Embedded System Tools Guide

Embedded Development Kit

EDK (v3.1 EA) September 24, 2002

XILINX®

Embedded System Tools Guide
EDK (v3.1 EA) September 24, 2002

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.

Preface: About This Guide

Chapter 1: Embedded System Tools (EST) Architecture

Chapter 2: Xilinx Platform Studio (XPS)

Chapter 3: Platform Generator

Chapter 4: Simulation Model Generator

Chapter 5: Bus Functional Model Generator

Chapter 6: Library Generator

Chapter 7: Format Revision Tool

Chapter 8: Platform Specification Format Utility

Chapter 9: GNU Compiler Tools

Chapter 10: GNU Debugger

Chapter 11: Xilinx Microprocessor Debugger

Chapter 12: Platform Specification Format (PSF)

Chapter 13: Microprocessor Hardware Specification (MHS)

Chapter 14: Microprocessor Peripheral Description (MPD)

Chapter 16: Black-Box Definition (BBD)

Chapter 17: Microprocessor Verification Specification (MVS)

Chapter 18: Microprocessor Software Specification (MSS)

Chapter 19: Microprocessor Driver Definition (MDD)

Chapter 20: Xilinx Libraries

Chapter 21: LibXil Standard C Libraries

Chapter 22: LibXil File

Chapter 23: LibXil Memory File System

Chapter 24: LibXil Net

Chapter 25: LibXil Kernel

Chapter 26: Device Drivers

Chapter 27: Stand-Alone Board Support Package

Chapter 28: Address Management

Chapter 29: Interrupt Management

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
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

Resource	Description/URL
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
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment http://www.support.xilinx.com/xlnx/xil_tt_home.jsp
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**
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.

Convention	Meaning or Use	Example
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 (EST) Architecture

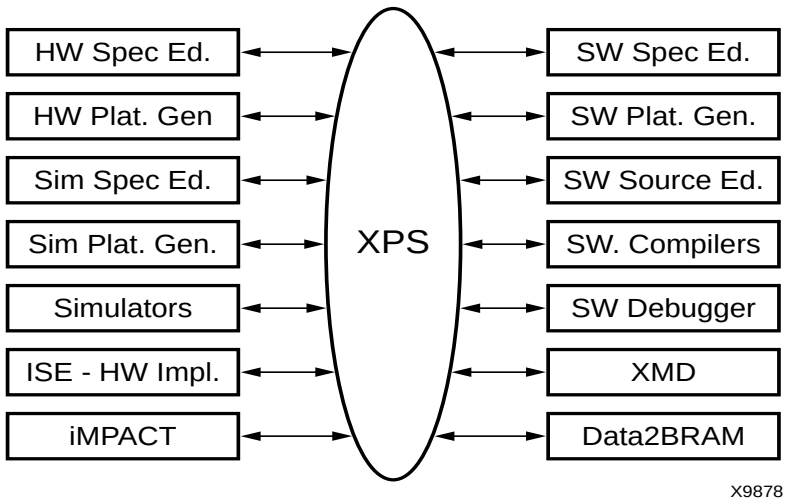
Summary

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:

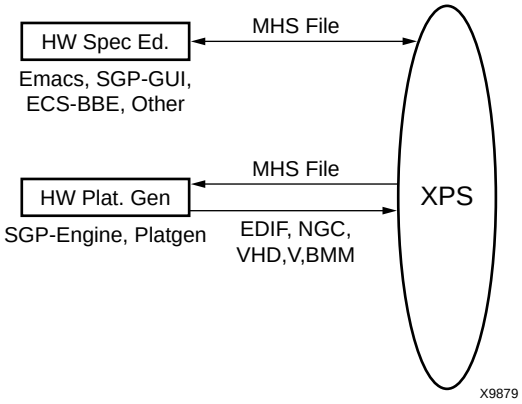
- hardware platform creation,
- hardware platform verification (simulation),
- software platform creation,
- software application creation, and
- software verification (debugging).

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 13, "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 EST tools. Users can define their own peripherals and include them in the MHS by following the guidelines in Chapter 12, "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

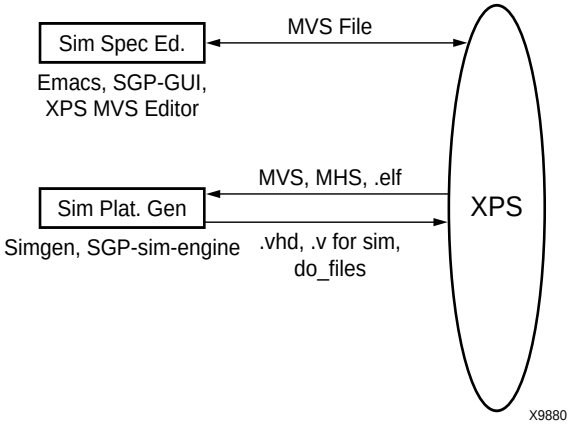
components to the automatically generated hardware platform. See Chapter 3, "Platform Generator," for more information.

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 Data2BRAM 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 MVS (Microprocessor Verification Specification) file is a simple text file, and can be created using any text editor. See Chapter 17, "Microprocessor Verification Specification (MVS)" for more information. XPS provides a GUI based method to create this file. The MVS 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 4, "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 MVS and 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.**



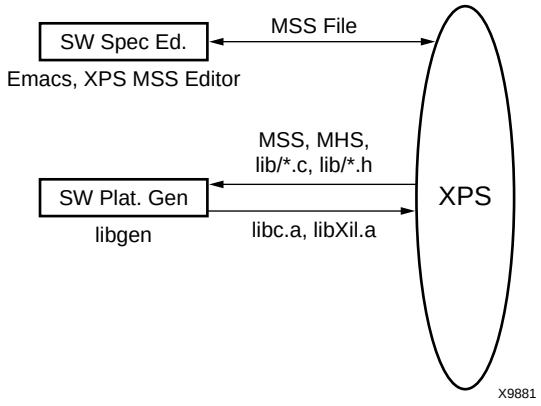
Software Platform Creation

The software platform is defined by the MSS (Microprocessor Software Specification) file (see Chapter 18, "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 (XPS)" 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 6, "Library Generator" for more information. The entire process of creating the software platform is shown in Figure 1-4.

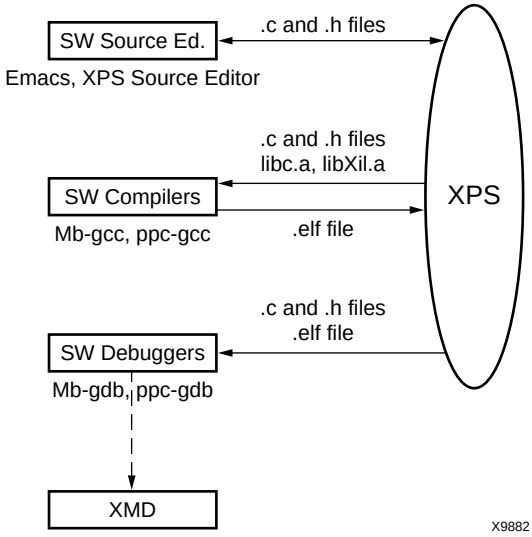
Figure 1-4: **Software Platform**



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 9, "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 11, "Xilinx Microprocessor Debugger" for more information on XMD and Chapter 10, "GNU Debugger" for more information on GDB.

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



Some Useful Tools

Xilinx Platform Studio

Once the hardware platform is defined and a stable MHS file is available, the Xilinx Platform Studio (XPS) tool provides a GUI for creating an MSS file for the 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 (XPS)” for more information. XPS is available only on Windows platform in this release.

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 3, “Platform Generator” for more information.

HDL Synthesis

Platgen generates hierarchal EDIF 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) and Synplify.

Platform Generator produces a synthesis vendor specific project file. This is done with **`-s`** option. The **`-s`** option builds the synthesis project file of the HDL files that were left untouched in default mode.

If the **`-flat`** option is specified, this synthesis step can be skipped since the top-level is also synthesized automatically.

The **`-i`** option disables IO insertion at the top-level, and also generates the HDL component stub with the name *system*_stub.vhd or *system*_stub.v. This allows the

processor system to be included as a macro in a top-level HDL design. Otherwise, the output from Platform Generator is the top-level netlist.

iSE XST

If Platform Generator is run without the **-flat** option and XST as the synthesis vendor, a synthesis script file for XST is created. This script can be executed under XST using the following command:

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

Synplicity Synplify

If Platform Generator is run without the **-flat** option and Synplicity as the vendor, a synthesis project file for Synplify is written. This project can be executed under Synplify using the following command:

```
synplify system.prj
```

Simulation Model Generator

The Simulation Platform Generation tool (simgen) generates and configures various simulation models for the hardware. It takes a Microprocessor Verification Specification (MVS) file as input. The MVS file has a reference to MHS file.

Users can specify the simulation tool to be used in MVS. The HDL language in which the simulation models need to be generated can also be specified. For each hardware instance, users can also specify the simulation model to be used. Please refer Chapter 4, “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 6, “Library Generator” for more information. For more information on Libraries and Device Drivers please refer to Chapter 20, “Xilinx Libraries” and Chapter 26, “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 9, "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 9, "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 **DEAFULT_INIT** parameter for that processor to **XMDSTUB** in **MSS** file. This creates a data2bram 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 3.1 release supports Xilinx iSE 5.1 Tools.

Chapter 2

Xilinx Platform Studio (XPS)

Summary

This chapter describes the Xilinx Platform Studio (XPS) tool used for customizing the software flow for the Xilinx Embedded Processors, MicroBlaze and PowerPC.

Overview

Xilinx Platform Studio (XPS) provides an integrated environment for creating the software specification and verification specification files for a Embedded Processor system. It also provides an editor and a project management interface to create and edit source code. The XPS offers software tool flow configuration options. The tools allows user to run the hardware flow. However, limited configuration of the hardware flow is supported. Currently, XPS-GUI is available only on Windows platforms. XPS-Batch is available for both Windows and Solaris users.

Processes Supported

XPS supports the creation of the MSS file (refer to the Microprocessor Software Specification chapter), the MVS file (refer to the Microprocessor Verification Specification chapter), and software tool flows associated with this software specification. It supports customization of software libraries, drivers, interrupt handlers and compilation of user programs. User can also choose the simulation model for the complete system. XPS also aids users in creating a MHS (refer to Microprocessor Hardware Specification chapter) template or add template core instances to an existing MHS file. The user can then edit this MHS file to convert template instances into a valid MHS block. User 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. XPS also supports customizing the hardware flow for the Platform Generation (platgen) tool. It performs process management and dependency checking between the hardware and software tool flow by calling the tools in the correct order using makefile mechanism. Please refer to Figure 2-1.

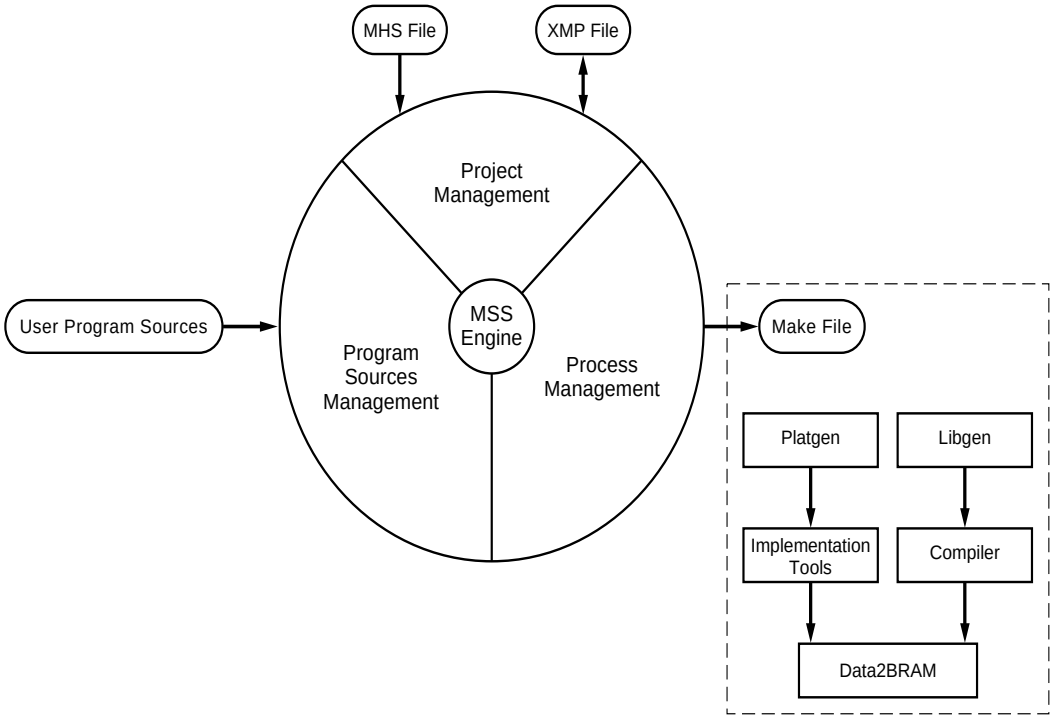


Figure 2-1: **XPS Process**

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 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.	Needs gmake on Solaris.

Features

XPS has the following features

- Adding core templates to Microprocessor Hardware Specification (MHS)
- Generation and modification of the Microprocessor Software Specification (MSS)
- Generation and modification of the Microprocessor Verification Specification (MVS)

- Support for all the tools described in Table 2-1.
- 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 location of the MHS file, the MSS file, the MVS file and the C source and header files that need to be compiled into an executable. The project also includes the FPGA architecture family and the device type for which the hardware tool flow needs to be run. XMP file also contains information about any aspect of the project which is not saved in MHS, MSS or MVS file.

Creating A New Project

A New Project is created using the **New Project** menu option in the Project submenu of the main menu. 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 xmp file resides is considered to be the project directory. Various tools are invoked from the project directory. All relative paths are assumed to be relative to the project directory. Optionally, users can also specify a MHS file to be used for the project. If the specified MHS file does not already exist in the project directory or does not have same name as project name, XPS copies it into the project directory with same base name as project name,

If you have created your hardware specification using SGP, you can specify a SGP Project to be imported instead of the MHS file. XPS will import all the required files from the SGP project directory into the **implementation** subdirectory of XPS project directory. However, before importing a SGP project, please ensure that the netlist generation has been successfully completed in SGP. If you are importing a SGP project, the MHS file **system_padded.mhs** is copied into the project directory, but its name is not changed.

You can also set the target device for which you intend to generate your system. You **must** set the correct target **architecture** before running any tool, since this is needed by all the tools. However, you can defer choosing the device size, the package and the speed grade till you are ready to generate a bitstream. These options can also be set/changed later in the **Set Project Options** dialog box in Options->Project Options menu.

If your MHS uses a peripheral which is not present either in the Xilinx EDK installation area or in **myip** directory of the XPS project directory, you **must** specify a **Peripheral Repository Directory** where the peripheral(s) resides before loading the project. The concept of a Peripheral Repository directory, and its subdirectory structure is explained in detail in PlatGen and LibGen chapters. This corresponds to the **-P option** of the two tools. Please note that all the tools automatically look into the **myip** and **drivers** directories in the project directory and that the project directory should **not** be specified as the Peripheral Repository Directory.

Opening An Existing Project

An existing **XMP** file should be opened and worked on using the **Open Project** menu option (**Project** submenu of Main menu) or using the Open Project button on the toolbar. If you are opening a XPS project that was created by importing a SGP project, XPS will ask you whether you want to copy over the relevant files before continuing with the project. If

there are any changes in the hardware specification or in the netlist which you want to be reflected in the XPS project, click 'Yes' so that XPS can copy over the files again. If you do not have any changes in the SGP project or if you do not want the changes made in SGP project to be reflected in XPS, click 'No' so that XPS continues to use the files which were imported the last time.

New source files and header files can be created and added 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

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

Main Window

The main window appears on the right in the XPS in Figure 2-2. MHS, source and header file editing can be performed in the main window of XPS. Users can also view and edit other text files in the main window. However, MPD, MDD, MSS and MVS files can be opened in a read-only mode. These file types can not be edited from the main window. Any number of files can be opened simultaneously.

Project View Window (Tree View)

This view appears on the left in the XPS window in Figure 2-2. The project view window shows system in a tree format. 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.

Transcript Window (Console)

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. XPS warnings and errors are displayed in blue color while status or informational text appears in black. For tools invoked by XPS, the STDOUT is shown in black while STDERR is shown in blue.

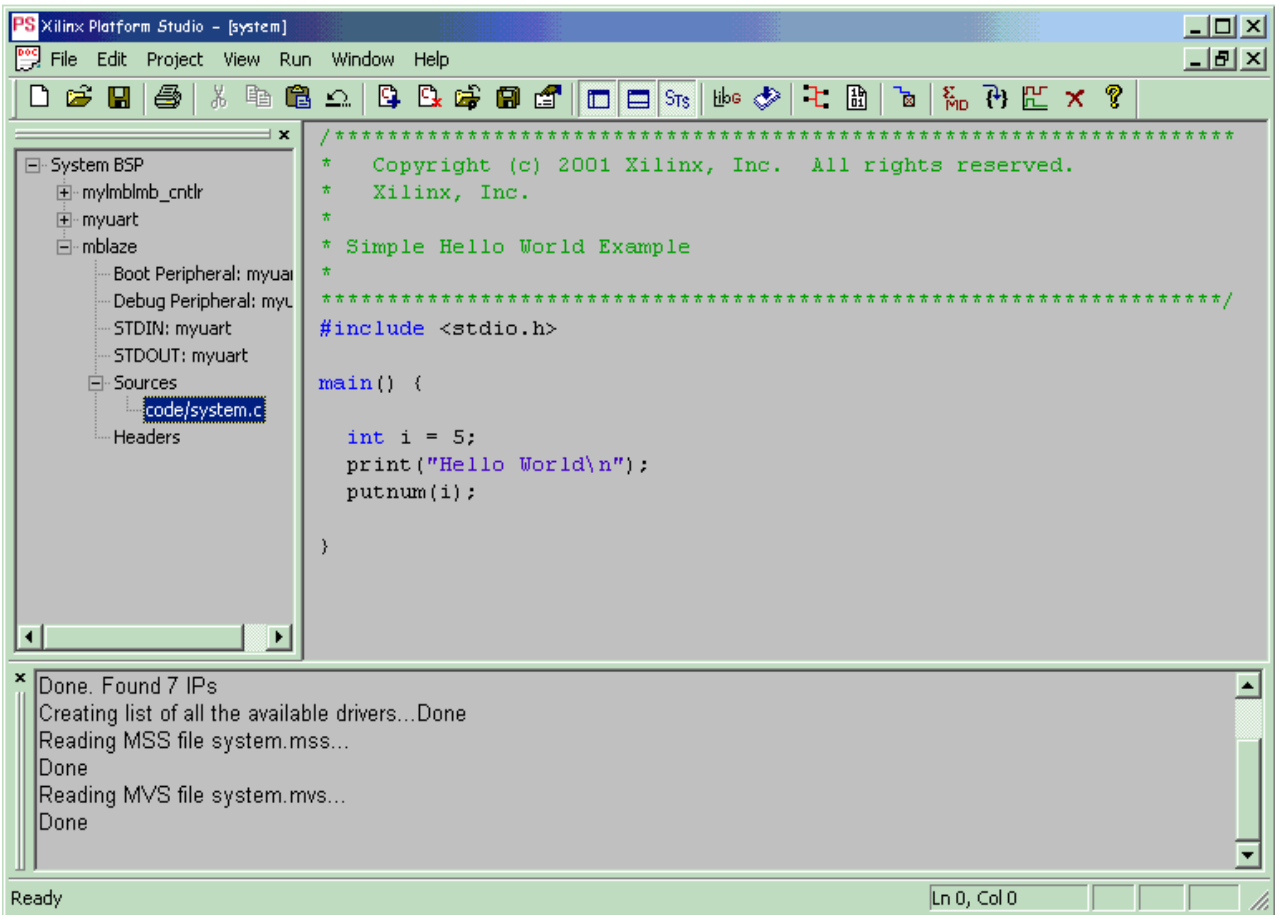


Figure 2-2: **XPS Screenshot**

Platform Management

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

Add Cores

Right click on "**System BSP**" item in the Project View window gives a menu option to "**Add Cores**" to the system. Selecting it brings up a dialog box which 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. If you click on button "**Add to MHS**", a stubbed instantiated of each of the selected core is added to the MHS file and the System BSP Tree is updated. Note that the stubbed instantiations in the MHS file are not complete and must be edited by hand. The window displaying list of available cores can also be brought up by using Project->Add Cores menu item in the Main menu.

Simulation Models

Right click on "**System BSP**" item in the Project View window gives a menu option to set "**Simulation Model**" for the system. User can choose between **Behavioral**, **Structural**, and **Timing** simulation models. The currently selected model has a check mark against it. The MVS file is updated anytime the simulation model is changed.

