COMPAQ PCI Development Platform Reconfigurable Hardware Device for the PCI Bus

User's Guide

Part Number: EK–PAMP1–UG. B01

September 1999

Compaq Computer Corporation
Houston, Texas

September 1999

Table of Contents

Preface

1 Introduction

2 System Overview

3 Configuration and Installation

4 Software Installation

5 Module Architecture

6 Address Map

7 DMA Engine

8 Interface Modes

9 Security Considerations

10 Software

11 Command Line Controls

12 Restoring Original PCI Interface Design or Upgrading Firmware

Figures

Tables

Preface

Overview

The *COMPAQ PCI Development Platform Reconfigurable Hardware Device for the PCI Bus User's Guide* describes hot to install and configure the PCI Development Platform reconfigurable hardware module.

Organization

This guide is organized as follows:

Chapter 1 provides an introduction to the PCI Development Platform reconfigurable hardware module.

Chapter 2 provides an overview of the hardware, device driver, software, and updates.

Chapter 3 describes the hardware installation and configuration.

Chapter 4 describes the PCI Development Platform software kit installation.

Chapter 5 provides information on the physical design of the PCI Development Platform module.

Chapter 6 describes the PCI Development Platform module address map.

Chapter 7 describes the PCI Development Platform module DMA engine.

Chapter 8 provides information on the supported interface modes.

Chapter 9 contains information on security considerations.

Chapter 10 describes the run-time library functions.

Chapter 11 describes the command line tools for controlling the PCI Development Platform module and other utilities in the software package.

Chapter 12 provides information on restoring the original PCI interface design.

Appendix A contains a table listing the major user signal buses along with pin connections and their use.

Appendix B contains a table listing the special purpose/restricted use signals along with the pin connections present, their special use, and restrictions for user application.

Appendix C describes the PAM driver interfaces for TRU64 UNIX.

Appendix D describes the PAM driver interfaces for Windows NT.

Appendix E describes the registers in the PCI interface.

Conventions

This document uses the following conventions:

Convention	Meaning
Note	A note calls the reader's attention to any item of information that may be of special importance.
Caution	A caution contains information essential to avoid damage to the equipment.
Warning	A warning contains information essential to the safety of personnel.
Italic type	*Italic type* emphasizes important information, indicates variables, and indicates complete titles of manuals.
Bold type	**Bold type** indicates text that is highlighted for emphasis.
`Monospaced`	In text, this typeface indicates the exact name of a command, routine, partition, pathname, directory, or file.

Note

The TRU64 UNIX commands used in this manual are case sensitive and must be entered as shown.

The PCI Development platform is also referred to as Pamette. This is can been seen in the source code and some of the software commands.

Getting Help

You can reach Compaq automated support services 24 hours a day, every day, at no charge. The services contain the most up-to-date information about Compaq products. You can access installation instructions, troubleshooting information, and general product information from the Compaq World Wide Web server or from the Compaq FTP server.

Compaq Worldwide Web Server

http://www.compaq.com/support

Navigate to a specific product, then look for support information from this list of supported resources.

Compaq FTP Server

ftp.compaq.com

Navigate to the following for a complete list of available Softpaq files:

ftp.compaq.con/pub/softpaq/allfiles.html

Compaq Internet Site

Send e-mail to:

Support@compaq.com

PAQFAX

To use fax-on-demand system, you must be in North America and you must have a fax machine or fax modem to receive the automated fax transmittals.

To use the fax-based information retrieval system that provides product-specific information, call 1-800-354-1518, option 1, and request a product catalog. When you receive it, call and order documents you wish to receive.

Calling the Compaq Support Line

Compaq has technical support centers worldwide. Many of the centers are staffed by technicians who speak the local languages. For a list of Compaq support centers to go:

http://www.compaq.com

From the Compaq Welcome page, select your country, then select Support.

Before You Call

Before call the Compact Support Line, have the following information ready:

- Your address and telephone number
- A description of the failure
- A description of any action(s) already taken to resolve the problem (e.g., changing mode switches, rebooting the unit, etc.)
- A description of your network environment (layout, cable type, etc.)
- Network load and frame size at the time of trouble (if known)
- The device history

Customers may receive product support as follows:

- In North America, call Technical Support at 1-800-354-9000
- In Europe, the Middle East, and Africa, contact Compaq at http://www.compaq.com or refer to the following table for the Technical Support numbers for your specific country:

Country	Phone Number	FAX Number
Austria	01-546521552	01-878-1682
Belgium	078/155.500	(32) 070/222.080
Denmark	4590-4545	4590-4595
Fom;amd	0961 55 98 00	09-6155-9899
France	0803-81-38-23	01-41-33-44-31
Germany	0180-5212-111	0180-5212-117
Italy	02 48 23 00 23	02 48 23 00 02
Netherlands	0900-1681-616	0900-8991-116
Norway	22 07 20 20	22 07 20 21
Portugal	1 4128460	1 4120654
Spain	09 02 20 24 00	091 590 9333
Sweden	08-7030150	08 703 5222
Switzerland	01 838 22 22	01 836 44 06
UK	0990-561643	0171-744 0068
Ireland	(01) 214-1407	(0)1-214-1251

1
Introduction

1.1 General

This document describes the installation and use of the PCI Development Platform reconfigurable hardware module.

The PCI Development Platform module is a reconfigurable hardware device comprised of five Xilinx Field Programmable Gate Arrays (FPGAs). One FPGA controls the PCI bus interface while the other four are available for user configuration. The module also contains SRAM and four DRAM connectors that can be utilized at the user's discretion.

This document describes the following:

- Hardware architecture and functionality
- Installing the PCI Development Platform module
- Installing and configuring the device driver
- Configuring the user area gate arrays
- Support software

1.2 Operating System Support

The PCI Development Platform reconfigurable hardware module is supported on version 4.0 or higher of TRU64 UNIX or version 4.0 or higher of Windows NT. The Windows NT version is compatible with Intel and Alpha based systems, while the TRU64 UNIX version is for Alpha systems only.

1.3 Platform Support

The PCI Development Platform reconfigurable hardware module is supported on the TRU64 UNIX (Alpha) platform and the Windows NT (Alpha and Intel) platform with a PCI-compliant 32-bit or 64-bit bus and a free full-size PCI slot. Refer to the Software Product Information for the specific platforms that are supported.

1.4 Differences between Firmware v2.0 and v1.10

For readers already familiar with firmware v1.10, the principal new features in v2.0 are:

- User configurable delay between address and first data on AUI type transactions (See 8.5)
- Ability to read and write the Rings from software (See 8.5.5.3).
- Ability to stop the user clock under hardware control from user-area (See 8.5.6).

- Ability to accept and generate 64-bit PCI transactions in Transaction mode from the user-area (See 8.5.4).
- New simplified methods to raise PCI interrupts from user-area (See 8.5.5).
- Ability to detect partial word writes in Transaction mode through an interrupt (See 8.5.5).
- Ability to detect loss of PLL lock through an interrupt (See 8.5.5)
- A new Promiscuous mode which mimics the PCI-side timing of Transaction mode (See 8.4).
- More control bits are included in data transmitted to the user-area in Promiscuous mode (See 8.3).
- Ability to disable address stepping on PCI transactions initiated by PCI Development platform (See 9).
- Wait states introduced by the other party in a data transfer are now supported for all transactions, including transactions where PCI Development platform sources data (See x 8.5).

2
System Overview

2.1 Hardware

The PCI Development Platform module is a generic PCI board based on reconfigurable logic. The hardware is built around SRAM based Field Programmable Gate Arrays (FPGAs) from Xilinx Inc. These components can be infinitely reprogrammed in circuit. Programming time is measured in tens of milliseconds. The reconfigurable nature of the board makes it useful for an extremely broad class of applications. Figure 2-1 shows a PCI Development Platform module overview and Figure 2-2 shows the PCI Development Platform module.

Caution

It is possible to program the FPGAs in a manner that can cause physical damage to the module or system.

