*integer*	Order retained as listed in the MHS	Position of peripheral on the bus. Use to define master request priority or DCR slave rank.

POSITION

Use the POSITION keyword to set the hardware version.

Format

```
BUS_INTERFACE MOPB=opb_bus_inst, POSITION=integer
```

Where *integer* is a positive integer. Highest position is "1".

Global Port

A global port outside of a BEGIN-END block can have the following keywords:

Table 15-5: **Global Port Keywords**

Keyword	Values	Default	Definition
DIR	IN, INPUT, I OUT, OUTPUT, O INOUT, IO	O	Direction mode
EDGE	RISING FALLING	No Default	Interrupt edge sensitivity (deprecated)
LEVEL	HIGH LOW	No Default	Interrupt level sensitivity (deprecated)
SENSITIVITY	EDGE_FALLING EDGE_RISING LEVEL_HIGH LEVEL_LOW	No Default	Interrupt sensitivity
SIGIS	CLK INTERRUPT RST	No Default	Signal classification
VEC	[A:B]	No Default	Vector dimension

DIR

The driver direction of a signal is specified by the DIR keyword.

Format

```
PORT mysignal = "", DIR=direction
```

Where *direction* is either INPUT, IN, I, OUTPUT, OUT, O, INOUT, or IO.

EDGE

The edge sensitivity of an interrupt signal is specified by the EDGE keyword. Its use is deprecated. Please use the SENSITIVITY keyword.

Format

```
PORT interrupt = “”, DIR=O, EDGE=edge_value, SIGIS=INTERRUPT
```

Where *edge_value* is either RISING or FALLING.

LEVEL

The level sensitivity of an interrupt signal is specified by the LEVEL keyword. Its use is deprecated. Please use the SENSITIVITY keyword.

Format

```
PORT interrupt = “”, DIR=O, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

SENSITIVITY

The interrupt sensitivity of an interrupt signal is specified by the SENSITIVITY keyword. This supercedes the EDGE and LEVEL keywords.

Format

```
PORT interrupt = “”, DIR=O, SENSITIVITY=value, SIGIS=INTERRUPT
```

Where the *value* is either EDGE_FALLING, EDGE_RISING, LEVEL_HIGH or LEVEL_LOW.

SIGIS

The class of a signal is specified by the SIGIS keyword.

Format

```
PORT mysig = “”, DIR=O, SIGIS=value
```

Where the *value* is either CLK, INTERRUPT, or RST. The following table lists SIGIS usage:

Table 15-6: **SIGIS Usage**

SIGIS	Usage
CLK	• Display purposes - XPS lists all clock signals • Clock buffer insertion - PlatGen infers clock buffers
INTERRUPT	• Display purposes - XPS lists all interrupt signals • Interrupt vector generation - PlatGen encodes the priority interrrupt vector
RST	• Display purposes - XPS lists all reset signals

VEC

The vector width of a signal is specified by the VEC keyword.

Format

```
PORT mysignal = “”, DIR=I, VEC=[A:B]
```

Local Port

A local port is a port defined between a BEGIN-END block. A local port does not have keywords.

Design Considerations

This section provides general design considerations.

Power Signals (net_gnd/net_vcc)

Power signals are signals that are constantly driven with either GND (net_gnd) or VCC (net_vcc).

Format

```
PORT mysignal = power_signal
```

In this example, *power_signal* is either “net_vcc” or “net_gnd”. PlatGen expands “net_vcc” or “net_gnd” to the appropriate vector size.

Unconnected Ports

Unconnected output ports are assigned open, and unconnected input ports are either set to GND (net_gnd) or VCC (net_vcc).

An unconnected port is identified as an empty double-quote (““) string.

PlatGen resolves the driver value on unconnected input ports by the INITIALVAL keyword as defined in the MPD.

Format

```
PORT mysignal = “”
```

Constant Assignments

Use 0b denotation to define a binary constant or 0x for a hex constant. An underscore (_) can be used for readability.

Format

```
PORT mysignal = 0b1010_0101 # mysignal is 8-bits
```

Or

```
PORT mysignal = 0xA5 # mysignal is 8-bits
```

Defining Memory Size

Memory sizes are based on C_BASEADDR and C_HIGHADDR settings. Use the following format when defining memory size:

```
PARAMETER C_HIGHADDR= 0xFFFF00FF
PARAMETER C_BASEADDR= 0xFFFF0000
```

All memory sizes must be 2^N where N is a positive integer, and 2^N boundary overlaps are not allowed.

The range specified by C_BASEADDR and C_HIGHADDR must comprise a complete, contiguous power-of-two range, such that range = 2^N, and the N least significant bits of C_BASEADDR must be zero.

Internal vs External Signals

By default, all signals defined between a BEGIN-END block are internal signals.

External signals are available through the port-declaration of the top-level module. Use the PORT command outside of a BEGIN-END block to declare the external signal.

External Interrupt Signals

For internal interrupts, each interruptible peripheral instance defines an interrupt signal locally.

For external interrupts, use the PORT command outside of a BEGIN-END block to declare the external signal and define the interrupt sensitivity.

Format

```
PORT my_int1 = my_int1, LEVEL=HIGH, DIR=INPUT
```

Internal Interrupt Signals

For the opb_intc and dcr_intc components, the interrupt vector will be a concatenation of the locally defined interrupt signals and / or external interrupts. The position of the interrupt signal defines the priority. The interrupt vector is in little-endian format, where the highest priority interrupt sits at the LSBit position.

Format

```
PORT intr = my_int1 & uart_intr & wdt_intr & tb_intr & int2
```

If there is only one interrupt defined in the platform, then you may be able to connect it directly to the MicroBlaze processor. The MicroBlaze processor's interrupt is level sensitive. Consequently, any other level sensitive interrupt line from a peripheral can be connected directly. However, if the peripheral's interrupt line is edge sensitive, then you must use the interrupt controller. If you connect an edge sensitive signal to a level sensitive signal, you may miss an interrupt.

Chapter 16

Microprocessor Peripheral Description (MPD)

Overview

The Microprocessor Peripheral Definition (MPD) file defines the interface of the peripheral.

An MPD file has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters and default values
- Any MPD parameter is overwritten by the equivalent MHS assignment (refer to the *Microprocessor Hardware Specification Format* document for more details)

Individual peripheral documentation contains information on all MPD file keywords.

This chapter includes the following sections:

MPD Syntax

MPD file syntax is case insensitive. Current version is 2.1.0.

MPD parameter / signal name must be HDL (VHDL, Verilog) compliant. VHDL and Verilog have certain naming rules and conventions that must be followed.

The MPD file is supplied by the IP provider and provides peripheral information. This file lists ports and default connectivity to the bus interface. Parameters that you set in this file are mapped to generics for VHDL or parameters for Verilog.

Comments

You can insert comments in the MPD file without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments continue to the end of the line
- Comments can be anywhere on the line

Format

Use the following format at the beginning of a component definition:

```
BEGIN peripheral_name
```

The BEGIN keyword signifies the beginning of a new peripheral.

Use the following format for assignment commands:

```
command name = value
```

Use the following format to end a peripheral definition:

```
END
```

Assignment Commands

There are five assignment commands:

1. bus_interface
2. io_interface
3. option
4. parameter
5. port

Signal Direction

Signals have three modes. Signal mode indicates its driver direction, and if the port can be read from within the peripheral.

The three modes and their accepted values are as follows:

- input - [input, in, i]
- output - [output, out, o]
- inout - [inout, io]

MPD Example

The following is an example MPD file:

```
BEGIN opb_gpio

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION SIM_MODELS = BEHAVIORAL:STRUCTURAL

## Bus Interfaces
```

```
BUS_INTERFACE BUS=SOPB, BUS_STD=OPB, BUS_TYPE=SLAVE

## Generics for VHDL or Parameters for Verilog
PARAMETER C_BASEADDR=0xFFFFFFFF, DT=std_logic_vector, MIN_SIZE=0x100, BUS=SOPB
PARAMETER C_HIGHADDR=0x00000000, DT = std_logic_vector, BUS=SOPB
PARAMETER C_OPB_DWIDTH=32, DT=integer, BUS=SOPB
PARAMETER C_OPB_AWIDTH=32, DT=integer, BUS=SOPB
PARAMETER C_GPIO_WIDTH=32, DT=integer
PARAMETER C_ALL_INPUTS=0, DT=integer

## Ports
PORT OPB_Clk = “”, DIR=IN, SIGIS=CLK, BUS=SOPB
PORT OPB_Rst = OPB_Rst, DIR=IN, BUS=SOPB
PORT OPB_ABus = OPB_ABus, DIR=IN, VEC=[0:C_OPB_AWIDTH-1], BUS=SOPB
PORT OPB_BE = OPB_BE, DIR=IN, VEC=[0:C_OPB_DWIDTH/8-1], BUS=SOPB
PORT OPB_DBus = OPB_DBus, DIR=IN, VEC=[0:C_OPB_DWIDTH-1], BUS=SOPB
PORT OPB_RNW = OPB_RNW, DIR=IN, BUS=SOPB
PORT OPB_select = OPB_select, DIR=IN, BUS=SOPB
PORT OPB_seqAddr = OPB_seqAddr, DIR=IN, BUS=SOPB
PORT GPIO_DBus = Sl_DBus, DIR=OUT, VEC=[0:C_OPB_DWIDTH-1], BUS=SOPB
PORT GPIO_errAck = Sl_errAck, DIR = OUT, BUS=SOPB
PORT GPIO_retry = Sl_retry, DIR = OUT, BUS=SOPB
PORT GPIO_toutSup = Sl_toutSup, DIR=OUT, BUS=SOPB
PORT GPIO_xferAck = Sl_xferAck, DIR=OUT, BUS=SOPB
PORT GPIO_IO = “”, DIR=INOUT, VEC=[0:C_GPIO_WIDTH-1], ENABLE=MULT

END
```

Bus Interface

A bus interface is a grouping of interface ports which are related.

Several components often have many of the same ports, requiring redundant port declaration for each component. Every component connected to a OPB bus, for example, must have the same ports defined and connected together.

A bus interface provides a high level of abstraction for component connectivity of a common interface. Components can use a bus interface the same as if it were a single port. In its simplest form, a bus interface can be considered a bundle of signals.

Bus Interface Keywords

A bus interface can have the following keywords:

Table 16-1: **Bus Interface Keywords**

Keyword	Values	Default	Definition
BUS	*string*	No Default	Bus label
BUS_STD	DCR FSL DSOCM ISOCM LMB OPB PLB TRANSPARENT	No Default	Bus standard
BUS_TYPE	MASTER MASTER_SLAVE SLAVE UNDEF	No Default	Bus type

BUS

The label of a bus interface is specified by the BUS keyword.

Format

```
BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type
```

Where *bus_label* is a string.

BUS_STD

The bus standard of a bus interface is specified by the BUS_STD keyword.

Format

```
BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type
```

Where *bus_std* is either DCR, LMB, OPB, PLB, or TRANSPARENT.

A TRANSPARENT bus interface is not tied to any physical bus component.

BUS_TYPE

The bus type of a bus interface is specified by the BUS_TYPE keyword.

Format

```
BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type
```

Where *bus_type* is either MASTER, MASTER_SLAVE, SLAVE, or UNDEF.

Bus Interface Naming Conventions

The following list are recommendations for bus labels:

Table 16-2: **Recommended Bus Labels**

Bus Label	Description
SDCR	Slave DCR interface
SLMB	Slave LMB interface
MOPB	Master OPB interface
MSOPB	Master-slave OPB interface
SOPB	Slave OPB interface
MPLB	Master PLB interface
MSPLB	Master-slave PLB interface
SPLB	Slave PLB interface

IO Interface

An IO interface defines an interface between at least one core and some hardware device on a board. One core may connect to more than one IO interface.

Physically is a set of PIN LOCs which are fixed on the chip and connected to a hardware device.

May imply that certain parameters on the core(s) connected to this interface has fixed values.

IO Interface Keywords

An IO interface can have the following keywords:

Table 16-3: **IO Interface Keywords**

Keyword	Values	Default	Definition
IO_IF	*string*	No Default	IO label
IO_TYPE	CLOCK GPIO RESET UART SDRAM ETHERNET	No Default	IO type

IO_IF

The label of an IO interface is specified by the IO_IF keyword.

Format

```
IO_INTERFACE IO_IF=io_label, IO_TYPE=io_type
```

Where *io_label* is a user defined string.

IO_TYPE

The IO type of an IO interface is specified by the IO_TYPE keyword.

Format

```
IO_INTERFACE IO_IF=io_label, IO_TYPE=io_type
```

Where *io_type* is either CLOCK, GPIO, RESET, UART, SDRAM, or ETHERNET.

Option

An option defines a tool directive.

Option Keywords

An option can have the following keywords:

Table 16-4: **Option Keywords**

Keyword	Values	Default	Definition
ADDR_SLICE	*integer*	No Default	Address slice of BRAM controller
ALERT	*string*	No Default	Alert message
ARCH_SUPPORT	*string*	ALL	List of supported FPGA architectures
AWIDTH	*integer*	No Default	Address width
BUS_STD	DCR FSL LMB OPB PLB	No Default	Define bus standard of BUS or BUS_ARBITER cores
CORE_STATE	ACTIVE DEPRECATED DEVELOPMENT OBSOLETE	ACTIVE	Core state
DESC	*string*	No Default	Allows a short description of the core to be displayed by the GUI tools
DWIDTH	*integer*	No Default	Data width
HDL	BOTH VERILOG VHDL	VHDL	HDL design availability.

Table 16-4: **Option Keywords**

Keyword	Values	Default	Definition
IMP_NETLIST	TRUE FALSE	FALSE	Synthesize HDL to a hardware implementation netlist
IP_GROUP	ALLIANCE INFRASTRUCTURE LOGICORE REFERENCE USER	USER	Core group classification
IPLEVEL_DRC_PROC	*string*	No Default	Tcl entry point for the IP-level DRC routine
IPTYPE	BRIDGE BUS BUS_ARBITER IP PERIPHERAL PROCESSOR	IP	Type of component
LONG_DESC	*string*	No Default	Allows a long description of the core to be displayed by the GUI tools
MAX_MASTERS	*integer*	No Default	Define maximum number of masters
MAX_SLAVES	*integer*	No Default	Define maximum number of slaves
NUM_WRITE_ENABLES	*integer*	No Default	Number of write enables of BRAM controller
SIM_MODELS	BEHAVIORAL STRUCTURAL TIMING	No Default	Simulation model availability
SPECIAL	BRAM BRAM_CNTLR	No Default	A class of components that require special handling
STYLE	BLACKBOX MIX HDL	HDL	Design style
SYSLEVEL_DRC_PROC	*string*	No Default	Tcl entry point for the system-level DRC routine
TCL_FILE	*string*	No Default	Define Tcl file name
TOP	*string*	No Default	Top-level name (deprecated)
USAGE_LEVEL	ADVANCED_USER ALL_USERS	ALL_USERS	Defines usage level

ADDR_SLICE

The address slice position supported by the BRAM controller is specified by the ADDR_SLICE keyword.

Format

```
OPTION ADDR_SLICE = 29
```

Used only by components of SPECIAL=BRAM_CNTLR.

ALERT

A message alert for the IP core is specified with the ALERT keyword.

Format

```
OPTION ALERT = “This belongs to Xilinx”
```

ARCH_SUPPORT

List of supported FPGA architectures. Valid values: all, spartan2, spartan2e, spartan3, virtex, virtexe, virtex2, virtex2p. Default is ALL. Supports a colon “:” separated list of elements. But, may also take a single element.

Format

```
OPTION ARCH_SUPPORT = virtex2:spartan2e
```

Format

```
OPTION ARCH_SUPPORT = virtex2
```

AWIDTH

The address width is specified by the AWIDTH keyword.

Format

```
OPTION AWIDTH = 32
```

BUS_STD

Define bus standard of BUS or BUS_ARBITER cores.

Format

```
OPTION BUS_STD = value
```

Where *value* is either DCR, FSL, LMB, OPB, or PLB. No default.

CORE_STATE

The state of the IP core is specified with the CORE_STATE keyword.

Format

```
OPTION CORE_STATE = ACTIVE
```

The following table lists CORE_STATE values:

Table 16-5: **CORE_STATE Values**

CORE_STATE	Definition
ACTIVE	Core is active (full uninhibited use) by EDK (default)
DEPRECATED	Core is deprecated. EDK tools allow use of core, but issues a warning that the core is deprecated
DEVELOPMENT	Core is in development and will be synthesized each time PlatGen is executed (no cache of synthesis results)
OBSOLETE	Core is obsolete. EDK tools issue an error that this core is no longer valid.

DESC

Allows a short description of the core to be displayed by the GUI tools. The short description replaces the core name in display field of the core.

Format

```
OPTION DESC = “OPB GPIO”
```

DWIDTH

The data width is specified by the DWIDTH keyword.

Format

```
OPTION DWIDTH = 32
```

HDL

The HDL keyword lists the HDL availability. The design is either completely written in VHDL, or completely written in Verilog. The BOTH value signifies that a design is available in VHDL or Verilog format.

Format

```
OPTION HDL = VERILOG
```

IMP_NETLIST

The IMP_NETLIST keyword directs PlatGen to write an implementation netlist file for the peripheral.

Format

```
OPTION IMP_NETLIST = TRUE
```

IP_GROUP

The IP_GROUP keyword defines the core group classification.

Format

```
OPTION IP_GROUP = USER
```

The following table lists IP_GROUP values:

Table 16-6: **IP_GROUP Values**

IP_GROUP	Definition
ALLIANCE	Third party IPs
INFRASTRUCTURE	All IPs in EDKInfrastructureLib
LOGICORE	All IPs in LogiCoreLib
REFERENCE	All IPs in XilinxReferenceDesigns
USER	User IPs (default)

IPLEVEL_DRC_PROC

The IPLEVEL_DRC_PROC keyword defines the Tcl entry point for the IP-level DRC routine. Do DRCs based only on IP-level settings.

Format

```
OPTION IPLEVEL_DRC_PROC = proc_name
```

IPTYPE

The IPTYPE keyword defines the type of the component.

Format

```
OPTION IPTYPE = PERIPHERAL
```

The following table lists IPTYPE values:

Table 16-7: **IPTYPE Values**

IPTYPE	Definition
BRIDGE	bridge component
BUS	bus component
BUS_ARBITER	combined bus and arbiter component
IP	component that is not address-mapped to a bus
PERIPHERAL	component that is address-mapped to a bus
PROCESSOR	processor component (MicroBlaze or PPC405)

LONG_DESC

Allows a long description of the core to be displayed by the GUI tools. The long description allows the GUI tools to display a hover help. No default.

Format

```
OPTION LONG_DESC = “OPB GPIO - IO only GPIO”
```

MAX_MASTERS

Define maximum number of masters allowed for cores marked as IPTYPE=BUS or IPTYPE=BUS_ARBITER. No default.

Format

```
OPTION MAX_MASTERS = 8
```

MAX_SLAVES

Define maximum number of slaves allowed for cores marked as IPTYPE=BUS or IPTYPE=BUS_ARBITER. No default.

Format

```
OPTION MAX_SLAVES = 8
```

NUM_WRITE_ENABLES

The number of write enables supported by the BRAM controller is specified by the NUM_WRITE_ENABLES keyword.

Format

```
OPTION NUM_WRITE_ENABLES = 8
```

For a byte-write 32-bit data memory, the NUM_WRITE_ENABLES = 4. For a byte-write 64-bit data memory, the NUM_WRITE_ENABLES = 8.

Used only by components of SPECIAL=BRAM_CNTLR.

SIM_MODELS

The simulation model availability is specified with the SIM_MODELS keyword.

Format

```
OPTION SIM_MODELS = BEHAVIORAL
```

If you have more than one model is available, then use the colon (:) to separate each model in the list. The first item in the list is the default setting.

Format

```
OPTION SIM_MODELS = BEHAVIORAL:STRUCTURAL:TIMING
```

SPECIAL

The SPECIAL keyword defines a class of components that require special handling.

Format

```
OPTION SPECIAL = BRAM_CNTLR
```

This keyword is reserved for internal use only.

STYLE

The STYLE keyword defines the design composition of the peripheral.

If you have only optimized hardware netlists, you must specify the BLACKBOX value within the MPD file. In this case, only the BBD file is read by the EDK tools.

If you have a mix of optimized hardware netlists and HDL files, you must specify the MIX value within the MPD file. In this case, the PAO and BBD files are read by the EDK tools. This indicates that VHDL with optimized hardware netlists or Verilog with optimized hardware netlists, but not both VHDL and Verilog along with optimized hardware netlists.

If you have only HDL files, you must specify the HDL value within the MPD file. In this case, only the PAO file is read by the EDK tools.

Format

```
OPTION STYLE = value
```

Where *value* is BLACKBOX, MIX, or HDL. The default value is HDL.

SYSLEVEL_DRC_PROC

The SYSLEVEL_DRC_PROC keyword defines the Tcl entry point for the system-level DRC routine. Do DRCs based only on system-level settings.

Format

```
OPTION IPLEVEL_DRC_PROC = proc_name
```

TCL_FILE

The TCL_FILE keyword defines the Tcl file name.

Format

```
OPTION TCL_FILE = opb_gpio_v2_1_0.tcl
```

USAGE_LEVEL

The USAGE_LEVEL keyword defines usage level of a core.

Format

```
OPTION USAGE_LEVEL = ALL_USERS
```

The following table lists USAGE_LEVEL values:

Table 16-8: **USAGE_LEVEL Values**

USAGE_LEVEL	Definition
ADVANCED_USER	Core can not be configured by BSB
ALL_USERS	Core can be configured by BSB (default)

Parameter

A parameter defines a constant that is passed into the entity (VHDL) or module (Verilog) declaration.

Parameter Keywords

An parameter can have the following keywords:

Table 16-9: **Parameter Keywords**

Keyword	Values	Default	Definition
ADDRESS	BASE HIGH NONE	C_BASEADDR is ADDRESS=BASE C_HIGHADDR is ADDRESS=HIGH	Identifies a named parameters as a valid address parameter
BRIDGE_TO	*string*	No Default	Allow address to be visible through the bridge
BUS	*string*	No Default	Bus label
CACHEABLE	TRUE FALSE	FALSE	Identify cacheable address
DESC	*string*	No Default	Allow a short description of the parameter to be displayed by the GUI tools
DT	integer string std_logic std_logic_vector	No Default	Datatype. See datatype translation table in the DT description for details.
IO_IF	*string*	No Default	IO label
IO_IS	*string*	No Default	
IPLEVEL_UPDATE_PROC	*string*	No Default	Tcl entry point for the IP-level update routine
LONG_DESC	*string*	No Default	Allow a long description of the parameter to be displayed by the GUI tools
MIN_SIZE	2^n	0	Minimum size address window
SYSLEVEL_UPDATE_PROC	*string*	No Default	Tcl entry point for the system-level update routine

ADDRESS

The ADDRESS keyword identifies a named parameters as a valid address parameter.

Format

```
PARAMETER C_BASEADDR=0xFFFFFFFF, MIN_SIZE=0x2000, ADDRESS=BASE
```

The following table lists ADDRESS values:

Table 16-10: **ADDRESS Values**

ADDRESS	Definition
BASE	Identify base address (default for C_BASEADDR)
HIGH	Identify high address (default for C_HIGHADDR)
NONE	Disable identification of address parameter

BRIDGE_TO

The BRIDGE_TO keyword Allows address to be visible through the bridge.

Format

```
PARAMETER C_BASEADDR=0xFFFFFFFF, BRIDGE_TO=SOPB
```

BUS

The bus interface of an parameter is specified by the BUS keyword.

Format

```
PARAMETER C_OPB_AWIDTH = 32, DT=datatype, BUS=bus_label
```

Where *bus_label* is a string.

If you have more than bus interface is sharing the parameter, then use the colon (:) to separate each bus interface in the list. The first item in the list is the default setting.

CACHEABLE

The CACHECABLE keyword identifies a cacheable address.

Format

```
PARAMETER C_BASEADDR=0xFFFFFFFF, CACHEABLE=TRUE
```

DESC

Allows a short description of the parameter to be displayed by the GUI tools. The short description replaces the parameter name in display field.

Format

```
PARAMETER C_HAS_EXTERNAL_XIN=0, DT=integer, DESC="HAS XIN"
```

DT

The datatype of a parameter is specified by the DT keyword.

Format

```
PARAMETER C_OPB_AWIDTH = 32, DT=datatype, BUS=bus_label
```

Where *datatype* can have the values in the following table:. The VHDL type and Verilog type columns describe how the DT value will be translated in the appropriate language.

Table 16-11: **DT Values**

DT Value	VHDL type	Verilog type
integer	integer	integer
string	string	string
std_logic	std_logic	bit
std_logic_vector	std_logic_vector	bit vector

IO_IF

IO interface association name.

Format

```
PARAMETER C_HAS_EXTERNAL_RCLK=0, IO_IF=uart_0, IO_IS=has_ext_rclk
```

IO_IS

A unique identifier name.

Format

```
PARAMETER C_FAMILY=virtex, IO_IF=uart_0, IO_IS=C_FAMILY
```

IPLEVEL_UPDATE_PROC

The IPLEVEL_UPDATE_PROC keyword defines the Tcl entry point for the IP-level update routine. Do update based on only IP-level settings.

Format

```
PARAMETER C_OPB_AWIDTH = 32, IPLEVEL_DRC_PROC = proc_name
```

LONG_DESC

Allows a long description of the parameter to be displayed by the GUI tools. The long description allows the GUI tools to display a hover help. No default.

Format

```
PARAMETER C_HAS_EXTERNAL_XIN=0, DT=integer, LONG_DESC=”XIN? What XIN?”
```

MIN_SIZE

The minimum size address window of an address is specified by the MIN_SIZE keyword.

Format

```
PARAMETER C_BASEADDR = 0xFFFFFFFF, DT=std_logic_vector, MIN_SIZE=0x100
```

SYSLEVEL_UPDATE_PROC

The SYSLEVEL_UPDATE_PROC keyword defines the Tcl entry point for the stem-level update routine. Do update based on only system-level settings.

Format

```
PARAMETER C_OPB_AWIDTH = 32, SYSLEVEL_DRC_PROC = proc_name
```

Parameter Naming Conventions

An MPD parameter correlates to a generic for VHDL or parameter for Verilog. The parameter name must be HDL (VHDL, Verilog) compliant. VHDL and Verilog have certain naming rules and conventions that must be followed.

Port

A port defines a data flow path that is passed into the entity (VHDL) or module (Verilog) declaration.

Port Keywords

A port can have the following keywords:

Table 16-12: **Port Keywords**

Keyword	Values	Default	Definition
3STATE	TRUE FALSE	No Default	Tri-state expansion (deprecated)
BUS	*string*	No Default	Bus label
DESC	*string*	No Default	Allow a short description of the port to be displayed by the GUI tools
DIR	IN, INPUT, I OUT, OUTPUT, O INOUT, IO	O	Direction mode
EDGE	RISING FALLING	No Default	Interrupt edge sensitivity (deprecated)
ENABLE	MULTI SINGLE	SINGLE	3-state enable control
ENDIAN	BIG LITTLE	BIG	Endianess
INTERRUPT_PRIORITY	HIGH LOW MEDIUM	LOW	Defines the relative priority of interrupt signals
INITIALVAL	VCC GND	GND	Driver value on unconnected inputs
IOB_STATE	BUF INFER REG	INFER	Identifies ports that instantiate or infer IOB primitives

Table 16-12: **Port Keywords**

Keyword	Values	Default	Definition
IO_IF	*string*	No Default	IO label
IO_IS	*string*	No Default	
LEVEL	HIGH LOW	No Default	Interrupt level sensitivity (deprecated)
LONG_DESC	*string*	No Default	Allow a long description of the port to be displayed by the GUI tools
SENSITIVITY	EDGE_FALLING EDGE_RISING LEVEL_HIGH LEVEL_LOW	No Default	Interrupt sensitivity
SIGIS	CLK INTERRUPT RST	No Default	Signal classification
THREE_STATE	TRUE FALSE	No Default	Tri-state expansion
VEC	[A:B]	No Default	Vector dimension. Where A and B are positive integer expressions.

3STATE

The 3STATE keyword enables / disables tri-state expansion. Its use is deprecated. Please use the THREE_STATE keyword.

Format

```
PORT PAR = “”, DIR=INOUT, 3STATE=FALSE, IOB_STATE=BUF
```

For output ports, the default value is FALSE. For inout ports, the default value is TRUE.

Please see the “3-state (InOut) Signals” section about designing tri-state signals at the HDL level.

BUS

Bus interface association name.

Format

```
PORT OPB_seqAddr = OPB_seqAddr, DIR=IN, BUS=bus_label
```

Where *bus_label* is a string.

If you have more than bus interface is sharing the parameter, then use the colon (:) to separate each bus interface in the list. The first item in the list is the default setting.

Format

```
PORT OPB_seqAddr = OPB_seqAddr, DIR=IN, BUS=MSOPB:SOPB
```

DESC

Allows a short description of the port to be displayed by the GUI tools. The short description replaces the port name in display field.

Format

```
PORT OPB_Clk="", DIR=IN, SIGIS=CLK, BUS=SOPB, DESC="OPB clock"
```

DIR

The driver direction of a signal is specified by the DIR keyword.

Format

```
PORT mysignal = "", DIR=direction
```

Where *direction* is either INPUT, IN, I, OUTPUT, OUT, O, INOUT, or IO.

EDGE

The edge sensitivity of an interrupt signal is specified by the EDGE keyword. Its use is deprecated. Please use the SENSITIVITY keyword.

Format

```
PORT interrupt = "", DIR=O, EDGE=edge_value, SIGIS=INTERRUPT
```

Where *edge_value* is either RISING or FALLING.

ENABLE

Tri-state signals can have multi-bit enable control, or a single bit enable control on the bus. This is specified with the ENABLE keyword.

Format

```
PORT mysignal = "", DIR=IO, VEC=[0:31], ENABLE=enable_value
```

Where *enable_value* is either SINGLE or MULTI. If there is no specification, then SINGLE is the default value.

Please see the "Design Considerations" section about designing tri-state signals at the HDL level.

ENDIAN

The endianess of a signal is specified by the ENDIAN keyword.

Format

```
PORT mysignal = "", DIR=I, VEC=[A:B], ENDIAN=endian_value
```

Where *endian_value* is either BIG or LITTLE. If there is no specification, then BIG is the default value. Where A and B are positive integer expressions.

INTERRUPT_PRIORITY

The INTERRUPT_PRIORITY keyword defines the relative priority of interrupt signals.

Format

```
PORT Intr="", DIR=O, SENSITIVITY=EDGE_RISING, SIGIS=INTERRUPT, INTERRUPT_PRIORITY=LOW
```

The level is dependent on the speed of the interface that the IP controls. For example, a UART runs at default 19200 baud, which gives a byte-rate of around 2000 bytes/s. An ethernet 100 runs at 100 MHz, which gives a byte-rate of 12 000 000 bytes/s. Therefore, UART is LOW and ethernet is HIGH.

CANBus runs at 1 MHz and gives a byte-rate of 120 000 bytes/s which would be MEDIUM. It is also dependent if the IP has FIFO or not. It is a judgment that the designer has to make.

IOB_STATE

The IOB_STATE keyword identifies ports that instantiate or infer IOB primitives.

Format

```
PORT DDR_Addr = “”, DIR=OUT, VEC=[0:C_DDR_AWIDTH-1], IOB_STATE=REG
```

The values are BUF, INFER, or REG. The default is INFER.

When a port requires an IOB primitive (IOB_STATE=INFER), PlatGen instantiates an IOB buffer. When a port has an IOB buffer (IOB_STATE=BUF) or IOB register (IOB_STATE=REG), PlatGen does not instantiate an IOB primitive.

IO_IF

IO interface association name.

Format

```
PARAMETER C_HAS_EXTERNAL_RCLK=0, IO_IF=uart_0, IO_IS=has_ext_rclk
```

IO_IS

A unique identifier name.

Format

```
PARAMETER C_FAMILY=virtex, IO_IF=uart_0, IO_IS=C_FAMILY
```

INITIALVAL

The signal driver value on unconnected input signals is specified by the INITIALVAL keyword.

Format

```
PORT mysignal = “”, DIR=INPUT, INITIALVAL=init_value
```

Where the *init_value* is either VCC or GND. If there is no specification, then GND is the default value.

LEVEL

The level sensitivity of an interrupt signal is specified by the LEVEL keyword. Its use is deprecated. Please use the SENSITIVITY keyword.

Format

```
PORT interrupt = “”, DIR=O, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

LONG_DESC

Allows a long description of the port to be displayed by the GUI tools. The long description allows the GUI tools to display a hover help. No default.

Format

```
PORT OPB_Clk="", DIR=I, SIGIS=CLK, BUS=SOPB, LONG_DESC="Clock from OPB"
```

SENSITIVITY

The interrupt sensitivity of an interrupt signal is specified by the SENSITIVITY keyword. This supersedes the EDGE and LEVEL keywords.

Format

```
PORT interrupt = “”, DIR=O, SENSITIVITY=value, SIGIS=INTERRUPT
```

Where the *value* is either EDGE_FALLING, EDGE_RISING, LEVEL_HIGH or LEVEL_LOW.

SIGIS

The class of a signal is specified by the SIGIS keyword.

Format

```
PORT mysig = “”, DIR=O, SIGIS=value
```

Where the *value* is either CLK, INTERRUPT, or RST. The following table lists SIGIS usage:

Table 16-13: **SIGIS Usage**

SIGIS	Usage
CLK	• Display purposes - XPS lists all clock signals • Clock buffer insertion - PlatGen infers clock buffers
INTERRUPT	• Display purposes - XPS lists all interrupt signals • Interrupt vector generation - PlatGen encodes the priority interrupt vector
RST	• Display purposes - XPS lists all reset signals

THREE_STATE

The THREE_STATE keyword enables / disables tri-state expansion. This supersedes the 3STATE keyword.