View MPD

Right click on a instance name give user 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 a instance name give user the option to “View MDD” for driver assigned to that core. If selected, the MDD file for that core is opened in the main window. If the MDD file is already open, focus is set on the file. This option is disabled if no driver is assigned to that core. 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. 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, you can close the project in XPS, hand edit the MSS file to add interrupt handler routine for other ports, and then reload the project.

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 file for the driver is used. XPS only supports driver interface levels 1 and 2. If a different driver interface level is specified, the new value is ignored and the last value for the driver interface level is retained. 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. Currently, XPS supports over-writing only 1 MDD parameter from the GUI. If you want to override any other MDD

parameter, close the XPS project, hand edit the MSS file and then load the project again. For more details, please refer to the chapter on LibGen.

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, Debug and Boot Peripherals. The Mode for a MicroBlaze instance (XMDSTUB, BOOTSTRAP, or EXECUTABLE) can also be specified in this tab. Please note that Debug and Boot peripherals 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 (-l option) 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 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
Boot Peripheral	Instance Name	Designates the peripheral instance as the Boot peripheral
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.
Debut	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.
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.

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

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. However, files of type MSS, MVS, MPD and MDD are opened in read-only mode and can not be edited in XPS editor.

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 your 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. 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. Please note that if your project was created by importing a SGP project, you should not change the target architecture here.

User can also specify the **Peripheral Repository Directory** here. If you change this option here, then you **must close the project and load it again** for the changes to be effective.This option corresponds to the **-P option** of LibGen and PlatGen 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 **-i** and **-ti options** of PlatGen tool.

User can also specify the option to generate netlist in Flat or Hierarchical mode. If hierarchical mode is chosen, user can choose to whether to run synthesis tool ("None") and which synthesis tool to run.

User can also specify the flow to use for running the Xilinx implementation tools. The available options are XPS (Xflow) and ISE (Project Navigator) flow. If the design is a sub-module, user must use the ISE flow. If the design in the top-level (not sub-module) and user chooses to generate netlist in flat mode, then XPS must be used for implementation flow. Only if the design is top-level and user chooses to generate netlist in hierarchical mode, the user can choose between XPS and ISE for implementation tools. 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 you to specify the HDL (VHDL or Verilog) to be used by PlatGen and SimGen. You can also specify the location of the Behavioral, Unisim and Simprim libraries required for simulation. These options are saved into the MVS 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, you 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 ask you 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 not present, XPS creates the etc directory and copies the default version of these two files in that directory from the EDK installation directory. To change options for Xilinx implementation tools, you can modify the two files.

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 four different stages of platform building for which the tools can be invoked

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.
3. **Generate Netlist**: This button calls the platform building tool PlatGen with the correct MHS file and produces the netlist files in NGC format. Please note that if you have imported a **SGP** Project, then you should have already generated the netlist using SGP and this button will not perform any function.
4. **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.
5. **Update Bitstream and Download**: This button invokes 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. So, you can use just a **single-button** to build both hardware and software flows and download their bitstream. File **download.cmd** is expected in etc directory.

XPS generates a makefile in the 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. The user can also save the project in another name by using **Save Project As** in the Project submenu of the Main menu.

ISE Project Navigator Interface

If ISE tools (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 copies any 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 **<proj_name>.bit** and **<proj_name>_bd.bmm** respectively.

Debug and Simulation

You 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)**: You can call the XMD tool. to debug your software. The XMD-button the XPS toolbar opens up a XMD shell in the project directory.
- **Software Debugger:** The debug button calls the software debugger corresponding to the compiler being used for the processor. If you have more than one processor, XPS asks you to choose the processor whose program sources you want to debug.
- **Hardware Simulation Model Generator (SimGen):** You can call 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 start the simulator with the compiled files.

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 and MVS files. It also provides a way to generate makefile. You can also create a XMP project file or load a XMP project file created by the XPS GUI. Please note that unlike XPS-GUI, XPS-Batch is available on both Solaris and Windows platforms.

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] <filename>	Loads the MHS/XMP file and opens/creates XPS project
load [mss|mvs]	Loads in the corresponding file into the project
save [mss|mvs|xmp|make|proj]	Saves the corresponding file. Option proj will save all files

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

Command	Description
xset [dev|package|speedgrade] sgpdir perdir netlist hdl	 Set arch, device, package and speedgrade SGP Project directory Peripheral Repository Directory Hierarchical or Flat netlist option HDL to be used
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 Project

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 names of MSS and MVS files from the XMP file, if specified. Otherwise, it will assume these files based on the XMP file name.

Note that during a single execution of XPS-Batch, you should either create a new project or open an existing one. You should not do both. XPS does not check whether an existing MHS or XMP has already been loaded and doing both might cause unknown results.

Reading MSS and MVS Files

You can read in a MSS or MVS file using command

load [mss | mvs].

Note that you can not specify the name of the file. It is assumed to be project basename with appropriate extension or taken from XMP file. Loading an MSS or MVS 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. However, if you do not specify a new driver, the old driver will be used.

Saving Files and Project

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

save [mss | mvs | xmp | make | proj].

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

Executing Flow Commands

You can run various flow tools by using the command

run option

XPS will run the project's makefile with appropriate targets. The valid options for

Table 3: Options for command run

netlist	Generate netlist
bits	Run Xilinx Implementation tools flow and generate bitstream
libs	Generate software libraries
prog	Compile user program into ELF file(s)
init_bram	Update bitstream with BRAM initialization information
sim	Generate simulation models and run simulator
dow	Download bitstream onto the FPGA
netlistclean	Delete netlist
hwclean	Delete implementation directory
libsclean	Delete software libraries
programclean	Delete ELF file(s)
simclean	Delete simulation directory
clean	Delete all tool generated files and directories

command run are shown in the table.

Closing A Project and Exiting

For closing the project, you can use the command

exit.

This will also 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 and MVS Changes

XPS-batch supports limited MVS or MSS editing. So, if you want to make any changes in these files, you will have to hand-edit the file, make the changes and load it in to XPS. Note that you do not have to close the project. You can save the MSS or MVS file, edit it and then just re-load it into the project by using load [mss | mvs] command.

XMP Changes

XPS-batch also does not support adding of source and header files to a processor. To do so, you must hand-edit the XMP file. To add a source file, open an existing XMP file, and find the line containing '**Processor: <instance_name>**'. Just below this line, introduce a line

containing '**Source: <file_path>**'. For adding a header file, you can add a line containing '**Header: <file_path>**'. Note that you must add a separate line for each source and header file you want to add to a processor instance.

Importing SGP Project

XPS-batch does not have any mechanism of importing an SGP project. To achieve the same, create a **implementation** directory in the directory where you intend to create XPS project. Copy all the files (*.*) in the SGP project directory into this implementation directory. Also copy the file **system_padded.mhs** into your intended XPS project directory. Use the command **'set sgpdir <sgp-dir>'** to set the SGP directory. This will let XPS generate the correct makefile. Now you can load the MHS file system_padded.mhs to create your new project.

Xilinx Microprocessor Project (XMP) File Format

XPS saves user options into Xilinx Microprocessor Project (XMP) file. Those options which are not saved in MSS or MVS files get saved in XMP file. When you open an already existing project, XPS loads these project options from the XMP file. XMP file is a formatted text file which XPS writes when saving a project.

This section describes various fields in XMP file. XMP file is a set of name-value pairs. The format is Field name immediately followed by a colon and then the value of that field. Filed Names can have space in it.

```
Field Name: Value
```

The directory in which the XMP file exists is assumed to be the Project Directory and all paths are assumed to be relative to this directory. For example, on an windows system, if XMP file exists in C:\myprojectdir\system1.xmp, Then the project directory is C:\myprojectdir and the name of the project is system1.

There are two types of fields:

- Global
- Processor Instance Specific

Global fields are those which apply to the complete system. Processor instance specific fields apply to that particular processor instance. We will first discuss Global Fields.

Peripheral Repository Directory

You can specify a Peripheral Repository Directory location as follows:

```
UsePeriphRepos: 1
PeriphReposDir: <dir_path>
```

Field UsePeriphRepos specifies whether to use any specified PeriphReposDir or not. A value of 0 indicates not to use the directory. Default value is 0.

Field PeriphReposDir specifies the Directory location. Note that these fields should specified before MHS File location.

MHS File Location

```
MHS File: <MHS file location>
```

If the MHS File does not exist in the project directory with same base name as project name, then XPS copies that MHS file into this name and location.

MSS File Location

```
MSS File: <MSS file location>
```

MSS Files are created by XPS in the project directory with project name as base.

MVS File Location

```
MVS File: <MVS file location>
```

MVS Files are created by XPS in the project directory with project name as base.

Project Navigator Options

```
UseProjNav: 1
AddToNPL: 1
NPL File: <ProjNav project file location>
```

The UseProjNav field specifies whether the XPS project should use Project Navigator or Xflow for implementation tools. A value of 0 indicates to use Xflow, otherwise ProjNav. The AddToNPL field specifies that if using ProjNav, whether XPS should overwrite the existing NPL file or add modules to the existing file. A value of 0 indicates that any existing NPL file should be overwritten. A value of 1 indicates that the NPL file already exists and XPS should add modules to the existing project. The Project Navigator Project (NPL) file location is specified by NPL File field.

SGP Project Location

```
SGP Dir: <SGP Proj directory location>
```

SGP Project was imported from this location.

Xilinx Target Family and Device

```
Architecture: <Target Family>
Device: <Target Device Name>
Package: <Package Name>
SpeedGrade: <Speed Grade>
```

The valid strings for target architecture families are: virtex2, spartan2, spartan2e, virtex, virtexe, and virtex2p. The field Device specifies the target device name. For example, if you are targeting Virtex2 100, then device name should be xc2v100. The field Package specifies the device package and SpeedGrade specifies the speed grade of the device. You have to make sure that you specify a valid device, package and speed grade for the specified family and device.

Netlist and Synthesis Tools Option

```
HierMode: 0
SynProj: 2
```

HierMode corresponds to the PlatGen option of whether to generate netlist in hierarchical mode or flat mode. A value of 0 (default) means netlist should be generated in flat mode. If hierarchical mode of netlist generation of chosen, then option SynProj specifies which synthesis tool script file is to be generated. Valid values are between 0 and 5. A value of 0 specifies not to generate any synthesis script file. Default value is 2, which generates script for XST.

Design Hierarchy

```
InsertNoPads: 0
TopInst: inst_system
```

This option specifies the design hierarchy. This corresponds to the PlatGen option of whether to insert pads at the toplevel of netlist. A value of 0 (default) specifies that pads should be inserted, any other value means that this is not the top-level and pads should not be inserted. If this is not the toplevel, TopInst specifies the instance name give to this design in the top-level module.

We will now discuss Processor Specified Fields.

Processor Instance

```
Processor: myppc
```

This field should be used atleast once before specifying any processor specific fields. For example, if you have a PowerPC instance called “myppc”, then you would use the above line to indicate that processor specific fields on following lines in XMP files apply to instance myppc. This field makes the instance name the current processor in XMP file. Then, all processor specific fields following this line will apply to the current processor instance until another line specifying a different processor instance is specified in the XMP file. Then that processor instance becomes the current processor instance.

Source and Header Files

```
Header: code/sys1.h
Source: code/a.c
Header: code/sys2.h
Source: code/b.c
```

The field Source specifies a source file for the current processor instance. The field Header specifies a header file for the current processor instance. If you have multiple source or header files, you should add one line for each file.

Compiler Flow

```
CompilerFlow: 3
```

This field specifies how far the compiler flow should be run. Valid values are between 0 and 3. The values correspond to the following flow:

- 0: Preprocess Only
- 1: Preprocess and Compile
- 2: Preprocess, Compile and Assemble
- 3: Preprocess, Compile, Assemble and Link.

Default value is 3, which means the compiler flow is run to the end.

Compiler Optimization Level

```
CompilerOptLevel: 2
```

This field specifies the compiler optimization level. Valid values are between 0 and 3, where 0 corresponds to no optimization and 3 corresponds to maximum optimization.

Use Hard Multiplier

```
HardMul: 0
```

This field specifies to use hard multiplier available on Virtex2 and Virtex2P devices for MicroBlaze instance.

Global Pointer Optimization

```
GlobPtrOpt: 0
```

This field specifies whether to use Global Pointer Optimization during compilation of program sources for the current processor. Value of 0 (default) indicates not to perform this optimization.

Debugging Information

```
DebugSym: 0
```

This field specifies whether to compile the program sources with debugging information or not. Value of 0 (default) indicates not to generate debugging information.

Compiler Search Path

```
SearchComp: ./ ../
```

This field specifies various directories (separated by space) for compiler search path (-B option).

Link Library Search Path

```
SearchLibs: ./ ../
```

This field specifies various directories (separated by space) where the linker should look for libraries for the program sources (-L option).

Include Files Search Path

```
SearchIncl: ./ ../
```

This filed specifies various directories (separated by space) where the compiler should look for various include files for the program sources (-I option).

Libraries to link

```
LFlags: a b
```

This field specifies libraries to be linked (separated by space) for compiling the program sources (-l option).

Preprocessor Options

```
PrepOpt:
```

This field specifies various options (separated by space) to be passed on to the preprocessor (-Wp option).

Assembler Options

```
AsmOpt:
```

This field specifies various options (separated by space) to be passed to the assembler (-Wa option).

Linker Options

```
LinkOpt:
```

This field specifies various options (separated by space) to be passed to linker (-Wl option).

Program Start Address

```
ProgStart: 0xffffc000
```

This field specifies the program start address for your application software. This field is ignored if dcc is the compiler.

Stack Size

```
StackSize: 0x400
```

This field specifies the stack size for your application software. This field is ignore if dcc is the compiler.

Heap Size

```
HeapSize: 0x400
```

This field specifies the heap size for your application software. This field is valid only for a PowerPC instance. This field is ignored if dcc is the compiler.

Linker Script

```
LinkerScript: <file_name>
```

This field specifies the linker script file to be used for compiling program sources.

Other Compiler Flags

```
ProgCCFlags: -save-temps
```

This field specifies various options to be passed to the top level compiler wrapper. You can use this field to specify those options which you could not specify through other fields.

Chapter 3

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:

Note: The EDK offers a format revision tool, RevUp, that converts any old MHS format to the new format. Please see Chapter 7, "Format Revision Tool" for more information.

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 Platform Generator as follows:

```
platgen system.mhs
```

Tool Options

The following are the options supported in the current version:

-a (Architecture family)

The **-a** option allows you to target a specific architecture family. The default family is virtex2.

-flat (Generate a flatten implementation netlist file)

The **-flat** option generates a flattened mplementation netlist file. A synthesis project file is not created.

By default, Platform Generator runs in hierarchal mode. In hierarchal mode, Platform Generator generates hierarchal implementation netlists. This means that each instance of a defined peripheral in the MHS file is synthesized. The default mode leaves the top-level HDL file untouched allowing you to synthesize it in any synthesizer of your choice. Currently, Platform Generator only supports XST and Synplify.

-h (Help)

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

-i (Do not insert IOs at top-level)

The **-i** option disables IO insertion at the top-level. This allows the processor system to be included as a macro in a top-level design. Otherwise, the output from Platform Generator is the top-level design.

-l (Specify the HDL format)

The **-l** option allows you to specify the HDL format. The default value is vhdl.

Options: [vhdl, verilog]

-p (Specify the Project Directory)

The **-p** option allows you to specify the project directory path. The default is the current directory.

-P (Peripheral repository load path)

The **-P** option allows you to specify the peripheral repository load path.

-s (Generate synthesis vendor project file)

With the -s option, Platform Generator produces a synthesis vendor specific project file. The **-s** option builds the synthesis project file for you of the HDL files that were left untouched in default mode (that is, not specifying the -flat option). The only supported values are 0, 2, and 4. The default value is 2.

Options: [0, 1, 2, 3, 4]

0 - None

1 - Exemplar - Leonardo

2 - iSE - XST - SCR / PRJ file

3 - Synopsys - FPGA Express

4 - Synplicity - Synplify - PRJ file

-v (Display version)

The -v option displays the version and quits.

Load Path

Refer to Figure 3-1 for a depiction of the peripheral directory structure. On a UNIX system, the processor cores reside in the following location:

$XILINX_EDK / hw / coregen / ip / xilinx / pcores* / com / xilinx / ip2 / processor

On a PC, the processor cores reside in the following location:

%XILINX_EDK% \ hw \ coregen \ ip \ xilinx \ pcores* \ com \ xilinx \ ip2 \ processor

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

- Current directory (where Platform Generator was launched; not where the MHS resides)
- Set the Platform Generator -P option, or the XIL_MYPERIPHERALS environment variable

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

1. Search current directory in the myip directory
2. Search $XIL_MYPERIPHERALS / myip (UNIX) or %XIL_MYPERIPHERALS% \ myip (PC)
3. Search $XILINX_EDK / hw / coregen / ip / xilinx / pcores* / com / xilinx / ip2 / processor (UNIX) or %XILINX_EDK% \ hw \ coregen \ ip \ xilinx \ pcores* \ com \ xilinx \ ip2 \ processor (PC)

The first two search areas (1 and 2) have the same underlying directory structure. The third search area has the CORE Generator directory structure. For search areas 1 and 2, the peripheral name is the name of the root directory. From the root directory, the underlying directory structure is as follows:

```
data
hdl/verilog
hdl/vhdl
simmodels
```

For example, if the XIL_MYPERIPHERALS environment is set, then the MPD, BBD, and PAO files are found in the following location:

$XIL_MYPERIPHERALS / myip / <*peripheral*> / data (UNIX)

%XIL_MYPERIPHERALS% \ myip \ <*peripheral*> \ data (PC)

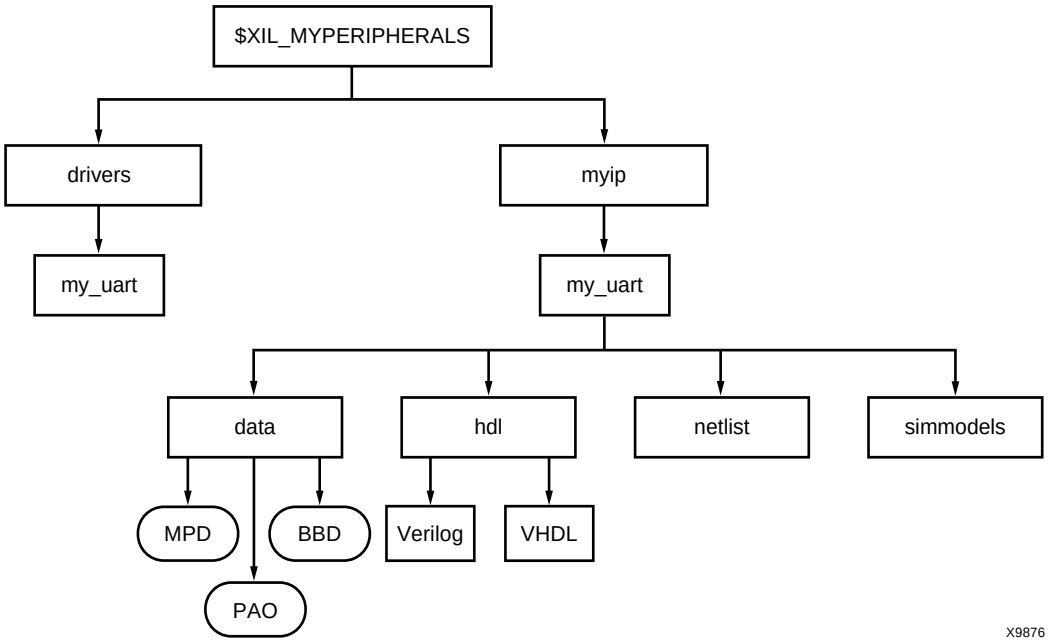


Figure 3-1: **Peripheral Directory Structure**

Output Files

Platform Generator 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 processor and its peripherals.

Implementation Directory

The implementation directory contains the following:

system.ngc

This is the top level implementation netlist of the processor and its peripherals. Only created if the **-flat** option is given.

*peripheral*_wrapper.ngc

Implementation netlist file of the peripheral. Only created if the **-flat** option is not given.

Synthesis Directory

The synthesis directory contains the following:

system.[prj|scr]

Synthesis project file.

About Memory Generation

Platform Generator generates the necessary banks of memory and the initialization files for the BRAM Block (bram_block_v1_00_a). The BRAM Block is coupled with a BRAM controller.

Current BRAM controllers include the following:

- DSOCM BRAM Controller (dsbram_if_cntlr_v1_00_a) - PowerPC only
- ISOCM BRAM Controller (isbram_if_cntlr_v1_00_a) - PowerPC only
- LMB LMB BRAM Controller (lmb_lmb_bram_if_cntlr_v1_00_a) - MicroBlaze only
- LMB OPB BRAM Controller (lmb_opb_bram_if_cntlr_v1_00_a) - MicroBlaze only
- OPB BRAM Controller (opb_bram_if_cntlr_v1_00_a)
- PLB BRAM Controller (plb_bram_if_cntlr_v1_00_a)

For the BRAM controllers the MHS options, C_BASEADDR and C_HIGHADDR (see the Chapter 13, “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 3-1: **Predefined Memory Sizes**

Architecture	Memory Size (kBytes) 32-bit byte-write	Memory Size (kBytes) 64-bit byte-write
Spartan-II	2, 4	4, 8
Spartan-IIE	2, 4, 8	4, 8, 16
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.