Figure 2-1 PCI Development Platform Module Overview

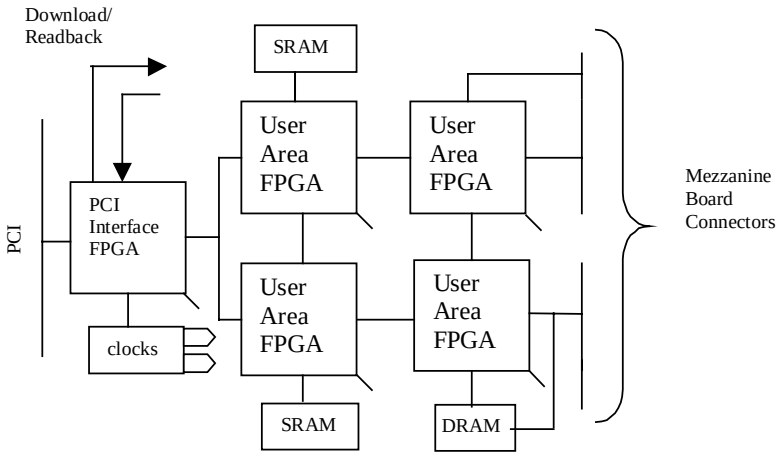
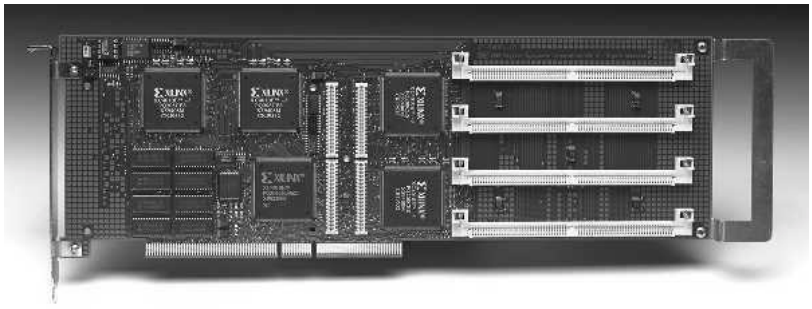


Figure 2-2 PCI Development Platform Module



2.1.1 PCI to User Area Interface FPGA

The board uses a Xilinx 4000 series FPGA to interface directly to a PCI bus. This FPGA is programmed to contain a proprietary Compaq PCI interface. The board may be used as a 5V, 33 MHz, 32-bit or 64-bit PCI target and initiator.

2.1.2 SROM and EEPROM

The PCI interface firmware is stored in both EEPROM and SROM. The firmware in the EEPROM is reconfigurable by the user. The PROM used at power-up is determined by the position of a failsafe jumper located in the top left corner of the board.

Caution

Although the firmware in the EEPROM is reconfigurable, any changes to this firmware may affect compatibility with the device driver and support software and will not be supported by COMPAQ.

2.1.3 User-Area FPGA

Behind the PCI interface FPGA (PIF) is a 2 x 2 user matrix of PQ208 footprints which, depending on the version, are populated with Xilinx 4000 series FPGA devices. The user matrix can be programmed from the host system via the interface FPGA. Devices in the user matrix can be programmed individually or in parallel with other user matrix devices. The matrix connects to the SRAM, DRAM modules, and IEEE P1386 daughter board connectors (Common Mezzanine Card/PCI Mezzanine Card).

2.1.4 SRAM and DRAM

In version one of the PCI development platform, two independent SRAM banks are provided, each is 64 k x 16-bits. These offer small fast scratch pad memory using 12ns SRAM. The version one board is identifiable by the eight SRAM part is the lower left corner. Version two has four SRAM parts in the same location.

In version two, the SRAM has been expanded to two independent SRAM banks that are each 128 k x 16-bits. The chip-select signals on version one are now used as address lines. This is possible because there is only one chip on each address and data line. The chip-enable signal is permanently enabled.

DRAM modules provide large amounts of local storage, which can be read/written in excess of 100 MB/s to provide bulk storage independent of the host memory system. The board has four angled 72-pin SIMM connectors. The data connections are shared. There are two address inputs, each going to two SIMMs. This allows the SIMMs to be operated in an interleaved manner. The data connection supports memories up to 36 bits wide. There are 12 address lines supporting up to four 64 MB modules. To meet the dimension limits of a single PCI slot, the

DRAM modules used should be 1-inch or less in depth. Deeper modules may be used provided the next slot is empty or it contains a short PCI card. Since all the logic controlling the SIMMs would be in user FPGAs, it would be possible to use any kind of 5V 72-pin SIMM.

2.1.5 Clocking Circuitry: System Clock and User Clock

The PCI Development Platform module has two independent clock systems, the user clock and the system clock, each of which is distributed to all of the FPGAs. The system clock is a copy of the PCI clock that has been recovered with a PLL. The system clock provides an in-phase copy of the PCI clock at the PCI frequency or double the PCI frequency.

The user clock is the output of a programmable frequency generator. The frequency range of the user clock is 400 kHz to 100 MHz with a resolution of about 0.5%. The user clock can be stopped, stepped, or double-stepped under software control. The user clock has no defined phase relation to the PCI clock.

2.1.6 Mezzanine Cards

The board provides a daughter card facility conformant with the IEEE P1386 Common Mezzanine Card (CMC) standard. Short, single width, mezzanine cards are supported.

With appropriate user FPGA programming, the mezzanine card may use PCI protocols (IEEE P1386.1) that are 32-bit or 64-bit wide with 64 extra uncommitted I/Os, or a custom protocol may be developed using only the P1386 layer. Thus, the mezzanine card allows electrical adaptation for external connections through the development of simple low-cost adapter cards using a custom protocol as well as the direct connection of standard commercially available PCI Mezzanine Cards (PMC).

2.1.7 DMA Engine

The PCI Development Platform module contains a simple but flexible DMA engine controlled by two registers. The DMA engine only supports 32-bit aligned addresses, but does support 32-bit and 64-bit data transfers. The following burst order and data widths are supported:

- Linear increment
- Intel cache-line wrap mode
- Cache-line wrap
- Linear increment (request 64-bit)

2.2 Software

The following sections describe the software used with the PCI Development Platform module.

Table 2-1 lists the software used with the PCI Development Platform module.

Table 2-1 PCI Development Platform Software

Software	Platforms	Distributed as	Language
Drivers	TRU64 UNIX (Alpha) Windows NT (Alpha & Intel)	Binary and Source	C
PamRT run-time library	TRU64 UNIX (Alpha) Windows NT (Alpha & Intel)	Binary and Source	C
Testing software	TRU64 UNIX (Alpha) Windows NT (Alpha & Intel)	Binary and Source	C
Support tools	TRU64 UNIX (Alpha) Windows NT (Alpha & Intel)	Binary and Source	C

2.2.1 Device Driver

The PCI Development Platform drivers allow the board to be mapped to a user application's address space and provide support for allocation and translation of memory, and fielding of interrupts.

2.2.2 Related Software

The accompanying software contains three major items: run-time libraries, testing software, and support tools.

The `PamRT.c` language run-time library allows the user to write applications to control the board and communicate with user developed circuits that have been downloaded into the board. It also supports download and read-back of user circuits, setting clocks, and other functions.

Testing software consists of `PamTest`, a verification test used to ensure that the board is performing some basic functions and the hardware is reliable. The PCI Development Platform module is a programmable gate array board; therefore, the functional test software focuses its' coverage on the communication between the board components. In other words, PamTest is not a design verification tool.

The Xilinx design tools create individual bit streams for each FPGA. These can be downloaded as a single entity, simplifying the downloading procedure of designs. The support tools provided by COMPAQ are used to concatenate the bit streams together when downloading to the Xilinx user FPGAs and for reading bit streams from the FPGAs and the firmware from the EEPROM. The support tools also include an application that can be used to update the PIF firmware in the EEPROM.

2.3 Updates

For updates on the current software and firmware kits, see the World Wide Web site at locations:

http://www.research.compaq.com/SRC/pamette/

http://www.compaq.com/customsystems/platforms/realtime_manu.html

This site also contains the answers to the most frequently asked questions and other miscellaneous information concerning the PCI Development Platform module.

3
Configuration and Installation

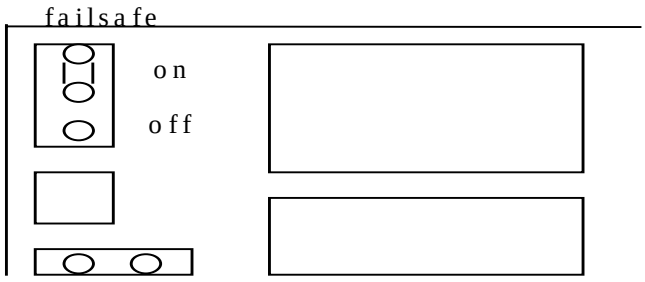
3.1 Hardware Configuration

The PCI Development Platform module communicates with the host system across the PCI bus via the PCI compliant PCI interface FPGA (PIF). This FPGA may be reprogrammed after initial boot-up, but it is recommended that the COMPAQ supplied PCI design is loaded into this FPGA during initial boot-up. This allows for initial confirmation of communication between the PCI Development Platform module and the platform.

3.1.1 Failsafe Jumper

A failsafe jumper is provided to select the source of the configuration bit stream for the PCI interface FPGA. As shipped, the SROM and EEPROM are loaded with the COMPAQ PCI compliant firmware. When the failsafe jumper is located in the "on" position (see **Error! Reference source not found.**), the PCI interface FPGA is loaded with the firmware from the SROM. When the failsafe jumper is located in the "off" position (default), the PCI interface FPGA is loaded from the EEPROM. The original PCI interface firmware can always be restored by rebooting after placing the failsafe jumper in the "on" position (see Chapter 12).

Figure 3–1 Failsafe Jumper and SROM Configuration for Boot-up



3.2 Hardware Installation

The PCI Development Platform module can be installed in either a 32-bit or a 64-bit PCI bus. A mezzanine card (CMC or PMC) can be connected to the PCI Development Platform module using the standoff pin and mezzanine connector. There is also space for four Dynamic RAM modules. For initial boot-up and testing, a mezzanine card and DRAM are not needed. Physical installation of the module should be performed with the host platform powered off.