Format

```
PORT PAR = “”, DIR=INOUT, THREE_STATE=FALSE, IOB_STATE=BUF
```

For output ports, the default value is FALSE. For inout ports, the default value is TRUE.

Please see the "3-state (InOut) Signals" section about designing tri-state signals at the HDL level.

VEC

The vector width of a signal is specified by the VEC keyword.

Format

```
PORT mysignal = "", DIR=INPUT, VEC=[A:B]
```

Where A and B are positive integer expressions.

Port Naming Conventions

This section provides naming conventions for bus interface signal names. These conventions are flexible to accommodate embedded processor systems that have more than one bus interface and more than one bus interface port per component.

The names must be HDL (VHDL or Verilog) compliant. As with any language, VHDL and Verilog have certain naming rules and conventions that you must follow.

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).

LMB - Clock and Reset

```
LMB_Clk
LMB_Rst
```

OPB - Clock and Reset

```
OPB_Clk
OPB_Rst
```

PLB - Clock and Reset

```
PLB_Clk
PLB_Rst
```

Slave DCR Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

DCR Slave Outputs

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

```
<Sln>_dcrDBus
<Sln>_dcrAck
```

Where <*Sln*> is a meaningful name or acronym for the slave output. An additional requirement on <*Sln*> is that it must not contain the string, "DCR" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

```
uart_dcrAck
intc_dcrAck
memcon_dcrAck
```

DCR Slave Inputs

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

```
<nDCR>_ABus
<nDCR>_Sl_DBus
<nDCR>_Read
<nDCR>_Write
```

Where <*nDCR*> is a meaningful name or acronym for the slave input. An additional requirement on <*nDCR*> is that the last three characters must contain the string, "DCR" (upper or lower case or mixed case).

```
DCR_Sl_DBus
bus1_DCR_Sl_DBus
```

Slave LMB Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

LMB Slave Outputs

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

```
<Sln>_DBus
<Sln>_Ready
```

Where <*Sln*> is a meaningful name or acronym for the slave output. An additional requirement on <*Sln*> is that it must not contain the string, "LMB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

```
d_Ready
i_Ready
```

LMB Slave Inputs

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

```
<nLMB>_ABus
<nLMB>_ReadStrobe
<nLMB>_AddrStrobe
<nLMB>_WriteStrobe
<nLMB>_WriteDBus
<nLMB>_BE
```

Where <*nLMB*> is a meaningful name or acronym for the slave input. An additional requirement on <*nLMB*> is that the last three characters must contain the string, "LMB" (upper or lower case or mixed case).

```
LMB_ABus
bus1_LMB_ABus
```

Master OPB Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

OPB Master Outputs

For interconnection to the OPB, all masters must provide the following outputs:

```
<Mn>_ABus
```

```
<Mn>_BE
<Mn>_busLock
<Mn>_DBus
<Mn>_request
<Mn>_RNW
<Mn>_select
<Mn>_seqAddr
```

Where <*Mn*> is a meaningful name or acronym for the master output. An additional requirement on <*Mn*> is that it must not contain the string, "OPB" (upper or lower case or mixed case), so that master outputs are not confused with bus outputs.

```
iM_request
bridge_request
o2ob_request
```

OPB Master Inputs

For interconnection to the OPB, all masters must provide the following inputs:

```
<nOPB>_DBus
<nOPB>_errAck
<nOPB>_MGrant
<nOPB>_retry
<nOPB>_timeout
<nOPB>_xferAck
```

Where <*nOPB*> is a meaningful name or acronym for the master input. An additional requirement on <*nOPB*> is that the last three characters must contain the string, "OPB" (upper or lower case or mixed case).

```
iOPB_DBus
OPB_DBus
bus1_OPB_DBus
```

Slave OPB Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

OPB Slave Outputs

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

```
<Sln>_DBus
<Sln>_errAck
<Sln>_retry
<Sln>_toutSup
<Sln>_xferAck
```

Where <*Sln*> is a meaningful name or acronym for the slave output. An additional requirement on <*Sln*> is that it must not contain the string, "OPB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

```
tmr_xferAck
uart_xferAck
intc_xferAck
```

OPB Slave Inputs

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

```
<nOPB>_ABus
<nOPB>_BE
<nOPB>_DBus
<nOPB>_RNW
<nOPB>_select
<nOPB>_seqAddr
```

Where <*nOPB*> is a meaningful name or acronym for the slave input. An additional requirement on <*nOPB*> is that the last three characters must contain the string, "OPB" (upper or lower case or mixed case).

```
OPB_DBus
iOPB_DBus
bus1_OPB_DBus
```

Master PLB Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

PLB Master Outputs

For interconnection to the PLB, all masters must provide the following outputs:

```
<Mn>_ABus
<Mn>_BE
<Mn>_RNW
<Mn>_abort
<Mn>_busLock
<Mn>_compress
<Mn>_guarded
<Mn>_lockErr
<Mn>_MSize
<Mn>_ordered
<Mn>_priority
<Mn>_rdBurst
<Mn>_request
<Mn>_size
<Mn>_type
<Mn>_wrBurst
<Mn>_wrDBus
```

Where <*Mn*> is a meaningful name or acronym for the master output. An additional requirement on <*Mn*> is that it must not contain the string, "PLB" (upper or lower case or mixed case), so that master outputs are not confused with bus outputs.

```
iM_request
bridge_request
o2ob_request
```

PLB Master Inputs

For interconnection to the PLB, all masters must provide the following inputs:

```
<nPLB>_MAddrAck
<nPLB>_MBusy
<nPLB>_MErr
<nPLB>_MRdBTerm
<nPLB>_MRdDAck
<nPLB>_MRdDBus
<nPLB>_MRdWdAddr
```

```
<nPLB>_MRearbitrate
<nPLB>_MWrBTerm
<nPLB>_MWrDAck
<nPLB>_MSSize
```

Where <*nPLB*> is a meaningful name or acronym for the master input. An additional requirement on <*nPLB*> is that the last three characters must contain the string, "PLB" (upper or lower case or mixed case).

```
iPLB_MBusy
PLB_MBusy
bus1_PLB_MBusy
```

Slave PLB Ports

Naming conventions should be followed for that part of the identifier following the last underscore in the name.

PLB Slave Outputs

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

```
<Sln>_addrAck
<Sln>_MErr
<Sln>_MBusy
<Sln>_rdBTerm
<Sln>_rdComp
<Sln>_rdDAck
<Sln>_rdDBus
<Sln>_rdWdAddr
<Sln>_rearbitrate
<Sln>_SSize
<Sln>_wait
<Sln>_wrBTerm
<Sln>_wrComp
<Sln>_wrDAck
```

Where <*Sln*> is a meaningful name or acronym for the slave output. An additional requirement on <*Sln*> is that it must not contain the string, "PLB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

```
tmr_addrAck
uart_addrAck
intc_addrAck
```

PLB Slave Inputs

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

```
<nPLB>_ABus
<nPLB>_BE
<nPLB>_PAValid
<nPLB>_RNW
<nPLB>_abort
<nPLB>_busLock
<nPLB>_compress
<nPLB>_guarded
<nPLB>_lockErr
<nPLB>_masterID
<nPLB>_MSize
<nPLB>_ordered
```

```
<nPLB>_pendPri
<nPLB>_pendReq
<nPLB>_reqPri
<nPLB>_size
<nPLB>_type
<nPLB>_rdPrim
<nPLB>_SAValid
<nPLB>_wrPrim
<nPLB>_wrBurst
<nPLB>_wrDBus
<nPLB>_rdBurst
```

Where <*nPLB*> is a meaningful name or acronym for the slave input. An additional requirement on <*nPLB*> is that the last three characters must contain the string, "PLB" (upper or lower case or mixed case).

```
PLB_size
iPLB_size
dPLB_size
```

Reserved Parameter Names

The EDK tools automatically expands and populates certain reserved parameters. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved parameter names:

Table 16-14: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_FAMILY	FPGA Device Family
C_INSTANCE	Instance name of component
C_KIND_OF_EDGE	Vector of edge sensitive (rising / falling) of interrupt signals
C_KIND_OF_LVL	Vector of level sensitive (high / low) of interrupt signals
C_KIND_OF_INTR	Vector of interrupt signal sensitivity (edge / level)
C_NUM_INTR_INPUTS	Number of interrupt signals
C_MASK	LMB Decode Mask (deprecated)
C_NUM_MASTERS	Number of OPB masters (deprecated)
C_NUM_SLAVES	Number of OPB slaves (deprecated)
C_DCR_AWIDTH	DCR Address width
C_DCR_DWIDTH	DCR Data width
C_DCR_NUM_SLAVES	Number of DCR slaves
C_LMB_AWIDTH	LMB Address width
C_LMB_DWIDTH	LMB Data width
C_LMB_MASK	LMB Decode Mask
C_LMB_NUM_SLAVES	Number of LMB slaves
C_OPB_AWIDTH	OPB Address width
C_OPB_DWIDTH	OPB Data width

Table 16-14: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_OPB_NUM_MASTERS	Number of OPB masters
C_OPB_NUM_SLAVES	Number of OPB slaves
C_PLB_AWIDTH	PLB Address width
C_PLB_DWIDTH	PLB Data width
C_PLB_MID_WIDTH	PLB master ID width
C_PLB_NUM_MASTERS	Number of PLB masters
C_PLB_NUM_SLAVES	Number of PLB slaves

Reserved Parameters

C_FAMILY

The C_FAMILY parameter defines the FPGA device family. This parameter is automatically populated by the EDK tools.

Format

PARAMETER C_FAMILY = *family*, DT=string

C_INSTANCE

The C_INSTANCE parameter defines the instance name of the component. This parameter is automatically populated by the EDK tools.

Format

PARAMETER C_INSTANCE = *instance_name*, DT=string

C_MASK

The C_MASK parameter defines the LMB decode mask. This parameter is automatically populated by the EDK tools. It's use is deprecated. Please use the C_LMB_MASK parameter.

Format

PARAMETER C_MASK = <*hex*>, DT=std_logic_vector(0 to 31)

Where <*hex*> is a hex value.

C_NUM_MASTERS

The C_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by the EDK tools. It's use is deprecated. Please use the C_NUM_OPB_MASTERS parameter.

Format

PARAMETER C_NUM_MASTERS = <*num*>, DT=integer

Where <*num*> is an integer value.

C_NUM_SLAVES

The C_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by the EDK tools. It's use is deprecated. Please use the C_NUM_OPB_SLAVES parameter.

Format

```
    PARAMETER C_NUM_SLAVES = <num>, DT=integer
```

Where <*num*> is an integer value.

C_DCR_AWIDTH

The C_DCR_AWIDTH parameter defines the DCR address width in bits. This parameter is automatically populated by the EDK tools.

Format

```
    PARAMETER C_DCR_AWIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_DCR_DWIDTH

The C_DCR_DWIDTH parameter defines the DCR data width in bits. This parameter is automatically populated by the EDK tools.

Format

```
    PARAMETER C_DCR_DWIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_DCR_NUM_SLAVES

The C_DCR_NUM_SLAVES parameter defines the number of DCR slaves on the bus. This parameter is automatically populated by the EDK tools.

Format

```
    PARAMETER C_DCR_NUM_SLAVES = <num>, DT=integer
```

Where <*num*> is an integer value.

C_LMB_AWIDTH

The C_LMB_AWIDTH parameter defines the LMB address width in bits. This parameter is automatically populated by the EDK tools.

Format

```
    PARAMETER C_LMB_AWIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_LMB_DWIDTH

The C_LMB_DWIDTH parameter defines the LMB data width in bits. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_LMB_DWIDTH = <num>, DT=integer
```

Where *<num>* is an integer value.

C_LMB_MASK

The C_LMB_MASK parameter defines the LMB decode mask. This parameter is automatically populated by the EDK tools.

The address mask indicates which bits are used in the LMB decode to decode that a valid address is present on the LMB. Any bits that are set to 1 in the mask indicate that the address bit in that position is used to decode a valid LMD access. All other address bits are considered don't cares for the purpose of decoding LMB accesses. The EDK tools may limit the users choice for the address mask: the most restrictive case is that only a single bit may be set in the mask.

Format

```
PARAMETER C_LMB_MASK = <hex>, DT=std_logic_vector(0 to 31)
```

Where *<hex>* is a hex value.

C_LMB_NUM_SLAVES

The C_LMB_NUM_SLAVES parameter defines the number of LMB slaves on the bus. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_LMB_NUM_SLAVES = <num>, DT=integer
```

Where *<num>* is an integer value.

C_OPB_AWIDTH

The C_OPB_AWIDTH parameter defines the OPB address width in bits. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_OPB_AWIDTH = <num>, DT=integer
```

Where *<num>* is an integer value.

C_OPB_DWIDTH

The C_OPB_DWIDTH parameter defines the OPB data width in bits. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_OPB_DWIDTH = <num>, DT=integer
```

Where *<num>* is an integer value.

C_OPB_NUM_MASTERS

The C_OPB_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_OPB_NUM_MASTERS = <num>, DT=integer
```

Where <*num*> is an integer value.

C_OPB_NUM_SLAVES

The C_OPB_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_OPB_NUM_SLAVES = <num>, DT=integer
```

Where <*num*> is an integer value.

C_PLB_AWIDTH

The C_PLB_AWIDTH parameter defines the PLB address width in bits. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_PLB_AWIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_PLB_DWIDTH

The C_PLB_DWIDTH parameter defines the PLB data width in bits. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_PLB_DWIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_PLB_MID_WIDTH

The C_PLB_MID_WIDTH parameter defines the PLB master ID width in bits. This is determined by the number of PLB masters as shown in the following table:

Table 16-15: **C_PLB_MID_WIDTH Calculation**

C_PLB_NUM_MASTERS (Number of PLB Masters)	C_PLB_MID_WIDTH
0 to 2	1
3 to 4	2
5 to 8	3
9 to 16	4

This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_PLB_MID_WIDTH = <num>, DT=integer
```

Where <*num*> is an integer value.

C_PLB_NUM_MASTERS

The C_PLB_NUM_MASTERS parameter defines the number of PLB masters on the bus. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_PLB_NUM_MASTERS = <num>, DT=integer
```

Where <*num*> is an integer value.

C_PLB_NUM_SLAVES

The C_PLB_NUM_SLAVES parameter defines the number of PLB slaves on the bus. This parameter is automatically populated by the EDK tools.

Format

```
PARAMETER C_PLB_NUM_SLAVES = <num>, DT=integer
```

Where <*num*> is an integer value.

Reserved Port Connections

Connectivity of the DCR, LMB, OPB and PLB busses to peripherals is done through a common set of signal connections.

Clock and Reset Ports

For interconnection to the clock and reset ports:

LMB - Clock and Reset

```
PORT LMB_Clk = “”, DIR=I, SIGIS=CLK
PORT LMB_Rst = LMB_Rst, DIR=I
```

OPB - Clock and Reset

```
PORT OPB_Clk = “”, DIR=I, SIGIS=CLK
PORT OPB_Rst = OPB_Rst, DIR=I
```

PLB - Clock and Reset

```
PORT PLB_Clk = “”, DIR=I, SIGIS=CLK
PORT PLB_Rst = PLB_Rst, DIR=I
```

Notice that the clock port has no default value. The clock port is an input to the bus and is assigned by the user in the MHS. Therefore, all peripherals on the bus must also be treated as a user input port. If a default value were given to LMB_Clk, OPB_Clk, or PLB_Clk, this would not match the user defined clock in the MHS, and the EDK tools would consider this a short in the system, and tie-off the sourceless ports.

The reset port which is an output from the bus, and has a default value. All peripherals on the bus share the same default LMB_Rst, OPB_Rst, or PLB_Rst. The user input to the bus is SYS_Rst which has no default value.

Slave DCR Ports

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

```
PORT <Sln>_dcrDBus = Sl_dcrDBus, DIR=O, VEC=[0:C_DCR_DWIDTH-1],
BUS=SDCR
PORT <Sln>_dcrAck = Sl_dcrAck, DIR=O, BUS=SDCR
PORT <nDCR>_ABus = DCR_ABus, DIR=I, VEC=[0:C_DCR_AWIDTH-1], BUS=SDCR
PORT <nDCR>_Sl_DBus = DCR_Sl_DBus, DIR=I, VEC=[0:C_DCR_DWIDTH-1],
BUS=SDCR
PORT <nDCR>_Read = DCR_Read, DIR=I, BUS=SDCR
PORT <nDCR>_Write = DCR_Write, DIR=I, BUS=SDCR
```

Slave LMB Ports

For interconnection to the LMB, all slaves must provide the following connections:

```
PORT <Sln>_DBus = Sl_DBus, DIR=O, VEC=[0:C_LMB_DWIDTH-1], BUS=SLMB
PORT <Sln>_Ready = Sl_Ready, DIR=O, BUS=SLMB
PORT <nLMB>_ABus = LMB_ABus, DIR=I, VEC=[0:C_LMB_AWIDTH-1], BUS=SLMB
PORT <nLMB>_ReadStrobe = LMB_ReadStrobe, DIR=I, BUS=SLMB
PORT <nLMB>_AddrStrobe = LMB_AddrStrobe, DIR=I, BUS=SLMB
PORT <nLMB>_WriteStrobe = LMB_WriteStrobe, DIR=I, BUS=SLMB
PORT <nLMB>_WriteDBus = LMB_WriteDBus, DIR=I, VEC=[0:C_LMB_DWIDTH-1],
BUS=SLMB
PORT <nLMB>_BE = LMB_BE, DIR=I, VEC=[0:C_LMB_DWIDTH/8-1], BUS=SLMB
```

Master OPB Ports

For interconnection to the OPB, all masters must provide the following connections:

```
PORT <Mn>_ABus = M_ABus, DIR=O, VEC=[0:C_OPB_AWIDTH-1], BUS=MOPB
PORT <Mn>_BE = M_BE, DIR=O, VEC=[0:C_OPB_DWIDTH/8-1], BUS=MOPB
PORT <Mn>_busLock = M_busLock, DIR=O, BUS=MOPB
PORT <Mn>_DBus = M_DBus, DIR=O, VEC=[0:C_OPB_DWIDTH-1], BUS=MOPB
PORT <Mn>_request = M_request, DIR=O, BUS=MOPB
PORT <Mn>_RNW = M_RNW, DIR=O, BUS=MOPB
PORT <Mn>_select = M_select, DIR=O, BUS=MOPB
PORT <Mn>_seqAddr = M_seqAddr, DIR=O, BUS=MOPB
PORT <nOPB>_DBus = OPB_DBus, DIR=I, VEC=[0:C_OPB_DWIDTH-1], BUS=MOPB
PORT <nOPB>_errAck = OPB_errAck, DIR=I, BUS=MOPB
PORT <nOPB>_MGrant = OPB_MGrant, DIR=I, BUS=MOPB
PORT <nOPB>_retry = OPB_retry, DIR=I, BUS=MOPB
PORT <nOPB>_timeout = OPB_timeout, DIR=I, BUS=MOPB
PORT <nOPB>_xferAck = OPB_xferAck, DIR=I, BUS=MOPB
```

Slave OPB Ports

For interconnection to the OPB, all slaves must provide the following connections:

```
PORT <Sln>_DBus = Sl_DBus, DIR=O, VEC=[0:C_OPB_DWIDTH-1], BUS=SOPB
PORT <Sln>_errAck = Sl_errAck, DIR=O, BUS=SOPB
PORT <Sln>_retry = Sl_retry, DIR=O, BUS=SOPB
PORT <Sln>_toutSup = Sl_toutSup, DIR=O, BUS=SOPB
PORT <Sln>_xferAck = Sl_xferAck, DIR=O
PORT <nOPB>_ABus = OPB_ABus, DIR=I, VEC=[0:C_OPB_AWIDTH-1], BUS=SOPB
PORT <nOPB>_BE = OPB_BE, DIR=I, VEC=[0:C_OPB_DWIDTH/8-1], BUS=SOPB
PORT <nOPB>_DBus = OPB_DBus, DIR=I, VEC=[0:C_OPB_DWIDTH-1], BUS=SOPB
PORT <nOPB>_RNW = OPB_RNW, DIR=I, BUS=SOPB
PORT <nOPB>_select = OPB_select, DIR=I, BUS=SOPB
PORT <nOPB>_seqAddr = OPB_seqAddr, DIR=I, BUS=SOPB
```

Master PLB Ports

For interconnection to the PLB, all masters must provide the following connections:

```
PORT <Mn>_ABus = M_ABus, DIR=O, VEC=[0:C_PLB_AWIDTH-1], BUS=MPLB
PORT <Mn>_BE = M_BE, DIR=O, VEC=[0:C_PLB_DWIDTH/8-1], BUS=MPLB
PORT <Mn>_RNW = M_RNW, DIR=O, BUS=MPLB
PORT <Mn>_abort = M_abort, DIR=O, BUS=MPLB
PORT <Mn>_busLock = M_busLock, DIR=O, BUS=MPLB
PORT <Mn>_compress = M_compress, DIR=O, BUS=MPLB
PORT <Mn>_guarded = M_guarded, DIR=O, BUS=MPLB
PORT <Mn>_lockErr = M_lockErr, DIR=O, BUS=MPLB
PORT <Mn>_MSize = M_MSize, DIR=O, VEC=[0:1], BUS=MPLB
PORT <Mn>_ordered = M_ordered, DIR=O, BUS=MPLB
PORT <Mn>_priority = M_priority, DIR=O, VEC=[0:1], BUS=MPLB
PORT <Mn>_rdBurst = M_rdBurst, DIR=O, BUS=MPLB
PORT <Mn>_request = M_request, DIR=O, BUS=MPLB
PORT <Mn>_size = M_size, DIR=O, VEC=[0:3], BUS=MPLB
PORT <Mn>_type = M_type, DIR=O, VEC=[0:2], BUS=MPLB
PORT <Mn>_wrBurst = M_wrBurst, DIR=O, BUS=MPLB
PORT <Mn>_wrDBus = M_wrDBus, DIR=O, VEC=[0:C_PLB_DWIDTH-1], BUS=MPLB
PORT <nPLB>_MAddrAck = PLB_MAddrAck, DIR=I, BUS=MPLB
PORT <nPLB>_MBusy = PLB_MBusy, DIR=I, BUS=MPLB
PORT <nPLB>_MErr = PLB_MErr, DIR=I, BUS=MPLB
PORT <nPLB>_MRdBTerm = PLB_MRdBTerm, DIR=I, BUS=MPLB
PORT <nPLB>_MRdDAck = PLB_MRdDAck, DIR=I, BUS=MPLB
PORT <nPLB>_MRdDBus = PLB_MRdDBus, DIR=I, VEC=[0:C_PLB_DWIDTH-1],
BUS=MPLB
PORT <nPLB>_MRdWdAddr = PLB_MRdWdAddr, DIR=I, VEC=[0:3], BUS=MPLB
PORT <nPLB>_MRearbitrate = PLB_MRearbitrate, DIR=I, BUS=MPLB
PORT <nPLB>_MWrBTerm = PLB_MWrBTerm, DIR=I, BUS=MPLB
PORT <nPLB>_MWrDAck = PLB_MWrDAck, DIR=I, BUS=MPLB
PORT <nPLB>_MSSize = PLB_MSSize, DIR=I, VEC=[0:1], BUS=MPLB
```

Slave PLB Ports

For interconnection to the PLB, all slaves must provide the following connections:

```
PORT <Sln>_addrAck = Sl_addrAck, DIR=O, BUS=SPLB
PORT <Sln>_MErr = Sl_MErr, DIR=O, VEC=[0:C_NUM_MASTERS-1], BUS=SPLB
PORT <Sln>_MBusy = Sl_MBusy, DIR=O, VEC=[0:C_NUM_MASTERS-1], BUS=SPLB
PORT <Sln>_rdBTerm = Sl_rdBTerm, DIR=O, BUS=SPLB
PORT <Sln>_rdComp = Sl_rdComp, DIR=O, BUS=SPLB
PORT <Sln>_rdDAck = Sl_rdDAck, DIR=O, BUS=SPLB
PORT <Sln>_rdDBus = Sl_rdDBus, DIR=O, VEC=[0:C_PLB_DWIDTH-1],BUS=SPLB
PORT <Sln>_rdWdAddr = Sl_rdWdAddr, DIR=O, VEC=[0:3], BUS=SPLB
PORT <Sln>_rearbitrate = Sl_rearbitrate, DIR=O, BUS=SPLB
PORT <Sln>_SSize = Sl_SSize, DIR=O, VEC=[0:1], BUS=SPLB
PORT <Sln>_wait = Sl_wait, DIR=O, BUS=SPLB
PORT <Sln>_wrBTerm = Sl_wrBTerm, DIR=O, BUS=SPLB
PORT <Sln>_wrComp = Sl_wrComp, DIR=O, BUS=SPLB
PORT <Sln>_wrDAck = Sl_wrDAck, DIR=O, BUS=SPLB
PORT <nPLB>_ABus = PLB_ABus, DIR=I, VEC=[0:C_PLB_AWIDTH-1], BUS=SPLB
PORT <nPLB>_BE = PLB_BE, DIR=I, VEC=[0:(C_PLB_DWIDTH/8)-1], BUS=SPLB
PORT <nPLB>_PAValid = PLB_PAValid, DIR=I, BUS=SPLB
PORT <nPLB>_RNW = PLB_RNW, DIR=I, BUS=SPLB
PORT <nPLB>_abort = PLB_abort, DIR=I, BUS=SPLB
PORT <nPLB>_busLock = PLB_busLock, DIR=I, BUS=SPLB
PORT <nPLB>_compress = PLB_compress, DIR=I, BUS=SPLB
PORT <nPLB>_guarded = PLB_guarded, DIR=I, BUS=SPLB
PORT <nPLB>_lockErr = PLB_lockErr, DIR=I, BUS=SPLB
PORT <nPLB>_masterID = PLB_masterID, DIR=I,VEC=[0:C_PLB_MID_WIDTH-1],
BUS=SPLB
PORT <nPLB>_MSize = PLB_MSize, DIR=I, VEC=[0:1], BUS=SPLB
PORT <nPLB>_ordered = PLB_ordered, DIR=I, BUS=SPLB
PORT <nPLB>_pendPri = PLB_pendPri, DIR=I, VEC=[0:1], BUS=SPLB
PORT <nPLB>_pendReq = PLB_pendReq, DIR=I, BUS=SPLB
PORT <nPLB>_reqPri = PLB_reqPri, DIR=I, VEC=[0:1], BUS=SPLB
PORT <nPLB>_size = PLB_size, DIR=I, VEC=[0:3], BUS=SPLB
PORT <nPLB>_type = PLB_type, DIR=I, VEC=[0:2], BUS=SPLB
PORT <nPLB>_rdPrim = PLB_rdPrim, DIR=I, BUS=SPLB
PORT <nPLB>_SAValid = PLB_SAValid, DIR=I, BUS=SPLB
PORT <nPLB>_wrPrim = PLB_wrPrim, DIR=I, BUS=SPLB
PORT <nPLB>_wrBurst = PLB_wrBurst, DIR=I, BUS=SPLB
PORT <nPLB>_wrDBus = PLB_wrDBus, DIR=I, VEC=[0:C_PLB_DWIDTH-1],BUS=SPLB
PORT <nPLB>_rdBurst = PLB_rdBurst, DIR=I, BUS=SPLB
```

Design Considerations

This section includes design considerations.

Unconnected Ports

Unconnected output ports are assigned open, and unconnected input ports are either set to GND or VCC.

An unconnected port is identified as an empty double-quote (““) string.

The EDK tools resolves the driver value on unconnected input ports by the INITIALVAL keyword.

Format

```
PORT mysignal = “”, DIR=OUTPUT
```

Scalable Data path

Using an MPD keyword declaration, you can automatically scale data path width. Bus expressions are evaluated as arithmetic equations.

Format

```
PORT name = default_connection, VEC=[A:B]
```

Where A and B are positive integer expressions.

MPD Example

The following is an example MPD file:

```
BEGIN my_peripheral
# Generics for vhdl or parameters for verilog
```

```
PARAMETER C_BASEADDR = 0xB00000, DT=std_logic_vector(0 to 31)
PARAMETER C_MY_PERIPH_AWIDTH = 17, DT=integer
# Global ports
PORT OPB_Clk = “”, DIR=I
PORT OPB_Rst = “”, DIR=I
# My peripheral signals
PORT MY_ADDR = “”, DIR=O, VEC=[0:C_MY_PERIPH_AWIDTH-1]
# OPB signals
.
.
END
```

By default, if the vectors are larger than one bit, EDK tools determine the range specification on buses as either big-endian or little-endian. However, if the vector is one-bit width, then the range cannot be determined, and the EDK tools default to big-endian style notation.

To change this default behavior, use the ENDIAN keyword.

Format

```
PORT mysignal = “”, DIR=I, VEC=[0:0], ENDIAN=LITTLE
```

This builds the VHDL equivalent:

```
mysignal: in std_logic_vector(0 downto 0);
```

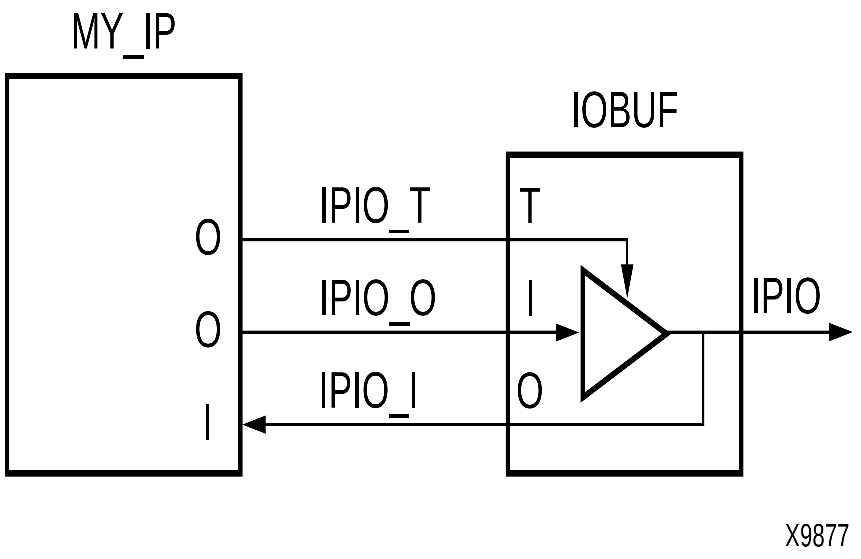
Interrupt Signals

Interrupt signals are identified by the SIGIS=INTERRUPT name-value keyword.

3-state (InOut) Signals

At the MHS / MPD level, there is a listing for an inout port in the MPD file that allows you to map to it in the MHS file. In the MPD file, a 3-state signal is identified by the inout direction mode, and the port name must be ioname.

Figure 16-1: **IOBUF Implementation**



The EDK tools expands the inout port in the MPD file to three ports in the port declaration section of the HDL file, and writes out the RTL code to infer the IOBUF. This port expansion occurs because if the top-level is synthesized without IO insertion, the 3-states on the inout ports are inferred as BUFTs at the CLB level. However, they should be inferred as IOBUFs at the IOB level. The EDK tools infers the 3-states at the top-level to ensure that the inout ports are always associated to the IOBUF.

Inout ports are currently defined at the top-level since the only internal signals are those defined as an input or an output. There are no inout signals defined internally that need a BUFT.

It is important to note that the 3-state enables are all active-low to allow a direct connection to the OBUFT of the IOBUF.

VHDL 3-state (InOut) With Multi-Bit Enable Example

The following is a VHDL example that includes 3-state signal with a multi-bit enable:

```
entity tri_state_multi is
generic (C_WIDTH: integer:= 9);
port (
     -- tri-state signal
   tristate_I: in std_logic_vector(0 to C_WIDTH-1);
   tristate_O: out std_logic_vector(0 to C_WIDTH-1);
   tristate_T: out std_logic_vector(0 to C_WIDTH-1));
end entity tri_state_multi;
```

MPD 3-state (InOut) With Multi-Bit Enable Example

The following is a MPD example that includes 3-state signal with a multi-bit enable:

```
BEGIN tri_state_multi
OPTION IPTYPE=IP
PARAMETER C_WIDTH = 9, DT=integer
PORT tristate = “”, DIR=INOUT, VEC=[0:C_WIDTH-1], ENABLE=MULTI, THREE_STATE=TRUE
END
```

VHDL 3-state (InOut) With Single-Bit Enable Example

The following is a VHDL example that includes 3-state signal with a single-bit enable:

```
entity tri_state_single is
generic (C_WIDTH: integer:= 9);
port (
     -- tri-state signal
   tristate_I: in std_logic_vector(0 to C_WIDTH-1);
   tristate_O: out std_logic_vector(0 to C_WIDTH-1);
   tristate_T: out std_logic);
end entity tri_state_single;
```

MPD 3-state (InOut) With Single-Bit Enable Example

The following is a MPD example that includes 3-state signal with a single-bit enable:

```
BEGIN tri_state_single
OPTION IPTYPE=IP
PARAMETER C_WIDTH = 9, DT=integer
PORT tristate = “”, DIR=INOUT, VEC=[0:C_WIDTH-1], ENABLE=SINGLE, THREE_STATE=TRUE
END
```

Chapter 17

Peripheral Analyze Order (PAO)

Overview

A PAO (Peripheral Analyze Order) file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation.

The STYLE option in the MPD with the values of MIXED or HDL identify the core as having a PAO file.

This chapter includes the following sections:

"PAO Format"

"PAO Example"

PAO Format

Use the following format:

```
lib library hdl_file_basename
```

Library specifies the unique library for the peripheral, and HDL file names are specified without a file extension. All names are in lower-case.

If your peripheral requires a certain version of a library, then the library name is given with the version appended. For example, if you request version 1.00.a, then the library name is:

```
library_name_v1_00_a
```

Comments

You can insert comments without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments can continue to the end of the line
- Comments can be anywhere on the line

PAO Example

The following is an example PAO file:

```
lib common_v1_00_a common_types_pkg
lib common_v1_00_a pselect
lib opb_gpio_v1_00_a gpio_core
lib opb_gpio_v1_00_a opb_gpio
```

Black-Box Definition (BBD)

Overview

The Black Box Definition (BBD) file manages the file locations of optimized hardware netlists for the black-box sections of your peripheral design.

The STYLE option in the MPD with the values of MIXED or BLACKBOX identify the core as having a BBD file.

This chapter includes the following sections:

“BBD Format”

“BBD Examples”

BBD Format

The BBD format is a look-up table chart that lists netlist files. The first line is the header of the look-up table. There can be as many entries as necessary in the header to make a selection. Header entries are tailored by MPD parameters. The last column of the table must be the FILES column.

The netlist and simmodels directories in the IP directory can have their own underlying directory structure because the BBD file manages the relative file locations. However, the directories must mirror each other.

Each file is listed with the file extension of the hardware implementation netlist. Since implementation netlists have multiple file extensions (such as, .edn, .edf, .edo, .ngo), it is important to identify the format. For simulation, the Platform Generator uses the file extension .vhd for VHDL simulation and .v for Verilog.

The black-box simulation netlists for HDL simulation must be moved to the simmodels directory, and the black-box hardware netlists for implementation must be moved to the netlist directory. The simmodels and netlist directories can have their own underlying directory structure, however, they must mirror each other.

Comments

You can insert comments without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments can continue to the end of the line
- Comments can be anywhere on the line

Lists

If you have multiple hardware implementation netlists, then use a comma (,) to separate each individual netlist in the list.

BBD Examples

File Selection Without Options

The following is an example of a file selection without options. The NGC netlist is copied into the your implementation directory regardless of specific options set on the core.