Platform Generator creates four blocks of memory. Each bank of memory is byte addressable (8 bits wide). Depending on the pre-defined memory size, each bank will contain one or more Select BlockRAMs.

For example, for a memory size of 4 kBytes on a Virtex device, Platform Generator creates four banks of memory. Each bank is 8 bits wide and 1 kBytes deep. This configuration uses eight Select BlockRAMs, two Select BlockRAMs for each bank.

MHS Example (LMB LMB Controller with BRAM Block)

The following is an example of the LMB LMB Controller with BRAM Block:

```
##############################################################################
BEGIN lmb_lmb_bram_if_cntlr
PARAMETER INSTANCE = mylmblmb_cntlr
PARAMETER HW_VER = 1.00.a
PARAMETER C_BASEADDR = 0x00000000
PARAMETER C_HIGHADDR = 0x00000fff
BUS_INTERFACE ILMB = i_lmb
BUS_INTERFACE DLMB = d_lmb
BUS_INTERFACE PORTA = lmb_porta
BUS_INTERFACE PORTB = lmb_portb
END
##############################################################################
BEGIN bram_block
PARAMETER INSTANCE = bram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = lmb_porta
BUS_INTERFACE PORTB = lmb_portb
END
```

Reserved MHS Attributes

The Platform Generator 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 3-2: **Automatically Expanded Reserved Parameters**

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_NUM_MASTERS	Number of OPB masters
C_NUM_SLAVES	Number of OPB slaves
C_DCR_AWIDTH	DCR Address width
C_DCR_DWIDTH	DCR Data width

Table 3-2: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_DCR_NUM_SLAVES	Number of DCR slaves
C_LMB_AWIDTH	LMB Address width
C_LMB_DWIDTH	LMB Data width
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

Current Limitations

The current limitations of the Platform Generator flow are:

- Vector slicing is not allowed.

Chapter 4

Simulation Model Generator

Summary

This chapter describes the Simulation Model Generator utility usage.

Overview

The Simulation Model Generation tool (SimGen) generates and configures various simulation models for a specified hardware. It takes a Microprocessor Verification Specification (MVS) file as input. MVS files have a reference to an MHS file that describes the hardware. The simulation tool is specified in the MVS file. The HDL language in which the simulation models need to be generated can also be specified. For each hardware instance, you can specify the simulation model. Please refer to Chapter 17, "Microprocessor Verification Specification (MVS)" for more information about the MVS file.

The hardware component is defined by the Microprocessor Hardware Specification (MHS) file. Please refer to Chapter 13, "Microprocessor Hardware Specification (MHS)" for more information.

SimGen produces a simulation model and a compilation script for vendor specific simulators.

About Simulation

This section introduces the basic facts and terminology of HDL simulation. There are three stages in the FPGA design process in which you conduct simulation.

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 you get the required functionality. 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.

EDK Library

Used for behavioral simulation. It contans all the EDK IP components, precompiled for ModelSim.

UNISIM Library

Used for behavioral simulation and contains default unit delays. This library includes all of the Xilinx Unified Library components that are inferred by most popular synthesis tools. The UNISIM library also includes components that are commonly instantiated such as I/Os and memory cells.

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

SIMPRIM Library

Used for structural and 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.

Tool Requirements

Set up your system to use the Xilinx ISE 5.1 tools. 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

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

For example,

```
simgen [options] system_name.mvs
```

Note: SimGen will generate simulation models for platforms generated by Platform Generator. PlatGen should be executed before SimGen to generate all the files that SimGen uses.

Tool Options

The following options are supported in the current version:

-a (Architecture family)

The **-a** option allows you to target a specific architecture family.

Usage: -a <architecture>

Options: { spartan2 | spartan2e | virtex | virtexe | virtex2 | virtex2p }

Default: virtex2

-h (Help)

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

-f (Flat)

The **-f** option specifies that a flat EDIF file should be used for structural simulation. If it is not specified, hierarchical EDIF files will be used.

-i (Initialize)

The **-i** option allows memory initialization of previously created simulation models. If this option is specified, only initialization will be performed and no simulation models will be genrated.

Usage: -i <program>

-l (Language)

The **-l** option allows you to specify the HDL Language. This option will override the language specified in the MVS file.

Usage: -l <language>

Options: { vhdl | verilog }

Default: vhdl

-m (Simulation model type)

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). This option will override the simulation model specified in the MVS file.

Usage: -m <sim_model>

Options: { beh | str | tim }

Default: beh

-p (Project Directory)

The **-p** option allows you to specify the project directory path. The default is the current directory.

Usage: -p <path>

-s (Simulator)

The **-s** option allows you to specify for which simulator to produce a compilation script file. The supported simulators are Model Technology ModelSim (mti) and Cadence Verilog-XL (vxl). This option will override the simulator specified in the MVS file.

Usage: -s <simulator>

Options: { mti | vxl }

Default: mti

-v (Version)

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

-L (ModelSim Behavioral Library Path)

The **-L** option allows you to specify the ModelSim Behavioral Library directory path. This option will override the value specified in the MVS file.

Usage: -L <path>

-S (ModelSim Simprim Library Path)

The **-S** option allows you to specify the ModelSim Simprim Library directory path. This option will override the value specified in the MVS file.

Usage: -S <path>

-U (ModelSim Unisim Library Path)

The **-U** option allows you to specify the ModelSim Unisim Library directory path. This option will override the value specified in the MVS file.

Usage: -U <path>

Input files

SimGen searches for files in the following directories located in the project directory. These directories are created by Platform Generator:

```
<project_directory>/hdl/
<project_directory>/implementation/
```

HDL Directory

The hdl directory should contain the following:

*system*_name.[*vhd*|*v*]

This is the top level HDL file of the processor and its peripherals. Used for behavioral and for hierarchical structural simulation models.

*peripheral*_wraper.[*vhd*|*v*]

These are the wrapper HDL source files of each peripheral. Used for behavioral simulation.

Implementation Directory

The implementation directory should contain the following files. Depending on the simulation model to be used, only appropriate files will be taken.

peripheral_wrapper.ngc

Netlist file of each peripheral. Created by Platform Generator if the **-flat** option is not given. Used to generate hierarchical structural simulation models.

system_name.ngc

System netlist file. Created by Platform Generator if the **-flat** option is given. Used to generate flat structural simulation models.

system_name.ncd

System EDIF file. Created by Platform Generator if the **-flat** option is given. Used to generate timing simulation models.

Output Files

SimGen produces all simulation files in the simulation directory within the project directory.

```
<project_directory>/simulation/
```

Simulation Directory

The simulation directory contains the following:

peripheral_wrapper.[*vhd*|*v*]

Post-synthesis simulation files.

system_name.[*vhd*|*v*]

The top level HDL file of the processor and its peripherals.

system_name.do

The compilation script for the specified simulator.

Memory Initialization

Platform Generator creates the necessary banks of memory for a system. The corresponding memory simulation models generated by SimGen can be initialized with data using the -i option.

To initialize memory of simulation models already crated by SimGen with a compiled executable, you need:

- A compiled executable
- A simulation model for your system
- A BMM file for your system

The compiled executable is generated with the appropriate gcc compiler or assembler, from corresponding C or assembly source code. The simulation model is generated previously by executing PlatGen and then SimGen. The BMM file is a memory description file that allow memory initialization and is created by PlatGen in the implementation directory.

Verilog

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

```
simgen -l verilog -i executable system.mvs
```

This command takes an executable file as input to generate the verilog memory initialization file *system_init.v*. This file is used along with your system to initialize memory.

VHDL

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

```
simgen -i executable system.mvs
```

This command takes an executable file and MVS file as input to generate the VHDL system configuration un the file *system_init.vhd*. This file is used along with your system to initialize memory.

Current Limitations

SimGen does not support generation of mixed level simulation models.

XILINX®

Chapter 5

Bus Functional Model Generator

Summary

This chapter describes the Bus Functional Model Generator utility.

Overview

The Bus Functional Model Generation tool (BfmGen) generates and configures a peripheral under test for simulation and verification using the IBM CoreConnect Toolkit. The input to the tool is the MPD file corresponding to the peripheral under test. Further options such as the preferred language for the bus functional model generation, the bus interface to perform the test on and the path to the IBM CoreConnect Toolkit may be specified. BfmGen creates the bus functional model of the peripheral under test in the preferred language that may then be used for simulation using the IBM CoreConnect Toolkit.

For more information on the MPD format, please refer Chapter 14, “Microprocessor Peripheral Description (MPD)”. For information on using the IBM CoreConnect Toolkit, please refer the *IBM CoreConnect Toolkit User’s Manual.*

Tool Requirements

BfmGen requires a valid MPD file as input. A valid license for using the IBM CoreConnect Toolkit is required for simulation and generation of the top level testbench. BfmGen supports the IBM OPB ToolKit Version 2.0.X,the IBM PLB Toolkit Version 4.X for 64-bit PLB data bus and the IBM DCR Toolkit Version 2.X.

Tool Usage

The BfmGen tool is invoked as follows:

```
bfmgen [options] peripheral_name.mpd
```

Tool Options

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

-h (Display Help)

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

-v (Display Version)

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

-b (Bus Interface Name)

The **-b** option defines the bus interface for which the Bus Functional Model needs to be generated. This option must be specified when the MPD file for the peripheral under test has more than one bus interface defined.

-lang (HDL Language)

The **-lang** option defines the Hardware Description Language to use for BfmGen. Valid options are **ver** (for verilog) and **vhdl** (for vhdl). This option defaults to **vhdl**.

-tk (IBM CoreConnect Toolkit Dir Path)

The **-tk** option specifies the path for the IBM CoreConnect Toolkit. A valid license is required to use the toolkit. If this option is specified, the peripheral under test is declared and instantiated in the CoreConnect Toolkit testbench. The toolkit directory must have either the vhdl or the verilog subdirectory based on the preferred language for BfmGen output.

Input files

Bfmgen requires the MPD file corresponding to the peripheral under test as input.

Output Files

Bfmgen produces the Bus Functional Model for the peripheral under test based on the **-tk** option and the **-lang** specified to the tool.

When the **-tk** option is not specified and the **-lang** option is **vhd**, the following files are produced.

- **vhdl/*peripheral*_wrapper.vhd**: VHDL wrapper file for the peripheral under test.
- **vhdl/*peripheral*_comp.vhd**: VHDL file that contains the component and signal declarations of the peripheral under test to be inserted in the component declaration section in the test bench of the IBM CoreConnect Toolkit.
- **vhdl/*peripheral*_inst.vhd**: VHDL file that contains the instance of the peripheral under test to be inserted in the architecture body in the test bench of the IBM CoreConnect Toolkit

When the **-tk** option is not specified and the **-lang** option is **ver**, the following files are produced.

- **verilog/*peripheral*_wrapper.v**: Verilog wrapper file for the peripheral under test
- **verilog/*peripheral*_wire.v**: Verilog file containing signal declarations to be inserted in the test bench module.
- **verilog/*peripheral*_inst.v**: Verilog file that contains the instance of the peripheral under test to be inserted in the test bench module of the IBM CoreConnect Toolkit

When the **-tk** option is specified and the **-lang** option is **vhd**, the following files are produced.

- **vhdl/*peripheral*_wrapper.vhd**: VHDL wrapper file for the peripheral under test
- **vhdl/*tk_tb*.vhd**: Modified VHDL Toolkit testbench file with component declaration and instantiation of the peripheral under test.

When the **-tk** option is specified and the **-lang** option is **vhd**, the following files are produced.

- **verilog/*peripheral*_wrapper.v**: Verilog wrapper file for the peripheral under test
- **verilog/*tk_tb*.v**: Modified Verilog Toolkit testbench file with component declaration and instantiation of the peripheral under test.

Using BfmGen and IBM CoreConnect Toolkit

In order to use the output files generated by BfmGen for simulation and verification using the IBM CoreConnect Toolkit, do the following:

1. Copy all the toolkit vhdl or verilog files required for simulation, except the top-level testbench file in the vhdl or verilog directory created by BfmGen.
2. Describe the bus transactions to be simulated in the IBM Bus Functional Language (BFL).
3. Compile the BFL using the IBM Bus Functional Compiler (BFC) and set the appropriate simulation target.
4. Simulate and verify the functionality of the peripheral under test for the given bus interface.

Current Limitations

The current limitations of BfmGen are:

- The latest release of the IBM PLB 4.X toolkit does not contain the top level test bench for the 64-bit data bus version. BfmGen in this case ignores the **-tk** option, if specified.

XILINX®

Chapter 6

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 18, "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 7, "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, -ver (Display version information)

This option displays the version number of LibGen.

-a, -arch *family_name* (Architecture family)

This option defines the target architecture family. *Family_name* can be one of spartan2, spartan2e, virtex, virtexe, virtex2 or virtex2p. The default option is virtex2.

-p, -proj *proj_dir* (Specify project directory)

This option specifies the project directory *proj_dir*. The default is the current directory. All output files and directories are generated in the project directory. This project directory is also called ***USER_PROJECT*** for convenience in the documentation.

-P, -Per_Dir *per_dir* (Specify user peripherals and driver directory)

This option specifies user peripherals and drivers directory. LibGen looks for drivers in the directory ***per_dir*/drivers/**

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

-m, -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 bootstrap: (MicroBlaze only) This mode is used when the user wants to use a bootstub executable to load user programs. The bootstub is created automatically for each processor instance in the MSS file by libgen as the file ***proc_inst_name*/code/bootstub.elf**, relative to the ***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 ***USER_PROJECT*** directory

-x, -xmdstub *proc_inst_name_1* [, *proc_inst_name_2, ...*]

Note: Option valid for MicroBlaze only.

Specifies that one or more processors have their 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.

-b, -bootstub *proc_inst_name_1* [, *proc_inst_name 2, ...*]

Note: Option valid for MicroBlaze only.

Similar in functionality for bootstubs as the **-xmdstub** option.

-e, -executable *proc_inst_name_1* [, *proc_inst_name_2, ...*]

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

-l, -lib

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

-s, -stub

Creates the stub files (for MicroBlaze) and BRAM initialization script **run_download.sh** (solaris) or **run_download.bat** (windows) only. Using this option prevents the generation of libraries and drivers.

-bspgen *proc_inst_name_1* [*, proc_inst_name_2, ...*]

Runs the BspGen utility for each processor instance specified after drivers are configured. The option can be used only when using the PowerPC processor with the **OS** parameter in MSS file defined as VxWorks5_4. Please refer the *BspGen Users Guide* chapter in the *Processor IP Reference Guide* for more information.

-d, -do_not_warn

Disables printing of some warning messages. By default, all warnings are printed.

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

The include directory contains C header files that are needed by drivers. The include file **xparameters.h** is also created by LibGen in this directory. This file defines base addresses of the peripherals in the system, **#defines** needed by drivers 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 Chapter 19, “Microprocessor Driver Definition (MDD)” for more information.

lib

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 20, “Xilinx Libraries”.

libsrc

The libsrc directory contains intermediate files and makefiles that are needed to compile the libraries and drivers. The directory contains peripheral specific driver files that are copied from the EDK and user driver directories. Please refer the *Drivers* section of this document for more information. Note that this directory is overwritten each time libgen is run.

code

The code directory is used as a repository for EDK executables. Libgen creates xmdstub.elf (for MicroBlaze on-board debug) and bootstub.elf (for MicroBlaze bootstrap) in this directory. The code directory can also be used for other user ELF files.

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, please refer Chapter 18, “Microprocessor Software Specification (MSS)”

Drivers

Most peripherals require software drivers. The EDK peripherals are shipped with associated drivers. Please refer Chapter 26, “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.

For each processor in the system, and the connectivity between the processor and peripherals through various buses and bridges is specified in the MHS file. LibGen uses this information to analyze all the peripherals that can be accessed by each processor and customize only those drivers. Libgen copies the necessary files in the driver directory over to the ***USER_PROJECT/processor_instance_name/*****libsrc** directory for each processor in the system and runs **make** for compiling the drivers. The MDD file for each driver specifies all configurable options for the drivers. Please refer Chapter 19, “Microprocessor Driver Definition (MDD)” and Chapter 18, “Microprocessor Software Specification (MSS)” for more information.

Libgen also creates an include file **xparameters.h** in the ***USER_PROJECT/processor_instance_name/*****include** directory. This header file must be included in the driver source files. This file contains peripheral base address definitions and interrupt masks for the peripherals. This file also contains function prototypes and other useful defines. The contents generated in this file can be controlled through the MDD file for each driver.

Users can write their own drivers. These drivers must be in a specific directory under ***USER_PROJECT/*****drivers** or ***per_dir/*****drivers.** 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. Each driver must also contain an MDD file in the **data/** subdirectory. Please refer to the existing EDK drivers to get an understanding of the structure of the drivers.

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. 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 Customization

In the MSS file, the INT_HANDLER parameter allows an interrupt handler routine to be associated with the interrupt signal. Libgen 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, LibGen uses a default dummy handler routine for the peripheral.

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**, Libgen 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 29, "Interrupt Management" for more information.

Boot and Debug Peripherals (MicroBlaze Specific)

These are peripherals that are specifically used to download bootstub and xmdstub. The attributes BOOT_PERIPHERAL and DEBUG_PERIPHERAL are used for denoting the boot and debug peripheral instances. Libgen uses these attributes in xmdstub and bootstrap modes.

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. Please refer Chapter 19, "Microprocessor Driver Definition (MDD)" for more information. 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.

Chapter 7

Format Revision Tool

Overview

The Format Revision Tool (RevUp) reads older data files (MPD, MHS, BBD, PAO, or MSS), and revises them upward. The upgrade is a format update and not an IP upgrade.

Current PSF version is 2.0.0. Previous supported versions include 1.0.0.

The PSF version demands that the current version of EDK be at least as recent as that version, at run time. Therefore, 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

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.[mhs|mpd|bbd|pao|mss]
```

Tool Options

The following are the options supported in the current version:

-h (Help)

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

-p (Specify the Project Directory)

The **-p** option allows you to specify the project directory path. The default is the current directory.

-P (Peripheral repository load path)

The **-P** option allows you to specify the peripheral repository load path.

-v (Display version)

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

Current Limitations

The current limitations of the RevUp flow are:

- For an MHS that includes IP outside of the regularly released EDK peripherals (that is, user IP), RevUp must be run on the MPD, PAO, and BBD before updating the MHS.
- For an MSS file, RevUp must be run only after MHS, MPD, PAO and BBD files have been reved up.

XILINX®

Chapter 8

Platform Specification Format Utility

Summary

This chapter describes the PSF utility tool.

Overview

The PSF Utility (PsfUtil) may be used to generate template MHS specifications for a list of user specified peripherals or to generate a catalog of peripherals available in the repository.

Tool Requirements

Please install the Xilinx EDK tool set before using the PsfUtil.

Tool Usage

Run PsfUtil as follows:

```
psfutil [options]
```

Tool Options

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

-h (Display Help)

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

-v (Display Version)

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

-w (Overwrite output)

The **-w** option overwrites the output file specified.

-iplist (Print all IP with version in the repository)

The **-iplist** option prints the list of all available processor IP in the repository. The output is written to either stdout or the output file.

-mpd2mhs (List of Peripherals for MHS Template Generation)

The **-mp2mhs** option specifies the list of peripherals for which the MHS template is to be generated. The list of peripherals is specified in a text file. The output is written either to stdout or the output file.

-mhs2sch (Generate ECS schematic from MHS specification)

The **-mhs2sch** option specifies the MHS file for which the ECS schematic files are to be generated. The output is a ".sch" file that contains the schematic of the system. The schematic may be viewed using the ECS schematic viewer provided with the Xilinx ISE installation.

-o (Output File)

The **-o** option specifies the output file name.

Note: You must specify only one of -iplist or the -mpd2mhs or the -mhs2sch option.

Input files

PsfUtil requires no input file when using the **-iplist** option.

For generating MHS templates using the **-mpd2mhs** option, PsfUtil requires a text file as input that specifies the peripherals for which MHS templates need to be generated in a text file with each peripheral with the optional version number specified in a separate line. An example text file is shown below