4
Software Installation

4.1 Introduction

The PCI Development Platform Software Kit contains two CDs (one for Windows NT and one for TRU64 UNIX). Each disk contains a device driver, run-time library, schematics, and this user guide. The installation process for TRU64 UNIX and Windows NT is different, so they are covered separately in the following sections.

4.2 TRU64 UNIX Software Kit Installation

The TRU64 UNIX PCI Development Platform Software Kit is installed by using the `setld` command.

Note

The TRU64 UNIX commands are case sensitive and must be entered as shown. The PCI Development platform is also referred to as Pamette. This is can been seen in the source code and some of the software commands.

4.2.1 Deleting Old Device Driver Revisions

If there is an old version of the PCI development platform software kit on the system, it should be removed before installation or reinstallation of the new kit. The easiest way to do this is using the `setld` command. This command (`setld –i`) lists all the currently installed software subsets. The correct command and sample output are shown below:

```
# setld –i | grep –i pam
PRLPPAMDRV247        installed  PAM PCI Device Driver
```

An installed software subset can be removed using the following command:

```
# setld –d {software subset name as shown above}
```

For example, to remove the PAM PCI device driver above use:

```
# setld –d PRLPPAMDRV247
```

All installed software subsets associated with the PCI development platform software kit should be deleted before installing the new software kit. The number 247 on the software subset is just an example. The number will vary depending on the previously installed version.

Preparing for Kit Installation

After inserting the *Tru64 UNIX Device Driver for the PCI Development Platform Reconfigurable Hardware Array* CD-ROM in the system's CD-ROM device, mount it and change the working directory. For example:

```
# mount -r /dev/rz4c   /mnt
# cd /mnt/PAMDRV
```

Note

The actual CD-ROM device and mount point may be different from above.

4.2.2 Installing the Kit

The PCI development platform software kit is installed using the standard `setld` command. The command for installation and corresponding output are shown below. The command (`setld -l .`) should be executed from the same directory that the driver package is in.

```
# setld -l .
The subsets listed below are optional:
There may be more optional subsets than can be presented on a
single screen. If this is the case, you can choose subsets screen
by screen or all at once on the last screen. All of the choices you
make will be collected for your confirmation before any subsets are
installed.
    1) C++ Logic Description Library
    2) Miscellaneous Support Tools
    3) PAM PCI Device Driver
    4) PAM kit overview (html)
    5) Pam Source Packages
    6) Pamette Runtime Library
    7) Pamette Sample Applications
    8) Pamette Test Programs
--- MORE TO FOLLOW ---
Enter your choices or press RETURN to display the next screen.
Choices (for example, 1 2 4-6): 1-8
Or you may choose one of the following options:
     8) ALL of the above
     9) CANCEL selections and redisplay menus
     10) EXIT without installing any subsets
Add to your choices, choose an overriding action or press RETURN to
confirm previous selections.
Choices (for example, 1 2 4-6):  1-8
```

```
You are installing the following optional subsets:
     C++ Logic Description Library
     Miscellaneous Support Tools
     PAM PCI Device Driver
     PAM kit overview (html)
     Pam Source Packages
     Pamette Runtime Library
     Pamette Sample Applications
     Pamette Test Programs
Is this correct? (y/n): y
7 subset(s) will be installed.
Loading 1 of 8 subset(s)....
PAM PCI Device Driver
   Copying from . (disk)
   Verifying
Loading 2 of 8 subset(s)....
Pamette Test Programs
   Copying from . (disk)
   Verifying
Loading 3 of 8 subset(s)....
Pamette Sample Applications
   Copying from . (disk)
   Verifying
Loading 4 of 8 subset(s)....
Pamette Runtime Library
   Copying from . (disk)
   Verifying
Loading 5 of 8 subset(s)....
PAM kit overview (html)
   Copying from . (disk)
   Verifying
Loading 6 of 8 subset(s)....
Pamette Source Package
   Copying from . (disk)
   Verifying
Loading 7 of 8 subset(s)....
C++ Logic Description Library
   Copying from . (disk)
```

```
   Verifying
Loading 8 of 8 subset(s)....
Miscellaneous Support Tools
   Copying from . (disk)
   Verifying
8 of 8 subset(s) installed successfully.
Configuring "PAM PCI Device Driver" (PRLPPAMDRV250)
To enable this functionality, rebuild the system kernel using
doconfig with no command flags.
Configuring "Pamette Test Programs" (PRLPAMTEST222)
Configuring "Pamette Sample Applications" (PRLPAMSAMP218)
Configuring "Pamette Runtime Library" (PRLPAMRT161)
Configuring "PAM kit overview (html)" (PRLPAMDOC280)
Configuring "Pam Source Packages" (PRLPAMSRC280)
Configuring "C++ Logic Description Library" (PRLPAMDC238)
Configuring "Miscellaneous Support Tools" (PRL1UTL125)
```

4.2.3 File List

After the kit is installed, the `setld -i` command will display all of the currently installed subsets. The following subsets are installed during the PCI development platform installation:

```
PRL1UTL125Miscellaneous Support Tools
PRLPAMDC238      C++ Logic Description
PRLPAMRT161      Pamette Runtime Library
PRLPAMSAMP218    Pamette Sample Applications
PRLPAMSRC280     Pam Source Packages
PRLPAMDOC280     PAM kit overview (html)
PRLPAMTEST222    Pamette Test Programs
PRLPPAMDRV250    Pam PCI Device Driver
```

By appending the subset name to the `setld -i` command, information on content and location of the files within the subset is revealed. The following example shows the contents of the Pam PCI Device Driver. The device driver is in the `PRLPPAMDRV241` subset. This name is appended to the `setld -i` command as follows.

```
setld -i PRLPPAMDRV250
```

The resulting file list is show below.

```
./usr/opt/PRLPPAMDRV250
./usr/opt/PRLPPAMDRV250/config.file
./usr/opt/PRLPPAMDRV250/files
./usr/opt/PRLPPAMDRV250/pci_data
./usr/opt/PRLPPAMDRV250/ppam.c
./usr/opt/PRLPPAMDRV250/stanza.static
./usr/sys/include/sys/ppamio.h
```

4.2.4 Rebuilding the Kernel

After the driver is installed using the `setld command`, the kernel has to be rebuilt and the system has to be rebooted. The kernel can only be rebuilt by root, using the following command:

```
# doconfig
```

The user is then prompted to decide what options to be included with the new kernel. This configuration is typically left up to the system administrator. After the kernel is rebuilt, it should be moved to the root directory.

```
# mv /sys/{system name}/vmunix  /
```

The *system name* may be the computer name or a different name assigned by the system administrator. After the `doconfig` command has finished building the new kernel, the path to the new kernel will be shown. It is the kernel in this path that should be moved to the root directory.

4.3 Windows NT Software Installation

The Windows NT PCI development platform software is installed by using an installation shield.

4.3.1 Installing the Software

To install the PCI Development Platform software, place the CD-ROM in the CD-ROM drive. From the **start** button, go to **run** and execute `D:\setup_alpha` or `D:\setup_intel` (this assumes the CD-ROM drive letter is D). An installation shield will load. If a version of the Development platform software is currently installed, the user is prompted for permission to remove the old version. It is recommended that the user remove older versions of the software kit before proceeding. The license agreement is presented next. After reading the agreement, click the **next** button. The software allows different subsets to be installed. It is recommended that the PCI Development platform Driver and PamRT subsets be installed. The PamDC and Samples subsets are optional. The installation can now proceed by clicking the **next** button. Follow the directions on the screen to complete the installation of the software kit.

4.3.2 Deleting old NT Software Revisions

The software kit can be removed by going from **start** -> **settings** -> **control panel,** on the Windows NT menu bar. In the control panel window, double click the **Add/Remove Programs** icon. Click on the **Pam Software Kits** line in the lower window, then click the **Add/Remove** button and confirm the removal of the program.

4.4 Installation Verification Procedure

The following sections provide the procedure for running tests to verify proper hardware and software installation.

4.4.1 PamTest

After the software is installed and the system is rebooted, the module's operating condition can be tested with the `PamTest` utility. The following code displays the options associated with `PamTest`. It is recommended that the user begin with `PamTest –C 0`. (Chapter 11 describes the other software utilities that can be used to control the module from the command line in Windows NT or TRU64 UNIX.) The executable for `PamTest` is located in the `/usr/bin` directory under TRU64 UNIX and Windows NT executable is located in `c:\Pam\bin` directory. The following statement describes the use of the `PamTest`. On Windows NT systems, `PamTest` is run from the MS-DOS prompt as shown below:

```
C:\pam\bin\PamTest –C 0
```

On TRU64 UNIX systems, the PamTest is run from the command line as shown below:

```
/usr/bin/PamTest –C 0
```

The appropriate result for TRU64 UNIX and Windows NT is shown below. If PamTest does not run and a runtime error results, please check the installation procedure.

```
--  PamTest of Jun 13 1999 15:18:11         --
Board : PCI Pamette V1R1      Firmware : 2.0      Serial Number : 0
Config : 4044XLA 4044XLA 4044XLA 4044XLA
Download  OK
Clock     OK
Connect   OK
Readback  OK
Sram      OK
Intr      OK
Ebus      OK
```

Note:

The PamTest application is currently designed for PCI buses that run at 33.3 MHz. Many PCI buses do not meet this requirement (the PCI specification only requires a PCI bus to operate at less than or equal to 33.3 MHz). This would cause the PamTest application to fail during the clock test. If PamTest reports a clock frequency less than 33.3 MHz, your module may be functioning correctly. To check the functionality of the board run the following PamTest option, which excludes the clock test.