```
FILES
blackbox.ngc
```

Multiple File Selections Without Options

The following is an example of multiple file selections without options. The set of NGC netlists are copied into the your implementation directory regardless of specific options set on the core.

```
FILES
blackbox1.ngc, blackbox2.ngc, blackbox3.edn
```

File Selection With Options

The following is an example of a file selection with options. The specific EDIF netlist is copied into the your implementation directory dependent on the C_FAMILY and C_BUS_CONFIG parameters set on the core.

```
C_FAMILY C_BUS_CONFIG FILES
virtex          1       virtex/ip1.edf
virtex          2       virtex/ip2.edf
spartan2        1       virtex/ip1.edf
spartan2        2       virtex/ip2.edf
virtexe         1       virtex/ip1.edf
virtexe         2       virtex/ip2.edf
spartan2e       1       virtex/ip1.edf
spartan2e       2       virtex/ip2.edf
virtex2         1       virtex2/ip1.edf
virtex2         2       virtex2/ip2.edf
virtex2p        1       virtex2/ip1.edf
virtex2p        2       virtex2/ip2.edf
```

Microprocessor Software Specification (MSS)

Summary

This chapter describes the Microprocessor Software Specification (MSS) format.

Overview

An MSS file is supplied by the user as an input to the Library Generator (Libgen). The MSS file contains directives for customizing libraries, drivers and file systems.

Note: RevUp tool provides a way to convert old MSS format to the new one used in this version of the EDK tools. Please see Chapter 9, “Format Revision Tool” for more information.

MSS Format

An MSS file is supplied by the user as an input to the Library Generator (Libgen). An MSS file is case insensitive. However, any reference to a file name or instance name in the MSS file is case sensitive.

Comments can be specified anywhere in the file. A ’**#**’ character denotes the beginning of a comment and all characters after the ’#’ till the end of the line are ignored. All white spaces are also ignored and carriage returns act as sentence delimiters.

Keywords

The keywords that are used in an MSS file are as follows:

Begin

The **begin** keyword begins a driver, processor, or file system definition block. The begin keyword should be followed by **driver, processor** or **filesys** keywords.

End

The **end** keyword signifies the end of a definition block.

Parameter

The MSS file has a simple ***name* = *value*** format for most statements. The **parameter** keyword is required before every such NAME, VALUE pairs. The format for assigning a value to a parameter is **parameter *name* = *value***. If the parameter is within a **begin-end** block, it is a local assignment, otherwise it is a global (system level) assignment.

Requirements

The MSS file has a dependency on the MHS file. This dependency has to be specified as a command line option to libgen using the -mhs option. Please refer Chapter 7, "Library Generator" for more information. Hence there is a dependency on hardware for the software flow. Please refer the Microprocessor Hardware Specification documentation for more information on hardware configuration.

NOTE :

Prior to EDK6.1 release this dependency was specified in the MSS file as **parameter HW_SPEC_FILE =** *file_name***.mhs**. This parameter will be deprecated for EDK6.1 release and eventually be removed for future releases.

The syntax of various files that the Embedded Development Tools use are described by the Platform Specification Format (PSF). Please refer Chapter 14, "Platform Specification Format (PSF)" for more information. The current PSF version is 2.1.0. The MSS file should also contain version information in the form of **parameter Version = 2.1.0** which represents the PSF version 2.1.0.

MSS Example

An example MSS file is given below:

```
parameter VERSION = 2.1.0

BEGIN PROCESSOR
parameter HW_INSTANCE = my_microblaze
parameter DRIVER_NAME = cpu
parameter DRIVER_VER = 1.00.a
parameter DEBUG_PERIPHERAL = my_jtag
parameter STDIN = my_uartlite_1
parameter STDOUT = my_uartlite_1
END

BEGIN PROCESSOR
parameter HW_INSTANCE = my_ppc
parameter DRIVER_NAME = cpu_ppc405
parameter DRIVER_VER = 1.00.a
parameter STDIN = my_uartlite_2
parameter STDOUT = my_uartlite_2
parameter EXECUTABLE = code/hello_world.elf
END

BEGIN DRIVER
parameter HW_INSTANCE = my_intc
parameter DRIVER_NAME = intc
parameter DRIVER_VER = 1.00.a
END

BEGIN DRIVER
parameter HW_INSTANCE = my_uartlite_1
parameter DRIVER_VER = 1.00.a
parameter DRIVER_NAME = uartlite
parameter INT_HANDLER = uart_1_handler, INT_PORT = Interrupt
END

BEGIN DRIVER
parameter HW_INSTANCE = my_uartlite_2
parameter DRIVER_VER = 1.00.a
parameter DRIVER_NAME = uartlite
parameter LIBRARY = XilFile
parameter INT_HANDLER = uart_2_handler, INT_PORT = Interrupt
END

BEGIN DRIVER
parameter HW_INSTANCE = my_timebase_wdt
parameter DRIVER_VER = 1.00.a
parameter DRIVER_NAME = timebase_wdt
parameter INT_HANDLER=my_timebase_hndl, INT_PORT = Timebase_Interrupt
parameter INT_HANDLER=my_timebase_hndl, INT_PORT = WDT_Interrupt
END

BEGIN LIBRARY
parameter LIBRARY_NAME = XilMfs
parameter LIBRARY_VER = 1.00.a
parameter NUMBYTES = 100000
parameter BASE_ADDRESS = 0x80f00000
END

BEGIN DRIVER
parameter HW_INSTANCE = my_jtag
parameter DRIVER_NAME = uartlite
parameter DRIVER_VER = 1.00.a
parameter INT_HANDLER = jtag_uart_handler, INT_PORT = Interrupt
END
```

Global Parameters

These parameters are system specific parameters and do not relate to a particular driver, file system or library.

PSF Version

This option specifies the PSF version of the MSS file. This option is mandatory for versions 2.1.0 and above.

Format

```
parameter VERSION = 2.1.0
```

Parameter INT_HANDLER

This option defines the interrupt handler software routine for an external interrupt port given in the MHS file.

Format

```
    parameter INT_HANDLER = my_int_handl, INT_PORT = Interrupt
```

The external interrupt port that raises the interrupt is specified after the attribute as shown above with the INT_PORT keyword. This port should match the port name (and not the signal name) specified in the MHS file as a global external port.

Instance Specific Parameters

These parameters are driver, library or file system specific parameters. The parameters have to be between a **Begin** and **End** block.

Driver, Library and Processor Block Parameters

Table 19-1: **Parameters Specified in Driver, Library and Processor Blocks Only**

Option	Values	Default	Definition
HW_INSTANCE	Instance name	None	Instance name specified in the MHS file.
DRIVER_NAME	Driver name	None	Driver name.
DRIVER_VER	1.00.a	No Version	Driver version.
LIBRARY_NAM E	Library name	None	Library name.
LIBRARY_VER	1.00.a	No Version	Library version.
INT_HANDLER	C Function Name	None	Specifies the interrupt handler function for the peripheral interrupt.
LEVEL	Number	Specified in MDD file	An MDD file parameter that can be overwritten in the MSS. Please see Chapter 19, "Microprocessor Library / Driver Definition (MLD / MDD)" for more information.

Table 19-1 provides the parameters that can be used in driver, library and processor blocks.

HW_INSTANCE Option

This option is required for drivers associated with peripheral instances specified in the MHS file.

Format

```
    parameter HW_INSTANCE = instance_name
```

All drivers in the EDK require instances to be associated with the drivers. Even a processor definition block should refer to the processor instance. The instance name that is given must match the name specified in the MHS file.

DRIVER_NAME Option

This option is needed for peripherals that have drivers associated with them.

Format

```
    parameter DRIVER_NAME = uartlite
```

Library Generator copies the driver directory specified to ***OUTPUT_DIR/processor_instance_name*****/libsrc** directory and compiles the drivers using makefiles provided. Please see the Chapter 7, "Library Generator" for more information.

DRIVER_VER Option

The driver version is set using the DRIVER_VER option.

Format

```
parameter DRIVER_VER = 1.00.a
```

This version is specified in the following format: **x.yz.a**, where **x,y** and **z** are digits, and **a** is a character. This is translated to the driver directory searched by LibGen as follows:

USER_PROJECT / drivers / *DRIVER_NAME*_vx_yz_a

XILINX_EDK / drivers / *DRIVER_NAME*_vx_yz_a

XILINX_EDK / hw / coregen / ip / xilinx / drivers / *DRIVER_NAME*_vx_yz_a

The MDD (Microprocessor Driver Definition) files needed by Libgen for each driver should be named *DRIVER_NAME*.mdd and should be present in a subdirectory **data/** within the driver directory. Please refer Chapter 19, "Microprocessor Library / Driver Definition (MLD / MDD)" for more information.

INT_HANDLER Option

This option defines the interrupt handler software routine for an interrupt port of the peripheral.

Format

```
parameter INT_HANDLER = my_int_handl, INT_PORT = Interrupt
```

The interrupt port of the peripheral instance that raises the interrupt is specified after the attribute as shown above with the INT_PORT keyword. This port should match the port name (and not the signal name) specified in the MHS file for that peripheral instance.

LEVEL Option

The driver level is set using the LEVEL option. The levels of drivers available in the EDK are levels 0 and 1. Level 0 drivers are small low level drivers, and level 1 drivers provide more functionality than the level 0 drivers. Please refer Chapter 28, "Device Drivers" for more information. The default level is specified in the MDD file for the driver. Please refer Chapter 19, "Microprocessor Library / Driver Definition (MLD / MDD)" for more information.

Format

```
parameter LEVEL = 1
```

Level is either 0 or 1 for EDK drivers

LIBRARY_NAME Option

This option is needed for libraries.

Format

```
parameter LIBRARY_NAME = xilmfs
```

Library Generator copies the library directory specified to ***OUTPUT_DIR/processor_instance_name/*****libsrc** directory and compiles the libraries using makefiles provided. Please see Chapter 7, "Library Generator" for more information.

LIBRARY_VER Option

The library version is set using the LIBRARY_VER option.

Format

```
parameter LIBRARY_VER = 1.00.a
```

This version is specified in the following format: **`x.yz.a`**, where **`x,y`** and **`z`** are digits, and **`a`** is a character. This is translated to the library directory searched by LibGen as follows:

`USER_PROJECT` / sw_services / *`LIBRARY_NAME`*_vx_yz_a

`XILINX_EDK` / sw / edklib / sw_services / *`LIBRARY_NAME`*_vx_yz_a

The MLD (Microprocessor Library Definition) files needed by Libgen for each library should be named *LIBRARY_NAME*_v_2_1_0.mld and should be present in a subdirectory **data/** within the library directory. Please refer Chapter 19, "Microprocessor Library / Driver Definition (MLD / MDD)" for more information.

MDD/MLD Specific Parameters

Parameters specified in the MDD / MLD file can be overwritten in the MSS file as

Format

```
parameter PARAM_NAME = PARAM_VALUE
```

Please refer Chapter 19, "Microprocessor Library / Driver Definition (MLD / MDD)" for information.

Processor Specific Parameters

Table 19-2: **Parameters Specified in Processor Blocks Only**

Option	Values	Default	Definition
EXECUTABLE	*directory/file*	code/executable.elf	Defines the user's executable file name and location. **NOTE:** This parameter has been integrated into the XMP file from EDK6.1.This parameter is **deprecated** for EDK6.1 release and will eventually be removed for future releases.
DEFAULT_INIT	XMDSTUB, EXECUTABLE	EXECUTABLE	Specifies which file should be used to initialize that processor's memory.
DEBUG_PERIP HERAL	Instance name	None	Peripheral instance used for On-board Debug.
STDIN	Instance name	None	Specifies standard input peripheral instance.
STDOUT	Instance name	None	Specifies standard output peripheral instance.
COMPILER	Name of the compiler	**mb-gcc** for MicroBlaze, **powerpc-eabi-gcc** for PPC405	Name of the compiler used for compiling drivers and libraries
OS	Name of the OS	standalone	Name of the OS supported (for example., VxWorks5_4)
ARCHIVER	Name of the archiver	mb-ar for MicroBlaze, powerpc-eabi-ar for PPC405	Name of the archiver used for archiving drivers and libraries.
COMPILER_FL AGS	Command line flags	Libgen generates default	Need not be specified if using EDT compilers
EXTRA_COMPI LER_FLAGS	Command line flags	None	User definable compiler flags used to compile libraries and drivers

Table 19-2 provides all the parameters that can be specified only in a processor definition block.

EXECUTABLE Option (Deprecated for EDK6.1)

TThis parameter has been integrated into the XMP file from EDK6.1. For information on XMP file refer Chapter 2, "Xilinx Platform Studio".This parameter is deprecated for EDK6.1 release and will eventually be removed for future releases.

DEFAULT_INIT Option

This option specifies whether XMDSTUB, BOOTSTRAP or EXECUTABLE is the program to load into the memory of that particular processor instance.

Format

```
parameter DEFAULT_INIT = XMDSTUB
```

The DEFAULT_INIT option can take EXECUTABLE, XMDSTUB or BOOTSTRAP as values. By default, the value is EXECUTABLE. For the PowerPC, the executable option is the only useful option.

STDIN Option

Identify standard input device with the STDIN option.

Format

```
parameter STDIN = instance_name
```

STDOUT Option

Identify standard output device with the STDOUT option.

Format

```
parameter STDOUT = instance_name
```

DEBUG_PERIPHERAL Option

The peripheral that is used to handle the xmdstub should be specified in the DEBUG_PERIPHERAL option. This is useful for MicroBlaze only.

Format

```
parameter DEBUG_PERIPHERAL = instance_name
```

COMPILER Option

This option specifies the compiler used for compiling drivers and libraries. The compiler defaults to **mb-gcc** or **powerpc-eabi-gcc** depending on whether the drivers are part of the microblaze instance or powerpc instance. Any other compatible compiler can be specified as an option.

Format

```
parameter COMPILER = dcc
```

This denotes the Diab compiler as the compiler to be used for drivers and libraries.

ARCHIVER Option

This option specifies the archive utility to be used for archiving object files into libraries. The archiver defaults to **mb-ar** or **powerpc-eabi-ar** depending on whether the drivers are part of the microblaze instance or powerpc instance. Any other compatible archiver can be specified as an option.

Format

```
parameter ARCHIVER = ar
```

This denotes the archiver **ar** to be used for drivers and libraries.

COMPILER_FLAGS Option

This option specifies compiler flags to be used for compiling drivers and libraries. If the option is not specified, Libgen automatically uses platform and processor specific options. It is recommended that this option *not* be specified in the MSS if the standard compilers and archivers in the EDK are used. COMPILER_FLAGS option can be defined in the MSS if there is a need for custom compiler flags that override Libgen generated ones. The EXTRA_COMPILER_FLAGS option is recommended if compiler flags have to be appended to the ones Libgen already generates.

Format

```
parameter COMPILER_FLAGS = ""
```

EXTRA_COMPILER_FLAGS Option

This option can be used whenever custom compiler flags need to be used in addition to the automatically generated compiler flags.

Format

```
parameter EXTRA_COMPILER_FLAGS = -g
```

This specifies that the drivers and libraries must be compiled with debugging symbols in addition to the LibGen generated COMPILER_FLAGS.

OS Option

This option denotes whether an RTOS is present (for example., VxWorks5_4) or not. By default, LibGen assumes a value of **standalone** as the OS.

Format

```
parameter OS = VxWorks5_4
```

This specifies that the VxWorks5_4 adaptation layer must be generated for the drivers. This option, although supported, is not currently used in Libgen.

Example MSS snippet showing processor options

```
BEGIN PROCESSOR
parameter HW_INSTANCE = my_microblaze
parameter DRIVER_NAME = cpu
parameter DRIVER_VER = 1.00.a
parameter DEFAULT_INIT = xmdstub
parameter DEBUG_PERIPHERAL = my_jtag
parameter STDIN = my_uartlite_1
parameter STDOUT = my_uartlite_1
parameter COMPILER = mb-gcc
parameter ARCHIVER = mb-ar
parameter EXTRA_COMPILER_FLAGS = -g -O0
parameter OS = standalone
END
```

Chapter 20

Microprocessor Library Definition (MLD)

Summary

This chapter describes the Microprocessor Library Definition(MLD) format, Platform Specification Format 2.1.0.

Overview

An MLD file contains directives for customizing software libraries. This document describes the MLD format and the parameters that can be used to customize libraries. For all EDK libraries, the user does not need to peruse this document. Reading this document is recommended for user-written libraries that need to be configured by libgen tool.

Requirements

Each library has an MLD file and a Tcl(Tool Command Language) file associated with it. The MLD file is used by the Tcl file to customize the library depending on different options in the MSS file. For more information on the MSS file format, please see Chapter 19, "Microprocessor Software Specification (MSS)".

The library source files and the MLD file for each library must be located at specific directories in order for Libgen to find the files and the libraries. Please refer Chapter 7, "Library Generator" for a list of directories searched for libraries.

Library Definition Files

Library Definition involves defining a Data Definition file (MLD) and a Data Generation file (Tcl file).

- Data Definition file - The MLD file (named as <library_name>_v2_1_0.mld) contains the configurable parameters . A detailed description of the various parameters and the MLD format is described in section"MLD Parameter Description Section" in this chapter.
- Data Generation file - The second file (named as <library_name>_v2_1_0.tcl, with the filename being the same as the mld filename) uses the parameters configured in the MSS file for the library to generate data. Data generated includes but not limited to generation of header files, C files, running DRCs for the library and generating

executables. The Tcl file includes procedures that are called by Libgen tool at various stages of its execution. Various procedures in a Tcl file includes **DRC** (name of DRC given in the MLD file), **generate** (Libgen defined procedure) called after library files are copied, **post_generate** (Libgen defined procedure) called after **generate** has been called on all drivers and libraries, **execs_generate** (Libgen defined procedure) called after the libraries and drivers have been generated. For more information on the working of libgen tool refer to Chapter 7, "Library Generator".

Note that a library need not have the data generation file (Tcl file).

MLD Format Specification

MLD format specification involves the MLD file Format specification and the Tcl file Format specification. These are described below.

MLD File Format Specifcation

MLD file format specification involves description of parameters defined in the Parameter Description section.

Parameter Description Section

This data section describes configurable parameters in a library. The format used to describe this section is discussed in section "MLD Parameter Description Section"of this chapter.

Tcl File Format Specification

Each library has a Tcl file associated with the MLD file. This Tcl file has the following sections :

DRC Section

This section contains Tcl routines which validate the library parameters provided by the user for consistency.

Generation Section

This section contains Tcl routines which generate the configuration header and 'C' files based on the library parameters

Example

This section explains the MLD format through an example MLD file and its corresponding Tcl file.

MLD file example

An example MLD file for the xilmfs library is given below:

```
OPTION psf_version = 2.1.0 ;
```

OPTION is a keyword identified by the libgen tool. The option name following the OPTION keyword is a directive to the libgen tool to do a specific action. Here psf_version of the MLD file is defined to be 2.1. This is the only option that can occur before a BEGIN LIBRARY construct now.

```
BEGIN LIBRARY xilmfs
```

The BEGIN LIBRARY construct defines the start of a library named "xilmfs".

```
  OPTION DRC = mfs_drc ;
  OPTION COPYFILES = all;
```

COPYFILES option indicates the files to be copied for the library. DRC option specifies the name of the Tcl procedure that the tool invokes while processing this library. Here "mfs_drc" is the Tcl procedure in the xilmfs_v2_1_0.tcl file that would be invoked by libgen while processing the xilmfs library.

```
 PARAM NAME = numbytes, DESC = "Number of Bytes", TYPE = int, DEFAULT =
100000, DRC = drc_numbytes ;
PARAM NAME = base_address, DESC = "Base Address", TYPE = int, DEFAULT =
0x10000, DRC = drc_base_address ;
 PARAM NAME = init_type, DESC = "Init Type", TYPE = enum, VALUES = ("New
file system"=MFSINIT_NEW, "MFS Image"=MFSINIT_IMAGE, "ROM
Image"=MFSINIT_ROM_IMAGE), DEFAULT = MFSINIT_NEW ;
 PARAM NAME = need_utils, DESC = "Need additional Utilities?", TYPE =
bool, DEFAULT =  false ;
```

PARAM defines a library parameter that can be configured. Each PARAM has the following properties associated with it whose meaning is self-explanatory - NAME, DESC, TYPE, DEFAULT, RANGE, DRC. The property VALUES defines the list of possible values associated with an ENUM type.

```
  BEGIN INTERFACE file
   PROPERTY HEADER="xilmfs.h" ;
   FUNCTION NAME=open, VALUE=mfs_file_open ;
   FUNCTION NAME=close, VALUE=mfs_file_close ;
   FUNCTION NAME=read, VALUE=mfs_file_read ;
   FUNCTION NAME=write, VALUE=mfs_file_write ;
   FUNCTION NAME=lseek, VALUE=mfs_file_lseek ;
  END INTERFACE
```

An Interface contains a list of standard functions. A library defining an interface should have values for the list of standard functions. It must also specify a header file where all the function prototypes are defined.

PROPERTY defines the properties associated with the construct defined in the BEGIN construct. Here "HEADER" is a property with value "xilmfs.h", defined by the "file" interface. FUNCTION defines a function supported by the interface. Here "open", "close", "read", "write", "lseek" are functions of "file" interface with values "mfs_file_open", "mfs_file_close", "mfs_file_read", "mfs_file_write", "mfs_file_lseek".These functions are defined in the header file "xilmfs.h".

```
  BEGIN INTERFACE filesystem
```

BEGIN INTERFACE defines an interface the library supports. Here "file" is the name of the interface.

```
  PROPERTY HEADER="xilmfs.h" ;
  FUNCTION NAME=cd, VALUE=mfs_change_dir ;
  FUNCTION NAME=opendir, VALUE=mfs_dir_open ;
  FUNCTION NAME=closedir, VALUE=mfs_dir_close ;
  FUNCTION NAME=readdir, VALUE=mfs_dir_read ;
  FUNCTION NAME=deletedir, VALUE=mfs_delete_dir ;
  FUNCTION NAME=pwd, VALUE=mfs_get_current_dir_name ;
  FUNCTION NAME=rename, VALUE=mfs_rename_file ;
  FUNCTION NAME=exists, VALUE=mfs_exists_file ;
  FUNCTION NAME=delete, VALUE=mfs_delete_file ;
 END INTERFACE

END LIBRARY
```

END is used with the construct name that was used in the BEGIN statement. Here END is used with INTERFACE and LIBRARY constructs to indicate the end of each of INTERFACE and LIBRARY constructs.

Example Tcl File

The following is the xilmfs_v2_1_0.tcl file corresponding the xilmfs_v2_1_0.mld file described in the previous section. The "mfs_drc" procedure would be invoked by libgen for xilmfs library while running DRCs for libraries. The **generate** routine generates constants in a header file and a c file for xilmfs library based on the library definition segment in the MSS file.

```
proc mfs_drc {lib_handle} {
    puts "MFS DRC ..."
}
proc mfs_open_include_file {file_name} {
    set filename [file join "../../include/" $file_name]
    if {[file exists $filename]} {
      set config_inc [open $filename a]
    } else {
       set config_inc [open $filename a]
       xprint_generated_header $config_inc "MFS Parameters"
    }
    return $config_inc
}
proc generate {lib_handle} {

    puts "MFS generate ..."
    file copy "src/xilmfs.h"  "../../include/xilmfs.h"
    set conffile  [mfs_open_include_file "mfs_config.h"]
    puts $conffile "#ifndef _MFS_CONFIG_H"
    puts $conffile "#define _MFS_CONFIG_H"
    set need_utils [xget_value $lib_handle "PARAMETER" "need_utils"]
    puts $conffile "#include <xilmfs.h>"
    set value  [xget_value $lib_handle "PARAMETER" "numbytes"]
    puts  $conffile "#define MFS_NUMBYTES  $value"
    set value  [xget_value $lib_handle "PARAMETER" "base_address"]
    puts  $conffile "#define MFS_BASE_ADDRESS $value"
    set value  [xget_value $lib_handle "PARAMETER" "init_type"]
    puts  $conffile "#define MFS_INIT_TYPE  $value"
    puts $conffile "#endif"
    close $conffile
}
```

MLD Parameter Description Section

This section gives a detailed description of the constructs used in the MLD file.

Conventions

[] - denote optional values.

<> - Value substituted by the MLD writer.

Comments

Comments can be specified anywhere in the file. A '**#**' character denotes the beginning of a comment and all characters after the '#' till the end of the line are ignored. All white spaces are also ignored and semi colon with carriage returns act as a sentence delimiter.

Library Definition

The library section includes library's name, options, dependencies and other global parameters.

Syntax:

```
OPTION psf_version = <psf version number>
BEGIN LIBRARY <library name>
  [OPTION drc = <global drc name>]
  [OPTION depends  = <list of directories>]
  [OPTION help = <help file>]
  [OPTION requires_interface = <list of interface names>]
  PARAM <parameter description>
  [BEGIN CATEGORY <name of category>
    <category description>
  END CATEGORY]
  BEGIN INTERFACE <interface name>
          .......
  END INTERFACE]
END LIBRARY
```

Keywords

The keywords that are used in an MLD / MDD file are as follows:

begin

The **begin** keyword begins one of the following - library, drive, block, category, interface, array.

end

The **end** keyword signifies the end of a definition block.

psf_version:

Specifies the psf version of the library.

drc:

Specifies the DRC function name. This is the global drc function, which is called by the GUI configuration tool or the command line libgen tool. This DRC function will be called once all the parameters have been entered by the user and MLD / MDD writers can verify that a valid library / driver can be generated with the given parameters.

option:

Specifies the name following the keyword **option** is an option to the tool libgen.

copyfiles:

Specifies the files to be copied for the driver / library. If ALL is used, then all of the library / driver files are copied by Libgen.

depends:

Specifies the list of directories that needs to be compiled before library is built.

requires_interface:

Specifies the interfaces that must be provided by other libraries / drivers in the system.

help:

Specifies the help file that describes the library / driver.

dep:

Specifies the condition that needs to be satisfied before processing an entity. For example to include a parameter that is dependent on another parameter (defined as a **dep** condition), the **dep** condition should be satisfied. Conditions of the form (operand1 OP operand2) is only supported for now. In future any expression can be given as condition.

interface:

Specifies the interfaces implemented by this library / driver. It describes the interface functions and header files used by the library / driver.

```
BEGIN INTERFACE <interface name>
  OPTION DEP=<list of dependencies>;
  PROPERTY HEADER=<name of header file where the function is declared>;
  FUNCTION NAME=<name of interface function>, VALUE=<function name of
library/driver implementation> ;
END INTERFACE
```

header:

Specifies the header file in which the interface functions would be defined.

function:

Specifies the function implemented by the interface. This is a name-value pair where name is the interface function name and value is the name of the function implemented by the

library / driver.

category:

The category block defines an unconditional block. This block gets included based on the default value of the category or if included in the MSS file.

```
BEGIN CATEGORY <category name>
  PARAM name = <category name>, DESC=<param description>,
TYPE=<category type>, DEFAULT=<default>, PERMIT=<value>, DEP =
<condition>
```

```
  OPTION DEPENDS=<list of dependencies>, DRC=<drc name>, HELP=<help
file>;
  < parameters or categories description>
END CATEGORY
```

Currently nested categories are not supporetd though the syntax specifies it. Its an enhancement for future. A category is selected in a MSS file by specifying the category name as a parameter with a boolean value TRUE. A category must have a PARAM with category name.

param

The MLD file has a simple ***name* = *value*** format for most statements. The **param** keyword is required before every such NAME, VALUE pairs. The format for assigning a value to a parameter is **param *name* = <*name*>, default= *value***. The param keyword specifies that the parameter can be overwritten in the MSS file.

property:

Specifies the variour properties of the entity defined with a **begin** statement

name:

Specifies the name of the entity in which it was defined(example: **param, property**).

desc:

Describes the entity in which it was defined(example: **param, property**).

type:

Specifies the type for the entity in which it was defined(example: **param)**. The following are the types that are supported:

bool - boolean (true or false)

int - integer

string - string value within " "

enum - list of possible values, that this parameter can take

library - specify other library that is needed for building the library / driver.

peripheral_instance - specify other hardware drivers that is needed for building the library.

default:

Specifies the default value for the entity in which it was defined.

permit:

Specifies the permissions for modification of values. The following permissions exist:

NONE - no modification

TOOL- may be modified by the tool

USER - may be modified by the user (default)

If permit = none, then the category is always active.

Array

```
BEGIN ARRAY <array name>
  PROPERTY desc = <array description> ;
  PROPERTY size = <size of the array>;
  PROPERTY default = <List of Values for each element based on the size
of the array>
  # array field description as parameters
  PARAM name = <name of parameter>, desc = "description of param", type
= <type of param>, default = <default value>
.....
END ARRAY
```

Array can have any number of PARAM's and only PARAM's. It cannot have CATEGORY as one of the field of an array element. Size of the array can be defined as one of the PROPERTY of the Array. An array with with default values specified in *default* property, leads to its *size* property being initialized to the number of values. If there is no *size* property defined, a *size* property is created before initializing it with the default number of elements. Each parameter in the array can have a default value. In case where *size* is defined with an integer value, an array of *size* elements would be created wherein the value of each element being the default value of each of the parameter.

Design Rule Check (DRC) Section

```
proc mydrc { handle } {

}
```

DRC function could be any Tcl code which checks the user parameters for correctness. The drc procedures can access (read-only) the Platform Specification Format database (built by the libgen tool using the MHS and the MSS files) to read the parameter values set by the user. The "handle" is a handle to the current library in the database. The drc procedure can get the library parameters from this handle. It can also get any other parameter from the database, by first requesting for a handle and using the handle to get the parameters.

For Errors, drc procedures would call the Tcl error command 'error "error msg"', which will be displayed to the user in an error Dialog box.

For Warnings, drc procedures return a string value which can be printed on the console.

On Success, drc procedures just return without any value.

Library Generation (Generate) Section

```
proc mygenerate { handle } {

}
```

generate could be any Tcl code which reads the user parameters and generates configuration files for the library. The configuration files can be C files, Header files, Makefiles, etc. The **generate** procedures can access (read-only) thePlatform Specification Format database (built by the libgen tool using the MHS and the MSS files) to read the parameter values of the library set by the user. The "handle" is a handle to the current library in the database. The generate procedure can get the library parameters from this handle. It can also get any other parameter from the database, by first requesting for a handle and using the handle to get the parameter

Microprocessor Driver Definition (MDD)

Summary

This chapter describes the Microprocessor Driver Definition (MDD) format, Platform Specification Format 2.1.0.

Overview

An MDD file contains directives for customizing software drivers. This document describes the MDD format and the parameters that can be used to customize drivers. For more information on drivers please refer Chapter 28, “Device Drivers”. For all EDK drivers, the user does not need to peruse this document. Reading this document is recommended for user-written drivers that need to be configured by libgen tool.

Requirements

Each device driver has an MDD file and a Tcl(Tool Command Language) file associated with it. The MDD file is used by the Tcl file to customize the driver depending on different options configured in the MSS file. For more information on the MSS file format, please see Chapter 19, “Microprocessor Software Specification (MSS)”.

The driver source files and the MDD file for each driver must be located at specific directories in order for Libgen to find the files and the drivers. Please refer Chapter 7, “Library Generator” for a list of directories searched for drivers.

Driver Definition Files

Driver Definition involves defining a Data Definition file (MDD) and a Data Generation file (Tcl file).

- Data Definition file - The MDD file (named as <driver_name>_v2_1_0.mdd) contains the configurable parameters . A detailed description of the various parameters and the MDD format is described in section, “MDD Parameter Description section” in this chapter.
- Data Generation file - The second file (named as <driver_name>_v2_1_0.tcl, with the filename being the same as the mdd filename) uses the parameters configured in the MSS file for the driver to generate data. Data generated includes but not limited to

generation of header files, C files, running DRCs for the driver and generating executables. The Tcl file includes procedures that are called by Libgen tool at various stages of its execution. Various procedures in a Tcl file includes **DRC** (name of DRC given in the MDD file), **generate** (Libgen defined procedure) called after driver files are copied, **post_generate** (Libgen defined procedure) called after **generate** has been called on all drivers and libraries, **execs_generate** (Libgen defined procedure) called after the libraries and drivers have been generated. For more information on the working of libgen tool refer to Chapter 7, "Library Generator".

Note that a driver need not have the data generation file (Tcl file).

MDD Format Specification

MDD format specification involves the MDD file Format specification and the Tcl file Format specification. These are described below.

MDD File Format Specifcation

MDD file format specification involves description of parameters defined in the Parameter Description section.

Parameter Description Section

This data section describes configurable parameters in a driver. The format used to describe this section is discussed in section "MDD Parameter Description section"of this chapter.

Tcl File Format Specification

Each driver has a Tcl file associated with the MDD file. This Tcl file has the following sections :

DRC Section

This section contains Tcl routines which validate the driver parameters provided by the user for consistency.

Generation Section

This section contains Tcl routines which generate the configuration header and 'C' files based on the driver parameters

Example

This section explains the MDD format through an example MDD file and its corresponding Tcl file.

MDD file example

An example MDD file for the uartlite driver is given below:

```
OPTION psf_version = 2.1;
```

OPTION is a keyword identified by the libgen tool. The option name following the OPTION keyword is a directive to the libgen tool to do a specific action. Here psf_version of the MDD file is defined to be 2.1. This is the only option that can occur before a BEGIN DRIVER construct now.