```
opb_v20 1.10.a
opb_uartlite
microblaze
bram_block
lmb_lmb_bram_if_cntlr
lmb_v10
lmb_v10
opb_gpio
```

When no version number is specified, the latest version present in the repository is selected.

For generating the ECS schematic files using the **-mhs2sch** option, PsfUtil requires the MHS file describing the system as input.

Output Files

The output is written to stdout if no -o option is specified, or it is written to the output file specified.

When using the **-iplist** option, the output file lists all the available IPs in the repository with their version numbers.

When using the **-mpd2mphs** option, the output file contains the MHS template specification of all the peripherals listed in the input text file.

When using the **-mhs2sch** option, the schematic files are written into the **SCH/** directory relative to the current working directory.

Chapter 9

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

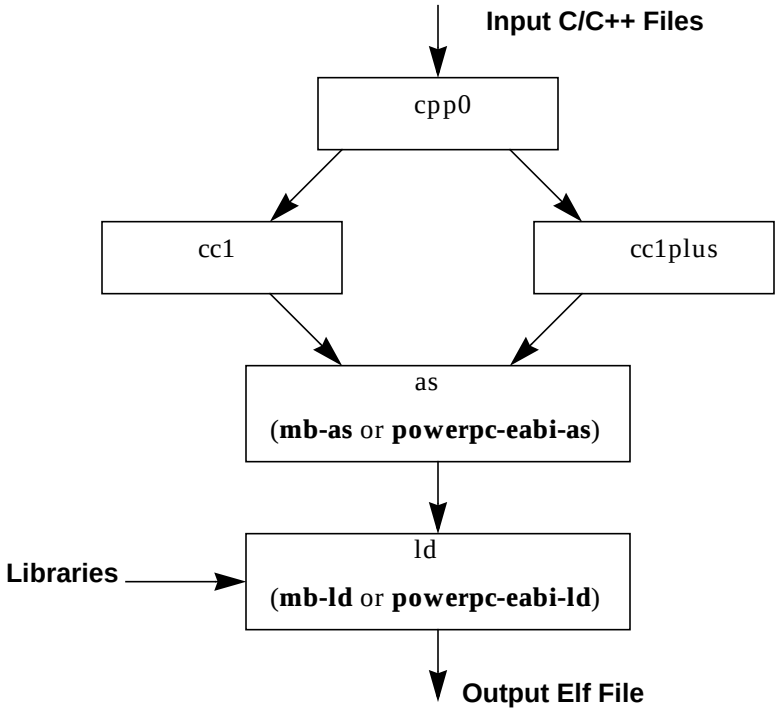


Figure 9-1: **GNU Tool Flow**

This section discusses the common features of both the MicroBlaze as well as PowerPC compiler. Figure 9-1shows 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**)

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 9-1 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 9-1: **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**)
-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.

-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 9-2

Table 9-2: **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 the chapter, "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`

Initialization files are searched in the following order[2]:

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.

1. Processor indicates **powerpc-eabi** for PowerPC and **microblaze** for MicroBlaze

2. 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*

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 9-3 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 9-3: **File Extensions**

Extension	File type
.c	C File
.C	C++ File
.cxx	C++ File
.cpp	C++ File
.c++	C++ File
.cc	C++ File
.S	Assembly File, but might have preprocessor directives
.s	Assembly File with no preprocessor directives

Libraries

Both the compiler (**powerpc-eabi-gcc** and **mb-gcc**) use certain libraries. The following libraries are needed for all the program.

Table 9-4: **Libraries used by the compilers**

Library	Particular
libxil.a	Contain drivers 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

These libraries are customized for every user's project and copied

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 9-3. 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 9-5: **MicroBlaze Specific Options**

Options	Explanation
-xl-mode-executable	Default mode for compilation.
-xl-mode-xmdstub	Intrusive hardware debugging on the board. Should be used only with xmdstub downloaded on to MicroBlaze
-xl-mode-bootstrap	Generate code, which can be downloaded using the boot strap loader
-xl-mode-bootstrap-reset	Same as bootstrap mode, but in this case, on reset, the control is transferred to the user program instead of the boot stub.
-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 multiplier. Use this option when the hardware multiplier is not present in the device. By default this option in turned ON.
-mno-xl-soft-mul	Do not use software multiplier. Compiler generates “mul” instructions.
-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 Options

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-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.

-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. The threshold value can be changed using the **-Gn** option discussed below.

-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.

-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 the chapter, "Xilinx Microprocessor Debugger"

-xl-mode-bootstrap

Certain programs are downloaded using the boot loader onto the device. This option links in crt2.o as the initialization file and starts the program at address location 0x100, leaving the first 100 words for the boot loader program. On a reset, the control is transferred back to the boot stub, which waits for loading a new program in the memory.

-xl-mode-bootstrap-reset

Same as the bootstrap mode above, but the reset location is overwritten to jump your code instead of the boot stub. Using this mode, the user does not have to reload the program on a reset, which is necessary in the previous mode.

-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.

Caution! mb-gcc will signal fatal error, if more than one mode of execution is supplied on the command line.

-Gn

The compiler stores certain data in the small data area of the code. Any global variable, which is equal to or lesser than 8 bytes will be stored in the small data area of the read-write or read-only section. This threshold value of 8 bytes can be changed using the above option in the command line.

MicroBlaze Linker Options

-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, which is 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

Pseudo-Ops

MicroBlaze supports a certain pseudo-ops making assembly programming easier for assembly writers. The supported pseudo-ops are listed in Table 9-6.

Table 9-6: **Pseudo-Opcodes supported by 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

Table 9-6: **Pseudo-Opcodes supported by Assembler**

Pseudo Opcodes	Explanation
neg Rd, Ra	Replace by instruction: **rsub** Rd, Ra, R0
sub Rd, Ra, Rb	Replace by instruction: **rsub** Rd, Rb, Ra

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 five C runtime routines:

crt0.o

This initialization file is to be used for programs which are to be executed standalone, i.e without **xmd**.

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.

crt2.o

In case of programs used with the boot-loader, crt2.o is used as the initialization file. The boot loader is used to load the program at runtime using the boot stub.

crt3.o

The source for crt2.o and crt3.o is the same as the functionality is the same except for the behavior on a reset. In crt3.o, address location 0x0 is overwritten, such that on a reset, the control is transferred to the user program instead of the boot stub.

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 file 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

One of the several tasks performed by operating systems is to pass arguments to a program. Since there is no operating system available for 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 9-7.

1. Functions which have calls to other sub-routines are called non-leaf functions.

Table 9-7: **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 0xffff8000, 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 10

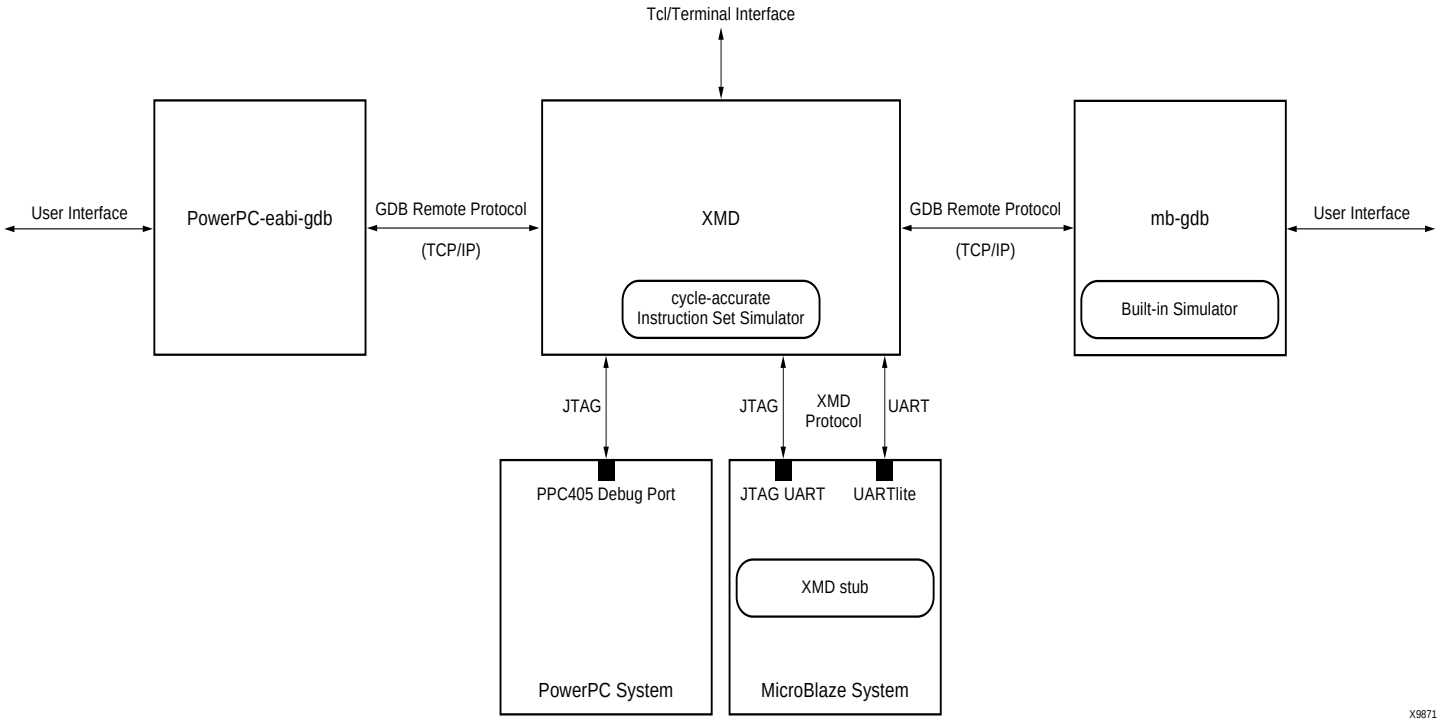
GNU Debugger

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.



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.

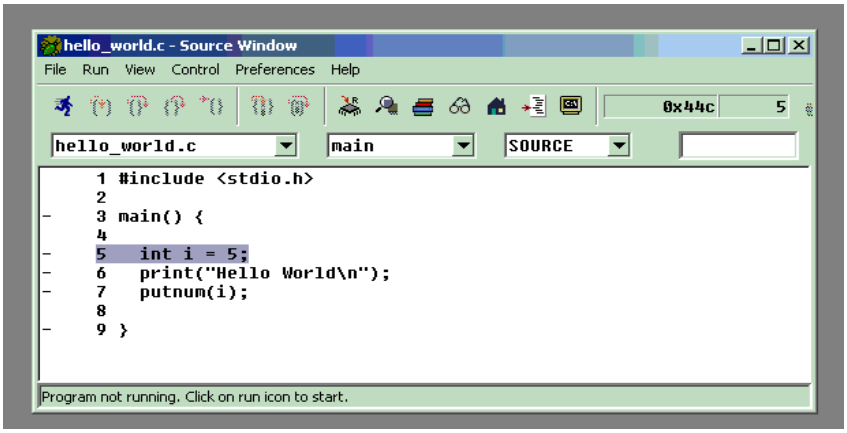
-w

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
```



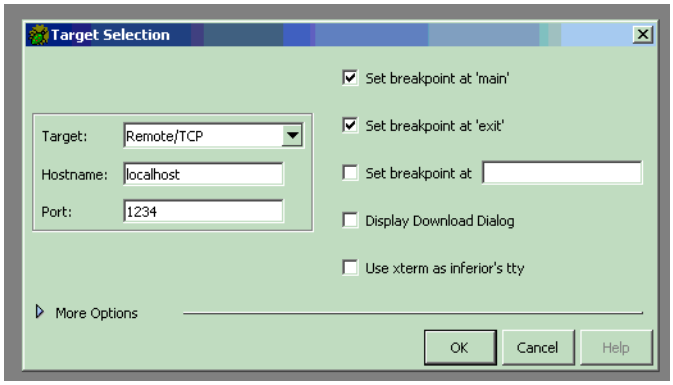
From the **Run** pull-down menu, select **Connect to target** in the mb-gdb window. In the Target Selection dialog, you can choose between the **`Simulator`** (built-in) and **`Remote/TCP`** (for XMD) targets.

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.



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

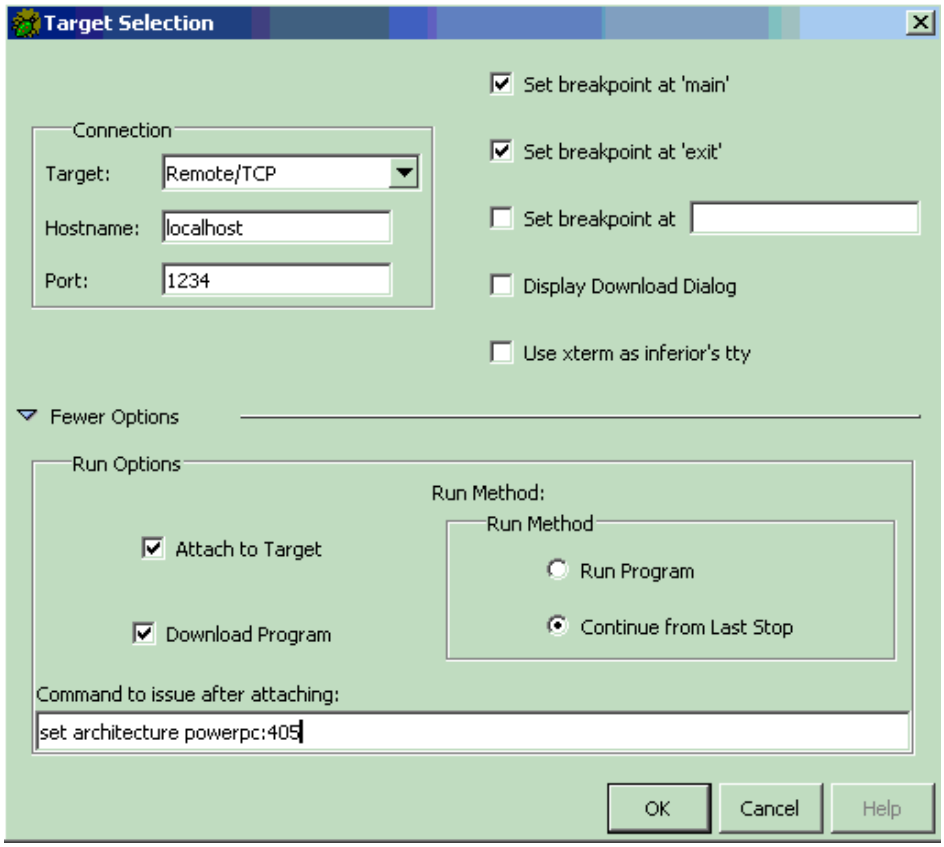
In order to debug a program, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The mb-gcc 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 on **More Options** in the bottom left corner of the target selection dialog. In the command field type:

set architecture powerpc:405

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) set architecture powerpc:405
(gdb) target remote localhost:1234
```

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 10-1 briefly describes the commonly used mb-gdb console commands. The

Table 10-1: **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)
monitor info	View the number of instructions and cycles executed (for the xmd 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.

Chapter 11

Xilinx Microprocessor Debugger

Overview

The Xilinx Microprocessor Debug (XMD) Engine is a program that facilitates a unified GDB interface as well as a Tcl (Tool Command Language) interface for debugging programs and verifying systems using the MicroBlaze or PowerPC (Virtex-II Pro) microprocessors. It supports debugging user programs on different targets such as

- Cycle-accurate MicroBlaze instruction set simulator
- MicroBlaze system running **`xmdstub`** on a hardware board
- PowerPC system on a hardware board

The XMD Engine is used along with MicroBlaze and PowerPC GDB (**`mb-gdb & powerpc-eabi-gdb`**) for debugging. **`mb-gdb`** and **`powerpc-eabi-gdb`** communicates with **`xmd`** using the Remote TCP protocol and control the corresponding targets. In either case, GDB can connect to **`xmd`** 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 the complete system.

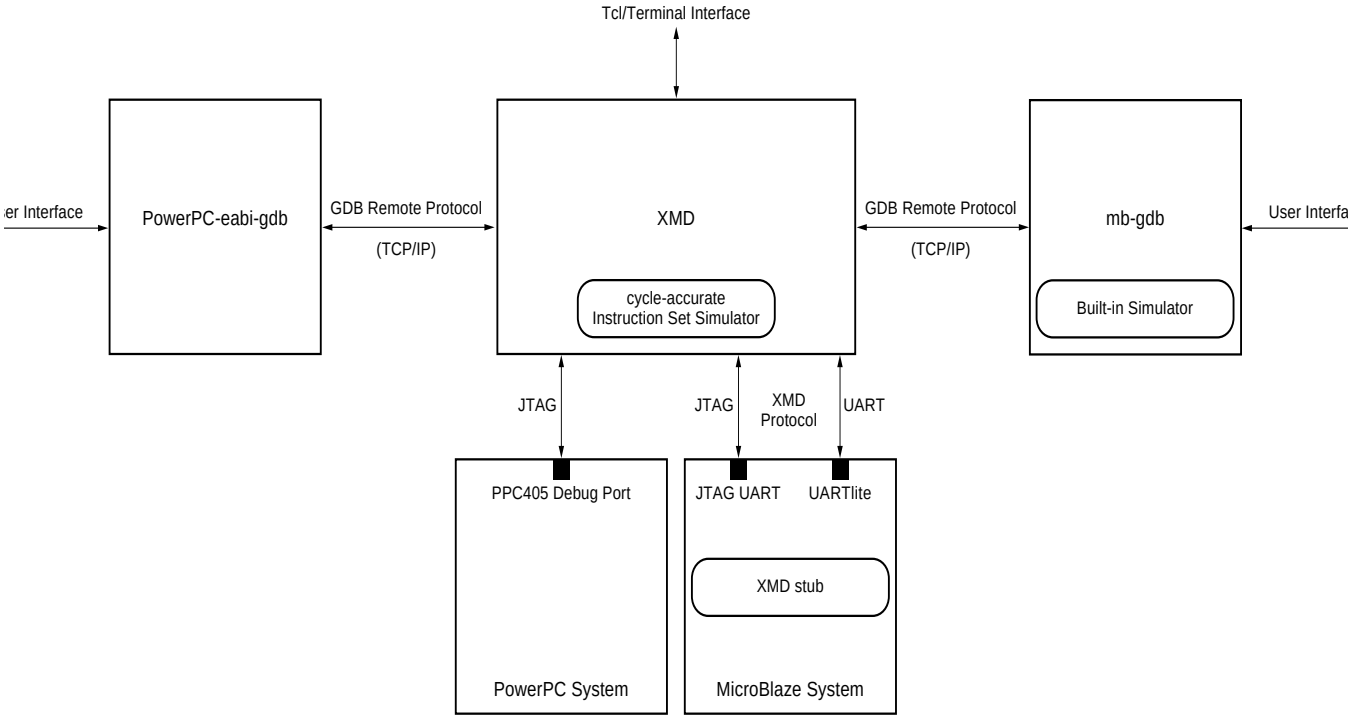


Figure 11-1: **Xilinx Microprocessor Debug (XMD) targets**

XMD usage

To start the XMD engine, simply execute xmd from a shell as follows.

```
> xmd
```

From the xmd Tcl prompt, **xmd** can be connected to the desired target using the commands described in Table 11-1.

MicroBlaze stub target

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 -comm jtag -posit 2
```

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).

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.

- **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

Table 11-1: **XMD commands[a]**

command [options]	Description
mbconnect <**sim** | **stub**> [*options*]	Connect to a MicroBlaze target as well as start a GDB server for the target. **xmd** supports non-intrusive debugging on the MicroBlaze simulator or intrusive debugging on hardware board running **xmdstub**. Use **sim** for simulator or **stub** for remote hardware. The default target is the simulator. • Simulator Target options ♦ **-memsize** *size* size of the memory allocated in the simulator. Programs can access the memory range from **0** to **size-1** • xmdstub Target options ♦ **-comm** <**serial** | **jtag**> Specify the **xmd** communication. Debugging is supported over JTAG (using opb_jtag_uart peripheral) or serial cable (using opb_uart peripheral). Default is JTAG communication ♦ **-posit** *device position* Specify the position of the FPGA device in the JTAG chain that contains the MicroBlaze system to be debugged. The JTAG chain positions are auto detected and displayed by **xmd** when no position is specified ♦ **-chain** *device count <list of BSDL files>* Specify the configuration of the JTAG chain on the target board by providing the BSDL files for all the devices that make up the JTAG chain in the same order as they occur in the chain. By default, xmd autodetects the JTAG chain. But if it fails to do so, then this option can be used to connect to the target board ♦ **-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** *baud rate* Specify the serial port baud rate in bps. The default value is *19200* bps.
mbdisconnect *target id*	Disconnect from the current MicroBlaze target, close the corresponding GDB server and revert to the previous MicroBlaze target.
ppcconnect	Connect to a hardware PowerPC target as well as start a remote GDB server.

Table 11-1: **XMD commands[a]**

command [options]	Description
rrd	Register Read
rwr *reg_num word*	Register Write
mrd *address* [*num_words*]	Memory Read
mwr *address word*	Memory Write
dis [*address*] [*num_words*]	Disassemble
con [*address*]	Continue from current PC or “address”
stp [*number*]	Step one or “number” instructions
rst	Reset target
bps *address*	Set Breakpoint at “address”
bpr *address*	Remote Breakpoint from “address”
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 alongwith 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).
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.

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.

Simulation Statistics

While **mb-gdb** is connected to XMD with the simulator target, the statistics of the cycle-accurate simulator can be viewed from mb-gdb as follows:

- In the **mb-gdb** GUI menu, select **View->Console**.
- In the console window, type **monitor info**
- To reset the simulation statistics, type **monitor reset**

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.