On Windows NT systems, run the following command from the MS-DOS prompt:

```
C:\pam\bin\PamTest –C 0 –e clock
```

On TRU64 UNIX systems, the appropriate command is shown below:

```
/usr/bin/PamTest –C 0 –e clock
```

The appropriate result for TRU64 UNIX and Windows NT is shown below.

```
--  PamTest of Jun 13 1999 15:18:11         --
Board : PCI Pamette V1R1      Firmware : 2.0      Serial Number : 0
Config : 4044XLA 4044XLA 4044XLA 4044XLA
Download  OK
Connect   OK
Readback  OK
Sram      OK
Intr      OK
Ebus      OK
```

4.4.2 SRAM Test

SRAM Test checks the functionality of the onboard SRAM. In testing the SRAM, the module programs the FPGAs to complete the connection between the PCI interface and the SRAM memory; therefore, this test adds coverage to the `PamTest` utility. The following statement describes the use of the SRAM Test. On Windows NT systems, the SRAM Test is run from the MS-DOS prompt as shown below:

```
C:\pam\bin\sramtest
```

On TRU64 UNIX systems, the SRAM Test is run from the command line as shown below:

```
/usr/bin/ppam_sramtest
```

The SRAM test returns a passed message if the SRAM is functioning correctly:

```
Sramtest passed
```

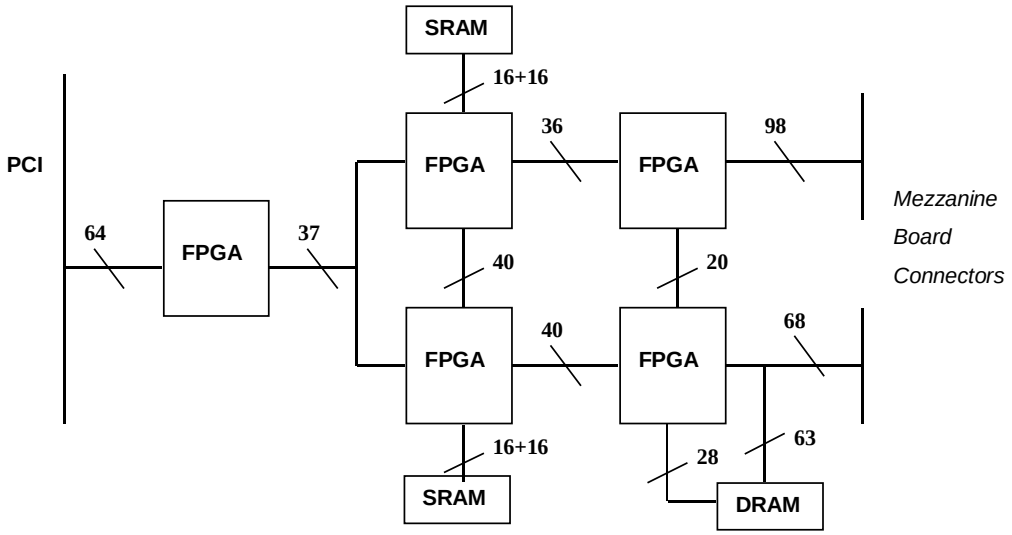
5
Module Architecture

5.1 Introduction

To completely utilize the functional and programmable capabilities of the PCI Development Platform module, the designer must have an in-depth knowledge of the interconnections on the module, the resources within the FPGA, and the accompanying software including run-time libraries and drivers. The accompanying schematics and software code, along with the technical specifications for the FPGAs, are necessary for implementing an actual design.

To understand the host interface of the PCI Development Platform module, one must first be familiar with the physical resources that exist for communication between the PCI interface FPGA (PIF), which has a relatively fixed configuration, and the four user-area FPGAs, which are programmed with application specific configurations. In the schematics[1], the PIF is called the `pcilca.` The `usrlca0, usrlca1, usrlca2,` and `usrlca3` comprise the user-area. The overall architecture of the PCI Development Platform module is shown in Figure 5-1. The available resources for the PIF user-area interface are presented in the following sections.

Figure 5-1 PCI Development Platform Module Architecture



[1] The schematics are contained on the installation disk as a postscript file, ppschema.ps.

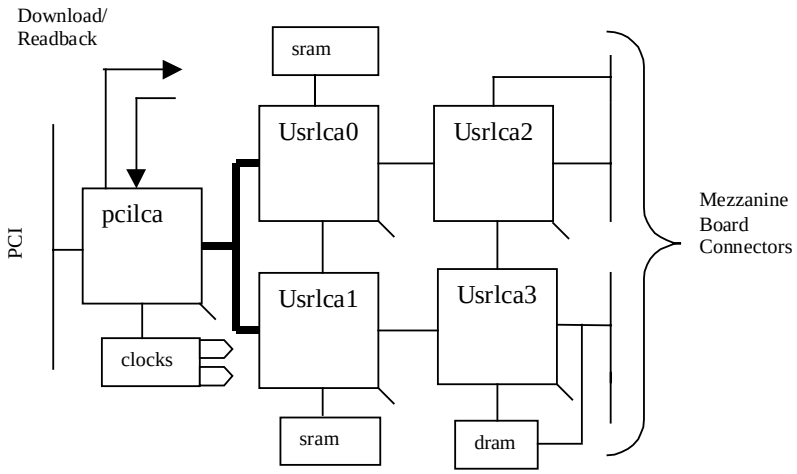
5.2 PCI Interface to User-Area Connections

The following sections describe the PCI interface to user-area connections.

5.2.1 EBus

`EBus<35:0>` (east bus) and `CnfgP_ld.din`[2] constitute a 37-bit wide bus joining the `pcilca, usrlca0,` and `usrlca1` FPGAs (see Figure 5-2). `CnfgP_ld.din` also connects to the download input of the PIF and its serial ROM and is used during PIF configuration and firmware upgrades, but during normal operation it is treated as an extension of `EBus<35:0>`.

Figure 5-2 EBus Connections

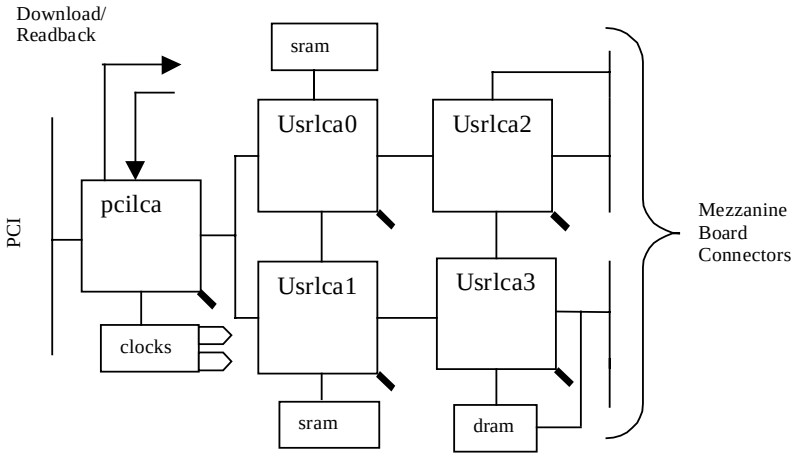


5.2.2 Rings

The `Ring<1:0>` is a 2-bit wide bus that connects to all five FPGAs (see Figure 5-3). The `Ring` pins are privileged in that they connect to pins that can directly drive FPGA global buffers. They can be used as general-purpose resources or programmed as interrupts. Refer to the interrupt section 8.5.5.3.

[2] Refer to the schematics for reference on all physical components.

Figure 5-3 Ring Bus



5.3 Clocks

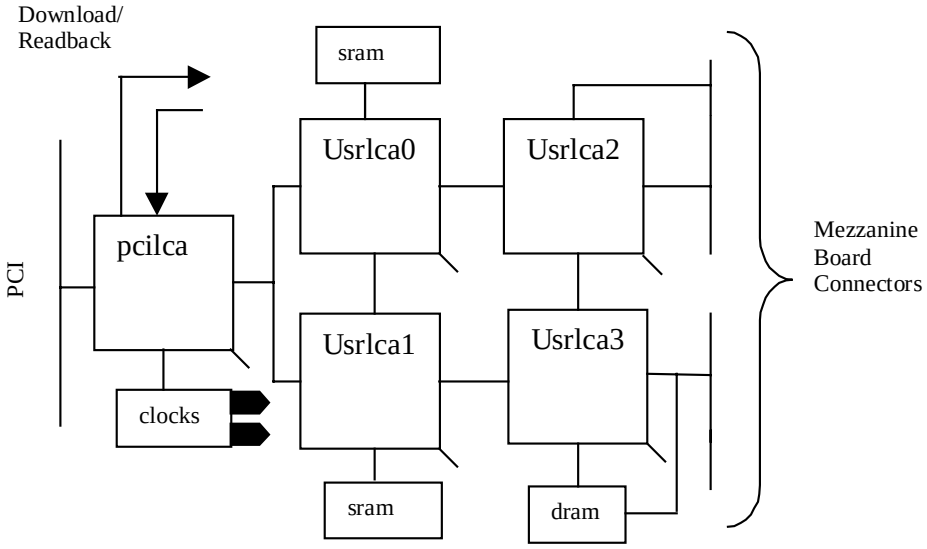
The PCI Development Platform module has two independent clock systems, each of which is distributed to all FPGAs (see Figure 5-4).