```
BEGIN DRIVER uartlite
```

The BEGIN DRIVER construct defines the start of a driver named "uartlite".

```
  PARAM NAME = level, DESC = "Driver Level", TYPE = int, DEFAULT = 0,
RANGE = (0, 1);
```

PARAM defines a driver parameter that can be configured. Each PARAM has the following properties associated with it whose meaning is self-explanatory - NAME, DESC, TYPE, DEFAULT, RANGE.

```
BEGIN BLOCK, DEP = (level = 0)
```

BEGIN BLOCK, dep allows conditional inclusion of a set of parameters subject to a condition fulfillmen. The condition is given by the DEP construct. Here the set of parameters defined inside the BLOCK would be processed by libgen tool only when "level" parameter has a value 0.

```
    OPTION DEPENDS = (common_v1_00_a);
    OPTION COPYFILES = (xuartlite_l.c xuartlite_l.h Makefile);
   OPTION DRC = uartlite_drc;
```

The DEPENDS option specifies that the driver depends on the sources of a directory named "common_v1_00_a". The area for searching the dependent directory is decided by the libgen tool. COPYFILES option indicates the files to be copied for a "level" 0 uartlite driver. DRC option specifies the name of the Tcl procedure that the tool invokes while processing this driver. Here "uartlite_drc" is the Tcl procedure in the uartlite_v2_1_0.tcl file that would be invoked by libgen while processing the uartlite driver.

```
    BEGIN INTERFACE stdin
```

BEGIN INTERFACE defines an interface the driver supports. Here "stdin" is the name of the interface.

```
      PROPERTY header = xuartlite_l.h;
      FUNCTION name = inbyte, value = XUartLite_RecvByte;
END INTERFACE
```

An Interface contains a list of standard functions. A driver defining an interface should have values for the list of standard functions. It must also specify a header file where all the function prototypes are defined.

PROPERTY defines the properties associated with the construct defined in the BEGIN construct. Here "header" is a property with value "xuartlite_l.h", defined by the "stdin" interface. FUNCTION defines a function supported by the interface. Here "inbyte" function of "stdin" interface has a value "XUartLite_RecvByte".This function is defined in the header file "xuartlite_l.h".

```
    BEGIN INTERFACE stdout
      PROPERTY header = xuartlite_l.h;
      FUNCTION name = outbyte, value = XUartLite_SendByte;
```

```
        END INTERFACE

        BEGIN INTERFACE stdio
          PROPERTY header = xuartlite_l.h;
          FUNCTION name = inbyte, value = XUartLite_RecvByte;
          FUNCTION name = outbyte, value = XUartLite_SendByte;
        END INTERFACE

        BEGIN ARRAY interrupt_handler
          PROPERTY desc = "Interrupt Handler Information";
          PROPERTY size = 1, permit = none;
         PARAM name = int_handler, default = XIntc_DefaultHandler, desc =
    "Name of Interrupt Handler", type = string;
         PARAM name = int_port, default = Interrupt, desc = "Interrupt pin
    associated with the interrupt handler", permit = none;
        END ARRAY
```

ARRAY construct is used to define an array of parameters. Here “interrupt_handler” is the name of the array. The description (DESC) of the array and the size (SIZE) are defined as properties of the array “interrupt_handler”. The construct PERMIT is a directive to the tool that the size of the array cannot be changed by the user. The array defines “int_handler” and “int_port” as parameters of an element of the array.

```
      END BLOCK

      BEGIN BLOCK, dep = (level = 1)
        OPTION depends = (common_v1_00_a uartlite_vxworks5_4_v1_00_a);
        OPTION copyfiles = all;

        BEGIN ARRAY interrupt_handler
          PROPERTY desc = "Interrupt Handler Information";
          PROPERTY size = 1, permit = none;
          PARAM name = int_handler, default = XUartLite_InterruptHandler,
    desc = "Name of Interrupt Handler", type = string;
         PARAM name = int_port, default = Interrupt, desc = "Interrupt pin
    associated with the interrupt handler", permit = none;
        END ARRAY

       PARAM name = connect_to, desc = "Connect to operationg system", type
    = enum, values = {"VxWorks5_4" = VxWorks5_4, "None" = none}, default =
    none;
      END BLOCK
    END DRIVER
```

END is used with the construct name that was used in the BEGIN statement. Here END is used with BLOCK and DRIVER constructs to indicate the end of each of BLOCK and DRIVER constructs.

Example Tcl File

The following is the uartlite_v2_1_0.tcl file corresponding the uartlite_v2_1_0.mdd file described in the previous section. The “uartlite_drc” procedure would be invoked by libgen for uartlite driver while running DRCs for drivers. The **generate** routine generates constants in a header file and a c file for uartlite driver based on the driver definition segment in the MSS file.

```
proc uartlite_drc {drv_handle} {
  puts “UartLite DRC”
}
```

```
proc generate {drv_handle} {
  set level [xget_value $drv_handle "PARAMETER" "level"]
  if {$level == 0} {
    xdefine_include_file $drv_handle "xparameters.h" "XUartLite"
"NUM_INSTANCES" "C_BASEADDR" "C_HIGHADDR"
  }
  if {$level == 1} {
    xdefine_include_file $drv_handle "xparameters.h" "XUartLite"
"NUM_INSTANCES" "C_BASEADDR" "C_HIGHADDR" "DEVICE_ID" "C_BAUDRATE"
"C_USE_PARITY" "C_ODD_PARITY"
    xdefine_config_file $drv_handle "xuartlite_g.c" "XUartLite"
"DEVICE_ID" "C_BASEADDR" "C_BAUDRATE" "C_USE_PARITY" "C_ODD_PARITY"
  }
}
```

MDD Parameter Description section

This section gives a detailed description of the constructs used in the MDD file.

Conventions

[] - denote optional values.

<> - Value substituted by the MDD writer.

Comments

Comments can be specified anywhere in the file. A '**#**' character denotes the beginning of a comment and all characters after the '#' till the end of the line are ignored. All white spaces are also ignored and semi colon with carriage returns act as a sentence delimiter.

Driver Definition

The driver section includes driver's name, options, dependencies and other global parameters.

Syntax:

```
OPTION psf_version = <psf version number>
BEGIN DRIVER <driver name>
  [OPTION drc = <global drc name>]
  [OPTION depends  = <list of directories>]
  [OPTION help = <help file>]
  [OPTION requires_interface = <list of interface names>]
  PARAM <parameter description>
  [BEGIN BLOCK,dep = <condition>
        .......
  END BLOCK]
  [BEGIN INTERFACE <interface name>
        .......
  END INTERFACE]
END DRIVER
```

Keywords

The keywords that are used in an MLD / MDD file are as follows:

begin

The **begin** keyword begins one of the following - library, drive, block, category, interface, array.

end

The **end** keyword signifies the end of a definition block.

psf_version:

Specifies the psf version of the library.

drc:

Specifies the DRC function name. This is the global drc function, which is called by the GUI configuration tool or the command line libgen tool. This DRC function will be called once all the parameters have been entered by the user and MLD / MDD writers can verify that a valid library / driver can be generated with the given parameters.

option:

Specifies the name following the keyword **option** is an option to the tool libgen.

copyfiles:

Specifies the files to be copied for the driver / library. If ALL is used, then all of the library / driver files are copied by Libgen.

depends:

Specifies the list of directories that needs to be compiled before library is built.

requires_interface:

Specifies the interfaces that must be provided by other libraries / drivers in the system.

help:

Specifies the help file that describes the library / driver.

dep:

Specifies the condition that needs to be satisfied before processing an entity. For example to enter into a **block**, the **dep** condition should be satisfied. Conditions of the form (operand1 OP operand2) is only supported for now. In future any expression can be given as condition.

block:

Specifies the block is to be entered into when the **dep** condition is satisfied. Note that nested blocks are not supported currently.

interface:

Specifies the interfaces implemented by this library / driver. It describes the interface functions and header files used by the library / driver.

```
BEGIN INTERFACE <interface name>
  OPTION DEP=<list of dependencies>;
  PROPERTY HEADER=<name of header file where the function is declared>;
  FUNCTION NAME=<name of interface function>, VALUE=<function name of
library/driver implementation> ;
```

```
    END INTERFACE
```

header:

Specifies the header file in which the interface functions would be defined.

function:

Specifies the function implemented by the interface. This is a name-value pair where name is the interface function name and value is the name of the function implemented by the library / driver.

param

The MLD / MDD file has a simple ***name = value*** format for most statements. The **param** keyword is required before every such NAME, VALUE pairs. The format for assigning a value to a parameter is **param *name* = <*name*>, default= *value*.** The param keyword specifies that the parameter can be overwritten in the MSS file.

property:

Specifies the variour properties of the entity defined with a **begin** statement

name:

Specifies the name of the entity in which it was defined(example: **param, property**).

desc:

Describes the entity in which it was defined(example: **param, property**).

type:

Specifies the type for the entity in which it was defined(example: **param)**. The following are the types that are supported:

bool - boolean (true or false)

int - integer

string - string value within " "

enum - list of possible values, that this parameter can take

library - specify other library that is needed for building the library / driver.

peripheral_instance - specify other hardware drivers that is needed for building the library / driver.

default:

Specifies the default value for the entity in which it was defined.

permit:

Specifies the permissions for modification of values. The following permissions exist:

NONE - no modification

TOOL- may be modified by the tool

USER - may be modified by the user (default)

If permit = none, then the category is always active. This property is still experimental. Tools do not perform any action for this property for EDK6.1 release.

Array

```
BEGIN ARRAY <array name>
  PROPERTY desc = <array description> ;
  PROPERTY size = <size of the array>;
  PROPERTY default = <List of Values for each element based on the size
of the array>
  # array field description as parameters
  PARAM name = <name of parameter>, desc = "description of param", type
= <type of param>, default = <default value>
.....
END ARRAY
```

Array can have any number of PARAM's and only PARAM's. It cannot have CATEGORY as one of the field of an array element. Size of the array can be defined as one of the PROPERTY of the Array. An array with with default values specified in *default* property, leads to its *size* property being initialized to the number of values. If there is no *size* property defined, a *size* property is created before initializing it with the default number of elements. Each parameter in the array can have a default value. In case where *size* is defined with an integer value, an array of *size* elements would be created wherein the value of each element being the default value of each of the parameter.

Design Rule Check (DRC) section

```
proc mydrc { handle } {

}
```

DRC function could be any Tcl code which checks the user parameters for correctness. The drc procedures can access (read-only) the Platform Specification Format database (built by the libgen tool using the MHS and the MSS files) to read the parameter values set by the user. The "handle" is a handle to the current driver in the database. The drc procedure can get the driver parameters from this handle. It can also get any other parameter from the database, by first requesting for a handle and using the handle to get the parameters.

For Errors, drc procedures would call the Tcl error command 'error "error msg"', which will be displayed to the user in an error Dialog box.

For Warnings, drc procedures return a string value which can be printed on the console.

On Success, drc procedures just return without any value.

Driver Generation section (Generate)

```
proc mygenerate { handle } {

}
```

generate could be any Tcl code which reads the user parameters and generates configuration files for the driver. The configuration files can be C files, Header files, Makefiles, etc. The **generate** procedures can access (read-only) thePlatform Specification Format database (built by the libgen tool using the MHS and the MSS files) to read the parameter values of the driver set by the user. The "handle" is a handle to the current driver in the database. The generate procedure can get the driver parameters from this handle. It can also get any other parameter from the database, by first requesting for a handle and using the handle to get the parameter

Xilinx Microkernel (XMK)

Scope

This chapter describes the organization of Xilinx Microkernel and the interaction of its components with the user application. Xilinx provides three libraries,

- Math Library **(libm)**
- Standard C language support **(libc)**
- Xilinx drivers and libraries (**libxil)**

Overview

The Standard C support library consists of the newlib libc, which contains the standard C functions such as strcpy, strcmp.

The Xilinx Microkernel contains the following components

- Xilinx file support functions **LibXil File**
- Xilinx memory file system **LibXil Mfs**
- Xilinx networking support **LibXil Net**
- Xilinx kernel support **LibXil Kernel**
- Xilinx device drivers **LibXil Driver**
- Xilinx Standalone Board Support Package (BSP)

Most of the routines in the library are written in C and can be ported to any platform. The Library Generator (LibGen) configures the libraries for an embedded processor, using the attributes defined in the Microprocessor Software Specification (MSS) file.

The math library is an enhancement over the newlib math library **libm.a .**

XMK Organization

The structure of **XMK** is outlined in Figure 22-1. The user application calls routines implemented in **LibXil** and / or **libm**. In addition to the standard C routines supported by **libc.a**, Xilinx Microkernel contains the following modules:

- Stream based file system and device access (LibXil File)
  - These set of libraries allow access to devices and file systems through system routines such as **open, close, read** and **write.**
  - For complete details refer to the Chapter 24, “LibXil File” chapter.
- Memory based file system (LibXil Mfs)

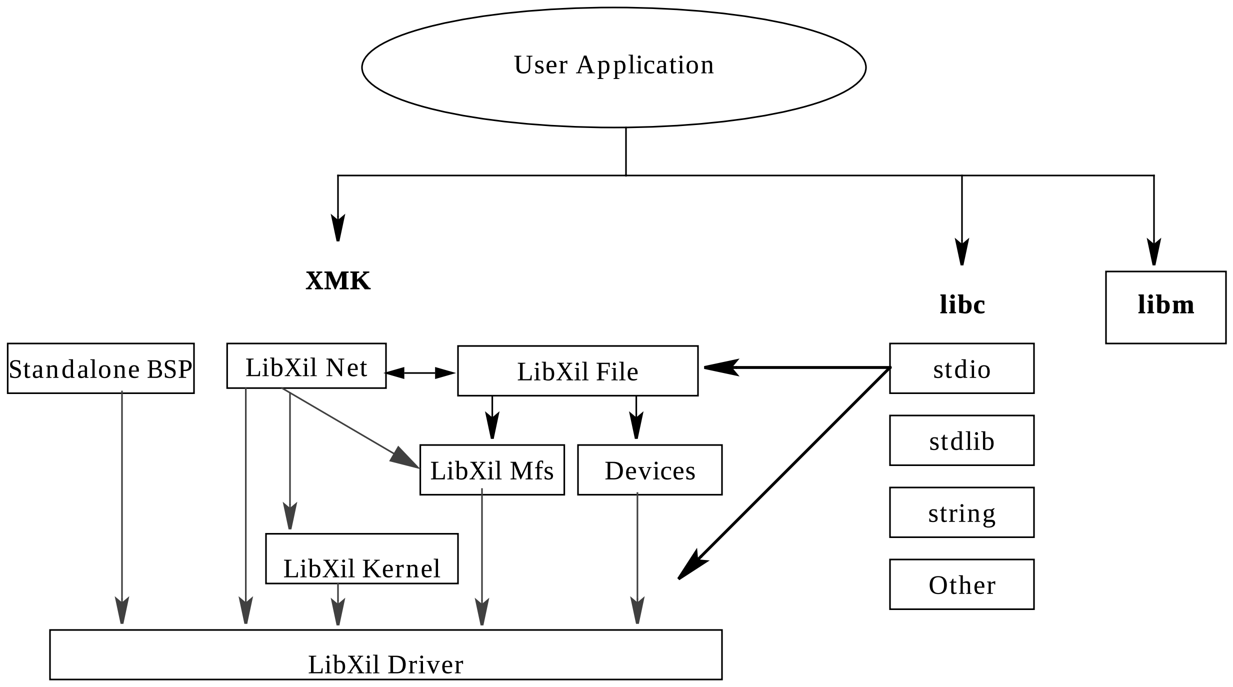


Figure 22-1: **Structure of XMK**

    - ♦ Xilinx provides a simple memory based file system, which allows easy access to data using file based input-output.
    - ♦ This system can be easily configured to meet project requirements by changing the source provided in the installation area.
    - ♦ This module is discussed in details in the Chapter 25, "LibXil Memory File System" chapter.
- Networking application support (LibXil Net)
    - ♦ EDK provides a simple TCP / IP stack based library, which can be used for network related projects.
    - ♦ For complete details, refer to Chapter 26, "LibXil Net".
- Kernel Support (LibXil Kernel)
    - ♦ EDK provides a simple embedded processor kernel, which can be customized for any system.
    - ♦ For complete details, refer to Chapter 27, "LibXil Kernel".
- Device drivers (LibXil Driver)
    - ♦ Some of the library modules interact with drivers. These drivers are provided in the Embedded Development Kit and are configured by libgen.
    - ♦ Drivers are detailed in the Chapter 28, "Device Drivers" chapter.
- Standalone Board Support Package (BSP)
    - ♦ Certain standalone board support files such as the crt0.S, boot.S and eabi.S are required for the powerpc processor. These files are provided in the EDK.

- For a detailed description, refer to the Chapter 29, “Stand-Alone Board Support Package” description.

These libraries and include files are created in the current project’s `lib` and `include` directories respectively. The **-I** and **-L** options of mb-gcc should be used to add these directories to its library search paths. Please refer to the Chapter 19, “Microprocessor Software Specification (MSS)”chapter and Chapter 7, “Library Generator” chapter for more information.

XMK Customization

The standard newlib libc contains dummy functions for most of the operating system specific function calls such as **open, close, read, write**. These routines are included in the **libgloss** component of the standard libc library. The LibXil File module contains routines to overwrite these dummy functions. The routines interact with file systems such as Xilinx Memory File System[1] and peripheral devices[2] such as UART, UARTLITE and GPIO.

LibXil Net routines provide support for networking applications via the ethernet. This module is discussed more in details in the Chapter 26, “LibXil Net” chapter. The module LibXil Net needs some support from the file system and hence calls other routines from the LibXil File and / or the LibXil Mfs modules. On the other hand, if an application requires opening files over the network, routines from the LibXil File module will need the support of the LibXil Net.

LibXilKernel has the key features of RTOS like multi-tasking, priority-driven preemptive scheduling, support for Inter-Process communication and synchronization. It is small, modular, user customizable and can be used in any system configuration. It also has system call interface, which allows a system to be built in different configurations. For details refer Chapter 27, “LibXil Kernel”.

LibGen is used to tailor the library compilation for a particular project using attributes in the MSS. These attributes are described in theChapter 24, “LibXil File” and Chapter 25, “LibXil Memory File System” chapters.

1. For more information on Memory File System, please refer to the chapter on LibXil Mfs

2. For more information on Device Drivers, please refer to the chapter on LibXil Driver

Chapter 23

LibXil Standard C Libraries

Summary

This chapter describes the software libraries available for the embedded processors.

Overview

The Embedded Processor Design Kit (EDK) libraries and device drivers provide standard C library functions, as well as functions to access peripherals. The EDK libraries are automatically configured by libgen for every project based upon the Microprocessor Software Specification file. These libraries and include files are saved in the current project's lib and include directories respectively. The **-I** and **-L** options of **`mb-gcc`** should be used to add these directories to its library search paths.

Standard C Library (libc.a)

The standard C library *libc.a* contains the standard C functions compiled for MicroBlaze or PowerPC. For a list of all the supported functions refer to the following files in *XILINX_EDK* / gnu / *processor* / *platform* / include

where

- ***processor* = powerpc-eabi** or **microblaze**
- ***platform* = sol** or **nt**
- ***XILINX_EDK*** = Installation directory

```
_ansi.h     fastmath.h  machine/    reent.h     stdlib.h    utime.h
_syslist.h  fcntl.h     malloc.h    regdef.h    string.h    utmp.h
ar.h        float.h     math.h      setjmp.h    sys/
assert.h    grp.h       paths.h     signal.h    termios.h
ctype.h     ieeefp.h    process.h   stdarg.h    time.h
dirent.h    limits.h    pthread.h   stddef.h    unctrl.h
errno.h     locale.h    pwd.h       stdio.h     unistd.h
```

Programs accessing standard C library functions must be compiled as follows:

For MicroBlaze

```
mb-gcc C files
```

For PowerPC

```
powerpc-eabi-gcc C files
```

The **`libc`** library is included automatically.

The **`-lm`** option should be specified for programs that access libm math functions.

Refer to Chapter "MicroBlaze Application Binary Interface" (ABI) in the "MicroBlaze Processor Reference Guide" for information on the C Runtime Library

Xilinx C Library (libxil.a)

The Xilinx C library *libxil.a* contains the following functions for the MicroBlaze Embedded processor:

```
_exception_handler.o
_interrupt_handler.o
xil_malloc.o
xil_sbrk.o
```

Default exception and interrupt handlers are provided. A memory management targeted for embedded systems is provided in *xil_malloc.o* file. The **`libxil.a`** library is included automatically.

Programs accessing Xilinx C library functions must be compiled as follows:

```
mb-gcc C files
```

Input/Output Functions

The EDK libraries contains standard C functions for I/O; such as printf and scanf. These are large and may not be suitable for embedded processors. In addition, the MicroBlaze processor library provides the following smaller I/O functions:

```
void print (char *)
```

This function prints a string to the peripheral designated as standard output in the MSS file.

```
void putnum (int)
```

This function converts an integer to a hexadecimal string and prints it to the peripheral designated as standard output in the MSS file.

```
void xil_printf (const *char ctrl1, ...)
```

This function is similar to *printf* but much smaller in size (only 1KB). It does not have support for floating point numbers. *xil_printf* also does not support printing of long long (i.e 64 bit numbers).

The prototypes for these functions are in **`stdio.h`**.

Please refer to Chapter 19, "Microprocessor Software Specification (MSS)" for information on setting the standard input and standard output devices for a system.

Memory Management Functions

MicroBlaze Processor

Memory management routines such as **`malloc, calloc`** and **`free`** can run the gamut of high functionality (with associated large size) to low functionality (and small size). This version of the MicroBlaze processor library only supports a simple, small `malloc`, and a dummy **`free`**. Hence when memory is allocated using `malloc`, this memory cannot be reused.

The **_STACK_SIZE** option to **mb-gcc** specifies the total memory allocated to stack and heap. The stack is used for function calls, register saves and local variables. All calls to *malloc* allocate memory from heap. The stack pointer initially points to the bottom (high end) of memory, and grows toward low memory while the heap pointer starts at low memory and grows towards high memory. The size of the heap cannot be increased at runtime. The return value of **malloc** must always be checked to ensure that it could actually allocate the memory requested.

Please note that whereas *malloc* checks that the memory it allocates does not overwrite the current stack pointer, updates to the stack pointer do not check if the heap is being overwritten.

Increasing the ***_STACK_SIZE*** may be one way to solve unexpected program behavior. Refer to the "Linker Options" section of Chapter 11, "GNU Compiler Tools" for more information on increasing the stack size.

PowerPC 405 Processor

PowerPC 405 processor supports all standard C library memory management functions such as *malloc(), calloc(), free().*

Arithmetic Operations

MicroBlaze Processor

Integer Arithmetic

Integer addition and subtraction operations are provided in hardware. By default, integer multiplication is done in software using the library function **mulsi3_proc**. Integer multiplication is done in hardware if the mb-gcc option **-mno-xl-soft-mul** is specified.

Integer divide and mod operations are done in software using the library functions **divsi3_proc** and **modsi3_proc**.

Double precision multiplication, division and mod functions are carried out by the library functions **muldi3_proc, divdi3_proc** and **moddi3_proc** respectively.

Floating Point Arithmetic

All floating point addition, subtraction, multiplication and division operations are also implemented using software functions in the C library.

PowerPC 405 Processor

Integer Arithmetic

Integer addition and subtraction operations are provided in hardware. Hence no specific software library is available for the PowerPC processor.

Floating Point Arithmetic

PowerPC supports all floating point arithmetic implemented in the standard C library.

Chapter 24

LibXil File

Scope

Xilinx libraries provide block access to file systems and devices using standard calls such as open, close, read, write etc. These routines form the LibXil File Module of the Libraries.

A system can be configured to use LibXil File module, using the Library Generator (libgen)

Overview

The LibXil library provides block access to files and devices through the **LibXil File** module. This module provides standard routines such as **open, close, read, write etc.** to access file systems and devices.

The module **LibXil File** can also be easily modified to incorporate additional file systems and devices. This module implements a subset of operating system level functions.

Module Usage

A file or a device is opened for read and write using the open call in the library. The library maintains a list of open files and devices. Read and write commands can be issued to access blocks of data from the open files and devices.

Module Routines

Functions
`int` **`open`** `(const char *`*`name`*`, int `*`flags`*`, int `*`mode`*`)`
`int` **`close`** `(int `*`fd`*`)`
`int` **`read`** `(int `*`fd`*`, char* `*`buf`*`, int `*`nbytes`*`)`
`int` **`write`** `(int `*`fd`*`, char* `*`buf`*`, int `*`nbytes`*`)`
`long` **`lseek`** `(int `*`fd`*`, long `*`offset`*`, int `*`whence`*`)`
`int` **`chdir`** `(const char *`*`buf`*`)`
`const char*` **`getcwd`** `(void)`

`int` **`open`** `(const char *`*`name`*`, int `*`flags`*`, int `*`mode`*`)`

Parameters	*name* refers to the name of the device / file *flags* refers to the permissions of the file. This field does not have any meaning for a device *mode* indicates whether the stream is opened in read, write or append mode.
Returns	file / device descriptor *fd assigned* by LibXil File
Description	This call registers the device or the file in the local device table and calls the underlying open function for that particular file or a device.
Includes	xilfile.h xparameters.h

`int` **`close`** `(int `*`fd`*`)`

Parameters	*fd* refers to the file descriptor assigned during by open()
Returns	If a file is being close, returns the status returned by the underlying file system. For devices, it returns 1, since devices can not be closed. 0 indicates success in closing a file. Any other value indicates error
Description	Close the file / device with the fd.
Includes	xilfile.h xparameters.h

`int` **`read`** `(int `*`fd`*`, char* `*`buf`*`, int `*`nbytes`*`)`

Parameters	*fd* refers to the file descriptor assigned by open() *buf* refers to the destination buffer where the contents of the stream should be copied *nbytes*: Number of bytes to be copied
Returns	The number of bytes read.
Description	Read *nbytes* from the file / device pointed by the file descriptor *fd* and store it in the destination pointed by *buf*.
Includes	xilfile.h xparameters.h

`int` **`write`** `(int` *`fd`*`, char*` *`buf`*`, int` *`nbytes`*`)`

Parameters	*fd*: refers to the file descriptor assigned by open() *buf:* refers to the source buffer *nbytes:* Number of bytes to be copied
Returns	The number of bytes written to the file.
Description	Write *nbytes* from the buffer, *buf* to the file pointed by the file descriptor *fd*
Includes	xilfile.h xparameters.h

`long` **`lseek`** `(int` *`fd`*`, long` *`offset`*`, int` *`whence`*`)`

Parameters	*fd*: file descriptor returned by open *offset*: Number of bytes to seek *whence*: Location to seek from. This parameter depends on the underlying File System being used.
Returns	New file pointer location
Description	The lseek() system call moves the file pointer for *fd* by *offset* bytes from *whence*.
Includes	xilfile.h xparameters.h

`int` **`chdir`** `(char*` *`newdir`*`)`

Parameters	*newdir*: Destination directory
Returns	The same value as returned by the underlying file system. -1 for failure.
Description	Change the current directory to *newdir*
Includes	xilfile.h xparameters.h

`const char*` **`getcwd`** `(void)`

Parameters	None
Returns	The current working directory.
Description	Get the absolute path for the current working directory.
Includes	xilfile.h xparameters.h

Libgen Support

LibXil File Instantiation

The users can write application to either interact directly with the underlying file systems and devices or make use of the **LibXil File** module to integrate with file systems and devices.

In order to instantiate LibXil File module in your system, use the following code in your MSS file.

```
BEGIN LIBRARY
 PARAMETER LIBRARY_NAME = xilfile
 PARAMETER LIBRARY_VER = 1.00.a
 PARAMETER peripherals = ( ("<peripheral 1 instance name>", "<mount 1>")
, ("<peripheral 2 instance name>", "<mount 2>") )
 PARAMETER filesys = (( "<Filesystem 1 name>" , "<mount 1>" ), (
"<Filesystem 2 name>" , "<mount 2>" ))
END
```

where

- *<peripheral 1 name>* : Instance name of the peripheral you need to access through XilFile.
- *<mount 1>* : Mount name for the *<peripheral 1>* or *<filesystem 1>*
- *<filesystem 1 name>* : Name of the filesystem, you need to access through XilFile. [1]

For example, to access two uarts (**myuart1** and **myuart2**) and the memory file system (**xilmfs**) , use the following code snippet.