PowerPC Target

xmd can connect to a hardware PowerPC target over a JTAG connection to a board containing a Virtex-II Pro device. Use the **ppcconnect** command to connect to the PowerPC target and start a remote GDB server. A sample session is shown below

```
XMD% ppcconnect

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

```
assumption: selected device 3 for debugging.

Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD%
```

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?".

- 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 alongwith 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.

- 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*

  Set a breakpoint at the given address. Note - Breakpoints on instructions immediately following **`imm`** instruction can lead to undefined results.

- xremove *target addr*

  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 11-2.

Table 11-2: **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.

- xppcserver

  Connect to a PPC405 target, start a remote OCD server and wait for GDB connections.

- xtargets [*target*]

  Print the target ID and target type of all curent targets or a specific target.

Chapter 12

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 16, "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 13, "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 14, “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 18, “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 15, “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 12-1 for a depiction of the peripheral directory structure. On a UNIX system, the processor cores reside in the following location:

$XILINX_EDK/hw/coregen/ip/xilinx/pcores*/com/xilinx/ip2/processor

On a PC, the processor cores reside in the following location:

%XILINX_EDK%\hw\coregen\ip\xilinx\pcores*\com\xilinx\ip2\processor

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

- Current directory (where Platform Generator was launched; not where the MHS resides)
- Set the Platform Generator -P option, or the XIL_MYPERIPHERALS environment variable

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

1. Search current directory in the myip directory
2. Search $XIL_MYPERIPHERALS/myip (UNIX) or %XIL_MYPERIPHERALS%\myip (PC)
3. Search $XILINX_EDK/hw/coregen/ip/xilinx/pcores*/com/xilinx/ip2/processor (UNIX) or
   %XILINX_EDK%\hw\coregen\ip\xilinx\pcores*\com\xilinx\ip2\processor (PC)

The first two search areas (1 and 2) have the same underlying directory structure. The third search area has the CORE Generator directory structure. For search areas 1 and 2, 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
```

For example, if the XIL_MYPERIPHERALS environment is set, then the MPD, BBD, and PAO files are found in the following location:

$XIL_MYPERIPHERALS/myip/<*peripheral*>/data (UNIX)

%XIL_MYPERIPHERALS%\myip\<*peripheral*>\data (PC)

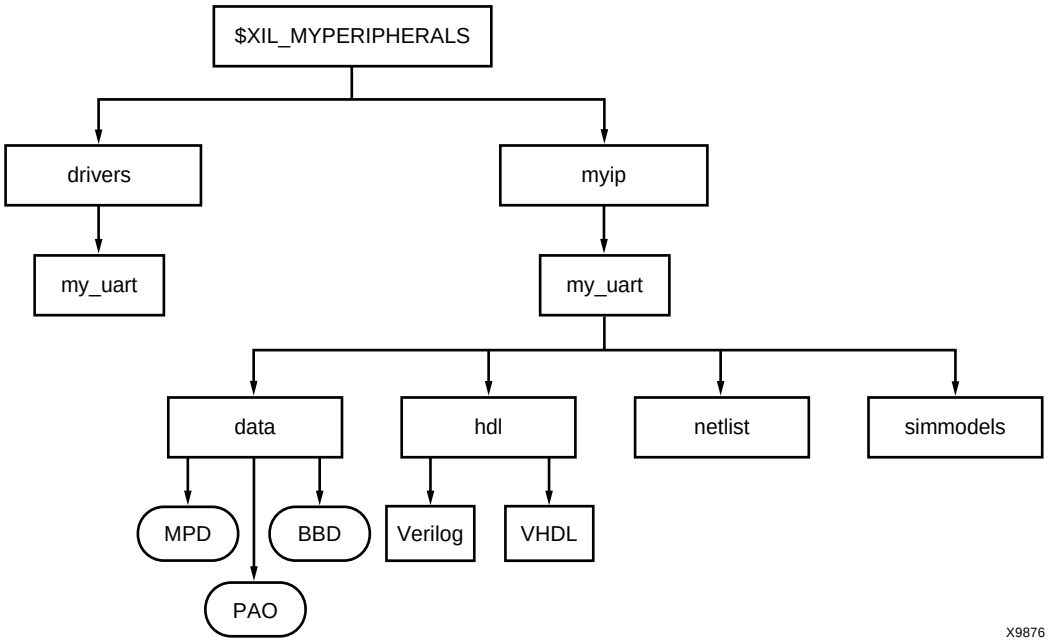


Figure 12-1: **Peripheral Directory Structure**

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:

$XIL_MYPERIPHERALS / myip / <*peripheral*>_v1_00_a / data (UNIX)

%XIL_MYPERIPHERALS% \ myip \ <*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.

Is Your IP Only A Black-Box Netlist?

Read about MPD and BBD files.

Is Your IP A Mixture Of Black-Box Netlists And HDL?

Read about MPD, BBD, and PAO files.

Chapter 13

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. Only connector names are case-sensitive.

Attribute settings in the MHS file have priority over the equivalent attribute setting in the Microprocessor Peripheral Definition (MPD) file. Refer to the *Microprocessor Peripheral Definition Format* document for more information on MPD file syntax.

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:

```
PARAMETER VERSION = 2.0.0

# Define external ports
PORT vcc_out = net_vcc, DIR=OUTPUT
PORT gnd_out = net_gnd, DIR=OUTPUT

PORT my_clk = sys_clk, DIR=INPUT
PORT fb_clk = sys_clk                    # Default is DIR=OUTPUT

PORT sys_rst = sys_rst, DIR=INPUT

# Define external interrupts
PORT my_int1 = my_int1, LEVEL=HIGH, DIR=INPUT, SIGIS=INTERRUPT
PORT my_int2 = int2, EDGE=FALLING, DIR=INPUT, SIGIS=INTERRUPT

PORT rx1 = rx1, DIR=INPUT
PORT tx1 = tx1, DIR=OUTPUT

##############################################################################
BEGIN opb_v20
PARAMETER HW_VER = 1.10.a
PARAMETER INSTANCE = myopb
```

```
PARAMETER C_HIGHADDR = 0x00FFA0FF
PARAMETER C_BASEADDR = 0x00FFA000
PARAMETER C_PARK = 0
PARAMETER C_PROC_INTRFCE = 0
PARAMETER C_REG_GRANTS = 1
PORT OPB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
###################################################################################
BEGIN lmb_v10
PARAMETER HW_VER = 1.00.a
PARAMETER INSTANCE = d_lmb
PORT LMB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
###################################################################################
BEGIN lmb_v10
PARAMETER HW_VER = 1.00.a
PARAMETER INSTANCE = i_lmb
PORT LMB_Clk = sys_clk
PORT SYS_Rst = sys_rst
END
###################################################################################
BEGIN microblaze
PARAMETER HW_VER = 1.00.c
PARAMETER INSTANCE = microblaze1
BUS_INTERFACE DOPB = myopb
BUS_INTERFACE IOPB = myopb
BUS_INTERFACE DLMB = d_lmb
BUS_INTERFACE ILMB = i_lmb
PORT Interrupt = intr
PORT Clk = sys_clk
END
###################################################################################
BEGIN lmb_lmb_bram_if_cntlr
PARAMETER INSTANCE = mylmblmb_cntlr
PARAMETER HW_VER = 1.00.a
PARAMETER C_HIGHADDR = 0x00007fff
PARAMETER C_BASEADDR = 0x00000000
BUS_INTERFACE ILMB = i_lmb
BUS_INTERFACE DLMB = d_lmb
BUS_INTERFACE PORTA = lmb_porta
BUS_INTERFACE PORTB = lmb_portb
END
###################################################################################
BEGIN bram_block
PARAMETER INSTANCE = lmbbram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = lmb_porta
BUS_INTERFACE PORTB = lmb_portb
END
###################################################################################
BEGIN opb_intc
PARAMETER HW_VER = 1.00.b
PARAMETER INSTANCE = myintc
PARAMETER C_HIGHADDR = 0xFFFF90FF
PARAMETER C_BASEADDR = 0xFFFF9000
BUS_INTERFACE SOPB = myopb
PORT OPB_Clk = sys_clk
```

```
PORT Intr = my_int1 & uart_intr & wdt_intr & tb_intr & int2
PORT Irq = intr
END
###################################################################
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
###################################################################
BEGIN opb_bram_if_cntlr
PARAMETER INSTANCE = myopbbram_cntlr
PARAMETER HW_VER = 1.00.a
PARAMETER C_HIGHADDR = 0xFFFF7FFF
PARAMETER C_BASEADDR = 0xFFFF4000
BUS_INTERFACE SOPB = myopb
BUS_INTERFACE PORTA = opb_porta
END
###################################################################
BEGIN bram_block
PARAMETER INSTANCE = opbbram1
PARAMETER HW_VER = 1.00.a
BUS_INTERFACE PORTA = opb_porta
END
###################################################################
BEGIN opb_timebase_wdt
PARAMETER HW_VER = 1.00.a
PARAMETER INSTANCE = mytimebase_wdt
PARAMETER C_HIGHADDR = 0x00FFD0FF
PARAMETER C_BASEADDR = 0x00FFD000
BUS_INTERFACE SOPB = myopb
PORT OPB_Clk = sys_clk
PORT Timebase_Interrupt = tb_intr
PORT WDT_Interrupt = wdt_intr
END
```

Bus Interface Definition

A bus interface is a grouping of interface ports which are related.

The following list are recommendations for bus labels:

Table 13-1: **Bus Labels**

Bus Name	Description
SDCR	Slave DCR interface
SLMB	Slave LMB interface
MOPB	Master OPB interface
MSOPB	Master-slave OPB interface

Table 13-1: **Bus Labels**

Bus Name	Description
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 Command

A global parameter is defined outside of a BEGIN-END block.

A global parameter can have the following options:

Table 13-2: **Global Parameter Options**

Option	Values	Default	Definition
VERSION	2.0.0	X	MHS version

VERSION Option

Use the VERSION option to set the MHS version.

Format

```
PARAMETER VERSION = 2.0.0
```

The version is specified as a literal of the form 2.0.0.

Local Parameter Command

A local parameter is defined between a BEGIN-END block.

A local parameter can have the following options:

Table 13-3: **Local Parameter Options**

Option	Values	Default	Definition
HW_VER	1.00.a	X	Hardware version
INSTANCE		X	User-defined instance name Must be lower-case

HW_VER Option

Use the HW_VER option 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 Option

Use the INSTANCE option to set the instance name of peripheral. This option is mandatory, and the instance name must be specified in lower-case.

Format

```
PARAMETER INSTANCE = my_uart0
```

Local Bus Interface Command

A local bus interface between a BEGIN-END block can have the following options:

Table 13-4: **Local Bus Interface Options**

Option	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 Option

Use the POSITION option 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 Command

A global port outside of a BEGIN-END block can have the following options:

Table 13-5: **Global Port PORTOptions**

Option	Values	Default	Definition
DIR	IN, INPUT, I OUT, OUTPUT, O INOUT, IO	O	Direction mode
EDGE	RISING FALLING	X	Interrupt edge sensitivity
LEVEL	HIGH LOW	X	Interrupt level sensitivity
SIGIS	CLK INTERRUPT	X	SIgnal classification
VEC	[A:B]	X	Vector dimension

DIR Option

The driver direction of a signal is specified by the DIR option.

Format

```
PORT mysignal = "", DIR=direction
```

Where *direction* is either INPUT, IN, I, OUTPUT, OUT, O, INOUT, or IO.

EDGE Option

The edge sensitivity of an interrupt signal is specified by the EDGE option.

Format

```
PORT interrupt = “”, DIR=O, EDGE=edge_value, SIGIS=INTERRUPT
```

Where *edge_value* is either RISING or FALLING.

LEVEL Option

The level sensitivity of an interrupt signal is specified by the LEVEL option.

Format

```
PORT interrupt = “”, DIR=O, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

SIGIS Option

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

Format

```
PORT interrupt = “”, DIR=O, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

VEC Option

The vector width of a signal is specified by the VEC option.

Format

```
PORT mysignal = “”, DIR=I, VEC=[A:B]
```

Local PORT Command

A local port is a port defined between a BEGIN-END block. A local port does not have options.

Design Considerations

This section provides general design considerations.

Assinging Constants

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.

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 component, 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.

Power Signals

Power signals are signals that are constantly driven with either VCC or GND.

Format

```
PORT mysignal = power_signal
```

In this example, *power_signal* is either "net_vcc" or "net_gnd". Platform Generator expands "net_vcc" or "net_gnd" to the appropriate vector size.

Chapter 14

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 options.

This chapter includes the following sections:

MPD Syntax

MPD file syntax is case insensitive. Only connector names are case-sensitive.

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 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 four assignment commands:

1. bus_interface
2. option
3. parameter
4. 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:

```
PARAMETER VERSION = 2.0.0

BEGIN opb_gpio, IPTYPE=PERIPHERAL, IMP_NETLIST=TRUE
```

```
OPTION SIM_MODELS = HDL

# Define 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

# Global ports
PORT OPB_Clk = "", DIR=IN, SIGIS=CLK, BUS=SOPB
PORT OPB_Rst = OPB_Rst, DIR=IN, BUS=SOPB

# OPB slave signals
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

# gpio signals
PORT GPIO_IO = "", DIR=INOUT, VEC=[0:C_GPIO_WIDTH-1], ENABLE=MULTI

END
```

Bus Interface Naming Conventions

A bus interface is a grouping of interface ports which are related.

The following list are recommendations for bus labels:

Table 14-1: **Recommended 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, use an intuitive naming convention. For example, the data-side OPB and instruction-side OPB are named DOPB and IOPB, respectively.

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.

The Platform Generator 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:

Figure 14-1: **Automatically Expanded Reserved Parameters**

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_NUM_MASTERS	Number of OPB masters
C_NUM_SLAVES	Number of OPB slaves
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_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

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.

Format

```
PARAMETER C_BUS_CONFIG = bus_config, DT=integer
```

C_FAMILY

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

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 Platform Generator.

Format

```
PARAMETER C_INSTANCE = instance_name, DT=string
```

C_NUM_MASTERS

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

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. This parameter is automatically populated by Platform Generator.

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 Platform Generator.

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. This parameter is automatically populated by Platform Generator.

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. This parameter is automatically populated by Platform Generator.

Format

```
PARAMETER C_LMB_DWIDTH = <num>, DT=integer
```

Where <*num*> is an integer 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 Platform Generator.

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. This parameter is automatically populated by Platform Generator.

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. This parameter is automatically populated by Platform Generator.

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 Platform Generator.

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 Platform Generator.

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. This parameter is automatically populated by Platform Generator.

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. This parameter is automatically populated by Platform Generator.

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. This is set to log2(S). This parameter is automatically populated by Platform Generator.

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 Platform Generator.

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 Platform Generator.

Format

```
PARAMETER C_PLB_NUM_SLAVES = <num>, DT=integer
```

Where <*num*> is an integer value.

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.

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.

Platform Generator is capable of dealing with a design of mixed HDL.

- VHDL top-level with lower-level VHDL / Verilog cores
- Verilog top-level with lower-level VHDL / Verilog cores

Due to this case, a Verilog core's signal interface must be written in lower-case. Verilog is a case sensitive language, and it's case is preserved in the synthesized netlist files (EDIF and NGC). However, VHDL is a case-insensitive language, thus synthesis vendors normalize all names to lower-case. So to have a VHDL core interface to a Verilog core, the ports must match.

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>_MRarbitrate
<nPLB>_MWrBTerm
<nPLB>_MWrDAck
<nPLB>_MSSize
<nPLB>_SMErr
<nPLB>_SMbusy
```

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 Signal Connections

Connectivity of the DCR, LMB, OPB and PLB busses to peripherals is done through a common set of signal connections.

Global Ports

For interconnection to the global ports:

LMB - Clock and Reset

```
PORT LMB_Clk = "", DIR=I, SIGIS=CLK
PORT LMB_Rst = OPB_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
```

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>_MRarbitrate = PLB_MRarbitrate, 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
PORT <nPLB>_SMErr = PLB_SMErr, DIR=I, BUS=MPLB
PORT <nPLB>_SMbusy = PLB_SMbusy, DIR=I, 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
```

Component Options

Components can have the following options:

Table 14-2: **MPD Peripheral Options**

Option	Values	Default	Definition
EDIF	TRUE FALSE	FALSE	Deprecated. Use the IMP_NETLIST option.
HDL	BOTH VERILOG VHDL	VHDL	HDL design availability.
IMP_NETLIST	TRUE FALSE	FALSE	Synthesize HDL to a hardware implementation netlist
IPTYPE	BRIDGE BUS BUS_ARBITER IP PERIPHERAL PROCESSOR	IP	Type of component
STYLE	BLACKBOX MIX HDL	HDL	Design style

HDL Option

The HDL option 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.

Format

```
BEGIN peripheral_name, HDL=VERILOG
```

IMP_NETLIST Option

In hierarchal mode, this option directs the Platform Generator to write an implementation netlist file for the peripheral. In flatten mode, the IMP_NETLIST option is ignored since the entire system is synthesized.

Format

```
BEGIN peripheral_name, IMP_NETLIST=TRUE
```

IPTYPE Option

The IPTYPE option lists defines the type of the component.

Format

```
BEGIN peripheral_name, IPTYPE=PERIPHERAL
```

The IPTYPE option can have the following Options:

- 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)

STYLE Option

The STYLE option 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 Platform Generator.

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 Platform Generator.

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 Platform Generator.

Format

```
BEGIN peripheral_name, STYLE=value
```

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

Global Parameter Command

A global parameter can have the following options:

Table 14-3: **Global Parameter Options**

Option	Values	Default	Definition
VERSION	2.0.0	X	MPD version

VERSION Option

Use the VERSION option to set the MPD version.

Format

```
PARAMETER VERSION = 2.0.0
```

The version is specified as a literal of the form 2.0.0.

Local Option Command

A local option defined between a BEGIN-END block can have the following options:

Table 14-4: **Local Option Options**

Option	Values	Default	Definition
SIM_MODELS	BEHAVIORAL STRUCTURAL TIMING	X	Simulation model availability

SIM_MODELS Option

The simulation model availability is specified with the SIM_MODELS option.

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
```

Local Parameter Command

A local parameter defined between a BEGIN-END block can have the following options:

Table 14-5: **Local Parameter Options**

Option	Values	Default	Definition
BUS	*string*	X	Bus label
DT	string integer std_logic	X	Datatype of VHDL generic
MIN_SIZE	2^n	0	Minimum size address window

BUS Option

The bus interface of an parameter is specified by the BUS option.

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.

Format

```
PARAMETER C_OPB_AWIDTH = 32, DT=datatype, BUS=MSOPB:SOPB
```

DT Option

The datatype of an parameter is specified by the DT option.

Format

```
PARAMETER C_OPB_AWIDTH = 32, DT=datatype, BUS=bus_label
```

Where *datatype* is an VHDL datatype.

MIN_SIZE Option

The minimum size address window of an address is specified by the MIN_SIZE option.

Format

```
PARAMETER C_BASEADDR = 0xFFFFFFFF, DT=std_logic_vector, MIN_SIZE=0x100
```

Local Bus Interface Command

A local bus interface between a BEGIN-END block can have the following Options:

Table 14-6: **Bus Interface Options**

Option	Values	Default	Definition
BUS	*string*	X	Bus label
BUS_STD	DCR LMB OPB PLB TRANSPARENT	X	Bus standard
BUS_TYPE	MASTER MASTER_SLAVE SLAVE UNDEF	X	Bus type

BUS Option

The label of a bus interface is specified by the BUS option.

Format

```
BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type
```

Where *bus_label* is a string.

BUS_STD Option

The bus standard of a bus interface is specified by the BUS_STD option.

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 Option

The bus type of a bus interface is specified by the BUS_TYPE option.

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.

Local Port Command

A local port defined between a BEGIN-END block can have the following options:

Table 14-7: **Local Port Options**

Option	Values	Default	Definition
BUS	*string*	X	Bus label
DIR	IN, INPUT, I OUT, OUTPUT, O INOUT, IO	O	Direction mode
EDGE	RISING FALLING	X	Interrupt edge sensitivity
ENABLE	MULTI SINGLE	SINGLE	3-state enable control
ENDIAN	BIG LITTLE	BIG	Endianess
INITIALVAL	VCC GND	GND	Driver value on unconnected inputs
LEVEL	HIGH LOW	X	Interrupt level sensitivity
SIGIS	CLK INTERRUPT	X	SIgnal classification
VEC	[A:B]	X	Vector dimension. Where A and B are positive integer expressions.

BUS Option

The bus interface of a signal is specified by the BUS option.

Format

```
PPORT 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
```

DIR Option

The driver direction of a signal is specified by the DIR option.

Format

```
PORT mysignal = "", DIR=direction
```

Where *direction* is either INPUT, IN, I, OUTPUT, OUT, O, INOUT, or IO.

EDGE Option

The edge sensitivity of an interrupt signal is specified by the EDGE option.

Format

```
PORT interrupt = “”, DIR=O, EDGE=edge_value, SIGIS=INTERRUPT
```

Where *edge_value* is either RISING or FALLING.

ENABLE Option

Tri-state signals can have multi-bit enable control, or a single bit enable control on the bus. This is specified with the ENABLE option.

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 “HDL Design Considerations” section about designing tri-state signals at the HDL level.

ENDIAN Option

The endianess of a signal is specified by the ENDIAN option.

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.

INITIALVAL Option

The signal driver value on unconnected input signals is specified by the INITIALVAL option.

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 Option

The level sensitivity of an interrupt signal is specified by the LEVEL option.

Format

```
PORT interrupt = “”, DIR=OUTPUT, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

SIGIS Option

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

Format

```
PORT interrupt = “”, DIR=OUTPUT, LEVEL=level_value, SIGIS=INTERRUPT
```

Where the *level_value* is either HIGH or LOW.