Clksys is a copy of the PCI clock that has been recovered with a Phase-Locked Loop (PLL). The clock recovery circuit is a Motorola MC88915T. In addition to producing a buffered low skew copy of the input, Clksys can, under software control, be put into a mode in which its output is double the frequency of the PCI clock. Thus Clksys provides an in-phase copy of the PCI clock at the PCI frequency or double the PCI frequency. This PLL can also produce an in-phase copy of an externally supplied clock (over the mezzanine connector). The clock source selection signal is driven from pin 5 on `Usrlca3`.

Clkusr is the output of an ICD2053B programmable frequency generator. This clock can be set to frequencies in the range 400 kHz to 100 MHz in steps of about 0.5%. Additionally, Clkusr clock can be *stopped*, *stepped*, or *double-stepped*[3] under software control. Clkusr has no defined phase relation to the PCI clock.

[3] Double-stepped means from the stopped state two clicks are issued by Clkusr after which the clock returns to the stopped state.

Figure 5-4 Clocks



5.4 SRAM

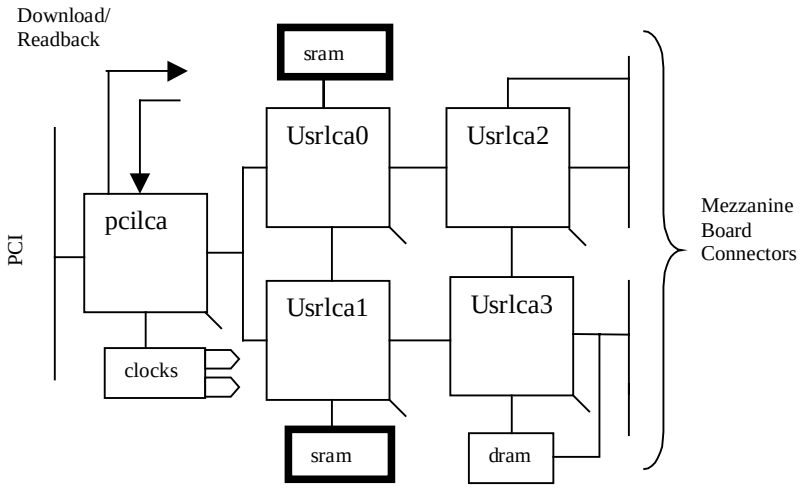
The Development Platform module has two separate SRAM banks (see Figure 5-5), one for `Usrlca0` and the other for `Usrlca1`.

For version 1, each bank contains 128 Kbytes of storage. This SRAM is fast asynchronous memory capable of operating at clock speed above 80 MHz.

For version 2, each bank contains 256 Kbytes of storage. This SRAM is also fast asynchronous memory and it is capable of operating at clock speed above 80 MHz. See section 2.1.4 for an explanation of the difference between the two versions.

The SRAM can be used to hold state bits, data in intermediate stages of computation or other necessary storage. Larger amounts of data can be stored in DRAM modules, which may be placed directly on the module and is discussed in Section 5.5.

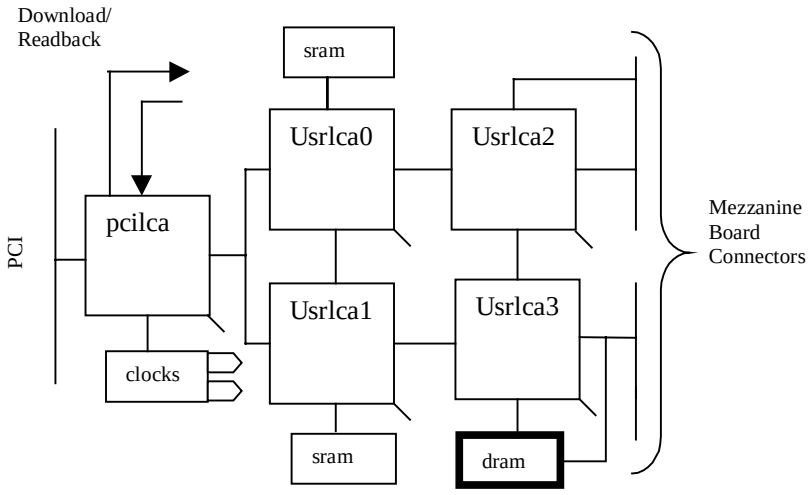
Figure 5-5 SRAM Banks



5.5 DRAM

Four 72-pin SIMMs DRAM connectors are present on the Development Platform module. The DRAM is connected between the mezzanine card connectors and Usrlca3 (see Figure 5-6). There is no DRAM controller on the module; consequently, any 72-pin SIMMs can be used on the board. The appropriate control logic for the SIMMs needs to be placed in Usrlca3 or on the daughter board. The DRAM is meant for storage of bulk data, while the SRAM provides fast access to smaller amounts of data.

Figure 5-6 DRAM

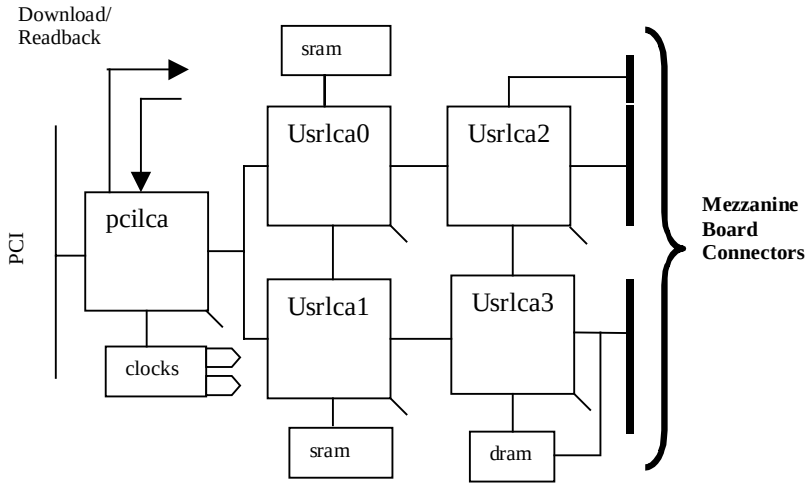


5.6 Mezzanine Connectors

The Development Platform module supports short, single width, mezzanine cards conformant with the IEEE P1386 Common Mezzanine Card Standard. The module also has the capability of supporting PCI mezzanine Cards. Similar to the DRAM connectors, the control logic for the mezzanine cards has to be developed for each application. `Usrlca2` and `Usrlca3` connect directly to the mezzanine card.

When the mezzanine card is placed on the Development Platform module, one end of the card adjoins the PCI bracket. This allows external connectors (see Figure 5-7) and cabling to attach directly to the daughter board through an opening in the PCI bracket. The use of a custom daughter board allows the Development Platform module to adapt to a variety of applications. Daughter boards can be used for data collection and other I/O communications. Specialized functions, such as JPEG and MPEG compression, can be accomplished by fitting daughter boards with application specific chips or processors.

Figure 5-7 Mezzanine Connectors



6
Address Map

6.1 Introduction

PCI Development Platform decodes its configuration space (mandatory for all PCI devices), and a single 16 MB memory space region. The `pam` value returned by `PamOpen` is a pointer to the base of the memory space region. The memory space region is subdivided as follows:

Address range	Contents
0x0 ... 0x3f	`struct PamRegs`
0x40 ... 0x3ff	15 more aliased copies of *PamRegs*
0x400 ... 0x1fff	< reserved >
0x2000 ... 0x200b	DMA engine
0x200c ... 0xffff	< reserved >
0x10000 ... 0x1003f	aliased copy of *PamRegs* with *secure* access
0x10040 ... 0x103ff	15 more aliased copies of *PamRegs*
0x10400 ... 0x11fff	< reserved >
0x12000 ... 0x1200b	aliased copy of DMA engine
0x1200c ... 0x1ffff	< reserved >
0x20000 ... 0x1000000	user transaction region

Note

Most users only need to know that the first 128 KB of address space is treated differently from the rest.

- `struct PamRegs` is defined in <`PamRegs.h`>, which is part of the PCI Development platform run-time software kit.
- The regions labeled *reserved* in the memory space address map are used primarily for PCI target performance testing. In these regions, writes are NOPs and reads return unpredictable data.

  The *reserved* regions below 0x20000 which have address bit 14 set (0x4000), have the special property that they accept 64-bit transactions. (In PCI protocol terms, when transactions are addressed to these regions with the PCI signal REQ64 asserted, ACK64 is asserted in response, in all other regions ACK64 is never asserted).