```
BEGIN LIBRARY
 PARAMETER LIBRARY_NAME = xilfile
 PARAMETER LIBRARY_VER = 1.00.a
 PARAMETER peripherals = ( ("myuart", "/dev/myuart") , ("myuart2",
"/dev/myuart2") )
 PARAMETER filesys = (( "xilmfs" , "/dev/mfs" ))
END
```

System Initialization

LibGen also generates the system initialization file, which is compiled into the LibXil library. This file initialized the data structure required by the **LibXil File** module, such as the Device tables and the File System table. This routine also initializes the STDIN, STDOUT and STDERR if present.

Limitations

LibXil File module currently enforces the following restrictions:

1. Note that there is no instance name for filesystem and other Xilinx Microkernel moudles. Hence, the name of the filesystem should be used instead of the instance name.

- Only one instance of a File System can be mounted. This file system and the mount point has to be indicated in the Microprocessor Software Specification (MSS) file.
- Files cannot have names starting with **/dev**, since it is a reserved word to be used only for accessing devices
- Currently LibXil File has support only for 1 file system (LibXil Memory File System) and 3 devices (UART, UARTlite and GPIO).
- Only devices can be assigned as STDIN, STDOUT and STDERR

Chapter 25

LibXil Memory File System

Scope

This document describes the Memory File System (MFS). This file system resides in RAM / ROM / Flash memory and can be accessed through LibXil File module or directly. Memory File System is integrated with a system using the Library Generator.

Overview

The Memory File System (MFS) component, **LibXil MFS**, provides users the capability to manage program memory in the form of file handles. Users can create directories, and can have files within each directory. The file system can be accessed from the high level C-language through function calls specific to the file system. Alternatively, the users can also manage files through the standard C language functions like **open** provided in **XilFile.**

MFS Functions

Quick Glance

This section presents a list of functions provided by the MFS. Table 25-1 provides the function names with signature at a glance. C-like access.

Table 25-1: **MFS functions at a glance**

Functions
`void` **`mfs_init_fs`** `(int numbytes, char *address, int status)`
`int` **`mfs_change_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_get_current_dir_name`** `(char *`*`dirname`*`)`
`int` **`mfs_create_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_delete_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_exists_file`** `(char *`*`filename`*`)`
`int` **`mfs_delete_file`** `(char *`*`filename`*`)`
`int` **`mfs_rename_file`** `(char *`*`from_file`*`, char *`*`to_file`*`)`
`int` **`mfs_get_usage`**`(int *`*`num_blocks_used`*`, int *`*`num_blocks_free`*`)`
`int` **`mfs_dir_open`** `(char *`*`dirname`*`)`
`int` **`mfs_dir_close`** `(int` *`fd`*`)`

Table 25-1: **MFS functions at a glance**

Functions
`int` **`mfs_dir_read`** `(int` *`fd`*`, char **`*`filename, int *filesize, int *filetype`*`)`
`int` **`mfs_file_open`** `(char *`*`filename`*`, int` *`mode`*`)`
`int` **`mfs_file_read`** `(int` *`fd`*`, char *`*`buf`*`, int` *`buflen`*`)`
`int` **`mfs_file_write`** `(int` *`fd`*`, char *`*`buf`*`, int` *`buflen`*`)`
`int` **`mfs_file_close`**`(int` *`fd`*`)`
`long` **`mfs_file_lseek`** `(int` *`fd`*`, long` *`offset`*`, int` *`whence)`*
`int` **`mfs_ls`** `(void)`
`int` **`mfs_cat`** `(char *`*`filename`*`)`
`int` **`mfs_copy_stdin_to_file`** `(char *`*`filename`*`)`
`int` **`mfs_file_copy`** `(char *`*`from_file`*`, char *`*`to_file`*`)`

Detailed summary of MFS Functions

`int` **`mfs_init_fs`** `(int numbytes, char *address, int status)`

Parameters	*numbytes* is the number of bytes of memory available for the file system *address* is the starting(base) address of the file system memory *status* is one of **MFSINIT_NEW, MFSINIT_IMAGE** or **MFSINIT_ROM_IMAGE**
Returns	1 for success 0 for failure
Description	Initialize the memory file system. This function must be called before any file system operation. The status / mode parameter determines certain filesystem properties: **MFSINIT_NEW** creates a new, empty file system for read / write **MFSINIT_IMAGE** initializes a filesystem whose data has been previously loaded into memory at the base address **MFSINIT_ROM_IMAGE** initializes a Read-Only filesystem whose data has been previously loaded into memory at the base address
Includes	xilmfs.h

`int` **`mfs_change_dir`** `(char *`*`newdir`*`)`

Parameters	*newdir* is the chdir destination.
Returns	1 for success 0 for failure
Description	If *newdir* exists, make it the current directory of MFS. Current directory is not modified in case of failure.
Includes	xilmfs.h

`int` **`mfs_create_dir`** `(char *`*`newdir`*`)`

Parameters	*newdir:* Directory name to be created
Returns	On success, return index of new directory in the file system On failure, return 0
Description	Create a new empty directory called *newdir* inside the current directory.
Includes	xilmfs.h

`int` **`mfs_delete_dir`** `(char *`*`dirname`*`)`

Parameters	*dirname:* Directory to be deleted
Returns	On success, return index of new directory in the file system On failure, return 0
Description	Delete the directory dirname, if it exists and is empty,
Includes	xilmfs.h

`int` **`mfs_get_current_dir_name`** `(char *`*`dirname`*`)`

Parameters	*dirname:* Current directory name is returned in this pointer
Returns	On Success return 0 On failure return 1
Description	Return the name of the current directory in a pre allocated buffer, *dirname*, of at least 16 chars.Note that it does not return the absolute path name of the current directory, but just the name of the current directory
Includes	xilmfs.h

`int` **`mfs_delete_file`** `(char *`*`filename`*`)`

Parameters	*filename:* file to be deleted
Returns	1 for success 0 for failure
Description	Delete *filename* from its directory.
Includes	xilmfs.h

`int` **`mfs_rename_file`** `(char *`*`from_file`*`, char *`*`to_file`*`)`

Parameters	*from_file:* Original filename *to_file:* New file name
Returns	On success, return 1 On failure, return 0
Description	Rename *from_file* to *to_file*. Rename works for directories as well as files. Function fails if *to_file* already exists.
Includes	xilmfs.h

`int` **`mfs_exists_file`** `(char *`*`filename`*`)`

Parameters	*filename:* file / directory to be checked for existence
Returns	0: if *filename* does not exist 1: if *filename* is a file 2: if *filename* is a directory
Description	Check if the file / directory is present in current directory.
Includes	xilmfs.h

`int` **`mfs_get_usage`** `(int *`*`num_blocks_used`*`, int *`*`num_blocks_free`*`)`

Parameters	*num_blocks_used:* Number of blocks used *num_blocks_free:* Number of free blocks
Returns	On Success return 0 On failure return 1
Description	Get the number of used blocks and the number of free blocks in the file system through pointers.
Includes	xilmfs.h

`int` **`mfs_dir_open`** `(char *`*`dirname`*`)`

Parameters	*dirname:* directoryto be opened for reading
Returns	The index of dirname in the array of open files or -1 on failure.
Description	Open directory dirname for reading. Reading a directory is done using mfs_dir_read()
Includes	xilmfs.h

`int` **`mfs_dir_close`** `(int `*`fd`*`)`

Parameters	*fd:* File descriptor return by open
Returns	On success return 1 On failure return 1
Description	Close the dir pointed by *fd*. The file system regains the fd and uses it for new files.
Includes	xilmfs.h

`int` **`mfs_dir_read`** `(int `*`fd`*`, char **`*`filename`*`, int *`*`filesize`*`, int *`*`filetype`*`)`

Parameters	*fd:* File descriptor return by open; passed to this function by caller *filename:* Pointer to file name at the current position in the directory in MFS; this value is filled in by this function *filesize:* Pointer to a value filled in by this function: Size in bytes of filename, if it is a regular file; Number of directory entries if filename is a directory *filetype:* Pointer to a value filled in by this function: **MFS_BLOCK_TYPE_FILE** if *filename* is aregular file; **MFS_BLOCK_TYPE_DIR** if *filename* is a directory
Returns	On Success return number of bytes read. On Failure return 1
Description	Read the current directory entry and advance the internal pointer to the next directory entry. *filename*, *filetype* and *filesize* are pointers to values stored in the current directory entry
Includes	xilmfs.h

`int` **`mfs_file_open`** `(char *`*`filename`*`, int `*`mode`*`)`

Parameters	*filename:* file to be opened *mode:* Read / Write or Create mode.
Returns	The index of filename in the array of open files or -1 on failure.
Description	Open file filename with given mode. The function should be used for files and not directories: **MODE_READ**, no error checking is done (if file or directory). **MODE_CREATE** creates a file and not a directory. **MODE_WRITE** fails if the specified file is a **DIR**.
Includes	xilmfs.h

`int` **`mfs_file_read`** `(int `*`fd`*`, char *`*`buf`*`, int `*`buflen`*`)`

Parameters	*fd:* File descriptor return by open *buf:* Destination buffer for the read *buflen:* Length of the buffer
Returns	On Success return number of bytes read. On Failure return 1
Description	Read *buflen* number bytes and place it in *buf*. *fd* should be a valid index in “open files” array, pointing to a file, not a directory. *buf* should be a pre-allocated buffer of size *buflen* or more. If fewer than *buflen* chars are available then only that many chars are read.
Includes	xilmfs.h

`int` **`mfs_file_write`** `(int `*`fd`*`, char *`*`buf`*`, int `*`buflen`*`)`

Parameters	*fd:* File descriptor return by open *buf:* Source buffer from where data is read *buflen:* Length of the buffer
Returns	On Success return 1 On Failure return 1
Description	Write *buflen* number of bytes from *buf* to the file. *fd* should be a valid index in open_files array. *buf* should be a pre-allocated buffer of size buflen or more.
Includes	xilmfs.h

`int` **`mfs_file_close`** `(int` *`fd`*`)`

Parameters	*fd:* File descriptor return by open
Returns	On success return 1 On failure return 1
Description	Close the file pointed by *fd*. The file system regains the fd and uses it for new files.
Includes	xilmfs.h

`long` **`mfs_file_lseek`** `(int` *`fd`*`, long` *`offset`*`, int` *`whence`*`)`

Parameters	*fd:* File descriptor return by open *offset:* Number of bytes to seek *whence:* File system dependent mode: If *whence* is **MFS_SEEK_END**, the *offset* can be either 0 or negative, otherwise *offset* should be non-negative. If *whence* is **MFS_SEEK_CURR**, the offset is calculated from the current location If *whence* is **MFS_SEEK_SET**, the offset is calculated from the start of the file
Returns	On success, return 1 On failure, return 0
Description	Seek to a given *offset* within the file at location *fd* in open_files array. It is an error to seek before beginning of file or after the end of file.
Includes	xilmfs.h

Utility Functions

The following few functions are utility functions that can be used along with Xilinx MFS. These functions are defined in mfs_filesys_util.c and are not declared in xilmfs.h. They must be declared by the user if needed.

`int` **`mfs_ls`** `(void)`

Parameters	None
Returns	On success return 1 On failure return 0
Description	List contents of current directory on **STDOUT**.
Includes	xilmfs.h

`int` **`mfs_cat`** `(char *`*`filename`*`)`

Parameters	*filename:* File to be displayed
Returns	On success return 1 On failure return 0
Description	Print the file to **STDOUT**.
Includes	xilmfs.h

`int` **`mfs_copy_stdin_to_file`** `(char *`*`filename`*`)`

Parameters	*filename:* Destination file.
Returns	On success return 1 On failure return 0
Description	Copy from **STDIN** to named file.
Includes	xilmfs.h

`int` **`mfs_file_copy`** `(char *`*`from_file`*`, char *`*`to_file`*`)`

Parameters	*from_file:* Source file *to_file*: Destination file
Returns	On success return 1 On failure return 0
Description	Copy *from_file* to *to_file*. It fails if *to_file* already exists, or if either could not be opened.
Includes	xilmfs.h

Additional Utilities

A programcalled **mfsgen** is provided along with the MFS library. **mfsgen** can be used to create an MFS memory image on a host system that can subsequently be downloaded to the embedded system memory. mfsgen is provided as source code in the file mfsgen.c in the utils sub-directory. This source can be compiled on the host platform (Windows, Solaris or other) to generate the executable for mfsgen. An entire directory hierarchy on the host system can be copied to a local MFS file image using mfsgen. This file image can then be downloaded on to the memory of the emebedded system for creating a pre-loaded file system. A few test programs are included to show how this is done. More information can be found in the readme.txt file in the utils sub-directory.

C-like access

The user can choose not to deal with the details of the file system by using the standard C-like interface provided by **Xil File.** It provides the basic C stdio functions like **open**, **close**, **read**, **write**, and **seek**. These functions have identical signature as those in the standard ANSI-C. Thus any program with file operations performed using these functions can be easily ported to MFS by interfacing the MFS in conjunction with library Xilfile.

LibGen Customization

Memory file system can be integrated with a system using the following snippet in the mss file. The memory file system should be instantiated with the name **xilmfs.** The attributes used by libgen and their descriptions are given in Table 25-2

```
BEGIN LIBRARY
parameter LIBRARY_NAME = xilmfs
parameter LIBRARY_VER = 1.00.a
parameter numbytes= 50000
parameter base_address = 0xffe00000
parameter init_type = MFSINIT_NEW
parameter need_utils = false
END
```

Table 25-2: **Attributes for including Memory File System**

Attributes	Description
numbytes	Number of bytes allocated for file system.
base_address	Starting address for file system memory
init_type	MFSINIT_NEW (default) or MFSINIT_IMAGE or MFSINIT_ROM_IMAGE
need_utils	true or false (default = false)

XILINX®

Chapter 26

LibXil Net

Summary

This chapter describes the network library for Embedded processors, libXilNet. The library includes functions to support the TCP/IP stack and the higher level application programming interface (Socket APIs).

Overview

The Embedded Development Kit (EDK) networking library, **libXilNet**, allows a processor to connect to the internet. LibXilNet includes functions for handling the TCP/IP stack protocols. It also provides a simple set of Sockets Application Programming Interface (APIs) functions enabling network programming. Lib Xil Net supports multiple connections (through Sockets interface) and hence enables multiple client support. This chapter describes the various functions of LibXilNet.

LibXilNet Functions

Quick Glance

Table 26-1 presents a list of functions provided by the LibXilNet at a glance.

Table 26-1: **LibXilNet functions at a glance**

Functions
`int` **`xilsock_init`** `(`*`void`*`)`
`void` **`xilsock_rel_socket`** `(int` *`sd`*`)`
`int` **`xilsock_socket`** `(int` *`domain,`* `int` *`type,`* `int` *`proto`*`)`
`int` **`xilsock_bind`** `(int` *`sd,`* `struct sockaddr*` *`addr,`* `int` *`addrlen`*`)`
`int` **`xilsock_accept`** `(int` *`sd,`* `struct sockaddr*` *`addr,`* `int` *`addrlen`*`)`
`int` **`xilsock_recvfrom`** `(int` *`s,`* `unsigned char*` *`buf,`* `int` *`len`*`)`
`int` **`xilsock_sendto`** `(int` *`s,`* `unsigned char*` *`buf,`* `int` *`len`*`)`
`int` **`xilsock_recv`** `(int` *`s,`* `unsigned char*` *`buf,`* `int` *`len`*`)`
`int` **`xilsock_send`** `(int` *`s,`* `unsigned char*` *`buf,`* `int` *`len`*`)`
`void` **`xilsock_close`** `(int` *`s`*`)`
`void` **`xilnet_mac_init`** `(unsigned int` *`baseaddr`*`)`
`void` **`xilnet_eth_init_hw_addr`**`(unsigned char *`*`addr`*`)`

Table 26-1: **LibXilNet functions at a glance**

Functions
int **xilnet_eth_recv_frame** (unsigned char* *frame*, int *len*)
int **xilnet_eth_send_frame** (unsigned char* *frame*, int *len*, void* *daddr*, unsigned short *type*)
void **xilnet_eth_update_hw_tbl** (unsigned char* *frame*, int *proto*)
void **xilnet_eth_add_hw_tbl_entry** (unsigned char* *ip*, unsigned char* *hw*)
int **xilnet_eth_get_hw_addr** (unsigned char* *ip*)
int **xilnet_eth_init_hw_addr_tbl** (void)
int **xilnet_arp** (unsigned char* *buf*, int *len*)
void **xilnet_arp_reply** (unsigned char* *buf*, int *len*)
void **xilnet_ip_init** (unsigned char* *ip_addr*)
int **xilnet_ip** (unsigned char* *buf*, int *len*)
void **xilnet_ip_header** (unsigned char* *buf*, int *len*, int *proto*)
unsigned short **xilnet_ip_calc_chksum** (unsigned char* *buf*, int *len*, int *proto*)
int **xilnet_udp** (unsigned char* *buf*, int *len*)
void **xilnet_udp_header** (struct xilnet_udp_conn *conn*, unsigned char* *buf*, int *len*)
unsigned short **xilnet_tcp_udp_calc_chksum** (unsigned char* *buf*, int *len*, unsigned char* *saddr*, unsigned char* *daddr*, unsigned short *proto*)
void **xilnet_udp_init_conns** (void)
int **xilnet_udp_open_conn** (unsigned short *port*)
int **xilnet_udp_close_conn** (struct xilnet_udp_conn* *conn*)
int **xilnet_tcp** (unsigned char* *buf*, int *len*)
void **xilnet_tcp_header** (struct xilnet_tcp_conn *conn*, unsigned char* *buf*, int *len*)
void **xilnet_tcp_send_pkt** (struct xilnet_tcp_conn *conn*, unsigned char* *buf*, int *len*, unsigned char *flags*)
void **xilnet_tcp_init_conns** (void)
int **xilnet_tcp_open_conn** (unsigned short *port*)
int **xilnet_tcp_close_conn** (struct xilnet_tcp_conn* *conn*)
int **xilnet_icmp** (unsigned char* *buf*, int *len*)
void **xilnet_icmp_echo_reply** (usigned char* *buf*, int *len*)

Protocols Supported

LibXilNet supports drivers and functions for the Sockets API and protocols of TCP/IP stack. The following list enumerates them.

- Ethernet Encapsulation (RFC 894)
- Address Resolution Protocol (ARP - RFC 826)

- Internet Protocol (IP - RFC 791)
- Internet Control Management Protocol (ICMP - RFC 792)
- Transmission Control Protocol (TCP - RFC 793)
- User Datagram Protocol (UDP - RFC 768)
- Sockets API

Library Architecture

Figure 26-1 gives the architecture of libXilNet. Higher Level applications like HTTP server, TFTP (Trivial File Transfer Protocol), PING etc., uses API functions to use the libXilNet library.

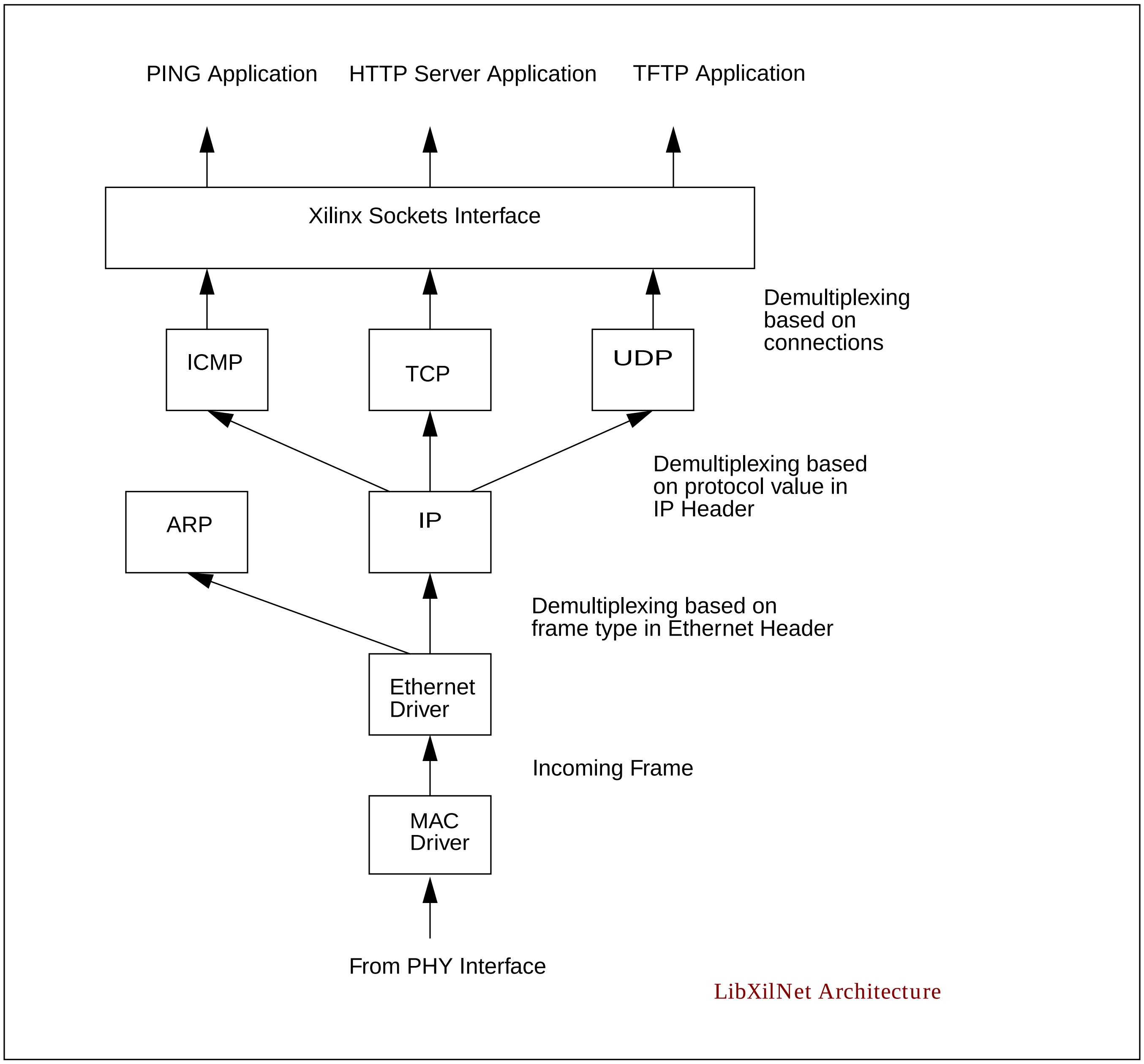


Figure 26-1: **Schematic Diagram of LibXilNet Architecture**

Protocol Function Description

A detailed description of the drivers and the protocols supported is given below.

Media Access Layer (MAC) Drivers Wrapper

MAC drivers wrapper initializes the base address of the mac instance specified by the user.This base address is used to send and receive frames. Ths initialization must be done before using other functionalites of LibXil Net library. The details of the function prototype is defined in the section"Functions of LibXilNet".

Ethernet Drivers

Ethernet drivers perform the encapsulation / removal of ethernet headers on the payload in accordance with the RFC 894. Based on the type of payload (IP or ARP), the drivers call the corresponding protocol callback function. A Hardware Address Table is maintained for mapping 48-bits ethernet address to 32-bits IP address.

ARP (RFC 826)

Functions are provided for handling ARP requests. An ARP request (for the 48-bit hardware address) is acknowledged with the 48-bit ethernet address in the ARP reply. Currently, ARP request generation for a desired IP address is not supported. The Hardware address table is updated with the new IP / Ethernet address pair if the ARP request is destined for the processor.

IP (RFC 791)

IPv4 datagrams are used by the higher level protocols like ICMP, TCP, and UDP for receiving / sending data. A callback function is provided for ethernet drivers which is invoked whenever there is an IP datagram as a payload in an ethernet frame. Minimal processing of the source IP address check is performed before the corresponding higher level protocol (ICMP, TCP, UDP) is called. Checksum is calculated on all the outgoing IP datagrams before calling the ethernet callback function for sending the data. An IP address for a Embedded Processor needs to be programmed before using it for communication. An IP address initializing function is provided. Refer to the table describing the various routines for further details on the function. Currently no IP fragmentation is performed on the outgoing datagrams. The Hardware address table is updated with the new IP / Ethernet address pair if an IP packet was destined for the processor.

ICMP (RFC 792)

ICMP functions handling only the echo requests (ping requests) are provided. Echo requests are issued as per the appropriate requirements of the RFC (Requests For Comments).

UDP (RFC 768)

UDP is a connectionless protocol. The UDP callback function, called from the IP layer, performs the minimal check of source port and strips off the UDP header. It demultiplexes from the various open UDP connections. A UDP connection can be opened with a given source port number through Socket functions. Checksum calculation is performed on the outgoing UDP datagram. The number of UDP connections that can be supported simultaneously is configurable.

TCP (RFC 793)

TCP is a connection-oriented protocol. Callback functions are provided for sending and receiving TCP packets. TCP maintains connections as a finite state machine. On receiving a TCP packet, minimal check of source port correctness is done, before demultiplexing the TCP packet from the various TCP connections. Necessary action for the demultiplexed connection is taken based on the current machine state. A status flag is returned to indicate the kind of TCP packet received to support connection management. Connection management has to be done at the application level using the status flag received from TCP. Checksum is calculated on all outgoing TCP packets. The number of TCP connections that can be supported simultaneously is configurable.

Sockets API

Functions for creating sockets (TCP / UDP), managing sockets, sending and receiving data on UDP and TCP sockets are provided. High level network applications need to use these functions for performing data communication. Refer to Table 26-1 for further details.

Current Restrictions

Certain restrictions apply to the EDK libXilNet library software. These are

- Only server functionalities for ARP - This means ARP requests are not being generated from the processor
- Only server functionalities in libXilNet - This means no client application development support provided in libXilNet.
- No timers in TCP - Since there are no timers used, every "send" over a TCP connection waits for an "ack" before performing the next "send".

Functions of LibXilNet

The following table gives the list of functions in libXilNet and their descriptions

int **xilsock_init** (void)

Parameters	None
Returns	1 for success and 0 for failure
Description	Initialize the xilinx internal sockets for use.
Includes	xilsock.h

`void` **`xilsock_rel_socket`** `(int` *`sd`*`)`

Parameters	*sd* is the socket to be released.
Returns	None
Description	Free the system level socket given by the socket descriptor *sd*
Includes	xilsock.h

`int` **`xilsock_socket`** `(int` *`domain`*`, int` *`type`*`, int` *`proto`*`)`

Parameters	*domain:* Socket Domain *type*: Socket Type *proto*: Protocol Family
Returns	On success, return socket descriptor On failure, return -1
Description	Create a socket of type, domain and protocol proto and returns the socket descriptor. The type of sockets can be: SOCK_STREAM (TCP socket) SOCK_DGRAM (UDP socket) *domain* value currently is AF_INET *proto* refers to the protocol family which is typically the same as the *domain*.
Includes	xilsock.h

`int` **`xilsock_bind`** `(int` *`sd`*`, struct sockaddr*` *`addr`*`, int` *`addrlen`*`)`

Parameters	*sd:* Socket descriptor *addr:* Pointer to socket structure *addrlen*: Size of the socket structure
Returns	On success, return 1 On failure, return -1
Description	Bind socket given the descriptor *sd* to the ip address / port number pair given in structure pointed to by *addr* of len *addrlen*. *addr* is the typical socket structure.
Includes	xilsock.h

`int xilsock_accept (int sd, struct sockaddr* addr, int *addrlen)`

Parameters	*sd:* Socket descriptor *addr:* Pointer to socket structure *addrlen*: Pointer to the size of the socket structure
Returns	On success, return socket descriptor On failure, return -1
Description	Accepts new connections on socket *sd*. If a new connection request arrives, it creates a new socket *nsd*, copies properties of *sd* to *nsd*, returns *nsd*. If a packet arrives for an existing connection, returns 0 and sets the xilsock_status_flag global variable. The various values of the is flag are: XILSOCK_NEW_CONN XILSOCK_CLOSE_CONN XILSOCK_TCP_ACK for new connection, closed a connection and acknowledgment for data sent for a connection correspondingly. This function implicitly polls / waits on a packet from MAC. Arguments *addr* and *addrlen* are in place to support the standard Socket accept function signature. At present, they are not used in the accept function.
Includes	xilsock.h

`int xilsock_recvfrom (int s, unsigned char* buf, int len)`

Parameters	*s:* UDP socket descriptor *buf*: Buffer to receive data *len*: Buffer size
Returns	Number of bytes received
Description	Receives data (maximum length of *len*) from the UDP socket *s* in *buf* and returns the number of bytes received.
Includes	xilsock.h

`int xilsock_sendto (int s, unsigned char* buf, int len)`

Parameters	*s:* UDP socket descriptor *buf*: Buffer containing data to be sent *len*: Buffer size
Returns	Number of bytes received
Description	Sends data of length *len* in *buf* on the UDP socket *s* and returns the number of bytes sent.
Includes	xilsock.h

`int` **`xilsock_recv`** `(int` *`s`*`, unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*s:* TCP socket descriptor *buf*: Buffer to receive data *len*: Buffer size
Returns	Number of bytes received
Description	Receives data (maximum length of *len*) from the TCP socket *s* in *buf* and returns the number of bytes received.
Includes	xilsock.h

`int` **`xilsock_send`** `(int` *`s`*`, unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*s:* TCP socket descriptor *buf*: Buffer containing data to be sent *len*: Buffer size
Returns	Number of bytes received
Description	Sends data of length *len* in *buf* on the UDP socket *s* and returns the number of bytes sent.
Includes	xilsock.h

`void` **`xilsock_close`** `(int` *`s`*`)`

Parameters	*s:* socket descriptor
Returns	None
Description	Closes the socket connection given by the descriptor *s*. This function has to be called from the application for a smooth termination of the connection after a connection is done with the communication.
Includes	xilsock.h

`void` **`xilnet_mac_init`** `(unsigned int` *`baseaddr`*`)`

Parameters	*baseaddr*: Base address of the MAC instance used in a system
Returns	None
Description	Initialize the MAC base address used in the libXil Net library to *baseaddr*. This function has to be called at the start of a user program with the base address used in the MHS file for ethernet before starting to use other functions of libXil Net library.
Includes	mac.h

`void` **`xilnet_eth_init_hw_addr`** `(unsigned char*` *`addr`*`)`

Parameters	*addr*: 48-bit colon separated hexa decimal ethernet address string
Returns	None
Description	Initialize the source ethernet address used in the libXil Net library to *addr*. This function has to be called at the start of a user program with a 48-bit, colon separated, hexa decimal ethernet address string for source ethernet address before starting to use other functions of libXil Net library. This address will be used as the source ethernet address in all the ethernet frames.
Includes	xilsock.h mac.h

`int` **`xilnet_eth_recv_frame`** `(unsigned char*` *`frame`*`, int` *`len`*`)`

Parameters	*frame*: Buffer for receiving an ethernet frame *len*: Buffer size
Returns	Number of bytes received
Description	Receives an ethernet frame from the MAC, strips the ethernet header and calls either *ip* or *arp* callback function based on frame type. This function is called from *accept /receive* socket functions. The function receives a frame of maximum length *len* in buffer *frame*.
Includes	xilsock.h mac.h

`void` **`xilnet_eth_send_frame`** `(unsigned char*` *`frame`*`, int` *`len`*`, unsigned char*` *`dipaddr`*`, void *`*`dhaddr`*`, unsigned short` *`type`*`)`

Parameters	*frame*: Buffer for sending a ethernet frame *len*: Buffer size *dipaddr*: Pointer to the destination ip address *dhaddr*: Pointer to the destination ethernet address *type*: Ethernet Frame type (IP or ARP)
Returns	None
Description	Creates an ethernet header for payload *frame* of length *len*, with destination ethernet address *dhaddr*, and frame type, *type*. Sends the ethernet frame to the MAC. This function is called from *receive/send* (both versions) socket functions.
Includes	xilsock.h mac.h

void **xilnet_eth_update_hw_tbl** (unsigned char* *frame*, int *proto*)

Parameters	*frame*: Buffer containing an ethernet frame *proto*: Ethernet Frame type (IP or ARP)
Returns	None
Description	Updates the hardware address table with ipaddress/hardware address pair from the ethernet frame pointed to by *frame*. *proto* is used in identifying the frame (ip/arp) to get the ip address from the ip/arp packet.,
Includes	xilsock.h mac.h

void **xilnet_eth_add_hw_tbl_entry** (unsigned char* *ip*, unsigned char* *hw*)

Parameters	*ip*: Buffer contains ip address *hw*: Buffer containing hardware address
Returns	None
Description	Add an ip / hardware pair entry given by *ip/hw* into the hardware address table
Includes	xilsock.h mac.h

int **xilnet_eth_get_hw_addr** (unsigned char* *ip*)

Parameters	*ip*: Buffer containing ip address
Returns	Index of entry in the hardware address table that matches the *ip* address
Description	Receives an ethernet frame from the MAC, strips the ethernet header and calls either *ip* or *arp* callback function based on the frame type. This function is called from *accept /receive* socket functions. The function receives a frame of maximum length *len* in buffer *frame*.
Includes	xilsock.h mac.h

`void` **`xilnet_eth_init_hw_addr_tbl`** `(void)`

Parameters	None
Returns	None
Description	Initializes Hardware Address Table. This function must be called in the user program before using other functions of LibXilNet.
Includes	xilsock.h mac.h

`int` **`xilnet_arp`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf*: Buffer for holding the ARP packet *len*: Buffer size
Returns	0
Description	This is the *arp* callback function. It gets called by the ethernet driver for *arp* frame type. The *arp* packet is copied onto the *buf* of length *len*.
Includes	xilsock.h

`void` **`xilnet_arp_reply`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf*: Buffer containing the ARP reply packet *len*: Buffer size
Returns	None
Description	This function sends the *arp* reply, present in *buf* of length *len*, for *arp* requests. It gets called from the *arp* callback function for *arp* requests.
Includes	xilsock.h

`void` **`xilnet_ip_init`** `(unsigned char*` *`ip_addr`*`)`

Parameters	*ip_addr*: Array of four bytes holding the ip address to be configured
Returns	None
Description	This function initializes the ip address for the processor to the address represented in *ip_addr* as a dotted decimal string. This function must be called in the application before any communication.
Includes	xilsock.h

`int` **`xilnet_ip`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf*: Buffer for holding the IP packet *len*: Buffer size
Returns	0
Description	This is the ip callback function. It gets called by the ethernet driver for ip frame type. The *ip* packet is copied onto the *buf* of length *len*. This function calls in the appropriate protocol callback function based on the protocol type.
Includes	xilsock.h

`void` **`xilnet_ip_header`** `(unsigned char*` *`buf`*`, int` *`len`*`, int` *`proto`*`)`

Parameters	*buf*: Buffer for the ip packet *len*: Length of the ip packet *proto*: Protocol Type in IP packet
Returns	None
Description	This function fills in the ip header from the start of *buf*. The ip packet is of length *len* and *proto* is used to fill in the protocol field of ip header. This function is called from the *receive/send* (both versions) functions.
Includes	xilsock.h

`unsigned short` **`xilnet_ip_calc_chksum`** `(unsigned char*` *`buf`*`, int` *`len`*`, int` *`proto`*`)`

Parameters	*buf*: Buffer containing ip packet *len*: Length of the ip packet
Returns	checksum calculated for the given ip packet
Description	This function calculates the checksum for the ip packet *buf* of length *len*. This function is called from the ip header creation function.
Includes	xilsock.h

`int` **`xilnet_udp`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf*: Buffer containing the UDP packet *len*: Length of the UDP packet
Returns	Length of the data if packet is destined for any open UDP connections else returns 0
Description	This is the *udp* callback function which is called when ip receives a udp packet. This function checks for a valid udp port, strips the udp header, and demultiplexes from the various UDP connections to select the right connection.
Includes	xilsock.h

`void` **`xilnet_udp_header`** `(struct xilnet_udp_conn` *`conn`*`, unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*conn:* UDP connection *buf*: Buffer containing udp packet *len*: Length of udp packet
Description	This function fills in the *udp* header from the start of *buf* for the UDP connection *conn*. The udp packet is of length *len*. This function is called from the *receivefrom/sendto* socket functions.
Includes	xilsock.h

`unsigned short` **`xilnet_udp_tcp_calc_chksum`** `(unsigned char*` *`buf`*`, int` *`len`*`, unsigned char*` *`saddr`*`, unsigned char*` *`daddr`*`, unsigned short` *`proto`*`)`

Parameters	*buf*: Buffer containing UDP / TCP packet *len*: Length of udp / tcp packet *saddr*: IP address of the source *daddr*: Destination IP address *proto*: Protocol Type (UDP or TCP) Returns the
Returns	Checksum calculated for the given udp / tcp packet
Description	This function calculates and fills the *checksum* for the *udp/tcp* packet *buf* of length *len*. The source ip address (*saddr*), destination ip address(*daddr*) and protocol (*proto*) are used in the checksum calculation for creating the pseudo header. This function is called from either the udp header or the tcp header creation function.
Includes	xilsock.h

`void` **`xilnet_udp_init_conns`** `(void )`

Parameters	None
Returns	None
Description	Initialize all UDP connections so that the states of all the connections specify that they are usable.
Includes	xilsock.h

`int` **`xilnet_udp_open_conn`** `(unsigned short` *`port`*`)`

Parameters	*port:* UDP port number
Returns	Connection index if able to open a connection. If not returns -1.
Description	Open a UDP connection with port number *port*.
Includes	xilsock.h

`int` **`xilnet_udp_close_conn`** `(struct xilnet_udp_conn *`*`conn`*`)`

Parameters	*conn*: UDP connection
Returns	1 if able to close else returns -1.
Description	Close a UDP connection *conn*.
Includes	xilsock.h

`int` **`xilnet_tcp`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf*: Buffer containing the TCP packet *len*: Length of the TCP packet
Returns	A status flag based on the state of the connection for which the packet has been received
Description	This is the *tcp* callback function which is called when ip receives a tcp packet. This function checks for a valid tcp port and strips the tcp header. It maintains a finite state machine for all TCP connections. It demultiplexes from existing TCP open / listening connections and performs an action corresponding to the state of the connection. It returns a status flag which identifies the type of TCP packet received (data or ack or fin).
Includes	xilsock.h

void **xilnet_tcp_header** (struct xilnet_tcp_conn *conn*, unsigned char* *buf*, int *len*)

Parameters	*conn:* TCP connection *buf*: Buffer containing tcp packet *len*: Length of tcp packet
Returns	None
Description	This function fills in the *tcp* header from the start of *buf* for the TCP connection *conn*. The tcp packet is of length *len*. It sets the flags in the tcp header.
Includes	xilsock.h

void **xilnet_tcp_send_pkt** (struct xilnet_tcp_conn *conn*, unsigned char* *buf*, int *len*, unsigned char *flags*)

Parameters	*conn:* TCP connection *buf*: Buffer containing TCP packet *len*: Length of tcp packet
Returns	The checksum calculated for the given udp / tcp packet
Description	This function sends a tcp packet, given by *buf* of length *len*, with *flags* (ack / rst / fin / urg / psh) from connection *conn*.
Includes	xilsock.h

void **xilnet_tcp_init_conns** (void)

Parameters	None
Returns	None
Description	Initialize all TCP connections so that the states of all the connections specify that they are usable.
Includes	xilsock.h

int **xilnet_tcp_open_conn** (unsigned short *port*)

Parameters	*port:* TCP port number
Returns	Connection index if able to open a connection. If not returns -1.
Description	Open a TCP connection with port number *port*.
Includes	xilsock.h

`int` **`xilnet_tcp_close_conn`** `(struct xilnet_tcp_conn *`*`conn`*`)`

Parameters	*conn*: TCP connection
Returns	1 if able to close else returns -1.
Description	Close a TCP connection *conn*.
Includes	xilsock.h

`int` **`xilnet_icmp`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf:* Buffer containing ICMP packet *len:* Length of the ICMP packet
Returns	*0*
Description	This is the icmp callback function which is called when ip receives a icmp echo request packet (ping request). This function checks only for a echo request and sends in an icmp echo reply.
Includes	xilsock.h

`void` **`xilnet_icmp_echo_reply`** `(unsigned char*` *`buf`*`, int` *`len`*`)`

Parameters	*buf:* Buffer containing ICMP echo reply packet *len:* Length of the ICMP echo reply packet
Returns	None
Description	This functions fills in the icmp header from the start of buf. The icmp packet is of length *len*. It sends the icmp echo reply by calling the ip, ethernet send functions. This function is called from the icmp callback function.
Includes	xilsock.h

LibGen Customization

XilNet library is customized through LibGen tool. Here is a snippet from system.mss file for specifying LibXilNet.