VEC Option

The vector width of a signal is specified by the VEC option.

Format

```
PORT mysignal = “”, DIR=INPUT, VEC=[A:B]
```

Where A and B are positive integer expressions.

HDL Design Considerations

This section includes HDL design considerations.

Unconnected Signals

Unconnected output signals are assigned open, and unconnected input signals are either set to GND or VCC.

An unconnected signal is identified as an empty double-quote (“”) string.

Platform Generator resolves the driver value on unconnected input signals by the INITIALVAL option.

Format

```
PORT mysignal = “”, DIR=OUTPUT
```

Scalable Data path

Using an MPD option 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, the Platform Generator determines 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 Platform Generator defaults to big-endian style notation.

To change this default behavior, use the ENDIAN option.

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 EDGE or LEVEL option.

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.

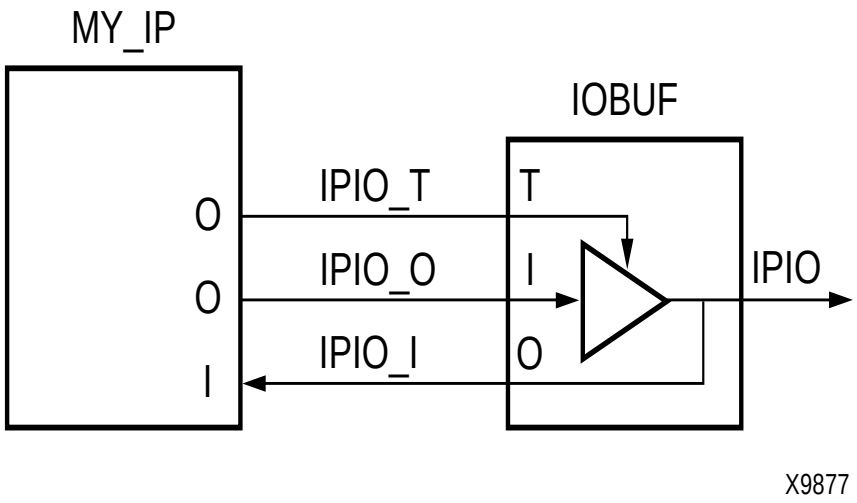


Figure 14-2: **IOBUF Implementation**

The Platform Generator 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. Platform Generator 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 an 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 an MPD example that includes 3-state signal with a multi-bit enable:

```
PARAMETER VERSION = 2.0.0

BEGIN tri_state_multi, IPTYPE=IP

PARAMETER C_WIDTH = 9, DT=integer

PORT tristate = "", DIR=INOUT, VEC=[0:C_WIDTH-1], ENABLE=MULTI

END
```

VHDL 3-state (InOut) With Single-Bit Enable Example

The following is an 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 an MPD example that includes 3-state signal with a single-bit enable:

```
PARAMETER VERSION = 2.0.0

BEGIN tri_state_single, IPTYPE=IP

PARAMETER C_WIDTH = 9, DT=integer

PORT tristate = "", DIR=INOUT, VEC=[0:C_WIDTH-1], ENABLE=SINGLE

END
```

Chapter 15

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

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
```

XILINX®

Chapter 16

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

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 options. The last column of the table must be the FILES column.

For implementation, the last column lists the relative path to the file from:
$XIL_MYPERIPHERALS/myip/<*ip*>/netlist (UNIX)

%XIL_MYPERIPHERALS%\myip\<*ip*>\netlist (PC)

For simulation, the last column lists the relative path to the file from:
$XIL_MYPERIPHERALS/myip/<*ip*>/simmodels (UNIX)

%XIL_MYPERIPHERALS%\myip\<*ip*>\simmodels (PC)

The netlist and simmodels directories 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 options set on the core.

```
C_FAMILY C_BUS_CONFIG FILES
virtex          1       virtex/microblaze_1.edf
virtex          2       virtex/microblaze_2.edf
virtex          3       virtex/microblaze_3.edf
virtex          4       virtex/microblaze_4.edf
virtex          5       virtex/microblaze_5.edf
virtex          6       virtex/microblaze_6.edf
spartan2        1       virtex/microblaze_1.edf
spartan2        2       virtex/microblaze_2.edf
spartan2        3       virtex/microblaze_3.edf
spartan2        4       virtex/microblaze_4.edf
spartan2        5       virtex/microblaze_5.edf
spartan2        6       virtex/microblaze_6.edf
virtexe         1       virtex/microblaze_1.edf
virtexe         2       virtex/microblaze_2.edf
virtexe         3       virtex/microblaze_3.edf
```

```
virtexe         4       virtex/microblaze_4.edf
virtexe         5       virtex/microblaze_5.edf
virtexe         6       virtex/microblaze_6.edf
spartan2e       1       virtex/microblaze_1.edf
spartan2e       2       virtex/microblaze_2.edf
spartan2e       3       virtex/microblaze_3.edf
spartan2e       4       virtex/microblaze_4.edf
spartan2e       5       virtex/microblaze_5.edf
spartan2e       6       virtex/microblaze_6.edf
virtex2         1       virtex2/microblaze_1.edf
virtex2         2       virtex2/microblaze_2.edf
virtex2         3       virtex2/microblaze_3.edf
virtex2         4       virtex2/microblaze_4.edf
virtex2         5       virtex2/microblaze_5.edf
virtex2         6       virtex2/microblaze_6.edf
virtex2p        1       virtex2/microblaze_1.edf
virtex2p        2       virtex2/microblaze_2.edf
virtex2p        3       virtex2/microblaze_3.edf
virtex2p        4       virtex2/microblaze_4.edf
virtex2p        5       virtex2/microblaze_5.edf
virtex2p        6       virtex2/microblaze_6.edf
```

Chapter 17

Microprocessor Verification Specification (MVS)

Summary

This chapter describes the Microprocessor Verification Specification (MVS) format.

Overview

You supply MVS file as an input to the Simulation Model Generator (SimGen) tool. The MVS file contains directives for customizing a simulation model for a defined system.

MVS Format

An MVS file is the input to the SimGen. Its semantics are case insensitive, however, any reference to a file name or instance name in the MVS 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 '#' though 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 MVS file are as follows:

Begin

The **begin** keyword begins a simulation model definition.

End

The **end** keyword signifies the end of a definition block.

Parameter

The MVS 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 MVS file has a dependency on the hardware and software specification (MHS and MSS) files. This dependency has to be specified in the MVS file as **parameter HW_SPEC_FILE =** *file_name.mhs* or as **parameter SW_SPEC_FILE =** *file_name.mss*

respectively. Hence, a hardware platform has to be defined in order to configure the simulation flow. Please refer to Chapter 13, "Microprocessor Hardware Specification (MHS)" for more information on hardware configuration.

The syntax of various files that the Embedded Software Tools use are described by the Platform Specification Format (PSF). The current PSF version is 2.0.0. The MVS file should also contain version information in the form of **parameter Version = 2.0.0** which represents the PSF version 2.0.0.

MVS Example

An example MVS file is given below:

```
# PSF Version
PARAMETER VERSION = 2.0.0

# Define the location of the Hardware Specification file
PARAMETER HW_SPEC_FILE = filename.mhs

# Define the location of the Software Specification file
PARAMETER SW_SPEC_FILE = filename.mss

# Define simulation language
# Options: vhdl / verilog
PARAMETER LANGUAGE = vhdl

# Define simulator
# Options: mti / vxl
PARAMETER SIMULATOR = mti

# Define simulation model
PARAMETER SIM_MODEL = behavioral
                    # Options: behavioral / structural / timing

# Specify path to ModelSim Behavioral Library
PARAMETER MTI_NODEBUG_LIB = /home/user/directory/behavioral

# Specify path to ModelSim Unisim Library
PARAMETER MTI_UNISIM_LIB = /home/user/directory/unisim

# Specify path to ModelSim Simprim Library
PARAMETER MTI_SIMPRIM_LIB = /home/user/directory/simprim
```

Global Parameters

These parameters are system specific parameters and do not relate to a particular driver, file system or library.

PSF Version

This parameter specifies the PSF version of the MSS file. It is mandatory for versions 2.0.0 and above.

Format

```
parameter VERSION = 2.0.0
```

Hardware Specification File Pointer

This parameter points to the MHS file. The path can be a relative path from the project directory or can be an absolute path. This parameter is mandatory.

Format

```
parameter HW_SPEC_FILE = filename.mhs
```

Software Specification File Pointer

This parameter points to the MSS file. The path can be a relative path from the <USER_PROJECT> directory or can be an absolute path. This parameter is optional.

Format

```
parameter SW_SPEC_FILE = filename.mss
```

Simulation Language

This parameter specifies the simulation language to be used for the generated HDL simulation files. The available options are `vhdl` and `verilog`. This parameter is optional.

Format

```
parameter LANGUAGE = { vhdl | verilog }
```

Simulator

This parameter specifies the simulator to be used. SimGen generates a compilation script for the specified simulator. The supported simulators are Model Technology ModelSim and Cadence Verilog-XL. This parameter is optional.

Format

```
parameter SIMULATOR = { mti | vxl }
```

Simulation Model

This parameter specifies the simulation model to be generated. The supported simulation model types are behavioral, structural and timing. This parameter is optional.

Format

```
parameter SIM_MODEL = { behavioral | structural | timing }
```

ModelSim Behavioral Library

This parameter specifies the path to the ModelSim behavioral library. This parameter is optional.

Format

```
parameter MTI_NODEBUG_LIB = /path/to/modelsim/behavioral/library
```

ModelSim Unisim Library

This parameter specifies the path to the ModelSim unisim library. This parameter is optional.

Format

```
parameter MTI_UNISIM_LIB = /path/to/modelsim/unisim/library
```

ModelSim Simprim Library

This parameter specifies the path to the ModelSim simprim library. This parameter is optional.

Format

```
parameter MTI_SIMPRIM_LIB = /path/to/modelsim/simprim/library
```

XILINX®

Chapter 18

Microprocessor Software Specification (MSS)

Summary

This chaprter 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 7, “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 in the MSS file as **parameter HW_SPEC_FILE =** *file_name***.mhs**. Hence, a hardware platform has to be defined in order to configure the software flow. Please refer the Microprocessor Hardware Specification documentation for more information on hardware configuration.

The syntax of various files that the Embedded Development Tools use are described by the Platform Specification Format (PSF). Please refer Chapter 12, "Platform Specification Format (PSF)" for more information. The current PSF version is 2.0.0. The MSS file should also contain version information in the form of **parameter Version = 2.0.0** which represents the PSF version 2.0.0.

MSS Example

An example MSS file is given below:

```
parameter HW_SPEC_FILE = system.mhs
parameter VERSION = 2.0.0

BEGIN PROCESSOR
parameter HW_INSTANCE = my_microblaze
parameter DRIVER_NAME = cpu
parameter DRIVER_VER = 1.00.a
parameter BOOT_PERIPHERAL = my_jtag
parameter DEBUG_PERIPHERAL = my_jtag
parameter EXECUTABLE = code/hello_world.elf
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 FILESYS
parameter FILESYS_NAME = XilMfs
parameter PROC_INSTANCE = my_microblaze
parameter MOUNT = "/dev/mfs"
parameter LIBRARY = XilFile
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.

Hardware Specification File Pointer

This option points to the MHS file. The path can be a relative path from the ***USER_PROJECT*** directory or can be an absolute path. This option is mandatory.

Format

```
parameter HW_SPEC_FILE = system.mhs
```

PSF Version

This option specifies the PSF version of the MSS file. This option is mandatory for versions 2.0.0 and above.

Format

```
parameter VERSION = 2.0.0
```

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 and Processor Block Parameters

Table 18-1: **Parameters Specified in Driver 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.
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 Driver Definition (MDD)" for more information.
LIBRARY	XilFile, XilNet	None	Specifies that the device can be accessed through this library. Please seeChapter 20, "Xilinx Libraries" for more information.

Table 18-1 provides the parameters that can be used both in driver 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 ***USER_PROJECT/processor_instance_name*/libsrc** directory and compiles the drivers using makefiles provided. Please see the Library Generator document 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

XIL_MYPERIPHERALS / 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 ***XIL_MYPERIPHERALS*** variable is set when a **-P** option is given to LibGen.

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 Driver Definition (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 26, "Device Drivers" for more information. The default level is specified in the MDD file for the driver. Please refer Chapter 19, "Microprocessor Driver Definition (MDD)" for more information.

Format

```
parameter LEVEL = 1
```

Level is either 0 or 1 for EDK drivers

LIBRARY Option

The device driver functions (that support I/O) can be accessed through a library that provides block access functions for read and write. This option provides a way to specify that the device is accessed through XilFile library shipped with the EDK. For more information on libXil libraries for MicroBlaze, please refer Chapter 20, "Xilinx Libraries".

Format

```
parameter LIBRARY = XilFile
```

Please refer to the section on *Libraries and File System Parameters* in this document for details on file systems (particularly Memory File Systems), and their access through the XilFile library.

MDD Specific Parameters

Parameters specified in the MDD file can be overwritten in the MSS file as

Format

```
parameter PARAM_NAME = PARAM_VALUE
```

Please refer Chapter 19, "Microprocessor Driver Definition (MDD)" for information.

Processor Specific Parameters

Table 18-2: **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.
DEFAULT_INIT	XMDSTUB, BOOTSTRAP, EXECUTABLE	EXECUTABLE	Specifies which file should be used to initialize that processor's memory.
BOOT_PERIPHERAL	Instance name	None	Peripheral instance used for downloading bootstub.
DEBUG_PERIPHERAL	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_FLAGS	Command line flags	Libgen generates default	Need not be specified if using EDT compilers
EXTRA_COMPILER_FLAGS	Command line flags	None	User definable compiler flags used to compile libraries and drivers

Table 18-2 provides all the parameters that can be specified only in a processor definition block.

EXECUTABLE Option

The executable image is set using the EXECUTABLE option.

Format

```
parameter EXECUTABLE = code/a.elf
```

This is the executable file used for populating memories of the particular processor instance. By default, libgen assumes the EXECUTABLE to be *processor_instance_name* / code / executable.elf

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
```

BOOT_PERIPHERAL Option

Identify the boot peripheral with the BOOT_PERIPHERAL option. This is useful for MicroBlaze only. The boot peripheral is used for download of the bootstub.

Format

```
parameter BOOT_PERIPHERAL = 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.

Library and File System Parameters

The MSS file also includes directives to configure libraries and file systems for the MicroBlaze processor. For more information on EDK libraries and EDK file systems, please refer Chapter 20, "Xilinx Libraries".

The parameters that are supported for Libraries and File systems are as follows:

FILESYS_NAME Option

This option specifies the name of the file system. The file system that is supported in the EDK is XilMfs (Memory File System). Please refer Chapter 20, “Xilinx Libraries” for more information.

Format

```
parameter FILESYS_NAME = XilMfs
```

If the user has any other file system that is compatible with EDK file systems, that name can be used.

PROC_INSTANCE Option

This option specifies the name of the processor instance that can access the file system.

Format

```
parameter PROC_INSTANCE = my_microblaze
```

This specifies that *my_microblaze* processor needs to access the file system functions.

MOUNT Option

Specifies the mount name as a directory string

Format

```
parameter MOUNT = “/dev/mfs”
```

LIBRARY Option

The file system functions can be accessed through a library that provides block access functions for read and write. This option provides a way to specify that the file system is accessed through the library.

Format

```
parameter LIBRARY = XilFile
```

Chapter 19

Microprocessor Driver Definition (MDD)

Summary

This chapter describes the Microprocessor Driver Definition (MDD) format.

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 26, “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.

Requirements

Each device driver has an MDD file associated with it. This file is used by the Library Generator (Libgen) to customize the driver depending on different options in the MSS file. For more information on the MSS file format, please see Chapter 18, “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 drivers. Please refer Chapter 6, “Library Generator” for a list of directories searched for drivers.

MDD Format

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 a sentence delimiter.

Keywords

The keywords that are used in an MDD file are as follows:

Begin

The **begin** keyword begins a driver block or a driver level block. Device drivers in the EDK come in three levels. Level 0 are the low-level drivers with basic functionality and small size, level 1 drivers have more functionality but are larger in size and OS level drivers assume the existence of an RTOS. The begin keyword should be followed by **driver,** or **level *number*** (0 or 1).

End

The **end** keyword signifies the end of a definition block.

Parameter

The MDD 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***. The parameter keyword specifies that the parameter can be overwritten in the MSS file.

Constant

The **constant** keyword is similar to the parameter keyword, except that the constant cannot be overwritten in the MSS file.The format for assigning a value to a constant is **constant *name* = *value***.

MDD Example

An example MDD file for the uartlite driver follows:

```
BEGIN driver XUartLite
constant VERSION = 2.0.0 # uses PSF 2.0.0
parameter LEVEL = 0 # default level
END

BEGIN level 0
constant INBYTE = XUartLite_RecvByte, DEFINED_IN = xuartlite_l
constant OUTBYTE = XUartLite_SendByte, DEFINED_IN = xuartlite_l
constant COPYFILES = (xuartlite_l.c xuartlite_l.h Makefile)
constant DEPENDS = (common_v1_00_a)
constant CONFIG_INCLUDE = xparameters, VALUES = ( NUM_INSTANCES
C_BASEADDR )
parameter INT_HANDLER = XIntc_DefaultHandler, INT_PORT = Interrupt
END

BEGIN level 1
constant COPYFILES = ALL
constant DEPENDS = (common_v1_00_a)
constant CONFIG_INCLUDE = xparameters, VALUES = ( NUM_INSTANCES
C_BASEADDR DEVICE_ID )
constant CONFIG_FILE = xuartlite_g, VALUES = ( DEVICE_ID C_BASEADDR )
parameter INT_HANDLER = XUartLite_InterruptHandler, INT_PORT=Interrupt
END
```

Driver Block

The driver block begins with a **Begin driver** followed by the name of the driver, in the above example, XUartLite. The name is significant, since it is used as a prefix for a configuration table in the driver configuration C file by libgen. This name is case sensitive.

Driver Block Specific Parameters and Constants

VERSION

Every MDD file has a PSF (Platform Specification Format) version number. This is the same number that is used in all file formats (MSS, MPD, PAO, MHS) in the EDK. The version signifies the version of the file that libgen will accept. Please refer Chapter 12, "Platform Specification Format (PSF)" for more information.

Format

```
constant VERSION = 2.0.0
```

This indicates that the PSF version is 2.0.0

LEVEL

This parameter specifies the default level of the driver used by libgen. This parameter can be overwritten

Format

```
parameter LEVEL = 0
```

This indicates that the default version is 0.

Other Generic Parameters

The user can define their own parameter name and specify a value for the parameter. This parameter name can then be used in the VALUES, DEFINE_IF_ANY or DEFINE_IF_ALL tags as explained in subsequent sections in the document. For DEFINE_IF_ANY and DEFINE_IF_ALL, the value must be either a 0 (FALSE) or 1(TRUE).

Format

```
parameter MY_OWN_PARAMETER = my_value
```

The literal *my_value* is used wherever it is referenced in one of the tags specified.

The Generic parameters can also take an optional tag TYPE. TYPE can have values DRIVER and INSTANCE (default).

```
parameter MY_OWN_DRV_PARAMETER = my_value, TYPE = DRIVER
```

By default, these parameters are peripheral instance specific. That is, each peripheral instance has the parameter defined. TYPE = DRIVER means that the parameter should be treated as a driver specific parameter. Please see the CONFIG_INCLUDE Option subsection of this document for a better understanding.

Level Block

The Level block begins with a **begin level**, followed by a number (0 or 1). This number indicates the driver level that the block describes. For more information on the levels of drivers, please refer Chapter 26, "Device Drivers".

Level Block Specific Parameters and Constants

INBYTE

This attribute specifies the driver function for input of a byte of data. The signature of the function should be **char** *function_name* **(int** **base_address***)*. When the peripheral using this driver is specified as a *standard input* (STDIN) peripheral in the MSS file, this function is called by library functions such as *scanf*. Only drivers for peripherals that can be used as standard input should have this attribute defined.

Format

```
constant INBYTE = XUartLite_RecvByte, DEFINED_IN = xuartlite_l
```

This indicates that the INBYTE *function_name* is **XUartLite_RecvByte** and it is defined in **xuartlite_l.h** file

OUTBYTE

This attribute specifies the driver function for output of a byte of data. The signature of the function should be **void** *function_name* **(int** **base_address***,* **char** *outchar***)**. This function is called by *printf* when the peripheral using this driver is specified as *standard output* (STDOUT) peripheral. Only drivers for peripherals that can be used as standard output should have this attribute defined.