User software that writes to currently reserved regions must be reviewed with each new firmware revision to ensure that new functions have not been assigned to these regions. A prudent coding practice for such code would be to refuse to run on firmware revisions higher than the current released revisions, thereby obliging the programmer to review it at each new revision. Recall that the only code that should need to access reserved regions is PCI target performance testing code.

- The address range 0x10000 ... 0x1ffff is essentially an alias for the range 0x0 ... 0xffff with the additional property that certain security sensitive bits in `struct PamRegs` can only be written by accesses to this range. PCI Development platform device drivers may only permit privileged users to map the range 0x10000 ... 0x1ffff. See Chapter 9 for more discussion of the security bits.
- Access to some fields in aliased copies of `PamRegs` may use `addr<9..6>` as an argument to the access. See the source code of PamRT for details.

7
DMA Engine

7.1 Introduction

The PCI Development Platform module contains a simple but flexible DMA engine controlled by two control registers starting at address 0x2000. These DMA registers are also accessible from the user-area FPGA (see Section 8.5.3). Access to the DMA engine is disabled unless the appropriate security bit is set (see Chapter 9).

7.2 DMA Address Register: 0x2000

Although PCI Development platform's DMA Engine can generate IO READ and IO WRITE PCI commands, it is primarily designed to work with 32-bit aligned addresses. In particular, it is not possible to load the DMA Engine with an address in which the low two bits are both one. In memory space commands the low two bits of the address register are used to encode the burst order and data width, as follows:

0x0 Linear increment.

0x1 Intel cacheline wrap mode (now deprecated in PCI spec).

0x2 Cacheline wrap (linear increment within cacheline).

0x3 Linear increment (request 64 bit). Address emitted on bus has low two bits cleared.

The DMA Engine only makes requests for 64 bit transactions (asserts the PCI signal REQ64) when the low two bits are **0x3**, or when the DMA Engine is programmed from the user-area and the interface is in 64-bit Transaction mode. All other transactions initiated by the DMA Engine request the use of 32 bit data cycles.

The DMA address register is incremented on each data cycle. Increment order is always linear even if the specified burst order is not linear. The ability to specify non-linear burst orders (encoding 0x1 and 0x2) only exists for the purposes of testing other devices. The DMA address register counter only applies to bits 2..12, thus if a DMA causes the address to increment to the boundary of an aligned 8kB region, it wraps back to the beginning of that region.

7.3 DMA Command Register: 0x2000+8

DMA commands are encoded in an arcane format whose primary motivation is to simplify the PIF circuit. The fields in the DMA command register are as follows.

Bit Range	Contents
31..28	PCI command
27..16	0x1000 – length
15..12	delay to next request
11..2	0x400 - burst length
1	Unused
0	Unused

Note that in the preceding table “0x1000 - length” means the numerical value of 0x1000 minus length. Likewise for “0x400 - burst length”. For example, if the user wants a length of 8, bits 27..16 should be loaded with “0x1000 – 8" which is equal to 0xf f 8.

Length and burst lengths always count in 32-bit words. When requesting 32-bit transactions, a burst length of 1, 2 or 3 are treated as a minimum burst length due to internal pipelining restrictions. When requesting 64-bit transactions, a burst length of 2, 4 or 6 is similarly treated as a minimum burst length. Any odd burst length, odd start address or odd total length forces a transaction to 32-bit. For 32-bit DMA, the minimum burst length is one 32-bit word, for 64-bit DMA, minimum burst length is one 64-bit word.

The DMA length must be a multiple of the burst length. Failure to observe this restriction may cause the DMA Engine to overrun beyond length up to the next multiple of burst length.

The interface will start requesting DMA when bit 27 in the DMA command register is set. Bits 27..16 are a counter which will count up by one for each DWORD(4 bytes) transferred —when bit 27 returns to zero the DMA Engine stops requesting transactions. Bit 27 is automatically zeroed whenever the PROG bits controlling the configuration of `usrlca0` and `usrlca1` (bits 0 and 1 in register 0x8 in *struct PamRegs*) are zero. This feature is designed to abort any active DMA if the user-area if deconfigured, as for example happens on last close of the PCI Development platform device under UNIX.

Software wishing to abruptly stop the DMA Engine should not simply clear the DMA command register. At the moment the host access updating the DMA command register arrives, a DMA transaction may already have been initiated that will issue in the cycles immediately following the update. If the DMA command register were indiscriminately cleared, the DMA transaction could proceed with invalid values for the PCI command and other fields in the DMA command register. Instead, software should use a read modify write sequence to clear only bit 27 and the rest of the length field.

The burst length field (11..2) is the desired burst length in data cycles. If the burst is disconnected the next request will try to complete the DMA up to the burst boundary then stop and start a new request for the next burst. The maximum burst size is attained by setting bits (11..2) to all zeroes. Hence maximum burst size is 0x400 DWORDs, or 4096 bytes.

The *delay to next request* (bits 15..12) specifies a delay between bursts. This lets us throttle back the rate, it also lets us optimize throughput on host bridges that introduce their own wait states.

When the remaining length is less than twice the *burst length* and *delay to next request* is set to 0 (respectively 1), the interface increases delay to next request by 2 to 2 (respectively 3). If this were not done the DMA Engine could issue an extra burst (beyond the rules stated above) because the decrement of length is delayed by a couple of cycles from the cycle when the actual data is transferred.

Typically the PCI commands used with the DMA Engine are the memory commands:

Command Code	Meaning
0x6	Memory read
0x7	Memory write
0xC	Memory Read Multiple
0xE	Memory read line
0xF	Memory write and invalidate

Refer to the PCI specification for more details on the precise meanings of these commands.

7.4 Simultaneous Initiator and Target

The DMA engine is well partitioned from the PCI target state machine in the PIF. One consequence of this is that it is perfectly possible for PCI Development Platform to be the target of cycles that it initiates. Allowing PCI Development Platform to be the target of its own cycles makes it possible to access any PIF control register from the user-area.

8
Interface Modes

8.1 Introduction

Four distinct interface modes are supported in the current firmware. These are selected through the `PamRT.h` function `PamSetMode` or `PamSetModeAndDelay` which sets the appropriate value in the decode register at address 0x30 in PCI Development Platform memory space. The three modes are:

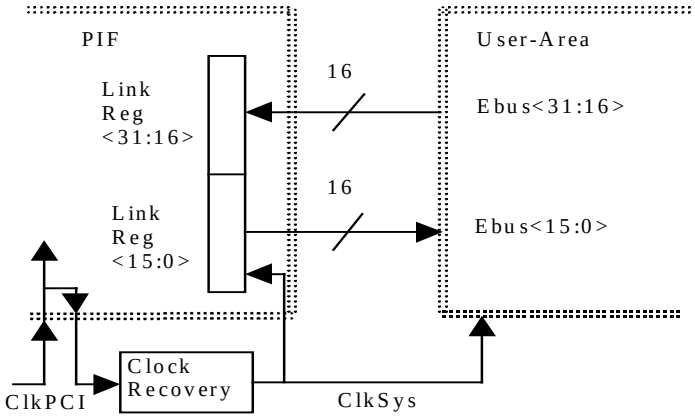
- **Static mode** is a simple low-performance interface that provides statically configured 16-bit paths to and from the user-area.
- **Promiscuous mode** transmits a selection of data and control values present on the PCI bus to the user-area. The flow of data is one-way. This is similar to promiscuous mode on Ethernet adapters.
- **Transaction mode** is a high performance transaction oriented mode that supports both target and master operation. This is the preferred mode for all but the simplest designs.
- **PromiscuousTransaction mode** combines the protocol state machine of the Transaction mode with the trace collection capabilities of Promiscuous mode. It can aid in the debug of Transaction mode applications.

The four modes are described in more detail in the following sections.

8.2 Static Mode

From the host side, Static Mode consists of a single 32-bit *link* register (address 0x38 in *struct PamRegs*) that can be read or written. The high 16 bits of write data are ignored. From the user-area side it consists of a 16-bit input port driven by the low 16 bits of the link register and a 16-bit output port which loads the high 16 bits of the link register on each Clksys cycle. The user-area input port is `EBus<15:0>` and the output port is `EBus<31:16>`. The following figure shows the Static Mode link register flow.

Figure 8-1 Static Mode Link Register



8.2.1 Coding guidelines for host software

Although Static mode is conceptually the most simple interface to PCI Development platform, an appreciation of the host CPU to PCI interface, particularly on Alpha systems, is required to ensure correct operation when writing software that communicates with PCI Development platform in Static mode.

To improve performance modern microprocessors may buffer or aggregate write data destined for memory or the I/O subsystem. As a consequence the program should consider time on the PCI bus as completely elastic with respect to CPU instruction issue. Inserting a string of NOPs to implement a short delay between successive accesses to the Link register may indeed introduce a delay with respect to CPU instruction issue. However writes to the Link register that are issued on either side of the string of NOPs may be subject to differing amounts of delay in write buffers allowing the two access to appear back to back on the PCI bus. Worse still some processors may squash, merge or reorder writes. Usually such processors provide explicit serialization commands, which may be used by software in cases where order matters. The `PamFlush()` primitive in the PCI Development platform runtime library issues the appropriate serialization for the current host.