```
BEGIN LIBRARY
PARAMETER LIBRARY_NAME = xilnet
PARAMETER LIBRARY_VER = 1.00.a
PARAMETER device_type = ethernet
PARAMETER emac_type = emac
END
```

LibXilNet has the infrastructure to support different networking devices. The parameter **device_type** can take **- ethernet, serial, host** as values. Currently however ethernet is the device type that has been tested.

With **ethernet** device type, LibXilNet can be used with either the regular ethernet core or the lite version, ethernetlite.

The parameter **emac_type** can take **- emac, emaclite** as values. This configures XilNet to be used with either Emac of EmacLite driver.

Table 26-2: **Configurable Parameters for XilNet in MSS**

Parameter	Description
device_type	Networking Device type used in the system Possible values are **ethernet, serial, host**.
emac_type	Type of ethernet core used. Possible values are **emac, emaclite**

Using XilNet in Application

Libgen generates a configuration file xilnet_config.h based on the parameter selection in MSS file.

In order to use the XilNet functions in your application, you need to do the following:

- Define "`#include <net/xilnet_config.h>`" in your C-file.
- Define "`#include <net/xilsock.h>`" in your C-file.

LibXil Kernel

Summary

This chapter describes the kernel for Embedded processors, libXil Kernel.

Overview

LibXilKernel has the key features of RTOS like multi-tasking, priority-driven preemptive scheduling, support for Inter-Process communication and synchronization. It is small, modular, user customizable and can be used in any system configuration. It also has system call interface, which allows a system to be built in different configurations. The user applications can be part of a single kernel executable or be separate executables.

Features

LibXilKernel supports the following features:

- Process Management
- Thread Management
- Interrupt Handling
- System Call Interface
- Semaphore
- Message Queue
- Shared Memory
- Dynamic Buffer Allocation

LibXilKernel Blocks

The kernel is highly modular. The user can select and customize the kernel modules that are needed for the application.The customizing of the kernel is discussed in "LibGen Customization" section in detail. Figure 27-1 shows the various modules of the Xilinx embedded kernel.

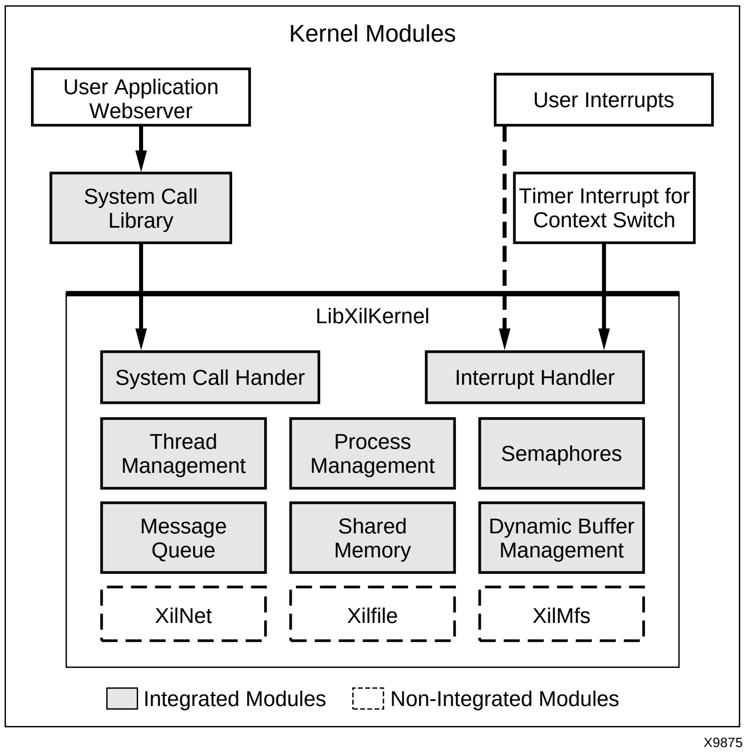


Figure 27-1: **Kernel Modules**

Process Management

The kernel supports multi-processing and has two different scheduling schemes. A process (thread) is an unit of scheduling in the kernel. Each process is associated with a Process Control Block (PCB), that contains information about the process. A process is created and handled using the APIs. Each process is in any of the following four states.

- PROC_NEW
- PROC_READY
- PROC_RUN
- PROC_WAIT
- PROC_DEAD

Figure 27-2 shows the process state flow in the system.

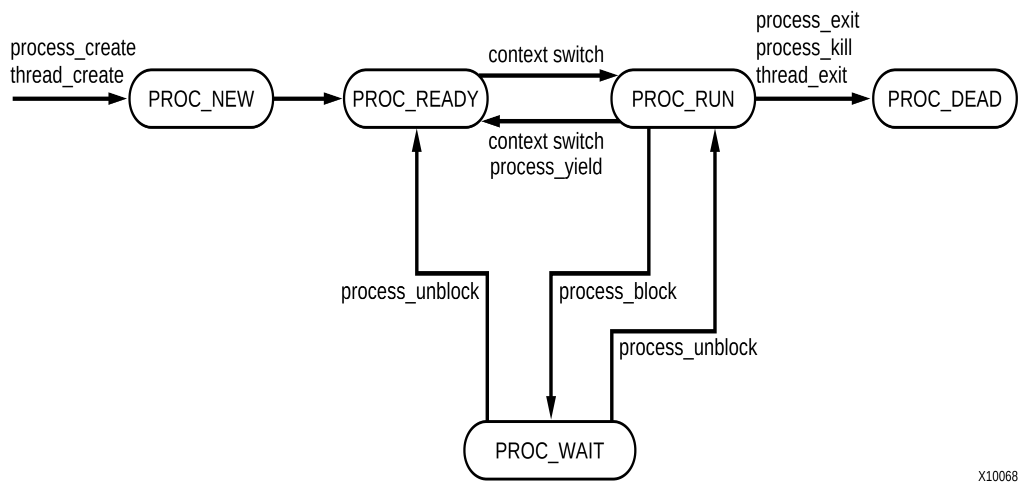


Figure 27-2: **Process State Flow**

The kernel supports the following two scheduling scheme.

- Round Robin scheduling (SCHED_RR)
- Pre-emptive Priority scheduling (SCHED_PRIO)

The scheduling scheme is selected during system initialization and cannot be changed dynamically.

Functions of Process Management

The following functions relate to process management. Most of the functions are optional and can be selected during system initialization. Refer "Customizing Process Management" section for more details.

void **sys_init**(void)

Parameters	None
Returns	None
Description	Initialize the system. This is called at the start of the system. • Initialize the Process Vector Table • Create an idle task (PID - 0) • Create the initial set of processes
Includes	sys/process.h

int **process_create**(unsigned int *start_addr*, int *priority*)

Parameters	*start_addr* is the start address of the process *priority* is the priority of the process in the system. The priority cannot be changed when the process is active
Returns	On success, return the PID of the new process On failure, return -1
Description	Create a new process. Allocate a new PID and Process Control Block (PCB) for the process.The process is placed in the Ready Queue.
Includes	sys/process.h

int **process_exit**(void)

Parameters	None
Returns	None
Description	Remove the process from the system. This function is optional.
Includes	sys/process.h

int **process_kill**(char *pid*)

Parameters	*pid* is the PID of process to kill
Returns	On success, return 0 On failure, return -1
Description	Remove or kill the process with process ID, *pid*. This function should be used with care, as any process can kill other process. This function is optional.
Includes	sys/process.h

int **process_status**(int *pid*, p_stat **ps*)

Parameters	*pid* is the PID of process *ps* is the buffer where the process status is returned
Returns	On success, return process status in *ps* On failure, return NULL in *ps*
Description	Get the process status. The status is returned in structure p_stat which has the following fields: ♦ pid is the process ID ♦ state is the current state of the process
Includes	sys/process.h

int **process_yield**(void)

Parameters	None
Returns	None
Description	Yield the processor to the next process.The current process goes to PROC_READY state. This function is optional.
Includes	sys/process.h

int **process_getpriority**(void)

Parameters	None
Returns	Priority of the current process or thread
Description	Get the priority of process or thread.
Includes	sys/process.h

int **process_setpriority**(int *priority*)

Parameters	*priority* is the new priority of process or thread
Returns	On success, return 0 On failure, return -1
Description	Set the priority of current process or thread to new value.
Includes	sys/process.h

int **get_currentPID**(void)

Parameters	None
Returns	Process ID of current process
Description	Get the Process ID of current process.
Includes	sys/process.h

Thread Management

Threads are light weight processes.They share the same code segment with other threads but have their own thread context, which is allocated when the threads are created. A thread is handled in the same way as a process.

Functions of Thread Management

The following functions relate to thread management. The thread module is optional and can be selected during system initialization. Refer "The user can select the maximum number of processes in the system, the different functions needed to handle processes." section for more details.

int **thread_create**(void **funcp*, unsigned int *arg*, int *priority*)

Parameters	*funcp* is the start address of the function from which the thread starts to execute *arg* is the argument to the thread function *priority* is the priority of the thread
Returns	On success, return the thread ID (PID) of the new thread On failure, return -1
Description	Create a new thread. The thread starts its execution from the start function. This function is optional.
Includes	sys/process.h

int **thread_exit**(void)

Parameters	None
Returns	None
Description	Remove the current thread from the system This function is optional.
Includes	sys/process.h

Interrupt Handling

The interrupt handler can be specified in the MSS file. Libgen generates the interrupt controller routine for handling interrupts. The kernel only supports timer interrupt. This interrupt is used as a timer tick to perform context switching between processes. The timer interrupt is initialized and started during system start. The timer tick interval can be customized by the user based on the application. Refer "LibGen Customization" section for more details.

System call interface

The system can be built in two different configuration.

The user application can be built as part of the kernel; as a single application. Threads can be used to support concurrent processing. In this case the kernels system call's can be directly accessed by the user application. Each system call name is prefixed by *sys_* when called directly. This configuration can be used if the system has only a single application running.

If the system has multiple application's running; then each application can be built as a separate process. The kernel is built as a separate central process in this configuration. The application can access the kernel services through the system call interface. The application should be linked to **libw.a** library, which has the system call wrappers. The kernel services can be configured during system initialization. Refer "LibGen Customization" section for more details.

Semaphore

Semaphore is used for Inter-Process Communication and Synchronization. A semaphore can be used as a binary or integer semaphore.The number of semaphores and the length of semaphore wait queue can be configured during system initialization. Refer "Functions of Semaphore" section for more details.

The semaphore structure is declared in sys/sema.h. It contains the following fields.

- sema_id - semaphore ID
- count - available resource count
- wait_q - queue of processes waiting for the resource

Functions of Semaphore

The following functions relate to semaphores. The semaphore module is optional and can be configured during system initialization.

Note: Message Queue module uses semaphores, so this needs to be included if message queue is to be used.

int **sema_init**(semaphore ***sema*, char *count*)

Parameters	*sema* is the semaphore structure which is returned when a new semaphore is created *count* is the resource count for the semaphore
Returns	On success, *sema* is assigned a new semaphore and 0 is returned On failure, return -1
Description	Initialize and create the semaphore.
Includes	sys/sema.h

int **sema_wait**(semaphore **sema*)

Parameters	*sema* is the semaphore structure returned by calling sema_init
Returns	On success, return 0 On failure, return -1
Description	Get the semaphore resource. If the resource is available then get the resource else **block** the process.
Includes	sys/sema.h

int **sema_trywait**(semaphore *sema*)

Parameters	*sema* is the semaphore structure returned by calling sema_init
Returns	On success, return 0 On failure, return -1
Description	Try to get the semaphore resource. If the resource is available then get the resource else return error. This is a **non-blocking** function.
Includes	sys/sema.h

int **sema_post**(semaphore *sema*)

Parameters	*sema* is the semaphore structure returned by calling sema_init
Returns	On success, return 0 On failure, return -1
Description	Free the semaphore resource or signal the availability of semaphore resource. If any process is waiting on this resource, then **unblock** the process.
Includes	sys/sema.h

int **sema_destroy**(semaphore *sema*)

Parameters	*sema* is the semaphore structure returned by calling sema_init
Returns	On success, return 0 On failure, return -1
Description	Release the semaphore.
Includes	sys/sema.h

Message Queue

Message Queue is used for Inter-Process Communication. The message queue size and number can be configured during system initialization. Refer "Customizing Message Queue" section for more details. Message queue internally uses semaphores, so semaphore module should be included to use message queue.

The message queue structure **struct msgid_ds** has the following fields.

- *msgid* - the message queue ID.
- *key* - key used to identify the message queue.
- *msgsize* - the message size in the queue.
- *maxmsg* - message queue maximum length.

Functions of Message Queue

The following functions relate to message queue. Message queue module is optional and can be included when the system is built.

int **msgget**(int *key*, int *msgsize*, int *maxmsg*, int *flag*)

Parameters	*key* is used to uniquely identify the Message Queue *msgsize* is the size of the message *maxmsg* is the maximum number of messages in the queue *flag* is used to identify IPC options
Returns	On success, return unique message queue ID On failure, return -1
Description	Create a new message queue, if none with the given key exists. If **flag = IPC_CREAT**, then return existing message queue ID for the given key If **flag = IPC_EXCL**, then return -1 if message queue for the key exists.
Includes	sys/msg.h sys/ipc.h

int **msgctl**(int *msgid*, int *cmd*, struct msgid_ds **buf*)

Parameters	*msgid* is the message queue ID got from msgget *cmd* is the command to the control function *buf* is the buffer where the status is returned
Returns	On success, return 0. Status is returned in *buf* for IPC_STAT On failure, return -1
Description	Control the message queue. If **cmd = IPC_STAT**, the return the message queue status in buf If **cmd = IPC_RMID**, then remove the message queue
Includes	sys/msg.h sys/ipc.h

int **msgsend**(int *msgid*, const void **msg*, int *nbytes*, int *flag*)

Parameters	*msgid* is the message queue ID got from msgget
	msg is the message to send
	nbytes is the size of the message
	flag is used to specify IPC options
Returns	On success, return 0
	On failure, return -1
Description	Send the message, if space is available on the message queue.
	If queue is full, then wait for queue space.This is a **blocking** function.
	If **flag = IPC_NOWAIT** and queue is full, then return error.
	Note: nbytes is not used. The message size specified during msgget is used for a message.
Includes	sys/msg.h
	sys/ipc.h

int **msgrecv**(int *msgid*, void **msg*, int *nbytes*, int *type*, int *flag*)

Parameters	*msgid* is the message queue ID got from msgget
	msg is the buffer where the message is received
	nbytes is the size of the message
	type is used to specify receiving options
	flag is used to specify IPC options
Returns	On success, return 0
	On failure, return -1
Description	Receive the message in the message queue. The message is received in a FIFO fashion. If queue is empty, then wait for message in queue. **If flag = IPC_NOWAIT** and queue is empty, then return error.
	Note:
	nbytes is not used. The message size specified during msgget is used for a message.
	type is not used.
Includes	sys/msg.h
	sys/ipc.h

Shared Memory

Shared memory is used for Inter-Process Communication. The number of shared memory and its size can be configured. Refer "Customizing Shared Memory" section for more details.

The shared memory structure **struct shmid_ds** has the following fields.

- *shmid* - shared memory ID.
- *key* - key to identify the shared memory segment.
- *size* - the size of the shared memory segment.
- *nattach* - number of processes currently attached to the shared memory.

Functions of Shared Memory

The following functions relate to shared memory. Shared memory module is optional and can be included when the system is built.

int **shmget**(int *key*, int *size*, int *flag*)

Parameters	*key* is used to uniquely identify the shared memory *size* is the size of the shared memory segment *flag* is used to specify IPC options
Returns	On success, return unique shared memory ID On failure, return -1
Description	Create a new shared memory segment, if none with the given key exists. If **flag = IPC_CREAT**, then return existing shared memory ID for the given key If **flag = IPC_EXCL**, then return -1 if shared memory for the key exists.
Includes	sys/shm.h sys/ipc.h

int **shmctl**(int *shmid*, int *cmd*, struct shmid_ds **buf*)

Parameters	*shmid* is the shared memory got from shmget *cmd* is the command to the control function *buf* is the buffer where the status is returned
Returns	On success, return 0. Status is returned in buf for IPC_STAT On failure, return -1
Description	Control the shared memory. If **cmd = IPC_STAT**, the return the shared memory status in buf If **cmd = IPC_RMID**, then remove the shared memory
Includes	sys/shm.h sys/ipc.h

void ***shmat***(int *shmid*, void **addr*, int *flag*)

Parameters	*shmid* is the shared memory got from shmget *addr* is used to specify the location, to attach shared memory segment *flag* is used to specify IPC options
Returns	On success, return the start address of the shared memory segment On failure, return NULL
Description	Returns the shared memory segment for shmid. Note: addr and flag arguments are not used.
Includes	sys/shm.h sys/ipc.h

int **shmdt**(void **addr*)

Parameters	*addr* is the shared memory address got from shmat
Returns	On success, return 0 On failure, return -1
Description	Detach the shared memory segment. The memory segment is not removed from the system and can be attached later.
Includes	sys/shm.h sys/ipc.h

Dynamic Buffer Management

The kernel provides a simple buffer management scheme, which can be used by applications that need dynamic memory allocation. The application can use the standard 'c' memory allocation routines.

The user can select different memory blocks sizes and number of such memory blocks required for the application. The memory blocks and the total memory needed by the system is allocated statically and can be configured by the user. Refer , "Customizing Dynamic Buffer Management" section for more details.

This method of buffer management provides user the flexibility of using dynamic memory allocation functions. And also a simple, small and fast way of allocating memory.

Functions of Dynamic Buffer Management

The following functions relate to buffer allocation. This module is optional and can be included during system initialization.

void ***bufmalloc**(unsigned int *size*)

Parameters	*size* is the size of memory to allocate
Returns	On success, return the start address of memory block On failure, return NULL
Description	Allocate memory to the user process
Includes	sys/mem.h

void **buffree**(void **mem*)

Parameters	*mem* is the address of the memory block got from bufmalloc
Returns	*None*
Description	Free the memory allocated by bufmalloc.
Includes	sys/mem.h

LibXilKernel modes

LibXilKernel and user applications can be built in three different modes:

- **Single Executable mode**

In this mode, the user applications and kernel are built as single executable. The entry point for an application is a function that is specified in **parameter single_elf_process_table**, in libgen configuration. Building xilkernel using libgen generates *libxilkernel.a* library, which is linked with user applications to generate the single executable.

Since all user applications are part of a single executable, the global variables and function names should be unique. All user applications can be debugged using GDB. Refer , "Debugging XilKernel" section for more details.

- **Multiple Executable mode**

In this mode, the user applications and kernel are built as separate multiple executables. The entry point for an application is its start address. This can be specified in **parameter multi_elf_process_table**, in libgen configuration, for startup applications or loaded dynamically using shell utility. Building xilkernel using libgen generates xilkernel.elf executable and *libxilkernel.a* library. The user application can be linked with the library separately.

This mode provides the flexibility of dynamically loading application separately. Each user application can debugged using GDB. Refer , "Debugging XilKernel" section for more details

- **Mixed Executable mode**

In this mode, some user applications are built as single executable and some as separate multiple executables. This mode offers the flexibility of both modes. Building xilkernel using libgen generates *libxilkernel.a* library. The library is linked with user applications to generate single kernel executable and separate user executable files.

LibGen Customization

LibXilKernel is highly customizable. Most of the modules and individual parameters can be changed to suit the user application. In order to customize the various modules in kernel, a parameter with the name of the category set to true must be defined in the MSS. The MSS snippet for including the library LibXil Kernel is given below:

```
begin library
   parameter library_name = xilkernel
   parameter library_ver = 1.00.a
   parameter max_procs = 2
   parameter config_sema = true
   parameter config_msgq = true
   parameter config_thread_support = true
  parameter multi_elf_process_table = ((0xffffe000, 1), (0x0000400, 3))
  parameter single_elf_process_table = ((user_main, 4, 600))
  parameter num_msgq = 2
   parameter msgq_table = ( (10, 10), (15, 155) )
  parameter linker_script_specification = true
  parameter vec_mem_start = 0xffff0000
  parameter vec_mem_size = 9k
  parameter data_mem_start = 0xffff7000
  parameter data_mem_size = 6k
  parameter code_mem_start = 0xffffc000
  parameter code_mem_size = 12k
  parameter copyoutfiles = true
  parameter copyfromdir = .
  parameter copytodir = /usr/kernelsrc
end
```

Customizing Process Management

The user can select the maximum number of processes in the system, the different functions needed to handle processes.

Table 27-1: **Attributes for including Process Management parameters**

Attribute	Description
config_process	Need process management
max_procs	maximum number of processes in the system
max_readyq	maximum size of Ready queue for each priority
config_process_exit	Include process_exit function
config_process_kill	Include process_kill() function
config_process_yield	Include process_yield() function

Table 27-1: **Attributes for including Process Management parameters**

Attribute	Description
multi_elf_process_table	Configure Separate Executable applications Process Table. This is defined to be an array with each element containing the parameters process_start and process_prio. Note: Process_table parameter has been deprecated.
single_elf_process_table	Configure Single Executable applications Process Table. This is defined to be an array with each element containing the parameters process_start, process_prio and process_stack_size.
process_start	Process start address
process_prio	Process priority

Customizing Scheduler Module

The user can configure the type of scheduler used and the number of priorities used for the Scheduler module. The configurable parameters are given below.

Table 27-2: **Attributes for Including Schedule Module**

Attribute	Description
config_sched	Need scheduler module
sched_type	Type of Scheduler to be used
no_prio	Number of priorities

Customizing Thread Management

The user can optionally select to include thread support, the maximum number of threads and size of the thread context. The configurable parameters are given below.

Table 27-3: **Attributes for Including Thread Module**

Attribute	Description
config_thread_support	Need thread module
max_threads	Maximum number of threads
thread_stack_size	Thread stack size (in bytes)

Customizing Semaphore

The user can optionally select to include semaphores, maximum number of semaphores and semaphore queue length. The following are the parameters used for configuration.

Table 27-4: **Attributes for Including Semaphore module**

Attribute	Description
config_sema	Need Semaphore module
max_sema	Maximum number of Semaphores
max_sema_ waitq	Semaphore Wait Queue Length

Customizing Message Queue

The user can optionally select to include message queue module, number of message queue and size of each message queue. In order to include the message queue module, the semaphore module must be selected using the config_sema parameter in the MSS. The following parameters definitions are used for configuration.

Table 27-5: **Attributes for Including Message Queue module**

Attribute	Description
config_msgq	Need Message Queue
max_msgsize	Maximum message size
max_msgq_size	Maximum message queue length
num_msgq	Number of message queue
msgq_table	Message Queue Table. This is defined as an array with each element having parameters msg_size and msgq_len
msg_size	Message size
msgq_len	Message Queue length

Customizing Shared Memory

The user can optionally select to include shared memory and size of each shared memory. The following parameter are used for configuration.

Table 27-6: **Attributes for including Shared Memory module**

Attribute	Description
config_shm	Need Shared Memory module
shm_table	Shared Memory table. This is defined as an array with each element having shm_size parameter
shm_size	Shared Memory size
num_shm	Number of Shared Memories.

Customizing Dynamic Buffer Management

The user can optionally select to include dynamic buffer management module, size of memory blocks and number of memory blocks. The following parameters are used for configuration.

Table 27-7: **Attributes for including Memory management module**

Attribute	Description
config_mem	Need Memory Management
num_mem	Number of memory blocks
mem_table	Memory block table. This is defined as an array with each element having mem_bsize, mem_nblks parameters.
mem_bsize	Memory block size.
mem_nblks	Number of memory blocks

Customizing Linker Script (PPC405)

The user can customize the default linker script (linker_include.sh and linker_script.sh) files for building PowerPC kernel. User can specify the start address and size of three program sections - vector table, code section and data section. The following parameters are used for configuration.

Table 27-8: **Attributes for Linker Script Configuration in PPC405**

Attribute	Description
linker_script_specification	Need Custom Linker Script specification
vect_mem_start	Start address of Vector Table Program section, The default value is 0x0
vect_mem_size	Size of Vector table section. The default value is 9K
code_mem_start	Start address of Code section. The default value is 0xffffd000

Attribute	Description
code_mem_size	Size of Code section. The default size is 12K
data_mem_start	Start address of Data section. The default value is 0x2400
data_mem_size	Size of Data section. The default size is 9K

Copy Kernel Source Files

The user can copy the configured kernel source files to his repository for further editing and using them for building the kernel. The following parameters are used.

Table 27-9: **Attributes for Copying Kernel Source files**

Attribute	Description
copyoutfiles	Need to Copy source files
copyfromdir	The Kernel source directory path. The path is relative to <project_direc>/<systemname>/libsrc/xilkernel_v1_00_a/src directory
copytodir	User repository directory. The path is relative to <project_direc>/<systemname>/libsrc/xilkernel_v1_00_a/src directory

Debugging XilKernel

The user application can be debugged using GDB and XMD. The following steps should be followed.

- Download the user applications (multiple executable mode) and kernel using XMD.
- Run the kernel from XMD.
- Start gdb and open the application to debug. In Single Executable mode, open the kernel executable for debugging. This mode allows debugging of multiple applications at the same time.
- Connect to XMD target. Insert Breakpoints in the code and Continue gdb debugging section.

Memory footprint

The size of libxilkernel depends on the user configuration. It is small in size and can fit in different configurations. The following is the memory size output from GNU size utility for the kernel.

- Basic kernel functionality with multi-tasking **- ~3k**
- Full kernel functionality with all modules included **- ~8k**

Chapter 28

Device Drivers

Summary

This chapter describes device drivers present in the EDK.

Overview

The purpose of this chapter is to describe the Xilinx device driver environment. This includes the device driver architecture, the Application Programmer Interface (API) conventions, the scheme for configuring the drivers to work with reconfigurable hardware devices, and the infrastructure that is common to all device drivers.

This document is intended for the software engineer that is using the Xilinx device drivers. It contains design and implementation details necessary for using the drivers.

Goals and Objectives

The Xilinx device drivers are designed to meet the following goals and objectives:

- Provide maximum portability

  The device drivers are provided as ANSI C source code. ANSI C was chosen to maximize portability across processors and development tools. Source code is provided both to aid customers in debugging their applications as well as allow customers to modify or optimize the device driver if necessary.

  A layered device driver architecture additionally separates device communication from processor and Real Time Operating System (RTOS) dependencies, thus providing portability of core device driver functionality across processors and operating systems.

- Support FPGA configurability

  Since FPGA-based devices can be parameterized to provide varying functionality, the device drivers must support this varying functionality. The configurability of device drivers should be supported at compile-time and at run-time. Run-time configurability provides the flexibility needed for future dynamic system reconfiguration.

  In addition, a device driver supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per instance basis.

- Support simple and complex use cases

  Device drivers are needed for simple tasks such as board bring-up and testing, as well as complex embedded system applications. A layered device driver architecture provides both simple device drivers with minimal memory footprints and more robust, full-featured device drivers with larger memory footprints.

- Ease of use and maintenance

  Xilinx makes use of coding standards and provides well-documented source code in order to give developers (i.e., customers and internal development) a consistent view of source code that is easy to understand and maintain. In addition, the API for all device drivers is consistent to provide customers a similar look and feel between drivers.

Note: A detailed description of the Xilinx driver functions are given in the documentation area of the EDK installation *(XILINX_EDK*/doc/xilinx_driver_api*)*

Device Driver Architecture

The architecture of the device drivers is designed as a layered architecture as shown in Figure . The layered architecture accommodates the many use cases of device drivers while at the same time providing portability across operating systems, toolsets, and processors. The layered architecture provides seamless integration with an RTOS (Layer 2), high-level device drivers that are full-featured and portable across operating systems and processors (Layer 1), and low-level drivers for simple use cases (Layer 0). The following paragraphs describe each of the layers. The user can choose to use any and all layers.

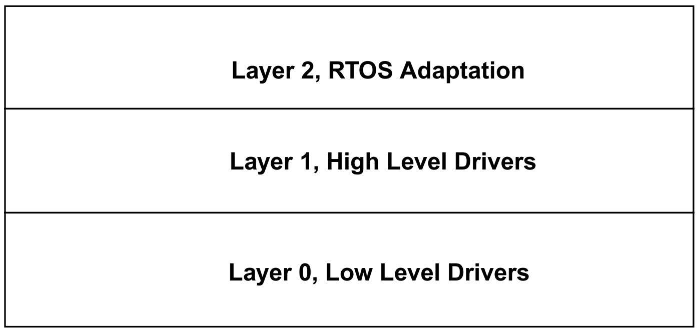


Figure 28-1: **Layered Architecture**

Layer 2, RTOS Adaptation

This layer consists of adapters for device drivers. An adapter converts a Layer 1 device driver interface to an interface that matches the requirements of the device driver scheme for an RTOS. Unique adapters may be necessary for each RTOS. Adapters typically have the following characteristics.

- Communicates directly to the RTOS and the Layer 1, high-level driver.
- References functions and identifiers specific to the RTOS. This layer is therefore not portable across operating systems.
- Can use memory management
- Can use RTOS services such as threading, inter-task communication, etc.

- Can be simple or complex depending on the RTOS interface and requirements for the device driver

Layer 1, High Level Drivers

This layer consists of high level device drivers. They are implemented as macros and functions and are designed to allow a developer to utilize all features of a device. These high-level drivers are independent of operating system and processor, making them highly portable. They typically have the following characteristics.

- Consistent and high-level (abstract) API that gives the user an “out-of-the-box” solution
- No RTOS or processor dependencies, making them highly portable
- Run-time error checking such as assertion of input arguments. Also provides the ability to compile away asserts.
- Comprehensive support of device features
- Abstract API that isolates the API from hardware device changes
- Supports device configuration parameters to handle FPGA-based parameterization of hardware devices.
- Supports multiple instances of a device while managing unique characteristics on a per instance basis.
- Polled and interrupt driven I/O
- Non-blocking function calls to aid complex applications
- May have a large memory footprint
- Typically provides buffer interfaces for data transfers as opposed to byte interfaces. This makes the API easier to use for complex applications.
- Does not communicate directly to Layer 2 adapters or application software. Utilizes asynchronous callbacks for upward communication.

Layer 0, Low Level Drivers

This layer consists of low level device drivers. They are implemented as macros and functions and are designed to allow a developer to create a small system, typically for internal memory of an FPGA. They typically have the following characteristics.

- Simple, low-level API
- Small memory footprint
- Little to no error checking is performed
- Supports primary device features only
- Minimal abstraction such that the API typically matches the device registers. The API is therefore less isolated from hardware device changes.
- No support of device configuration parameters
- Supports multiple instances of a device with base address input to the API
- None or minimal state is maintained
- Polled I/O only
- Blocking functions for simple use cases

- Typically provides byte interfaces but can provide buffer interfaces for packet-based devices.

Object-Oriented Device Drivers

In addition to the layered architecture, it is important that the user understand the underlying design of the device drivers. The device drivers are designed using an object-oriented methodology. The methodology is based upon components and is described in the following paragraphs. This approach pertains particularly to the Layer 1, high-level device drivers.

Component Definition

A component is a logical partition of the software which provides a functionality similar to one or more classes in C++. Each component provides a set of functions that operate on the internal data of the component. In general, components are not allowed access to the data of other components. A device driver is typically designed as a single component. A component may consist of one or more files.

Component Implementation

The component contains data variables which define the set of values that instances of that type can hold and a set of functions that operate on those data variables. Components must utilize the functions of other components in order to access the data of other components, rather than accessing component data directly. Components provide data abstraction and encapsulation by gathering the state of an object and the functions that operate on that object into a single unit and by denying direct access to its data members.

Component Data Variables

The primary mechanism for implementing a component in C is the structure. The data variables for a component are grouped in a single structure such that instances of the component each have their own data. The structure and the prototypes for all component functions are declared in the header file which is shared between the implementing component and other components which utilize it. A pointer to this structure, referred to as the instance pointer, is passed into each function of the component which operates on the instance data.

Component Interface

Each component has a set of functions which are collectively referred to as the component interface. Every function of a component which operates on the instance data utilizes a pointer, named InstancePtr, to an instance of a component as the first argument. This argument emulates the *this* pointer in C++ and allows the component function to manipulate the instance data.

Component Instance

An instance of a component is created when a variable is created using the component data type. An instance of a component maps to each physical hardware device. Each instance may have unique characteristics such as it's memory mapped address and specific device capabilities.

Component Example

The following code example illustrates a device driver component.