Format

```
constant OUTBYTE = XUartLite_SendByte, DEFINED_IN = xuartlite_l
```

This indicates that the OUTBYTE *function_name* is **XUartLite_SendByte** and it is defined in **xuartlite_l.h** file

COPYFILES

This attribute specifies that files that should be copied for the specific level of the driver. The files are copied from the **src** directory inside the drivers directory.

Format

```
constant COPYFILES = (file1 file2 file3 ...)
```

This indicates that the files *file1, file2, file3, and so on*, should be copied in order to compile this driver into the project software platform. The list must be enclosed in parentheses (even if there is a single file) and separated by spaces. The keyword **ALL** can be used instead of listing the files to specify that all files in the directory should be used to compile the driver.

DEPENDS

This attribute specifies which drivers this particular level of the driver depends on. These dependent drivers are then included in the compilation.

Format

```
constant DEPENDS = (driver1 driver2 driver3 ...)
```

This indicates that the drivers *driver1, driver2, driver3, ...* should be copied in order to compile this driver into the project software platform. The list must be enclosed in parentheses (even if there is a single driver) and separated by spaces. The drivers should have the complete name (including version suffixes, if any).

CONFIG_INCLUDE

This attribute specifies the name of the include file (.h) that is configured with #defines specific to each driver level and also specific to how the peripheral has been configured. This file is generated in the **include** directory by libgen (refer Chapter 6, "Library Generator" for more information)

Format

```
constant CONFIG_INCLUDE = xparameters, VALUES = (C_BASEADDR DEVICE_ID
NUM_INSTANCES), DEFINE_IF_ANY = (C_HAS_IPR), DEFINE_IF_ALL = (MY_PARAM)
```

The name of the file that is configured in the preceeding example is **xparameters.h.** The values that should be defined are specified in the VALUES tag as a space separated list enclosed between parentheses. The parentheses are necessary even if there is a single item in the list.

LibGen compares the names given in the VALUES tag with names of parameters in the MPD or MHS files, or user defined parameters in the MDD file itself. If a match is found, the value of the attribute is defined as

```
#define XPAR_INSTANCE_NAME_PARAMETER_NAME PARAMETER_VALUE
```

if the parameter is an instance specific parameter (for example, C_BASEADDR). Any **C_** in the name is removed. For the example MDD given above, suppose the instance name for the uartlite peripheral in the MSS file is given as my_uart. Then the #define would be:

```
#define XPAR_MY_UART_BASEADDR 0xFFFF0100
```

On the other hand, if the parameter is a driver specific parameter and not peripheral instance specific (for e.g. NUM_INSTANCES, which is recognized by libgen as number of instances of peripherals that use this driver) then the define will be with the name of the driver (as given in the BEGIN DRIVER *DRIVER_NAME* in the MDD) instead of the INSTANCE_NAME. For example, if there are two instances of uart in the MSS referring to this driver, then the definition would be:

```
#define XPAR_XUARTLITE_NUM_INSTANCES 2
```

Note that all the names and parameters are upper case, and that all parameters having a prefix of **C_** are truncated to lose the prefix.

DEFINE_IF_ANY

This tag can be used to specify a list of parameters. Each parameter is defined if *any* instance of a peripheral that uses the driver has the parameter defined as TRUE (1) in the MPD or MHS file.

DEFINE_IF_ALL

This tag can be used to specify a list of parameters. Each parameter is defined only if *all* instances of a peripheral that uses the driver has the parameter defined as TRUE (1) in the MPD or MHS file.

CONFIG_FILE

This attribute specifies the name of the file (.c) that is configured by LibGen. The file ***CONFIG_FILE*.h** will always be included in the C file. This file is generated in this drivers **src/** directory when LibGen configures the drivers.

Format

```
constant CONFIG_FILE = xuartlite_g, VALUES = (DEVICE_ID C_BASEADDR)
```

The name of the file that is configured in the preceeding example is **xuartlite_g.c**. The file **xuartlite_g.h** is always included using **#include** in this file. Also, if a CONFIG_INCLUDE parameter is specified, that file is also included using **#include**. The values that should be defined are specified in the VALUES tag as a space separated list enclosed between parentheses. The parentheses are necessary even if there is a single item in the list. The list is an ordered list as is apparent from the following discussion.

LibGen compares the names given in the VALUES tag with names of parameters in the MPD or MHS files, or user defined parameters in the MDD file itself. LibGen creates a data structure in the C file as follows (with the example given above):

```
XUartLite_Config XUartLite_ConfigTable[] =
{
  {
    XPAR_MYUART_DEVICE_ID,
    XPAR_MYUART_BASEADDR
  },
  {
    XPAR_MYUART2_DEVICE_ID,
    XPAR_MYUART2_BASEADDR
  }
};
```

As seen in the code segment above, the MSS file contains two uartlite peripherals with instance names **myuart** and **myuart2**. The type of the table is ***DRIVER_NAME*_Config** and the name of the table is ***DRIVER_NAME*_ConfigTable**. The size of the table is the number of instances of the peripheral using this particular driver. Each element in the table is an ordered list of values given in the VALUES tag as shown above.

INT_HANDLER

This parameter defines the default interrupt handler software routine for an interrupt port of the peripheral. This parameter can be overwritten in the MSS file for this particular driver and peripheral instance.

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 INT_PORT keyword. This port should match the port name specified in the MHS file for that peripheral instance.

XILINX®

Chapter 20

Xilinx Libraries

Scope

This chapter describes the organization of Xilinx Libraries 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 C library contains the following components

- Xilinx file support functions **LibXil File**
- Xilinx memory file system **LibXil Mfs**
- Xilinx networking support **LibXil Net**
- 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 .**

Library Organization

The structure of **LibXil** is outlined in Figure 20-1. The user application calls routines implemented in **LibXil** and/or **libm**. In addition to the standard C routines supported by **libc.a**, Xilinx library LibXil 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 22, “LibXil File” chapter.
- Memory based file system (LibXil Mfs)

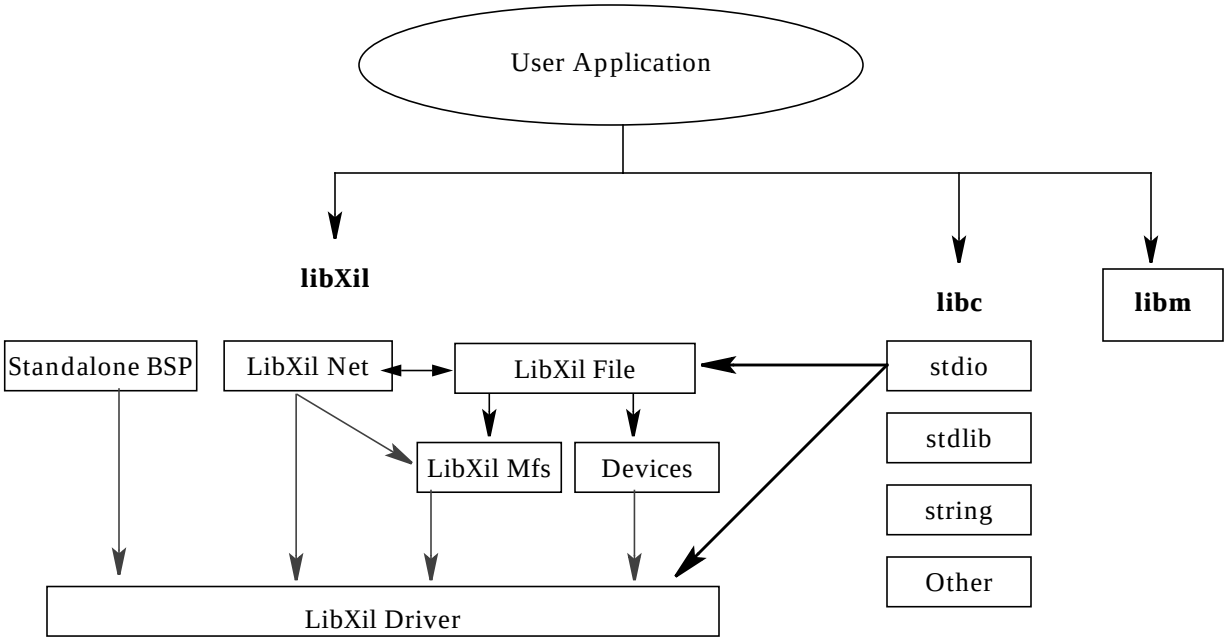


Figure 20-1: **Structure of LibXil library**

- 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 23, “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 the Chapter 24, “LibXil Net” chapter.
- 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 26, “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 27, “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 18, “Microprocessor

Software Specification (MSS)"chapter and Chapter 6, "Library Generator" chapter for more information.

Library 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 24, "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.

LibGen is used to tailor the library compilation for a particular project using attributes in the MSS. These attributes are described in theChapter 22, "LibXil File" and Chapter 23, "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

XILINX®

Chapter 21

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:

`mb-gcc` *`C files`* (for MicroBlaze)

`powerpc-eabi-gcc` *`C files`* (for PowerPC)

The **libc** library is included automatically.

The **-lm** option should be specified for programs that access libm math functions.

Refer to Chapter 4, "MicroBlaze Application Binary Interface" 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 18, "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 9, "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.

XILINX®

Chapter 22

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`*`)`
`int` **`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

`int 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.

The Libgen attribute **`LIBRARY`** indicates that **LibXil File** module should be compiled into the project specific Libraries.

To use Memory File System with LibXil File component, use the following code:

```
BEGIN FILESYS
parameter FILESYS_NAME = XilMfs
parameter PROC_INSTANCE = proc1
parameter MOUNT = "/dev/mfs"
parameter LIBRARY = XilFile
END
```

To access a device through Xilfile add the following to the peripheral description in the mss file

```
parameter LIBRARY = XilFile
```

All devices which have stream based input/output mechanism are supported through LibXil File.

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:

- 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

XILINX®

Chapter 23

LibXil Memory File System

Scope

This document describes the Memory File System (MFS). This file system resides on the 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 23-1 provides the function names with signature at a glance. C-like access.

Table 23-1: **MFS functions at a glance**

Functions
`void` **`mfs_init_fs`** `(void)`
`int` **`mfs_change_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_delete_file`** `(char *`*`filename`*`)`
`int` **`mfs_create_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_delete_dir`** `(char *`*`newdir`*`)`
`int` **`mfs_rename_file`** `(char *`*`from_file`*`, char *`*`to_file`*`)`
`int` **`mfs_exists_file`** `(char *`*`filename`*`)`
`int` **`mfs_get_current_dir_name`** `(char *`*`dirname`*`)`
`int` **`mfs_get_usage`**`(int *`*`num_blocks_used`*`, int *`*`num_blocks_free`*`)`
`int` **`mfs_file_open`** `(char *`*`filename`*`, int `*`mode`*`)`
`int` **`mfs_file_read`** `(int `*`fd`*`, char *`*`buf`*`, int `*`buflen`*`)`

Table 23-1: **MFS functions at a glance**

Functions
`int mfs_file_write (int fd, char *buf, int buflen)`
`int mfs_file_close(int fd)`
`int mfs_file_lseek (int fd, int 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 (void)`

Parameters	None
Returns	1 for success 0 for failure
Description	Initialize the memory file system. This function must be called before any file system operation.
Includes	xilmfs.h xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.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 xio.h

`int` **`mfs_file_lseek`** `(int` *`fd`*`, int` *`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
	xio.h

`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
	xio.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
	xio.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 xio.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 xio.h

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 23-2

```
BEGIN FILESYS
parameter FILESYS_NAME = XilMfs
parameter PROC_INSTANCE = proc1
parameter MOUNT = "/dev/mfs"
parameter LIBRARY = XilFile
END
```

Table 23-2: **Attributes for including Memory File System**

Attributes	Description
MOUNT	Mount name for the file system.
LIBRARY	Set this attribute to **XilFile** if the file system is accessed through XilFile component of the Libraries

XILINX®

Chapter 24

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 24-1 presents a list of functions provided by the LibXilNet at a glance.

Table 24-1: **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 24-1: **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 24-1 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.

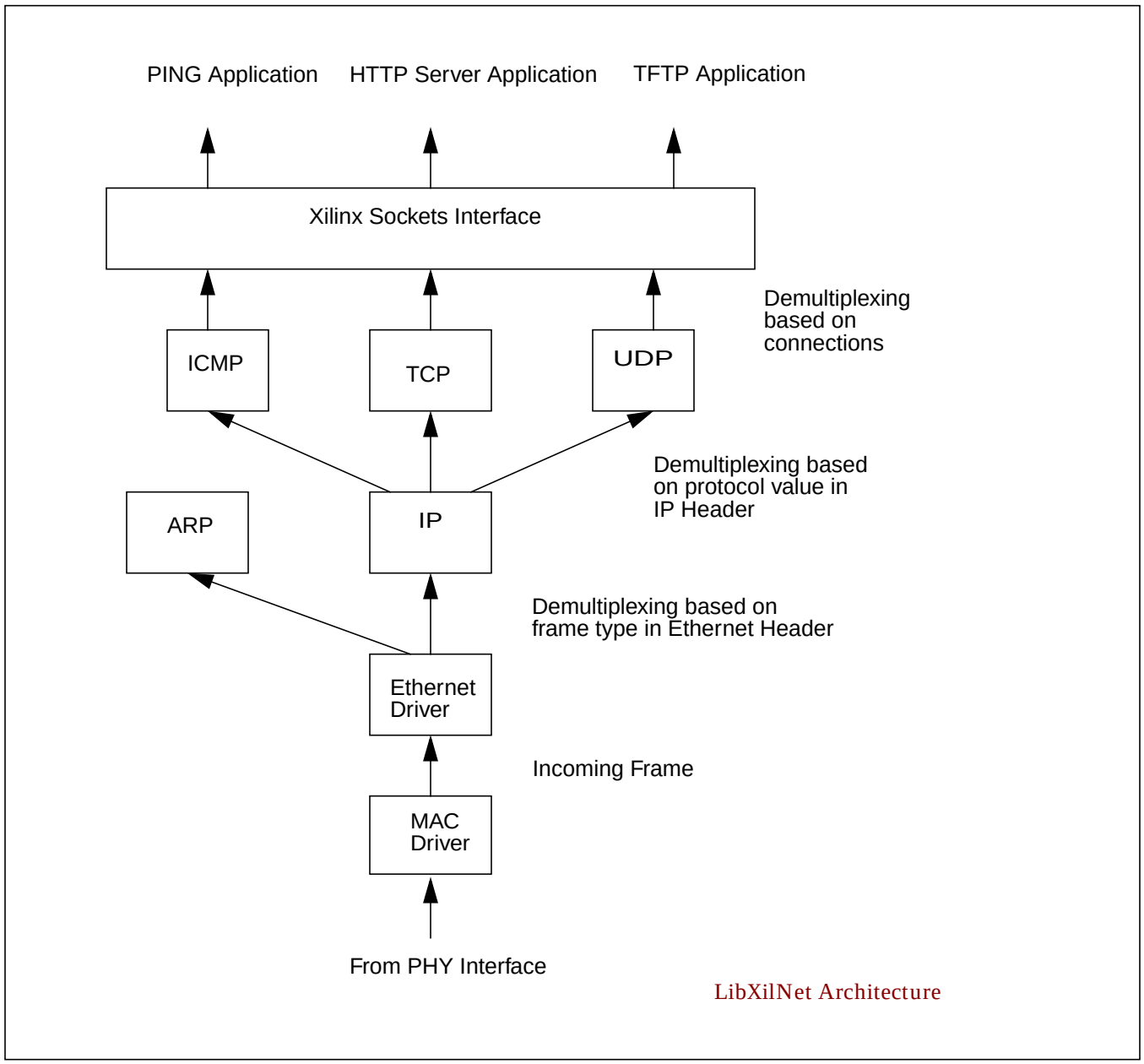


Figure 24-1: **Schematic Diagram of LibXilNet Architecture**

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 24-1 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 DRIVER
PARAMETER HW_INSTANCE = myether
PARAMETER DRIVER_NAME = emac
PARAMETER DRIVER_VER = 1.00.b
PARAMETER LIBRARY = XilNet
END

BEGIN PROCESSOR
 PARAMETER HW_INSTANCE = PPC405_i
 PARAMETER DRIVER_NAME = cpu_ppc405
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER EXECUTABLE = ppc405_i/code/executable.elf
 PARAMETER COMPILER = powerpc-eabi-gcc
 PARAMETER ARCHIVER = powerpc-eabi-ar
```

```
 PARAMETER DEFAULT_INIT=executable
 PARAMETER STDIN = myuart
 PARAMETER STDOUT = myuart
 PARAMETER EXTRA_COMPILER_FLAGS = "-D_CONFIG_EMAC_"
END
```

LibXilNet can be used with either the regular ethernet core or the lite version, ethernetlite.

When used with the the regular emac core, the following line should be added to the processor declaration block in system.mss

```
PARAMETER EXTRA_COMPILER_FLAGS = "-D_CONFIG_EMAC_"
```

When XilNet is used with ethernetlite core, the following line should be added to the processor declaration block in system.ms.