Serialization does not imply that a pending write is issued immediately from the processor or flushed through subsequent levels of buffering in the path to the PCI Development platform. To force writes to the PCI Development platform as quickly as possible, the programmer may follow the write by an explicit read of the Link register or some other low addressed PCI Development platform register in addition to a call to `PamFlush()`. For this purpose the Info register at address offset 0x0 is a good candidate.

The software programmer should also be aware that updates to the link register are subject to several cycles of pipeline delay in the PIF before they appear on `EBus<0:15>`. In addition the circuit loaded into the user-area may itself incur several cycles of pipeline delay between a new value appearing on `EBus<0:15>` and the resulting change propagating to `EBus<16:31>`. Therefore a write to the Link register followed immediately by a read of the Link register may yield a value in `EBus<16:31>` which has been computed by the user-area circuit in response to the previous value of the low sixteen bits of the Link register. This value may appear inconsistent with its current value. Several reads may be required before the value read from the high bits of Link register reflects a value computed by the user-area circuit that is based on the last value written to its low bits.

The following code fragment demonstrates the application of the above guide-lines to write two successive values to the low 16 bits of Link register and then read the resulting value that the user circuit has driven on to EBus<16:31> in response to these writes:

```
volatile int force_read;
/* update link register with new_link_val_0 */
PAMREGS(pam)->link = new_link_val_0;
PamFlush();
force_read = PAMREGS(pam)->info;
PamFlush();
/* update link register with new_link_val_1 */
PAMREGS(pam)->link = new_link_val_1;
PamFlush();
/* dummy read to allow time for propagation
through internal pipeline */
force_read = PAMREGS(pam)->link;
PamFlush();
/* get result of computation from EBus<16:31> */
return_link_val_hi = (PAMREGS(pam)->link >> 16) & 0xfff;
```

8.2.2 Static Mode and Clkusr

All signals on the PIF side of the EBus are clocked on Clksys. The user-area interface may be clocked on any clock of the user's choosing (in particular Clkusr), however the PIF side of the interface will continue to be clocked on Clksys. Potential metastability problems on the reception of EBus<31:16> are handled in the PIF, but if an asynchronous interface is used the user may see some bits in the input and output ports change one cycle ahead of others. One way to deal with this is to use sideband signals provided by the download path (see Section 5.2.1) as a strobe. The following code fragment illustrates this. It assumes that the user-area circuit is configured to only read from or update the EBus when a logic high (1) is present on Cntrl_.U_din<n> (pin 151). Circuits internal to the PIF cause the value written to bit *n* of the PamRegs field dwnld1 is driven onto Cntrl_.U_din<n>.

```
/* update link register */
PAMREGS(pam)->link = new_link_val;
PamFlush();
/* strobe all usrlca DIN (pin 151) */
PAMREGS(pam)->dwnld1 = 0xf;
PamFlush();
/* end strobe */
PAMREGS(pam)->dwnld1 = 0x0; PamFlush();
/* get new link register value */
new_link_result = PAMREGS(pam)->link;
```

In user-area applications that use Clksys set to double speed, some care is required to determine which cycles correspond to the first half of the PCI clock and which to the second half. See the PCI performance tests for an example (ppam_pciperf & pciperf).

8.3 Promiscuous Mode

In Promiscuous Mode, EBus<31:0> is always driven by the PIF. On each cycle it contains a copy of the low 32 bits of the PCI address/data bus delayed by a few cycles. During Promiscuous Mode the clock recovery circuit is configured such that Clksys runs at twice the PCI clock speed. The data/address from the PCI bus is driven on to EBus<31:0> during the first half of the PCI clock period. On the second half of each PCI clock period, the PCI control signals for that cycle are driven on to EBus<31:0>. This mode can be used for a variety of bus monitoring applications. Table 8-1 lists the bit assignments.

Table 8-1 Promiscuous Mode Bit Assignments

Bit range	Contents
31	FRAME
30	TRDY
29	IRDY
28	STOP
27	DEVSEL
26	GNT
25	REQ
24	ACK64
23	REQ64
22	PAR
21..18	C/BE < 3:0 >
17..12	< undefined >
11..0	AD < 43:32 >

The ability to enter Promiscuous Mode is disabled unless the appropriate security bit is set (see Chapter 9).

8.4 PromiscuousTransaction Mode

In PromiscuousTransaction mode EBus<31:0> is always driven by the PIF and the values transmitted are identical to regular Promiscuous mode. The difference is in regular Promiscuous mode the PIF responds to transaction as quickly as possible and attempts to burst for as many data cycles as possible to allow the other party in the PCI transaction to operate to its limits of performance. In PromiscuousTransaction mode, the PIF responds to transaction in exactly the same way as it would if it was in Transaction mode (see 8.5.). Whereas regular Promiscuous allows us to analyze the host system and other devices, the purpose of PromiscuousTransaction mode is to debug applications of PCI Development platform which use Transaction mode.

The pciperf (ppam_pciperf on UNIX) sample application in the PCI Development platform software kit includes a command line switch to operate in PromiscuousTransaction mode.

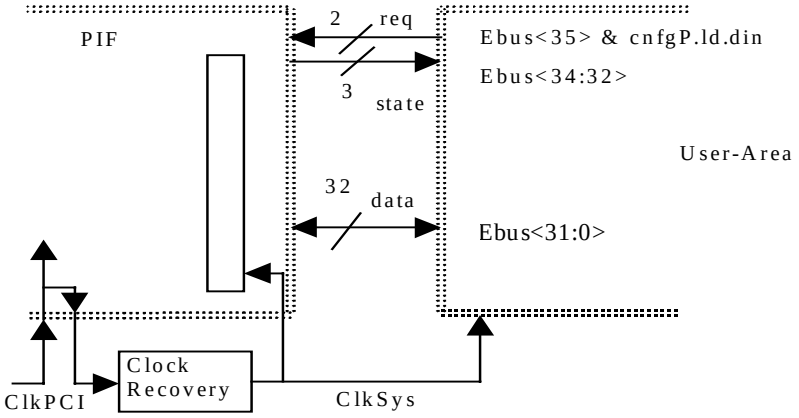
8.5 Transaction Mode

In Transaction Mode, the EBus carries multi-cycle transactions that reflect activity on the PCI bus. Since Clksys is used as the clock, all activity is synchronous with the PCI.

Transaction Mode has a 32-bit bi-directional address/data bus, a 3-bit state code from the PIF to the user-area, and a 2-bit request code from the user-area to the PIF. On each cycle the state code on EBus<34:32> tells the disposition of the data bus (EBus<31:0>) on the next cycle.

When the state code indicates that the data bus is idle, the user-area may read or write certain PIF internal registers using the request codes on EBus<35> and CnfgP_ld.din. The request codes are also used for coarse-grained flow control of DMA. The following figure shows the bit flow in Transaction Mode.

Figure 8-2 Bit Flow in Transaction Mode



At all times the PIF is master of the protocol and the user-area must satisfy its requests. Flow control must be handled at higher application specific levels, for instance through user-area initiated interrupts and polling of ready bits implemented in user-area status registers. The user-area cannot introduce delays or wait states within an individual transaction.

The user-area design must respect the direction of EBus<31:0> dictated by the state code on EBus<34:32> on each cycle. Failure to do so could result in conflicting values being driven onto EBus<31:0>, with possible damage to the module.

The state codes are listed in Table 8-2.

Table 8-2 State Codes

Mnemonic	Code	Meaning
AIF	7	Address cycle of a write to a register in the PCI Interface (address range 0x0 ... 0x1f f f f).
DW	6	Data Wait: transaction is active but no progress on this cycle.
AIU	5	Address cycle of transaction transferring data to user-area
AUI	4	Address cycle of transaction transferring data from user-area
DV	3	Data Valid on EBus
DS	2	Data Skip: like DV, but byte enables were off— should increment address and ignore data
MSTR	1	Used to prefix transactions that were initiated by PCI Development Platform
IDLE	0	No active transaction

The distinction between AIF and AIU or AUI is that the PIF claims the first 128 KB (0x20000 bytes) of address space so CPU reads to this region are invisible from the user-area and writes appear as a transaction headed by an AIF cycle. Any CPU read or writes beyond the first 128 KB produces an AIU or AUI cycle.

Note that the Transaction mode state codes offer no mechanism to accurately interpret a partial word write. The DS state code is used for any write data cycle that does not have all byte enables active. Software communicating with PCI Development platform in Transaction mode must ensure that writes are always whole words. Firmware v2.0 and above offers a mechanism to raise an interrupt if a partial word write occurs to addresses at or above 0x20000 in PCI Development address space (see 8.5). Even if the host never generates partial word writes, the DS state code may still appear and should be interpreted to signify a write data cycle with *no* byte enable active. Writes with no byte enable active interspersed in a stream of DV state codes are a natural consequence of host bridge transaction aggregation. Applications wishing to avoid or eliminate aggregation (for instance to simplify the user-area circuit state machine) may accomplish this by inserting reads between writes that could otherwise be subject to aggregation.