```
/* the device component data type */

typedef struct
{
    Xuint32 BaseAddress;  /* component data variables */
    Xuint32 IsReady;
    Xuint32 IsStarted;
} XDevice;

/* create an instance of a device */

XDevice DeviceInstance;

/* device component interfaces */

XStatus XDevice_Initialize(XDevice *InstancePtr, Xuint16 DeviceId);
XStatus XDevice_Start(XDevice *InstancePtr);
```

API and Naming Conventions

External Identifiers

External identifiers are defined as those items that are accessible to all other components in the system (global) and include functions, constants, typedefs, and variables.

An ’X’ is prepended to each Xilinx external so it does not pollute the global name space, thus reducing the risk of a name conflict with application code. The names of externals are based upon the component in which they exist. The component name is prepended to each external name. An underscore character always separates the component name from the variable or function name.

External Name Pattern:

```
X<component name>_VariableName;
X<component name>_FunctionName(ArgumentType Argument)
X<component name>_TypeName;
```

Constants are typically defined as all uppercase and prefixed with an abbreviation of the component name. For example, a component named XUartLite (for the UART Lite device driver) would have constants that begin with XUL_, and a component named XEmac (for the Ethernet 10/100 device driver) would have constants that begin with XEM_. The abbreviation utilizes the first three uppercase letters of the component name, or the first three letters if there are only two uppercase letters in the component name.

File Naming Conventions

The file naming convention utilizes long file names and is not limited to 8 characters as imposed by the older versions of the DOS operating system.

Component Based Source File Names

Source file names are based upon the name of the component implemented within the source files such that the contents of the source file are obvious from the file name. All file names must begin with the lowercase letter “x” to differentiate Xilinx source files. File extensions .h and .c are utilized to distinguish between header source files and implementation source files.

Implementation Source Files (*.c)

The C source files contain the implementation of a component. A component is typically contained in multiple source files to allow parts of the component to be user selectable.

Source File Naming Pattern:

```
x<component name>.c                        main source file
x<component name>_functionality.c          secondary source file
```

Header Source Files (*.h)

The header files contain the interfaces for a component. There will always be external interfaces which is what an application that utilizes the component invokes.

- The external interfaces for the high level drivers (Layer 1) are contained in a header file with the file name format *x<component name>.h.*
- The external interfaces for the low level drivers (Layer 0) are contained in a header file with the file name format *x<component name>_l.h.*

In the case of multiple C source files which implement the class, there may also be a header file which contains internal interfaces for the class. The internal interfaces allow the functions within each source file to access functions in the another source file.

- The internal interfaces are contained in a header file with the file name format *x<component name>_i.h.*

Device Driver Layers

Layer 1 and Layer 0 device drivers (i.e., high-level and low-level drivers) are typically bundled together in a directory. The Layer 0 device driver files are named *x<component name>_l.h* and *x<component name>_l.c*. The “*_l*” indicates low-level driver. Layer 2 RTOS adapter files include the word “adapter” in the file name, such as *x<component name>_adapter.h* and *x<component name>_adapter.c*. These are typically stored in a different directory name (e.g., one specific to the RTOS) than the device driver files.

Example File Names

The following source file names illustrates an example which is complex enough to utilize multiple C source files.

```
xuartns550.c          Main implementation file
xuartns550_intr.c     Secondary implementation file for interrupt
handling
xuartns550.h          High level external interfaces header file
xuartns550_i.h        Internal identifiers header file
xuartns550_l.h        Low level external interfaces header file
xuartns550_l.c        Low level implementation file
xuartns550_g.c        Generated file controlling parameterized
instances
```

```
and,

xuartns550_sio_adapter.c  VxWorks Serial I/O (SIO) adapter
```

High Level Device Driver API

High level device drivers are designed to have an API which includes a standard API together with functions that may be unique to that device. The standard API provides a consistent interface for Xilinx drivers such that the effort to use multiple device drivers is minimized. An example API follows.

Standard Device Driver API

Initialize

This function initializes an instance of a device driver. Initialization must be performed before the instance is used. Initialization includes mapping a device to a memory-mapped address and initialization of data structures. It maps the instance of the device driver to a physical hardware device. The user is responsible for allocating an instance variable using the driver's data type, and passing a pointer to this variable to this and all other API functions.

Reset

This function resets the device driver and device with which it is associated. This function is provided to allow recovery from exception conditions. This function resets the device and device driver to a state equivalent to after the Initialize() function has been called.

SelfTest

This function performs a self-test on the device driver and device with which it is associated. The self-test verifies that the device and device driver are functional.

Optional Functions

Each of the following functions may be provided by device drivers.

Start

This function is provided to start the device driver. Starting a device driver typically enables the device and enables interrupts. This function, when provided, must be called prior to other data or event processing functions.

Stop

This function is provided to stop the device driver. Stopping a device driver typically disables the device and disables interrupts.

GetStats

This function gets the statistics for the device and / or device driver.

ClearStats

This function clears the statistics for the device and / or device driver.

InterruptHandler

This function is provided for interrupt processing when the device must handle interrupts. It does not save or restore context. The user is expected to connect this interrupt handler to their system interrupt controller. Most drivers will also provide hooks, or callbacks, for the user to be notified of asynchronous events during interrupt processing (e.g., received data or device errors).

Configuration Parameters

Standard device driver API functions (of Layer 1, high-level drivers) such as Initialize() and Start() require basic information about the device such as where it exists in the system memory map or how many instances of the device there are. In addition, the hardware features of the device may change because of the ability to reconfigure the hardware within the FPGA. Other parts of the system such as the operating system or application may need to know which interrupt vector the device is attached to. For each device driver, this type of information is distributed across two files: *xparameters.h* and *x<component name>_g.c*.

Typically, these files are automatically generated by a system generation tool based on what the user has included in their system. However, these files can be hand coded to support internal development and integration activities. Note that the low-level drivers of Layer 0 do not require or make use of the configuration information defined in these two files. Other than the memory-mapped location of the device, the low-level drivers are typically fixed in the hardware features they support.

xparameters.h

This source file centralizes basic configuration constants for all drivers within the system. Browsing this file gives the user an overall view of the system architecture. The device drivers and Board Support Package (BSP) utilize the information contained here to configure the system at runtime. The amount of configuration information varies by device, but at a minimum the following items should be defined for each device:

- Number of device instances
- Device ID for each instance (Level 1 drivers only) A Device ID uniquely identifies each hardware device which maps to a device driver. A Device ID is used during initialization to perform the mapping of a device driver to a hardware device. Device IDs are typically assigned either by the user or by a system generation tool. It is currently defined as a 16-bit unsigned integer.
- Device base address for each instance
- Device interrupt assignment for each instance if interrupts can be generated.

File Format and Naming Conventions

Every device must have the following constant defined indicating how many instances of that device are present in the system (note that `<component name>` does not include the preceding “X”):

```
XPAR_X<component name>_NUM_INSTANCES
```

Each device instance will then have multiple, unique constants defined. The names of the constants typically match the hardware configuration parameters, but can also include other constants. For example, each device instance has a unique device identifier (DEVICE_ID), the base address of the device’s registers (BASEADDR), and the end address of the device’s registers (HIGHADDR).

```
XPAR_<component instance>_DEVICE_ID
XPAR_<component instance>_BASEADDR
XPAR_<component instance>_HIGHADDR
```

<component instance> is the instance name of the component. Note that the system generation tools may create these constants with a different convention than described here. Other device specific constants are defined as needed:

```
XPAR_<component instance>_<item description>
```

When the device specific constant applies to all instances of the device:

```
XPAR_<component name>_<item description>
```

For devices that can generate interrupts, a separate section within *xparameters.h* is used to store interrupt vector information. While the device driver implementation files do not utilize this information, their RTOS adapters, BSP files, or user application code will require them to be defined in order to connect, enable, and disable interrupts from that device. The naming convention of these constants varies whether an interrupt controller is part of the system or the device hooks directly into the processor.

For the case where an interrupt controller is considered external and part of the system, the naming convention (for a Level 1 interrupt controller) is as follows:

```
XPAR_<interrupt controller instance>_<component instance>_<interrupt
pin>_INTR
```

Where <interrupt controller instance> is the name of the interrupt controller component, <component instance> is the component instance of the component connected to the controller and <interrupt pin> is the name of the pin connected to the interrupt controller. Of course XPAR_XINTC must have the other required constants DEVICE_ID, BASEADDR, etc. This convention supports single and cascaded interrupt controller architectures.

For the case where an interrupt controller is considered internal to a processor, the naming convention changes:

```
XPAR_<proc name>_<component instance>_<interrupt pin>_INTR
```

Where <proc name> is the name of the processor.

When a Level 0 interrupt controller driver is used, xparameters.h contains the interrupt mask for each interrupting peripheral. The naming convention for this is:

```
XPAR_<component instance>_<interrupt pin>_MASK
```

x<component name>_g.c

The header file *x<component name>.h* defines the type of a configuration structure. The type will contain all of the configuration information necessary for an instance of the device. The format of the data type is as follows:

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;

   /* Other device dependent data attributes */

} X<component name>_Config;
```

The implementation file *x<component name>_g.c* defines an array of structures of `X<component name>_Config` type. Each element of the array represents an instance of the device, and contains most of the per-instance XPAR constants from *xparameters.h*. These files are only created for Level 1 drivers.

Example

To help illustrate the relationships between these configuration files, an example is presented that contains a single interrupt controller whose component name is MYINTC and a single UART whose component name is MYUART. Only xintc.h and xintc_g.c are illustrated, but xuart.h and xuart_g.c would be very similar. Both the uart and interrupt controller use Level 1 drivers.

xparameters.h

```
/* Constants for INTC */
XPAR_XINTC_NUM_INSTANCES     1
XPAR_MYINTC_DEVICE_ID 0
XPAR_MYINTC_BASEADDR        0xA0000100
XPAR_MYINTC_HIGHADDR 0xA0000200

/* Interrupt vector assignments for this instance */
XPAR_MYINTC_MYUART_INTERRUPT_INTR 0

/* Constants for UART */
XPAR_XUARTLITE_NUM_INSTANCES     1
XPAR_MYUART_DEVICE_ID       0
XPAR_MYUART_BASEADDR        0xB0001000
XPAR_MYUART_HIGHADDR 0xB0001100
```

xintc.h

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;
} XIntc_Config;
```

xintc_g.c

```
static XintcConfig[XPAR_XINTC_NUM_INSTANCES] =
{
  {
     XPAR_MYINTC_DEVICE_ID,
     XPAR_MYINTC_BASEADDR
  }
};
```

If the interrupt controller and the uart use Level 0 drivers, then the following files would be created. xintc_lg.c contains the interrupt vector tables that are used by the low level interrupt handler.

xparameters.h

```
/* Constants for INTC */
XPAR_XINTC_NUM_INSTANCES     1
```

```
XPAR_MYINTC_BASEADDR        0xA0000100
XPAR_MYINTC_HIGHADDR 0xA0000200
XPAR_INTC_MAX_NUM_INTR_INPUTS 1
XPAR_MYINTC_KIND_OF_INTR 0x00000000

/* Interrupt mask assignments for this instance */
XPAR_MYUART_INTERRUPT_MASK 0x00000001

/* Constants for UART */
XPAR_XUARTLITE_NUM_INSTANCES     1
XPAR_MYUART_BASEADDR        0xB0001000
XPAR_MYUART_HIGHADDR 0xB0001100
```

xintc_lg.c

```
#include "xbasic_types.h"
#include "xintc_l.h"
#include "xparameters.h"
extern void uartlite_int_handler (void *);


XIntc_VectorTableEntry
XIntc_InterruptVectorTable[XPAR_INTC_MAX_NUM_INTR_INPUTS] = {
        {
                uartlite_int_handler,
                (void *) XPAR_MYUART_BASEADDR
        }
};
Xuint32 XIntc_AckBeforeService = XPAR_MYINTC_KIND_OF_INTR;
```

Common Driver Infrastructure

Source Code Documentation

The comments in the device driver source code contain *doxygen* tags for *javadoc*-style documentation. *Doxygen* is a *javadoc*-like tool that works on C language source code. These tags typically start with “@” and provide a means to automatically generate HTML-based documentation for the device drivers. The HTML documentation contains a detailed description of the API for each device driver.

Driver Versions

Some device drivers may have multiple versions. Device drivers are usually versioned when the API changes, either due to a significant hardware change or simply restructuring of the device driver code. The version of a device driver is only indicated within the comment block of a device driver file. A modification history exists at the top of each file and contains the version of the driver. An example of a device driver version is “1.00b”, where 1 is the major revision, 00 is the minor revision, and b is a subminor revision. The hardware device and its device driver must match major and minor revisions in order to be compatible.

Currently, the user is not allowed to link two versions of the same device driver into their application. The versions of a device driver use the same function and file names, thereby preventing them from being linked into the same link image. As multiple versions of drivers are supported, the version name will be included in the driver file names, as in *x<component>_v1_00_a.c*.

Primitive Data Types

The primitive data types provided by C are minimized by the device drivers because they are not guaranteed to be the same size across processor architectures. Data types which are size specific are utilized to provide portability and are contained in the header file *xbasic_types.h*.

Device I/O

The method by which I/O devices are accessed varies between processor architectures. In order for the device drivers to be portable, this difference is isolated such that the driver for a device will work for many microprocessor architectures with minimal changes. A device I/O component, XIo, in *xio.c* and *xio.h* source files, contains functions and/or macros which provide access to the device I/O and are utilized for portability.

Error Handling

Errors that occur within device drivers are propagated to the application. Errors can be divided into two classes, synchronous and asynchronous. Synchronous errors are those that are returned from function calls (either as return status or as a parameter), so propagation of the error occurs when the function returns. Asynchronous errors are those that occur during an asynchronous event, such as an interrupt and are handled through callback functions.

Return Status

In order to indicate an error condition, functions which include error processing return a status which indicates success or an error condition. Any other return values for such functions are returned as parameters. Error codes are standardized in a 32-bit word and the definitions are contained in the file *xstatus.h*.

Asserts

Asserts are utilized in the device drivers to allow better debugging capabilities. Asserts are used to test each input argument into a function. Asserts are also used to ensure that the component instance has been initialized.

Asserts may be turned off by defining the symbol NDEBUG before the inclusion of the header file *xbasic_types.h*.

The assert macro is defined in *xbasic_types.h* and calls the function XAssert when an assert condition fails. This function is designed to allow a debugger to set breakpoints to check for assert conditions when the assert macro is not connected to any form of I/O.

The XAssert function calls a user defined function and then enters an endless loop. A user may change the default behavior of asserts such that an assert condition which fails does return to the user by changing the initial value of the variable XWaitInAssert to XFALSE in *xbasic_types.c*. A user defined function may be defined by initializing the variable XAssertCallbackRoutine to the function in *xbasic_types.c*.

Communication with the Application

Communication from an application to a device driver is implemented utilizing standard function calls. Asynchronous communication from a device driver to an application is accomplished with callbacks using C function pointers. It should be noted that callback functions are called from an interrupt context in many drivers. The application function called by the asynchronous callback must minimize processing to communicate to the application thread of control.

Reentrancy and Thread Safety

The device drivers are designed to be reentrant, but may not be thread-safe due to shared resources.

Interrupt Management

The device drivers use device-specific interrupt management rather than processor-specific interrupt management.

Multi-threading & Dynamic Memory Management

The device drivers are designed without the use of multi-threading and dynamic memory management.   This is expected to be accomplished by the application or by an RTOS adapter.

Cache & MMU Management

The device drivers are designed without the use of cache and MMU management. This is expected to be accomplished by the application or by an RTOS adapter.

Chapter 29

Stand-Alone Board Support Package

Overview

The Board Support Package (BSP) is a set of software modules used to access processor specific functions. The stand-alone BSP is used when an application accesses board / processor features directly (without an intervening Operating System layer).

MicroBlaze BSP

When the user system contains a MicroBlaze, and no Operating System, the Library Generator automatically builds the Stand-Alone BSP in the project library libxil.a.

Interrupt Handling

The `microblaze_enable_interrupts.s` and `microblaze_disable_interrupts.s` files contain functions to enable and disable interrupts on the MicroBlaze.

void microblaze_enable_interrupts(void)

This function enables interrupts on the MicroBlaze. When the MicroBlaze starts up, interrupts are disabled. Interrupts must be explicitly turned on using this function.

void microblaze_disable_interrupts(void)

This function disables interrupts on the MicroBlaze. This function may be called when entering a critical section of code where a context switch is undesirable.

Instruction Cache Handling

The `microblaze_enable_icache.s` and `microblaze_disable_icache.s` files contain functions to enable and disable the instruction cache on MicroBlaze.

void microblaze_enable_icache(void)

This functions enables the instruction cache on MicroBlaze. When MicroBlaze starts up, the instruction cache is disabled. The ICache must be explicitly turned on using this function.

void microblaze_disable_icache(void)

This function disables the instruction cache on MicroBlaze.

void microblaze_update_icache (int tag, int instr, int lock_valid)

This function updates the cache `tag` with the appropriate `instr`. The function disables the cache before updating the tag line. The MSR is restored back to its original value after the cache is updated.

This function can also be used to invalidate and lock the cahce depending on the value of the `lock_valid` parameter. The effect of this parameter is summarized in the Table 29-1 ,

Table 29-1: **Effect of lock_valid parameter**

Lock	Valid	lock_valid	Effect
0	0	0	Invalidate Cache
0	1	1	Valid, but unlocked cacheline. The same cacheline can be used for more than one addresses, if required.The previous address will be swapped out of the cache and written to the memory
1	0	2	Invalidate Cache, No effect of lock bit
1	1	3	Valid cache and locked cacheline. The cacheline is now locked to a particular address. Other addresses cannot use this cacheline.

void microblaze_init_icache_range (int cache_addr, int cache_size)

The icache can be initialized using the function `microblaze_init_icache_range`. This function can be used for initializing the entire icache or only a part of it. The parameter `cache_addr` indicate the beginning of the cache location, which is to be initialized. The `cache_size` represents the size from `cache_addr`, which needs to be initialized.

for eg. ***microblaze_init_icache_range*** *(0x00000300, 0x100) will initialize the instruction cache region between 0x300 to 0x3ff (0x100 bytes of cache memory is cleared starting from 0x300).*

Data Cache Handling

The `microblaze_enable_dcache.s` and `microblaze_disable_dcache.s` files contain functions to enable and disable the data cache on MicroBlaze.

void microblaze_enable_dcache(void)

This functions enables the data cache on MicroBlaze. When MicroBlaze starts up, the data cache is disabled. The Dcache must be explicitly turned on using this function.

void microblaze_disable_dcache(void)

This function disables the instruction cache on MicroBlaze.

void microblaze_update_dcache (int tag, int data, int lock_valid)

This function updates the cache `tag` with the appropriate `data.` The function disables the cache before updating the tag line. The MSR is restored back to its original value after the cache is updated.

This function can also be used to invalidate and lock the cahce depending on the value of the `lock_valid` parameter. The effect of this parameter is summarized in the Table 29-1

void microblaze_init_dcache_range (int cache_addr, int cache_size)

The icache can be initialized using the function `microblaze_init_dcache_range`. This function can be used for initializing the entire icache or only a part of it. The parameter `cache_addr` indicate the beginning of the cache location, which is to be initialized. The `cache_size` represents the size from `cache_addr`, which needs to be initialized.

for eg. ***microblaze_init_dcache_range*** *(0x00000300, 0x100) will initialize the data cache region between 0x300 to 0x3ff (0x100 bytes of cache memory is cleared starting from 0x300).*

Fast Simplex Link Interface Macros

The Fast Simplex Link (FSL) interfaces on MicroBlaze can be used in several ways.

microblaze_bread_datafsl(val, id)

This macro performs a blocking data get function on an input FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_bwrite_datafsl(val, id)

This macro performs a blocking data put function on an output FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_nbread_datafsl(val, id)

This macro performs a non-blocking data get function on an input FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_nbwrite_datafsl(val, id)

This macro performs a non- blocking data put function on an output FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_bread_cntlfsl(val, id)

This macro performs a blocking control get function on an input FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_bwrite_cntlfsl(val, id)

This macro performs a blocking control put function on an output FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_nbread_cntlfsl(val, id)

This macro performs a non-blocking control get function on an input FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

microblaze_nbwrite_cntlfsl(val, id)

This macro performs a non-blocking data control function on an output FSL of MicroBlaze. **id** is the FSL identifier and can range from 0 to 7.

PowerPC BSP

When the user system contains a PowerPC, and no Operating System, the Library Generator automatically builds the Stand-Alone BSP in the project library libxil.a.

The Stand-Alone BSP contains boot code, cache, file and memory management, configuration, exception handling, time and processor specific include functions.

Boot Code

The `boot.S`, `crt0.S`, and `eabi.S` files contain a minimal set of code for initializing the processor and starting an application.

boot.S

Code in the `boot.S` consists of the two sections **boot** and **boot0**. The boot section contains only one instruction that is labeled with **_boot**. During the link process, this instruction is mapped to the reset vector and the **_boot** label marks the application's entry point. The boot instruction is a jump to the **_boot0** label. The **_boot0** label must reside within a 23-bit address space of the **_boot** label. It is defined in the **boot0** section. The code in the **boot0** section calculates the 32-bit address of the **_start** label and jumps to it.

crt0.S

Code in the **crt0.S** file starts executing at the **_start** label. It initializes the **.sbss** and **.bss** sections to zero, as required by the ANSI C specification, sets up the stack, initializes some processor registers, and calls the main() function.

The program remains in an endless loop on return from main().

eabi.S

When an application is compiled and linked with the **-msdata=eabi** option, GCC inserts a call to the **__eabi** label at the beginning of the main() function. This is the place where register R13 must be set to point to the **.sdata** and **.sbss** data sections and register R2 must be set to point to the **.sdata2** read-only data section.

Code in **eabi.S** sets these two registers to the correct values. The **_SDA_BASE_** and **_SDA2_BASE_** labels are generated by the linker.

Cache

The `xcache_l.c` file and corresponding `xcache_l.h` include file provide access to cache and cache-related operations.

void XCache_WriteCCR0(unsigned int val);

The XCache_WriteCCR0() function writes an integer value to the CCR0 register. Below is a sample code sequence. Before writing to this register, the instruction cache must be enabled to prevent a lockup of the processor core. After writing the CCR0, the instruction cache can be disabled, if not needed.

```
...
XCache_EnableICache(0x80000000) /* enable instruction cache for first
128 MB memory region */
XCache_WriteCCR0(0x2700E00) /* enable 8 word pre-fetching */
XCache_DisableICache() /* disable instruction cache */
...
```

void XCache_EnableDCache(unsigned int regions);

The XCache_EnableDCache() function enables the data cache for a specific memory region. Each bit in the *regions* parameter represents 128 MB of memory.

A value of 0x80000000 enables the data cache for the first 128 MB of memory (0 - 0x7FFFFFF). A value of 0x1 enables the data cache for the last 128 MB of memory (0xF8000000 - 0xFFFFFFFF).

void XCache_DisableDCache(void);

The XCache_DisableDCache() function disables the data cache for all memory regions.

void XCache_FlushDCacheLine(unsigned int adr);

The XCache_FlushDCacheLine() function flushes and invalidates the data cache line that contains the address specified by the *adr* parameter. A subsequent data access to this address results in a cache miss and a cache line refill.

void XCache_StoreDCacheLine(unsigned int adr);

The XCache_StoreDCacheLine() function stores in memory the data cache line that contains the address specified by the *adr* parameter. A subsequent data access to this address results in a cache hit if the address was already cached; otherwise, it results in a cache miss and cache line refill.

void XCache_EnableICache(unsigned int regions);

The XCache_EnableICache() function enables the instruction cache for a specific memory region. Each bit in the *regions* parameter represents 128 MB of memory.

A value of 0x80000000 enables the instruction cache for the first 128 MB of memory (0 - 0x7FFFFFF). A value of 0x1 enables the instruction cache for the last 128 MB of memory (0xF8000000 - 0xFFFFFFFF).

void XCache_DisableICache(void);

The XCache_DisableICache() function disables the instruction cache for all memory regions.

void XCache_InvalidateICache(void);

The XCache_InvalidateICache() function invalidates the whole instruction cache. Subsequent instructions produce cache misses and cache line refills.

void XCache_InvalidateICacheLine(unsigned int adr);

The XCache_InvalidateICacheLine() function invalidates the instruction cache line that contains the address specified by the *adr* parameter. A subsequent instruction to this address produces a cache miss and a cache line refill.

Exception Handling

This section documents the exception handling API that is provided in the Board Support Package. For an in-depth explanation on how exceptions and interrupts work on the PPC405, please refer to the chapter "Exceptions and Interrupts" in the PPC *User's Manual*.

The exception handling API consists of a set of the files `xvectors.S`, `xexception_l.c`, and the corresponding header file `xexception_l.h`.

void XExc_Init(void);

This function sets up the interrupt vector table and registers a "do nothing" function for each exception. This function has no parameters and does not return a value.

This function must be called before registering any exception handlers or enabling any interrupts. When using the exception handler API, this function should be called at the beginning of your main() routine.

IMPORTANT: If you are not using the default linker script, you need to reserve memory space for storing the vector table in your linker script. The memory space must begin on a 64k boundary. The linker script entry should look like this example:

```
.vectors :
  {
    . = ALIGN(64k);
    *(.vectors)
  }
```

For further information on linker scripts, please refer to the Linker documentation.

void XExc_RegisterHandler(Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr);

This function is used to register an exception handler for a specific exception. It does not return a value. Please refer to Table 29-2 for a list of parameters.

Table 29-2: **Exception Handler Parameters**

Parameter Name	Parameter Type	Description
ExceptionId	Xuint8	Exception to which this handler should be registered. The type and the values are defined in the header file `xexception_l.h`. Please refer to Table 29-3 for possible values.
Handler	XExceptionHandler	Pointer to the exception handling function
DataPtr	void *	User value to be passed when the handling function is called

Table 29-3: **Registered Exception Types and Values**

Exception Type	Value
XEXC_ID_JUMP_TO_ZERO	0
XEXC_ID_MACHINE_CHECK	1
XEXC_ID_CRITICAL_INT	2
XEXC_ID_DATA_STORAGE_INT	3
XEXC_ID_INSTUCTION_STORAGE_INT	4
XEXC_ID_NON_CRITICAL_INT	5
XEXC_ID_ALIGNMENT_INT	6
XEXC_ID_PROGRAM_INT	7
XEXC_ID_FPU_UNAVAILABLE_INT	8
XEXC_ID_SYSTEM_CALL	9
XEXC_ID_APU_AVAILABLE	10
XEXC_ID_PIT_INT	11
XEXC_ID_FIT_INT	12
XEXC_ID_WATCHDOG_TIMER_INT	13
XEXC_ID_DATA_TLB_MISS_INT	14
XEXC_ID_INSTRUCTION_TLB_MISS_INT	15
XEXC_ID_DEBUG_INT	16

The function provided as the *Handler* parameter must have the following function prototype:

```
typedef void (*XExceptionHandler)(void * DataPtr);
```

This prototype is declared in the `xexception_l.h` header file.

When this exception handler function is called, the parameter *DataPtr* will contain the same value as you provided when you registered the handler.

void XExc_RemoveHandler(Xuint8 ExceptionId)

This function is used to deregister a handler function for a given exception. For possible values of parameter *ExceptionId***,** please refer to Table 29-3.

void XExc_mEnableExceptions (EnableMask);

This macro is used to enable exceptions. It must be called after initializing the vector table with function exception_Init and registering exception handlers with function XExc_RegisterHandler. The parameter *EnableMask* is a bitmask for exceptions to be enabled. The *EnableMask* parameter may have the values XEXC_CRITICAL, XEXC_NON_CRITICAL or XEXC_ALL.

void XExc_mDisableExceptions (DisableMask);

This macro is called to disable exceptions. The parameter *DisableMask* is a bitmask for exceptions to be disabled.The *DisableMask* parameter may have the values XEXC_CRITICAL, XEXC_NON_CRITICAL or XEXC_ALL.

Files

File support is limited to the **stdin** and **stdout** streams. In such an environment, the following functions do not make much sense:

- `open()` (in **`open.c`**)
- `close()` (in **`close.c`**)
- `fstat()` (in **`fstat.c`**)
- `unlink()` (in **`unlink.c`**)
- `lseek()` (in **`lseek.c`**)

These files are included for completeness and because they are referenced by the C library.

int read(int fd, char *buf, int nbytes);

The read() function in `read.c` reads *nbytes* bytes from the standard input by calling inbyte(). It blocks until all characters are available, or the end of line character is read. Read() returns the number of characters read. The parameter *fd* is ignored.

int write(int fd, char *buf, int nbytes);

The write() function in `write.c` writes *nbytes* bytes to the standard output by calling outbyte(). It blocks until all characters have been written. Write() returns the number of characters written. The parameter *fd* is ignored.

int isatty(int fd);

The isatty() function in `isatty.c` reports if a file is connected to a tty. This function always returns 1, since only the **stdin** and **stdout** streams are supported.

Memory Management

char *sbrk(int nbytes);

The sbrk() function in the `sbrk.c` file allocates `nbytes` of heap and returns a pointer to that piece of memory. This function is called from the memory allocation functions of the C library.

Process

The functions getpid() in `getpid.c` and kill() in `kill.c` are included for completeness and because they are referenced by the C library.

Processor-Specific Include Files

The `xreg405.h` include file contains the register numbers and the register bits for the PPC 405 processor.

The `xpseudo-asm.h` include file contains the definitions for the most often used inline assembler instructions.

These inline assembler instructions can be used from drivers and user applications written in C.

Time

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit time base counter inside the PowerPC core. The counter increases by one at every processor cycle.

The `sleep.c` file and corresponding `sleep.h` include file implement functions for tired programs. All sleep functions are implemented as busy loops.

typedef unsigned long long XTime;

The **XTime** type in `xtime_l.h` represents the Time Base register. This struct consists of the Time Base Low (TBL) and Time Base High (TBH) registers, each of which is a 32-bit wide register. The definition of **XTime** is as follows:

```
typedef unsigned long long XTime;
```

void XTime_SetTime(XTime xtime);

The XTime_SetTime() function in `xtime_l.c` sets the time base register to the value in *xtime*.

void XTime_GetTime(XTime *xtime);

The XTime_GetTime() function in `xtime_l.c` writes the current value of the time base register to variable *xtime*.

void XTime_TSRClearStatusBits(unsigned long Bitmask);

The XTime_TSRClearStatusBits() function in `xtime_l.c` is used to clear bits in the Timer Status Register (TSR). The parameter *Bitmask* designates the bits to be cleared. A one in any position of the Bitmask parameter clears the corresponding bit in the TSR. This function does not return a value.

The header file `xreg405.h` defines the following values for the *Bitmask* parameter:

Bitmask Parameter Values

Name	Value	Description
XREG_TSR_WDT_ENABLE_NEXT_WATCHDOG	0x80000000	Clearing this bit disables the watchdog timer event.
XREG_TSR_WDT_INTERRUPT_STATUS	0x40000000	Clears the Watchdog Timer Interrupt Status bit. This bit is set after a watchdog interrupt occurred, or could have occurred had it been enabled.
XREG_TSR_WDT_RESET_STATUS_11	0x30000000	Clears the Watchdog Timer Reset Status bits. These bits Specify the kind of reset that occurred as a result of a watchdog timer event.
XREG_TSR_PIT_INTERRUPT_STATUS	0x08000000	Clears the Programmable Interval Timer Status bit. This bit is set after a PIT interrupt has occurred.
XREG_TSR_FIT_INTERRUPT_STATUS	0x04000000	Clears the Fixed Interval Timer Status bit. This bit is set after a FIT interrupt has occurred.
XREG_TSR_CLEAR_ALL	0xFFFFFFFF	Clears all bits in the TSR. After a Reset, the content of the TSR is not specified. Use this Bitmask to clear all bits in the TSR.

`Example:`

```
XTime_TSRClearStatusBits(TSR_CLEAR_ALL);
```

void XTime_PITSetInterval(unsigned long interval);

The XTime_PITSetInterval() function in `xtime_l.c` is used to load a new value into the Programmable-Interval Timer Register. This register is a 32-bit decrementing counter clocked at the same frequency as the time-base register. Depending on the AutoReload setting the PIT is automatically reloaded with the last written value or has to be reloaded manually. This function does not return a value.

`Example:`

```
XTime_PITSetInterval(0x00ffffff);
```

void XTime_PITEnableInterrupt(void);

The XTime_PITEnableInterrupt() function in `xtime_l.c` enables the generation of PIT interrupts. An interrupt occurs when the PIT register contains a value of 1, and is then decremented. This function does not return a value. XExc_Init() must be called, the PIT interrupt handler must be registered, and exceptions must be enabled before calling this function.