```
PARAMETER EXTRA_COMPILER_FLAGS = "-D_CONFIG_EMACLITE_
```

Using XilNet in Application

In order to use the XilNet functions in your application, you need to do the following:

- Define "`#include <net/xilsock.h>`" in your C-file.
- XilNet is designed to be used with any Media Access Control Protocol (for example, ethernet, SLIP, PPP. Currently it supports only ethernet protocol). In order to choose the right MAC protocol, the compiler flag, **-D_CONFIG_ETH_**, should be added to the compiler flags used for compiling your sources.

Chapter 25

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.

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 "Customization" section in detail. Figure 25-1 shows the various modules of the Xilinx embedded kernel.

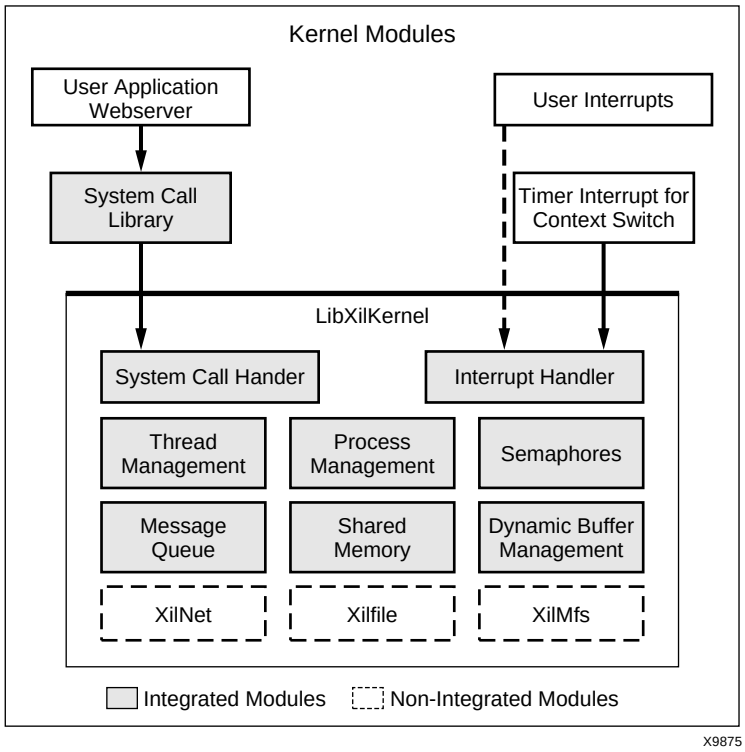


Figure 25-1: **Kernel Modules**

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

Figure 25-2 shows the process state flow in the system.

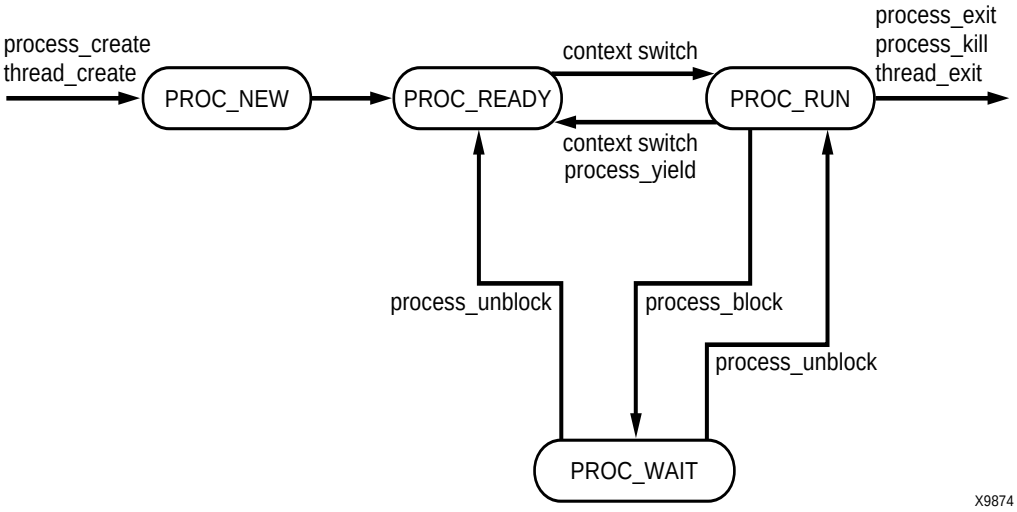


Figure 25-2: **Process State Flow**

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

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 "Customizing Thread Management" 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 "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 "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 memiry 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

Customization

LibXilKernel is highly customizable. Most of the modules and individual parameters can be changed to suit the user application. The user can directly modify the config files and build a system. Most configuration parameters are macros and some constant structures definitions. The following files should be modified to configure the system.

- os_config.h
- config_param.h
- config_cparam.h
- sys/init.c

The libxilkernel source is in lib/unsupported/xilkernel of EDK installation. All file paths are relative to this directory. The include files are in the include directory. Structures used for configuring start processes, message queue, shared memory and dynamic buffer management are declared in sys/init.h file. Following are the various structures:

struct _process_init - Information about the process.

- unsigned int *p_start_addr* - Start address of the process.
- int *priority* - Priority of the process.

struct _msgq_init - Information about each Message Queue.

- unsigned int *msg_size* - Size of the message.
- char *msgq_len* - Message queue length.

struct _shm_config - Information about each shared memory segment.

- unsigned int *shm_size* - Shared memory size.

struct _malloc_init - Information about the memory blocks.

- unsigned int *mem_bsize* - The size of the memory block.
- char *n_blocks* - Number of memory blocks to allocate.

The following sections describe customizing the different modules.

Customizing Process Management

The user can select the maximum number of processes in the system, the different functions needed to handle processes, the scheduling types, priorities and other parameters.

The os_config.h file can be used to configure the following.

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

The following macros are used to configure the scheduling scheme.

TIMER_TICKS - Timer tick value used for context switching

SCHED_TYPE - Select the type of scheduling scheme. The two different scheduling schemes and their configuration is specified in config_param.h file.

SCHED_RR - Round Robin Scheduling.

N_PRIO - Maximum number of priorities. This is always 1 for RR.

SCHED_PRIO - Pre-emptive Priority Scheduling.

N_PRIO - Maximum number of priorities.

MIN_PRIORITY - Lowest priority in the system.

MAX_PRIORITY - Highest priority in the system.

Processes can be statically created when the system is initialized. These processes can be specified in the following manner.

- The process start address and priority is specified in the variable struct _process_init *start_p[]*, which is defined in sys/init.c.
- The macro **N_INIT_PROCESS** is defined (config_cparam.h) to be the number of elements in start_p[] or the number of processes to create during system initialization.

Customizing Thread Management

The user can optionally select to include thread support, the maximum number of threads and size of the thread context. The following macros are used for configuration.

CONFIG_THREAD_SUPPORT - Include thread support modules. Defined in os_config.h.

MAX_THREADS - Maximum number of threads in the system. Defined in config_param.h.

THREAD_BSS_BSIZE - Size of the thread context memory. Defined in config_param.h.

Customizing Semaphore

The user can optionally select to include semaphores, maximum number of semaphores and semaphore queue length. The following macros are used for configuration.

CONFIG_SEMA - Include the semaphore module. Defined in os_config.h.

MAX_SEMA - Maximum number of semaphores in the system. Defined in config_param.h.

MAX_SEMA_WAITQ - Maximum length of the semaphore wait queue. Defined in config_param.h.

Customizing Message Queue

The user can optionally select to include message queue module, number of message queue and size of each message queue. The following macros and structure definitions are used for configuration.

CONFIG_MSGQ - Include message queue module. Defined in os_config.h.

The message size and message queue length is specified in the variable struct _msgq_init *msgq_config[]*, which is defined in sys/init.c.

N_MSGQ - is defined to be the number of elements in msgq_config[] or the number of message queue in the system.Defined in config_cparam.h.

MSG_QUEUE_MSIZE - is the total memory size for all message queue's in the system. The value is calculated from the definition of variable msgq_config[]. Defined in config_cparam.h.

$$\sum_{\forall(\text{msgq\_config[] elmts})} (\text{msg\_size} * \text{msgq\_len})$$

Customizing Shared Memory

The user can optionally select to include shared memory and size of each shared memory. The following macros and structure definitions are used for configuration.

CONFIG_SHM - Include shared memory module. Defined in os_config.h.

The shared memory size is specified in the variable struct _shm_init *shm_config[]*, defined in sys/init.c.

N_SHM - is defined to be the number of elements in shm_config[] or the number of shared memory segments in the system.Defined in os_config.h

SHM_MSIZE - is the total memory size for all shared memory segments in the system. The value is calculated from the definition of variable shm_config[]. Defined in config_cparam.h

$$\sum_{\forall(\text{shm\_config[] elmts})} (\text{shm\_size})$$

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 macros and structure definitions are used for configuration.

CONFIG_MALLOC - Include the buffer management module. Defined in os_config.h

The memory block size and number of memory blocks needed is specified in the variable struct _malloc_init *malloc_config[]*. Defined in sys/init.c

N_MALLOC_BLOCKS - is defined to be the number of elements in malloc_config[] or the number of different sized memory blocks. Defined in os_config.h

TOT_MALLOC_BLOCKS - is defined to be the total number of memory blocks in the system. Its value is calculated from the definition of variable malloc_config[].Defined in config_cparam.h

$$\sum_{\forall(\text{malloc\_config[] elmts})} (\text{n\_blocks})$$

MALLOC_MSIZE - is the total memory size for all memory blocks in the system.Its value is calculated from the definition of variable malloc_config[].Defined in config_cparam.h

$$\sum_{\forall(\text{malloc\_config[] elmts})} (\text{mem\_bsize} * \text{n\_blocks})$$

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 26

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.

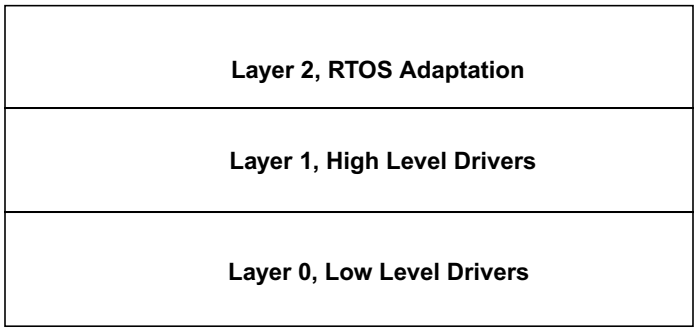


Figure 26-1: **Layered Architecture**

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
- 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 name>_<component instance>_DEVICE_ID
XPAR_<component name>_<component instance>_BASEADDR
XPAR_<component name>_<component instance>_HIGHADDR
```

<component instance> is typically a number between 0 and (XPAR_X<component name>_NUM_INSTANCES - 1). 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 name>_<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 is as follows:

```
XPAR_INTC_<instance>_<component name>_<component instance>_VEC_ID
```

Where INTC is the name of the interrupt controller component, <instance> is the component instance of the INTC, <component name> and <component instance> is the name and instance number of the component connected to the controller. Of course XPAR_INTC 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 name>_<component instance>_VEC_ID
```

Where <proc name> is the name of the processor.

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*.

Example

To help illustrate the relationships between these configuration files, an example is presented that contains a single interrupt controller whose component name is INTC and a single UART whose component name is (UART). Only xintc.h and xintc_g.c are illustrated, but xuart.h and xuart_g.c would be very similar.

xparameters.h

```
/* Constants for INTC */
XPAR_INTC_NUM_INSTANCES     1
XPAR_INTC_0_DEVICE_ID       21
XPAR_INTC_0_BASEADDR        0xA0000100

/* Interrupt vector assignments for this instance */
XPAR_INTC_0_UART_0_VEC_ID   0

/* Constants for UART */
XPAR_UART_NUM_INSTANCES     1
XPAR_UART_0_DEVICE_ID       2
XPAR_UART_0_BASEADDR        0xB0001000
```

xintc.h

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;
} XIntc_Config;
```

xintc_g.c

```
static XintcConfig[XPAR_INTC_NUM_INSTANCES] =
{
  {
     XPAR_INTC_0_DEVICE_ID,
     XPAR_INTC_0_BASEADDR,
  }
}
```

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.

XILINX®

Chapter 27

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.

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 x`cache_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 - 0x7FFFFF). 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 27-1 for a list of parameters.

Table 27-1: **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 27-2 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 27-2: **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

Table 27-2: **Registered Exception Types and Values**

Exception Type	Value
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 27-2.

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.

Chapter 28

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 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.

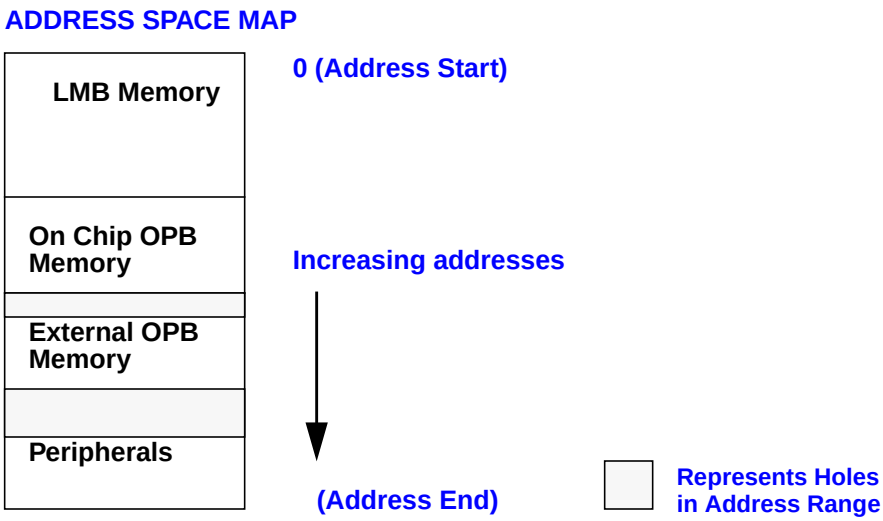


Figure 28-1: **A Sample Address Map for a MicroBlaze System**

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 4, "MicroBlaze Application Binary Interface" (ABI) 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 28-1 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 28-2(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.

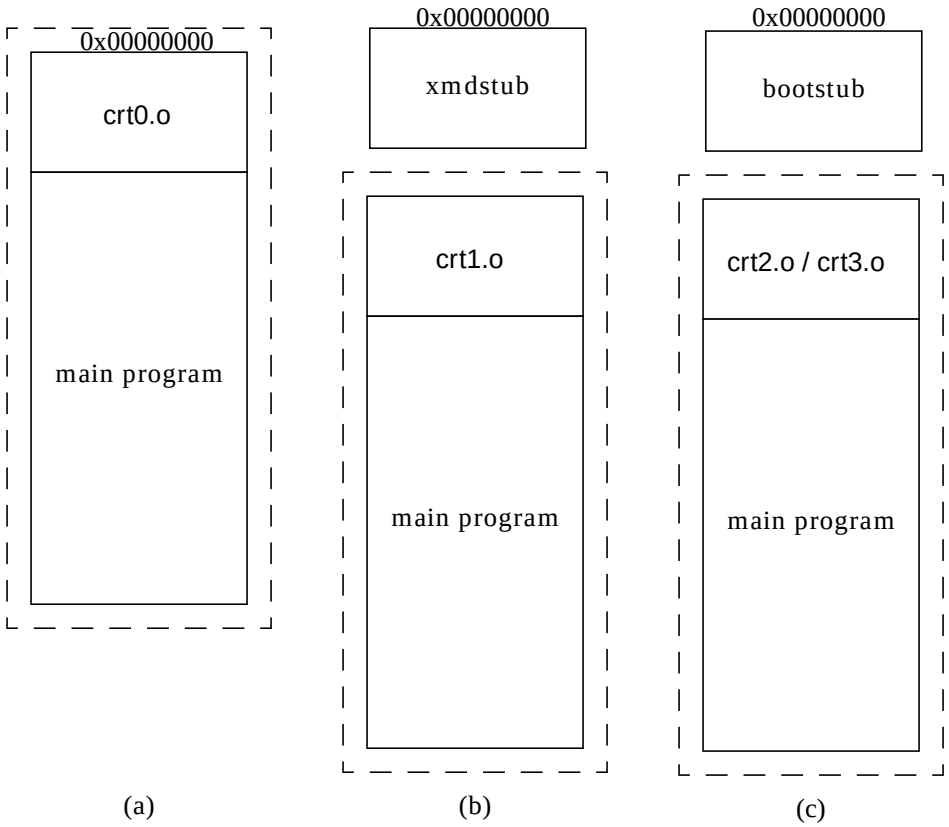


Figure 28-2: **Execution Scenarios**

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 28-2(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 28-2(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 28-1

Table 28-1: **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 28-1. 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 9, "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 28-3.:

.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 the Chapter 4, "MicroBlaze Application Binary Interface"chapter.

.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 the *Chapter 4, "MicroBlaze Application Binary Interface"* chapter.

.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

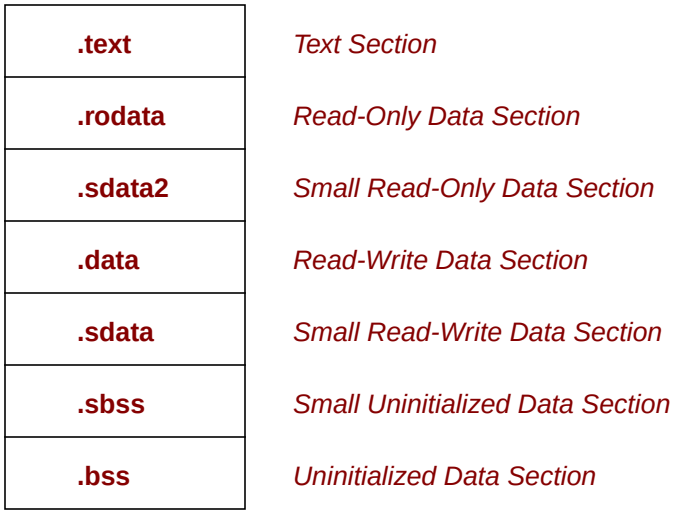


Figure 28-3: **Sectional layout of an object or executable file**

.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)
  } >OPB
  . = ALIGN(8);
  _essro = .;
  _ssro_size = _essro - _ssro;
 _SDA_BASE_ = _ssro + (_ssro_size / 2 );

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

Note that if you choose to write a linker script, you *must* do the following to ensure that your program will work correctly:

- 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 _SDA2_BASE_ variable points to the center of the .sdata2 area, and that _SDA2_BASE_ is aligned on a word boundary.
- 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.
- 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.

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 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 28-4 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.

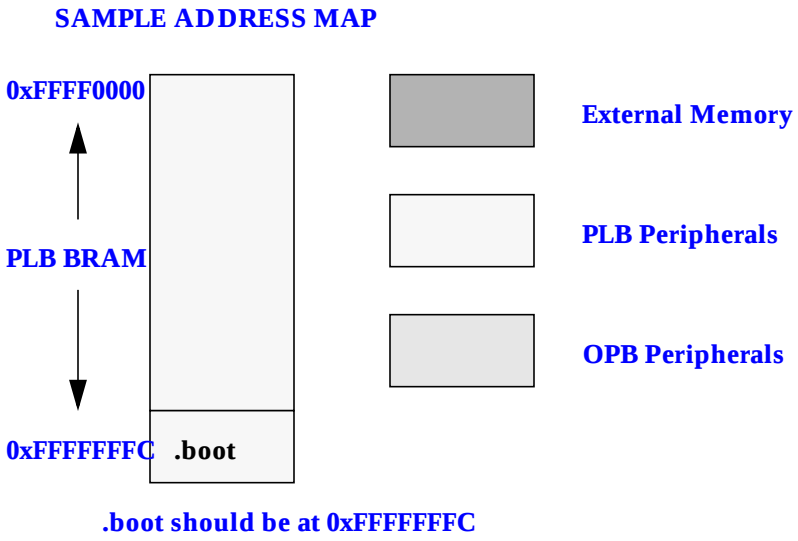


Figure 28-4: **A Sample Address Map for a PowerPC System**

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 9, "GNU Compiler Tools" chapter.

Linker Script

PowerPC Linker is built with default linker scripts. This script assumes a contiguous memory starting from address 0xFFFF0000. You can take a look at the default linker scripts used by the linker at:

$XILINX_EDK / gnu / powerpc-eabi / nt(orsol) / 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.

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:

- 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 script below for an example.
- Ensure that the variables __SDATA_START__. __SDATA_END__, SDATA2_START, __SDATA2_END__, __SBSS2_START__ , __SBSS2_END__, __sbss_start and __sbss_end are defined to the beginning and end of the sections sdata, sdata2, sbss2, sbss respectively.
- 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 .vector section is aligned on a 64k boundary
- 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 the default 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, LEN GTH = 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
{
  .boot0   : { *(.boot0)} > bram
  .text    : { *(.text) } > bram
  .boot    : { *(.boot) } > boot
  .data :
  {
    *(.data)
    *(.got2)
    *(.rodata)
    *(.fixup)
  } > bram

/*
 * .vectors section must be aligned on a 64k boundary
 * Hence the syntax BLOCK(64k)
 */

  .vectors BLOCK (64k):
  {
    *(.vectors)
  } > 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 29

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 26, “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 9, “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 29-1 shows peripheral interrupt signals with priorities 1 through 4 connected to the interrupt controller input.

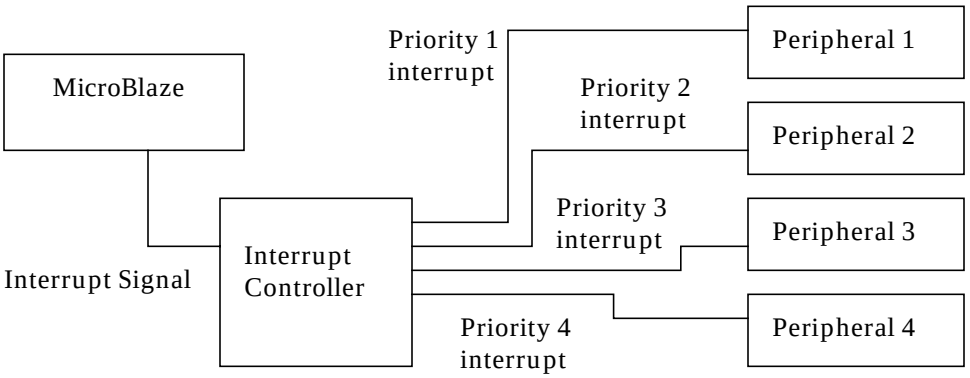


Figure 29-1: **Interrupt Controller and Peripherals**

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 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 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** (***USER_PROJECT/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 Chapter 18, "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.
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

```
parameter Version = 2.0.0

port sys_reset = sys_reset, DIR = input
port sys_Clk = sys_Clk, DIR = input

begin opb_v20
parameter HW_VER = 1.00.b
parameter INSTANCE = opb_bus
port SYS_Rst = sys_reset
port OPB_Clk = sys_Clk
end

BEGIN lmb_lmb_bram
parameter INSTANCE = lmb_lmb_bram_i
parameter HW_VER = 1.00.a
parameter C_BASEADDR = 0x00000000
parameter C_HIGHADDR = 0x00000fff
bus_interface ILMB = i_lmb
bus_interface DLMB = d_lmb
end

begin lmb_v10
parameter INSTANCE = d_lmb
parameter HW_VER = 1.00.a
port LMB_Clk = sys_Clk
port SYS_Rst = sys_reset
end

begin lmb_v10
parameter INSTANCE = i_lmb
parameter HW_VER = 1.00.a
port LMB_Clk = sys_Clk
port SYS_Rst = sys_reset
end

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 <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 */

/* 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)
      ;

}
```

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:
   - 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.
   - 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

```
parameter Version = 2.0.0

port sys_reset = sys_reset, DIR = input
port sys_Clk = sys_Clk, DIR = input

port rx = rx, DIR = input
port tx = tx, DIR = output

begin opb_v20
parameter HW_VER = 1.00.b
parameter INSTANCE = opb_bus
port SYS_Rst = sys_reset
port OPB_Clk = sys_Clk
end

BEGIN lmb_lmb_bram
parameter INSTANCE = lmb_lmb_bram_i
parameter HW_VER = 1.00.a
parameter C_BASEADDR = 0x00000000
parameter C_HIGHADDR = 0x00000fff
bus_interface ILMB = i_lmb
bus_interface DLMB = d_lmb
end

begin lmb_v10
parameter INSTANCE = d_lmb
parameter HW_VER = 1.00.a
port LMB_Clk = sys_Clk
port SYS_Rst = sys_reset
end
```

```
begin lmb_v10
parameter INSTANCE = i_lmb
parameter HW_VER = 1.00.a
port LMB_Clk = sys_Clk
port SYS_Rst = sys_reset
end

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 27, "Stand-Alone Board Support Package".