```
Example:
   XTime_PITEnableInterrupt();
```

void XTime_PITDisableInterrupt(void);

The XTime_PITDisableInterrupt() function in `xtime_l.c` disables the generation of PIT interrupts. It does not return a value.

```
Example:
   XTime_PITDisableInterrupt();
```

void XTime_PITEnableAutoReload(void);

The XTime_PITEnableAutoReload() function in `xtime_l.c` enables the auto-reload function of the PIT Register. When auto-reload is enabled the PIT Register is automatically reloaded with the last value loaded by calling the **XTime_PITSetInterval** function when the PIT Register contains a value of 1 and is decremented. When auto-reload is enabled, the PIT Register never contains a value of 0. This function does not return a value.

```
Example:
   XTime_PITEnableAutoReload();
```

void XTime_PITDisableAutoReload(void);

The XTime_PITDisableAutoReload() function in `xtime_l.c` disables the auto-reload feature of the PIT Register. When auto-reload is disabled the PIT decrements from 1 to 0. If it contains a value of 0 it stops decrementing until it is loaded with a non-zero value. This function does not return a value.

```
Example:
   XTime_PITDisableAutoReload();
```

void XTime_PITClearInterrupt(void);

```
The XTime_PITClearInterrupt() function in xtime_l.c is used to clear
PIT-Interrupt-Status bit in the Timer-Status Register. This bit
specifies whether a PIT interrupt occurred. You must call this function
in your interrupt-handler to clear the Status bit, otherwise another PIT
interrupt will occur immediately after exiting the interrupt –handler
function. This function does not return a value. Calling this function
is equivalent to calling
XTime_TSRClearStatusBits(XREG_TSR_PIT_INTERRUPT_STATUS).
```

```
Example:
   XTime_PITClearInterrupt();
```

unsigned int usleep(unsigned int __useconds);

The usleep() function in `sleep.c` delays the execution of a program by *__useconds* microseconds. It always returns zero. This function requires that the processor frequency (in Hz) is defined. The default value of this variable is 400MHz. This value can be overwritten in the mss file as follows:

```
BEGIN PROCESSOR
PARAMETER HW_INSTANCE = PPC405_i
PARAMETER DRIVER_NAME = cpu_ppc405
PARAMETER DRIVER_VER = 1.00.a
PARAMETER CORE_CLOCK_FREQ_HZ = 20000000
END
```

The file xparameters.h can also be modified with the correct value, as follows:

```
#define XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ 20000000
```

unsigned int sleep(unsigned int __seconds);

The sleep() function in `sleep.c` delays the execution of a program by *__seconds* seconds. It always returns zero.This function requires that the processor frequency (in Hz) is defined. The default value of this variable is 400MHz. This value can be overwritten in the mss file as follows:

```
BEGIN PROCESSOR
PARAMETER HW_INSTANCE = PPC405_i
PARAMETER DRIVER_NAME = cpu_ppc405
PARAMETER DRIVER_VER = 1.00.a
PARAMETER CORE_CLOCK_FREQ_HZ = 20000000
END
```

The file xparameters.h can also be modified with the correct value, as follows:

```
#define XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ 20000000
```

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

The nanosleep() function in *sleep.c* is currently not implemented. It is a placeholder for linking applications against the C library. It always returns zero.

Address Management

Summary

This chapter describes the embedded processor program address management techniques. For advanced address space management, a discussion on linker scripts is also included in this chapter.

MicroBlaze Processor

Programs and Memory

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

MicroBlaze address space is divided between the system address space and the user address space. In certain examples, users would need advanced address space management, which can be done with the help of linker script, described in this chapter.

Current Address Space Restrictions

Memory and Peripherals Overview

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

BRAM Size Limits

The amount of BRAM memory that can be assigned to the LMB address space or to each instance of an OPB mapped BRAM peripheral is limited. The largest supported BRAM memory size for Virtex / VirtexE is 16 kilobytes and for Virtex-II it is 64 kilobytes. It is important to understand that these limits apply to each separately decoded on-chip memory region only. The total amount of on-chip memory available to a MicroBlaze system may exceed these limits. The total amount of memory available in the form of

BRAMs is also FPGA device specific. Smaller devices of a given device family provide less BRAM than larger devices in the same device family.

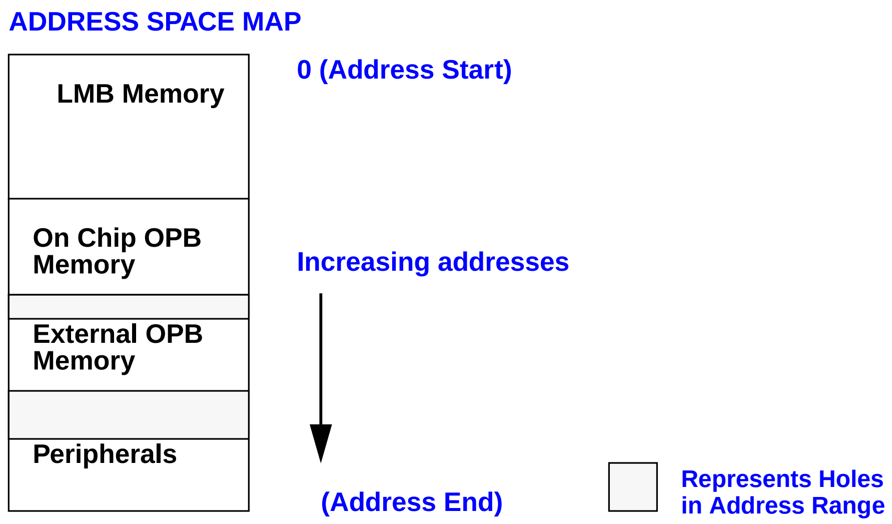


Figure 30-1: **A Sample Address Map for a MicroBlaze System**

Special Addresses

Every MicroBlaze system must have user writable memory present in addresses 0x00000000 through 0x00000018. These memory locations contain the addresses MicroBlaze jumps to after a reset, interrupt, or exception event occurs. This memory can be part of the LMB or the OPB BRAM address space. Please refer to Chapter "MicroBlaze Application Binary Interface" (ABI) in the "MicroBlaze Processor Reference Guide" for further details.

OPB Address Range Details

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

Address Map

Figure 30-1 shows a possible address map for a MicroBlaze System. The actual address map is defined in the MicroBlaze Hardware Specification (MHS) file. It contains an address map specifying the addresses of LMB memory, OPB memory, External memory and peripherals.

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

Memory Speeds and Latencies

MicroBlaze requires 2 clock cycles to access on-chip Block RAM connected to the LMB for *write* and 2 clock cycles for *read*. On chip memory connected to the OPB bus requires 3 cycles for *write* and 4 cycles for *read*. External memory access is further limited by off-chip memory access delays for read access, resulting in 5-7 clock cycles for *read*. Furthermore, memory accesses over the OPB bus may incur further latencies due to bus arbitration overheads. As a result, instructions or data that need to be accessed quickly should be stored in LMB memory when possible.

For more information on memory access times, see the *MicroBlaze Hardware Reference* chapter.

System Address Space

MicroBlaze programs can be executed in different scenarios. Each scenario needs a different set of system address space. The system address space is occupied by the xmdstub or the bootstub, when debug or boot support is required. System address space is also needed by the C-runtime routines.

System with only an executable [No debug, No Bootstrap]

The scenario is depicted in Figure 30-2(a). The C-runtime file crt0.o is linked with the user program. The system file, crt0.o starts at address location 0x0, immediately followed by user's program.

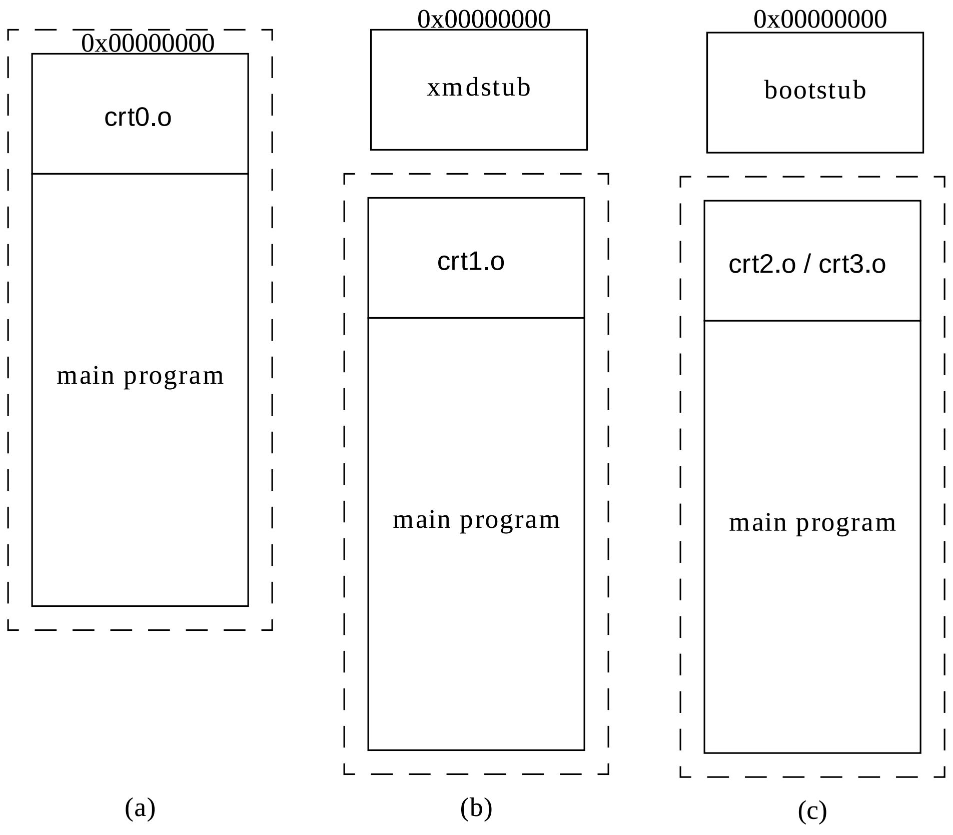


Figure 30-2: **Execution Scenarios**

System with debugging support

With systems requiring debug support, **xmdstub** must be downloaded at address location 0x0. The C-runtime file crt1.o is bundled with the user program and is place at a default location. This scenario is shown in Figure 30-2(b).

System with bootstrap support

The user can also bootstrap their program by using the bootstub. This bootstub occupies the system address space starting at address location 0x0. In addition to this system space, every user program is pre-pended with another C-runtime routine crt2.o or crt3.o depending on the compilation switch used. This scenario is shown in Figure 30-2(c).

Default User Address Space

The default usage of the compiler **mb-gcc** will place the users program immediately after the system address space. The user does not have to give any additional options in order to make space for the system files. The default start address for user programs is described in Table 30-1

Table 30-1: **Start address for different compilation switches**

Compile Option	Start Address
-xl-mode-executable	0x0
-xl-mode-xmdstub	0x400
-xl-mode-bootstrap	0x100
-xl-mode-bootstrap-reset	0x100

If the user needs to start the program at a location other than the default start address or if non-contiguous address space is required, advanced address space management is required.

Advanced User Address Space

Different Base Address, Contiguous User Address Space

The user program can run from any memory [that is, LMB memory or OPB memory]. By default, the compiler will place the user program at location defined in Table 30-1. To execute a program from any address location other than default, users must provide the compiler **mb-gcc** with an additional option.

The option required is

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

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

Different Base Address, Non-contiguous User Address Space

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

All such user programs need creation of non-contiguous executables. To facilitate creation of non-contiguous executable, linker scripts have to be modified. The default linker script provided with the MicroBlaze Distribution Kit will place all user code and data in one contiguous address space.

Linker scripts are defined in later sections in this document.

For more details on linker options see the Chapter 11, "GNU Compiler Tools" chapter.

Object-file Sections

The sections of an executable file are created by concatenating the corresponding sections in an object (.o) file. The various sections in the object file are given in Figure 30-3.:

.text

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

.rodata

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

.sdata2

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

.data

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

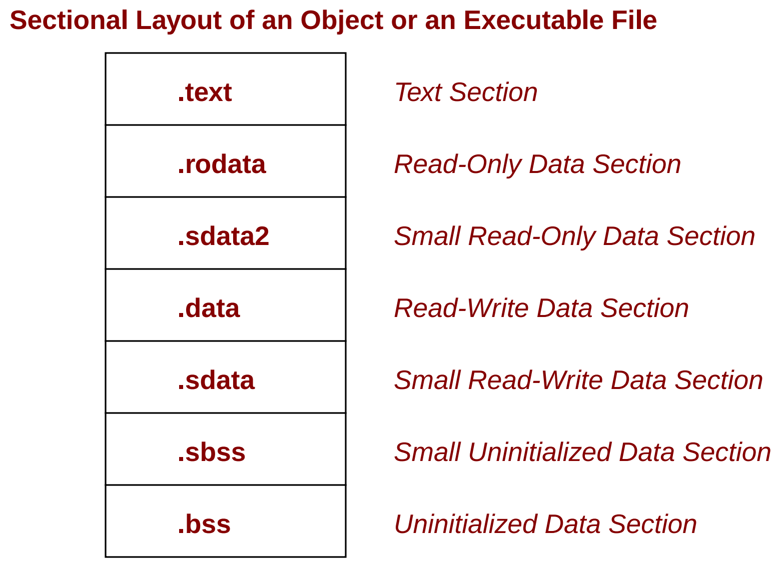

Sectional Layout of an Object or an Executable File
.text
Text Section
.rodata
Read-Only Data Section
.sdata2
Small Read-Only Data Section
.data
Read-Write Data Section
.sdata
Small Read-Write Data Section
.sbss
Small Uninitialized Data Section
.bss
Uninitialized Data Section

Figure 30-3: **Sectional layout of an object or executable file**

.sdata

This section contains small read-write data of a size less than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all data in the .sdata section uses a single instruction. (A preceding imm instruction will never be necessary). This section has the w (read-write) and the i (initialized) flags.

.sbss

This section contains small un-initialized data of a size less than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. This section has the w (read-write) flag.

.bss

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

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

If your address map specifies that the LMB, OPB and External Memory occupy contiguous areas of memory, you can use the default (built-in) linker script to generate your executable. This is done by invoking mb-gcc as follows:

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

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

$XILINX_EDK / gnu / microblaze / nt(orsol) / microblaze / lib / ldscripts, where $XILINX_EDK is the EDK installed directory. These scripts are imbibed into the linker and hence any changes to these scripts will not be reflected. To customize linker scripts, you must write your own linker script.

Minimal Linker Script

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

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

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

Your source files can now be compiled by specifying the minimal linker script as though it were a regular file, e.g.,

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

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

If you want more control over the layout of your memory, for example, if you want to split up your .text section between ILMB and IOPB, or if you want your stack and heap in DLMB and the rest of the .bss section in DOPB, you will need to write a full-fledged linker script.

Linker Script

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

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

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

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

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

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


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

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

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

   *(.text)
  } >LMB


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

  . = ALIGN(4);
  _heap = .;
  .bss : {
    _STACK_SIZE = 0x400;
    . += _STACK_SIZE;
    . = ALIGN(4);
  } >LMB
  _stack = .;
  /*                      */
  /* Start of OPB memory */
  /*                      */

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

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

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

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

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

  . = ALIGN(8);
  _ssro = .;
  .sdata : {
    *(.sdata)
  } >OPB
  . = ALIGN(4);
  .sbss : {
  __sbss_start = .;
    *(.sbss)
  __sbss_end = .;
  } >OPB
  . = ALIGN(8);
  _essro = .;
  _ssro_size = _essro - _ssro;
  _SDA_BASE_ = _ssro + (_ssro_size / 2 );

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

```
    _end = .;
  }
```

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

- Allocate space in the .bss section for stack and heap. Set the `_heap` variable to the beginning of this area, and the `_stack` variable to the end of this area. See the .bss section in the preceding script for an example. Ensure that the stack and heap space are contiguous. See example above to see how this is done.
- Ensure that the _SDA2_BASE_ variable points to the center of the .sdata2 area, and that _SDA2_BASE_ is aligned on a word boundary. See example above to see how this is done.
- Ensure that the .sdata and the .sbss sections are contiguous, that the _SDA_BASE_ variable points to the center of this section, and that _SDA_BASE_ is aligned on a word boundary. See example above to see how this is done.
- If you are not using the xmdstub, ensure that crt0 is always loaded into memory address zero. mb-gcc ensures that this is the first file specified to the loader, but the loader script needs to ensure that it gets loaded at address zero. See the .text section in the example above to see how this is done.
- Ensure that `__sbss_start, _sbss_end, __bss_start, __bss_end` variables are defined to the start and end of .sbss and .bss sections respectively. See the .bss, .sbss sections in the example above to see how this is done.
- Ensure that the .bss and .common sections from input files are contiguous. ANSI C requires that all uninitialized memory be initialized to startup (Not required for stack and heap). The standard crt0.s that we provide 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.
- In order to minimize your simulation time, make sure to point your `__bss_end` immediately after your declarations of all the .bss, .common sections from input files. See .opb_bss section in the above example to see how this is done.

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

PowerPC Processor

Programs and Memory

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

Figure 30-4 shows a sample address map for a PowerPC based EDK system. The figure shows that there can be various memories in the system. Here users need advanced address space management, which can be done with the help of linker script, described in "Linker Script" section.

Current Address Space Restrictions

Special Addresses

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

Default Linker Options

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

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

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

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

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

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

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

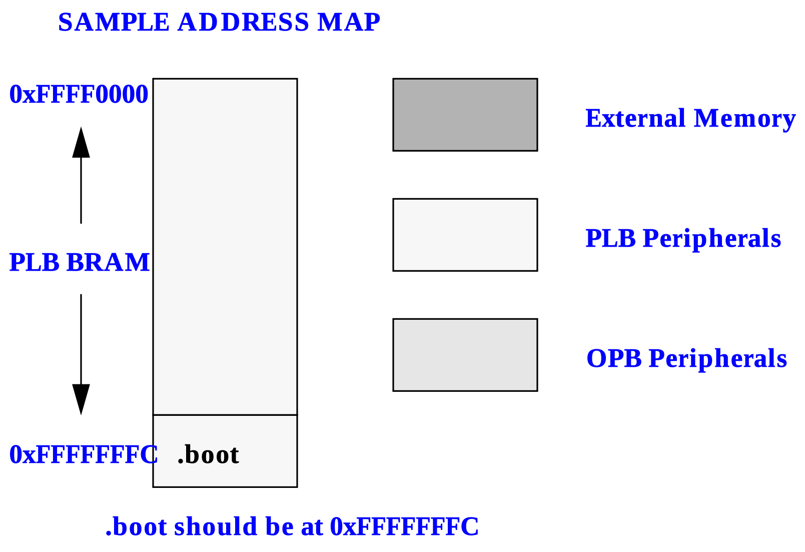


Figure 30-4: **A Sample Address Map for a PowerPC System**

Advanced User Address Space

Different Base Address, Contiguous User Address Space

The user program can run from any memory. By default, the compiler places the user program at location 0xFFFF0000. To execute the program from any address location other than the default, users must provide the compiler **powerpc-eabi-gcc** with additional option.

The option required is

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

where `start_address` is the new base address required for the user program.

Different Base Address, Non-contiguous User Address Space

The users can place different components of their program on different memories. For example, on PowerPC systems users can keep their code on instruction cache memory and the data on ZBT memory.

All such user programs need the creation of a non-contiguous executables. To facilitate creation of non-contiguous executable, linker scripts must be modified. The default linker script provided with the Embedded Distribution Kit will place all user code and data in one contiguous address space.

Linker scripts are defined in later sections in this chapter.

For more details on linker options, see the Chapter 11, "GNU Compiler Tools" chapter.

Linker Script

PowerPC Linker is built with default linker scripts. This script assumes a contiguous memory starting from address 0xFFFF0000. The script defines boot.o as the first file to be linked. boot.o is present in the libxil.a library which is created by the LibGen tool. The script defines the start address to be 0xFFFF000. If the user has given the start address through the linker option as:

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

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

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

You can take a look at the default linker scripts used by the linker at:

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

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

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

For a more detailed explanation of the linker options, please refer to the GNU linker documentation at (**http://www.gnu.org/manual**).

Minimal Linker Script

You must write a linker script if you want to control how your program is targeted to Instruction Cache, ZBTor External Memory.

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

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

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

Restrictions

Note that if you choose to write a linker script, you *must* do the following to ensure that your program will work correctly. An example linker script is given which incorporates these restrictions. Each of the restriction is highlighted in the example linker script.

- Allocate space in the .bss section for stack and heap. Set the `_stack` variable to the location after_ STACK_SIZE locations of this area, and the `_heap_start` variable to the next location after _STACK_SIZE location. Since the stack and heap need not be initialized for hardware as well as simulation, define `__bss_end` variable after the bss and COMMON defintions. See the .bss section in the example script below to see how this is done.
- Ensure that the variables `__SDATA_START__. __SDATA_END__, SDATA2_START, __SDATA2_END__, __SBSS2_START__ , __SBSS2_END__, __bss_start, __bss_end, __sbss_start and __sbss_end` are defined to the beginning and end of the sections sdata, sdata2, sbss2, bss, sbss respectively. See example below to see how this is done.
- Ensure that the .sdata and the .sbss sections are contiguous.
- Ensure that the .sdata2 and the .sbss2 sections are contiguous.
- Ensure that the .boot section starts at 0xFFFFFFFC.

- Ensure that boot.o is the first file to be linked (Check the STARTUP(boot.o) in the following script which achieves this)
- Ensure that the .vectors section is aligned on a 64k boundary. In order to ensure this, make .vectors as the first section defintion in the linker script. The memory where .vectors will be assigned to should start on a 64k boundary. Include this section definition only when your program uses interrupts/exceptions. See the example script given below to see how this is done.
- Each (physical) region of memory must use a separate program header. Two discontinuous regions of memory cannot share a program header
- Put all uninitialized sections (.bss, .sbss, .sbss2, stack, heap) at the end of a memory region. If this is impossible (eg., .sdata, .sbss and .sdata2, .sbss2 in same physical memory), start a new program header for the first initialized section after uninitialized sections.
- ANSI C requires that all uninitialized memory be initialized to startup (Not required for stack and heap). The standard crt0.s that we provide 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.

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

Here is a sample linker script.

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

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

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

STARTUP(boot.o)

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

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

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

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

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

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

  .boot0   : { *(.boot0)} > bram
  .text    : { *(.text) } > bram
  .boot    : { *(.boot) } > boot
  .data :
  {
    *(.data)
    *(.got2)
    *(.rodata)
    *(.fixup)
  } > bram
  /* small data area (read / write): keep together! */
.sdata : { *(.sdata) } > bram
.sbss :
 {
   . = ALIGN(4);
   *(.sbss)
   . = ALIGN(4);
 } > bram
 __sbss_start = ADDR(.sbss);
 __sbss_end   = ADDR(.sbss) + SIZEOF(.sbss);

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

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

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

__bss_end   = .;

   /* add stack and heap and align to 16 byte boundary */
   . = . + STACKSIZE;
   . = ALIGN(16);
   __stack = .;
```

```
      _heap_start = .;
      . = . + _HEAP_SIZE;
      . = ALIGN(16);
      _heap_end = .;
    } > bram
    __bss_start = ADDR(.bss);
  }
```

Chapter 31

Interrupt Management

Summary

This chapter outlines interrupt management in both MicroBlaze and PowerPC. It specifically details the role of LibGen for Low Level (Level 1) interrupt routines for MicroBlaze and PowerPC.

Levels of Interrupt Management

There are two levels of interrupt management possible using EDK. Level 0 is low level interrupt management and level 1 is a higher level interrupt management.

Level 0 (Low Level)

Level 0 interrupt management is charaterized by statically creating an interrupt vector table for the interrupt controller peripheral with the handler routines for all the peripherals that the interrupt controller is connected to. There is a statically determined priority ordering in the interrupt table. Once the platform is built and generated, users cannot register other interrupt handlers to handle peripheral interrupts. Currently there is a restriction of only one interrupt controller peripheral being connected to each processor in the system.

When using the level 0 procedure, LibGen can be used to statically configure interrupt handlers for peripherals. LibGen also configures an interrupt vector table for the interrupt controller peripheral to use. This is detailed in subsequent sections in this document.

Level 1 (High Level)

Level 1 interrupt management is characterized by having the flexibility of registering interrupt routines at program runtime.

When using the high level interrupt management, the user must dynamically register peripheral interrupt handler routines and enable / disable peripheral interrupts. Libgen does not configure interrupt vector tables, or the interrupt handlers when using the Level 1 management procedure. For more information please refer to the *Interrupt Controller Driver* specifics in Chapter 28, “Device Drivers”.

MicroBlaze Interrupt Management

This section describes interrupt management for MicroBlaze. Interrupt Management involves writing interrupt handler routines for peripherals and setting up the MHS and MSS files appropriately. MicroBlaze is capable of handling up to 32 interrupting devices. An interrupt controller peripheral is required for handling more than one interrupt signal. The mechanism of interrupt management is different if an interrupt controller is present than when it is not. This chapter describes both these management procedures.

Interrupt Handlers

Users are expected to write their own interrupt handlers (or Interrupt Service Routines) for any peripherals that raise interrupts. These routines can be written in C just like any other function. The interrupt handler function can have any name with the signature **void *func* (void *)**.

The main interrupt handler routine has to be tagged with *interrupt_handler* attributes so that mb-gcc can identify this as an interupt handler. Refer to the Interrupt Handlers section in Chapter 11, "GNU Compiler Tools", for more information on this attribute.

Libgen tags the interrupt controller interrupt routine automatically when the recommended interrupt management procedures as described in subsequent sections are followed.

The Interrupt Controller Peripheral

An interrupt controller peripheral should be used for handling multiple interrupts. In this case, the user is responsible for writing interrupt handlers for the peripheral interrupt signals only. The interrupt handler for the interrupt controller peripheral is automatically generated by LibGen. This handler ensures that interrupts from the peripherals are handled by individual interrupt handlers in the order of their priority. Figure 31-1 shows peripheral interrupt signals with priorities 1 through 4 connected to the interrupt controller input.

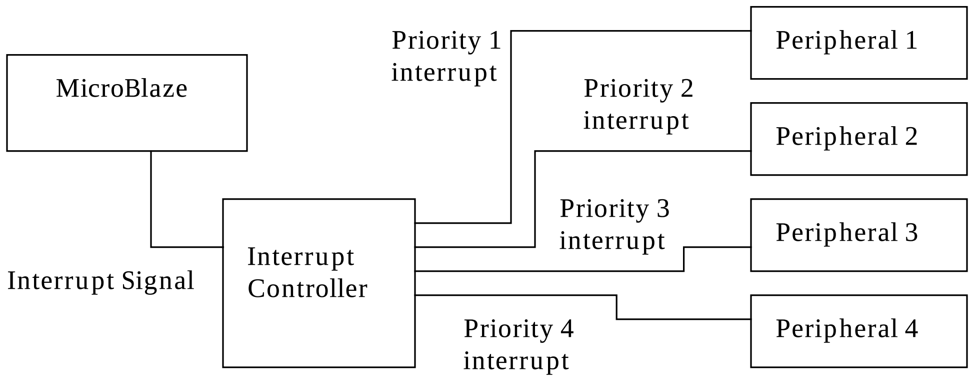


Figure 31-1: **Interrupt Controller and Peripherals**

The interrupt signal output of the controller is connected to the interrupt input of MicroBlaze. In the MSS file, each peripheral interrupt signal must be associated with interrupt handler routines (also called Interrupt Service Routines). LibGen automatically creates a vector table with the peripheral interrupt handler routines listed in the order of priority. When any peripheral raises an interrupt, the default handler for the interrupt controller is called. This handler then queries the interrupt controller to find out which peripheral raised the interrupt and then calls the peripheral specific interrupt handler. For a system where the interrupt controller is not present and only one interrupt signal is connected, the peripheral's interrupt handler (written by the user) gets called when an interrupt occurs.

MicroBlaze Enable Interrupts

The functions *microblaze_enable_interrupts* and *microblaze_disable_interrupts* are used to enable and disable interrupts on MicroBlaze. These functions are part of the MicroBlaze BSP and are described there.

System without Interrupt Controller (Single Interrupt Signal)

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

Procedure

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

1. The MHS and MSS file must be set up as follows:
   - The interrupt signal of the peripheral(or the external interrupt signal) must be connected to the interrupt input of the MicroBlaze in the MHS file.
   - The peripheral must be given an instance name using the INSTANCE keyword in the MHS file. Libgen creates a definition in **xparameters.h** (***`OUTPUT_DIR/PROC INST NAME/`*****`include`**) for XPAR_*INSTANCE_NAME*_BASEADDR mapped to the base address of this peripheral.
2. The interrupt handler routine that handles the signal should be written. The base address of the peripheral instance is accessed as XPAR_*INSTANCE_NAME*_BASEADDR.
3. The handler function is then designated to be an interrupt handler for the signal using the INT_HANDLER keyword in the MSS file (Refer to Chapter 19, "Microprocessor Software Specification (MSS)"). The peripheral instance is first selected in the MSS file, and then the INT_HANDLER attribute is given the function name. In case of an external interrupt signal, the INT_HANDLER attribute is given as a global parameter in the MSS file. The attribute is not part of any block in the MSS.
4. Libgen and mb-gcc are executed. This operation has the following implications:
   - the function is marked as an interrupt handler using the mb-gcc *interrupt_handler* attribute. All volatile registers used by this function are saved. Also, this function will return using the *rtid* instruction, rather than the normal *rtsd* instruction. Furthermore, this function will also be given the name *_interrupt_handler* by mb-gcc. By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.

- the startup code (crt0, crt1, crt2 or crt3) places the address of _interrupt_handler as the target address that MicroBlaze jumps to when an interrupt occurs. Therefore control will go to the interrupt handler when an interrupt occurs.

Example MHS File Snippet

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

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

Example MSS File snippet

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

Example C Program

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

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

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

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

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

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

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

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

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


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

  }
}

void
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

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

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

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

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

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

}
```

Example MHS File Snippet (for external interrupt signal)

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

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

Example MSS File snippet

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

Example xparameters.h snippet (generated by libgen)

```
extern void globint_handler(void) __attribute__ ((interrupt_handler));
```

System with an Interrupt Controller (One or More Interrupt Signals)

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

Procedure

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

1. The MHS and MSS files must be set up as follows:
   - The interrupt signals of all the peripherals must be assigned to the Intr port of the interrupt controller in the MHS file. The interrupt signal output of **intc** is then connected to the interrupt input of MicroBlaze.
   - The peripherals must be given instance names using the INSTANCE keyword in the MHS file. Libgen creates a definition in **xparameters.h** for XPAR_*INTC_INSTANCE_INSTANCE_NAME*_BASEADDR mapped to the base address of each peripheral for use in the user program. Libgen also creates an interrupt mask for each interrupt signal using the priorities as XPAR_*INTC_INSTANCE_INSTANCE_NAME_INTERRUPT_SIGNAL_NAME*_MASK. This can be used to enable or disable interrupts.
2. The interrupt handler functions for each interruptible peripheral must be written.
3. Each handler function is then designated to be the handler for an interrupt signal using the INT_HANDLER keyword in the MSS file. Note that **intc** interrupt signal must not be given an INT_HANDLER keyword. If the INT_HANDLER keyword is not present for a particular peripheral, a default dummy interrupt handler is used.
4. Libgen and mb-gcc is run to achieve the following:
   - Level of **intc** driver and drivers of peripherals connected to **intc** are zero.
   - ♦ The *XIntc_LowLevelInterruptHandler* function is marked as the main interrupt handler by mb-gcc using the *interrupt_handler* attribute. All volatile registers used

by this function are saved. Also, this function will return using the *rtid* instruction, rather than the normal *rtsd* instruction. Furthermore, this function will also be given the name *_interrupt_handler*. By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.

- ♦ An interrupt vector table is generated and compiled automatically by libgen. This table is accessed by the intc interrupt_handler to call peripheral interrupt handlers in order of priority.
- - Level of **intc** driver and drivers of peripherals connected to **intc** are one.
- ♦ The *XIntc_VoidInterruptHandler* function is marked as the main interrupt handler by mb-gcc using the *interrupt_handler* attribute. All volatile registers used by this function are saved. Also, this function will return using the *rtid* instruction, rather than the normal *rtsd* instruction. Furthermore, this function will also be given the name *_interrupt_handler*. By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.
- ♦ An interrupt vector table is generated only when each of the peripherals connected to intc registers its interrupt handlers with the intc interrupt handler. *XIntc_VoidInterruptHandler* calls the peripheral interrupt handler using the *dynamically* updated interrupt vector table to identify the handler.
- - The startup code (crt0, crt1, crt2 or crt3) places the address of _interrupt_handler as the target address that MicroBlaze jumps to when an interrupt occurs. Therefore control will go to the intc interrupt handler when an interrupt occurs.

Example MHS File Snippet

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

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

BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
```

```
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = timer1 & uart1
END

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

Example MSS File snippet

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

BEGIN DRIVER
parameter HW_INSTANCE = myuart
parameter DRIVER_NAME = uartlite
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = uart_int_handler, INT_PORT = Interrupt
END
```

Example C Program

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

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

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

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


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

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

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

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

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

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


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

  }
}

void
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

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

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

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

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

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

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

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

}
```

PowerPC Interrupt Management

For the PowerPC processor, LibGen can be used to statically configure Low Level (Level 1) interrupt vector tables with the peripheral interrupt handlers as described above for MicroBlaze. The only limitation is that LibGen does not automatically configure interrupt controller interrupt handler to be the exception handler for the PowerPC. The user has to register the interrupt controller handler as the exception handler.

Thus, for low level handlers, users can take advantage of LibGen's configuration of peripheral handlers and interrupt controller vector table. For more information on using the exception handlers in the PowerPC, please refer Chapter 29, "Stand-Alone Board Support Package".