8.5.1 Target Transactions

Transactions for which PCI Development Platform is the target will most commonly be generated by CPU accesses to the PCI Development Platform address space, although they can be generated by other PCI Development Platforms or any other master capable PCI device in the system. A basic transaction as seen from the user-area consists of an address cycle (AIU, AUI, or AIF), followed by one or more data cycles (DW, DV, or DS). A transaction is usually terminated by the IDLE state code, but may be terminated by the start of a new transaction (MSTR, AIU, AUI, or AIF). The PIF will never retry transactions[4] , so there will always be at least one DV or DS cycle, however user-area applications should not rely on this behavior as it may change in future firmware revisions.

AIU

Transactions started by an AIU cycle present address and data to the user-area. In the cycle following AIU, the PIF drives the address onto `EBus<31:0>` and continues to drive `EBus<31:0>` with data until the end of the transaction. The number of DW, DV, and DS cycles depends on the master of the transaction, usually the hostbridge. In the simplest case a transaction consists of an AIU cycle, one or more DW cycles, a DV cycle, and then either an IDLE cycle or a new transaction. Note that there will always be at least one DW cycle because the PCI Development Platform is a medium decode device.

Host bridges may aggregate neighboring writes, and Alpha based systems can reorder writes. The `PamRT` function `PamFlush` will ensure write ordering but will not guarantee that aggregation is disabled. One way to suppress aggregation is to perform a read to some harmless PCI location, for instance address 0x0 of the PCI Development Platform memory space. DS states may occur when aggregation of a noncontiguous sequence of writes occurs. For example, from the code sequence:

```
volatile int *user_area;
user_area = ((volatile int *) pam) + (1<<17)/sizeof(int);
user_area[0] = 0xaaaaaaaa;
user_area[2] = 0xbbbbbbbb;
user_area[3] = 0xcccccccc;
```

[4] *Retry* is PCI terminology for a transaction that transfers zero data because the target signals to the master that it is not currently ready, and that transaction should be attempted again at a later time.

The sequence of states could be:

Cycle	State	EBus<31:0>	PIF Drives
0	AIU	< undefined >	NO
1	DW	0xXX020000	YES
2	DW	< undefined >	YES
3	DV	< undefined >	YES
4	DS	0xaaaaaaaa	YES
5	DV	< undefined >	YES
6	DW	0xbbbbbbbb	YES
7	DW	< undefined >	YES
8	DV	< undefined >	YES
9	IDLE	0xcccccccc	YES
10	IDLE	< undefined >	NO

Observe that the high six address bits are reserved since these correspond to the location when the PCI Development Platform was mapped when the PCI was configured at boot time and hence are host specific. Note also that bit 17 of the address is set because the user-area address space is offset 128 KB from the base of memory space.

AIF

Transactions started by an AIF cycle are normally ignored by the user-area application apart from the utilization described in Section 9.5.3. In the cycle following AIF, the PIF drives the address onto EBus<31:0> and continues to drive EBus<31:0> with data until the end of the transaction. AIF transactions are in other respects like AIU transactions.

AUI

Transactions started by an AUI cycle present an address to the user-area and receive data from it. In the cycle following AUI, the PIF drives the address onto EBus<31:0> and on the cycle after, stops driving allowing the user-area to drive EBus<31:0> with result data. Whereas in the AIU case, address and data flow in the same direction and can be pipelined to arbitrary depth, transactions started by an AUI cycle will in general need to process the received address and return data.

Prior to v2.0 the delay between the receipt of address and the time the first data must be on EBus<31:0> was fixed at 6 cycles. In v2.0 and above the delay may be set to any value from 0 and 7 cycles, although practical values are restricted to 2 to 7 cycles. The choice of delay is a compromise between the shortest possible delay, which yields the best throughput, and longer delays, which allow more clock cycles for the user-area to compute the result. The delay is set from host software by writing the target delay field (bits 12..10) in the decode register at address 0x30. Writing a value of *n* in the target delay field means that the PIF expect the user-area circuit to drive result data onto EBus<31:0> *n* cycles after the address was on the EBus<31:0>, and *n+1* cycles after the AUI state code.

Assuming that a user-area access at address 0x2004C from the base of the PCI Development Platform memory space returns the value 0xaaaaaaaa, and the target delay is set to 6, a code sequence such as this:

```
volatile int *user_area;

user_area = ((volatile int *) pam) + (1<<17)/sizeof(int);

printf("User area [0x13] = %08x\n", user_area[0x13]);
```

Could produce the following sequence of states:

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	AUI	< undefined >	NO	NO
1	DW	0xXX02004c	YES	NO
2	DW	< undefined >	NO	NO
3	DW	< undefined >	NO	YES
4	DW	< undefined >	NO	YES
5	DW	< undefined >	NO	YES
6	DW	< undefined >	NO	YES
7	DW	0xaaaaaaaa	NO	YES
8	DW	< undefined >	NO	YES
9	DW	< undefined >	NO	YES
10	DW	< undefined >	NO	YES
11	DV	< undefined >	NO	YES
12	IDLE	< undefined >	NO	YES
13	IDLE	< undefined >	NO	NO

Due to internal pipelining in the PIF the DV cycles of an AUI-type transaction arrive a few cycles (5 in v2.0) after the result data should be driven onto EBus<31:0>. Thus the presence of DV cycles can not be used to decide when to transmit data. If the user-area circuit is required to respond correctly to read transactions that have been aggregated into a burst (several DV cycles for one AUI), then the successive result values should be emitted on each cycle following the first data cycle 5. Alternatively where the host initiates reads, driving software may be organized to avoid read aggregation. Lastly it should be noted that many hosts do not, in any case, aggregate reads. Although it occurs quite commonly on Alpha systems, x86 systems do not in general aggregate reads. When PCI Development platform is sourcing data, firmware v2.0 in Transaction mode cannot sustain a transfer after the other party of the PCI transaction introduces a wait state. On detecting a wait state the PIF will latch the current output word, wait for the completion of the current data cycle and then terminate the transaction without transferring any more data. To receive subsequent words, the initiator must rearbitrate for the bus and issue a new read starting after the last data word transferred. In terms of what passes across the EBus, the user-area circuit always transfers a contiguous stream of 32-bit data words.

Assuming:

- Software emits a sequence of 32-bit reads corresponding to user-area accesses at addresses 0x20008, 0x20010, 0x20014 and 0x20018 from the base of the PCI Development platform memory space (note that the address are not contiguous).
- The target delay is set to 4.
- The values at 0x20008, 0x2000c, 0x20010, etc. are equal to the low 16 address bits, namely 0x8, 0xc, 0x10, etc.
- We are on a host that does read aggregation, but it introduces a two-cycle wait state at the access to 0x20014.

We can expect to see the following sequence of states and values on EBus:

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	AUI	< undefined >	NO	NO
1	DW	0xXX020008	YES	NO
2	DW	< undefined >	NO	NO
3	DW	< undefined >	NO	YES
4	DW	< undefined >	NO	YES
5	DW	< undefined >	NO	YES
6	DW	< undefined >	NO	YES
7	DW	0x00000008	NO	YES
8	DW	0x0000000c	NO	YES
9	DW	0x00000010	NO	YES
10	DW	0x00000014	NO	YES
11	DV	0x00000018	NO	YES
12	DS	0x0000001c	NO	YES
13	DV	0x00000020	NO	YES
14	DW	0x00000024	NO	YES
15	DW	0x00000028	NO	YES
16	DV	0x0000002c	NO	YES
17	IDLE	0x00000030	NO	YES
18	IDLE	< undefined >	NO	NO

Note that, although the initiator was attempting to read 0x20008, 0x20010, 0x20014 and 0x20018, only the values of 0x20008, 0x20010, 0x20014 were returned due to the wait state at 0x20014. The user-area returned the value 0xc at 0x2000c even though the initiator did not request it. The user-area continued to return following values until the state code signaled IDLE. In many applications it may be inconvenient to support arbitrarily long read bursts. Application developers should always keep in mind that a PCI Development platform application is a compact between the user-area circuit and the driving software.

They are always free to adopt coding styles in the software parts of their application that will limit burst sizes, furthermore a given host will normally impose natural limits on burst size. Read bursts offer certain performance advantages, but there is little point making the user-area circuit unduly complex unless the performance gained is of clear value in the overall application.

In the simplest case the user-area circuit may choose to support only single word bursts and require software to never issue load sequences that could result in aggregation. Under these conditions all the user-area circuit need do in response to a AUI transaction is start to drive the return value onto EBus<31:0> before the minimum time dictated by the target delay setting and hold it there until the end of the transaction.

8.5.2 Master Transactions

Master transactions are transactions that were initiated by PCI Development platform. Such transactions are sometimes called DMA (Direct Memory Access) since the target of the transactions is often host memory. Master transactions are very similar to the target transaction described above except that they are prefixed by a MSTR cycle. An AIU or AUI cycle and a sequence of data cycles will always follow the MSTR cycle. Since target devices may retry, the transaction may have zero DV cycles. In v2.0 all byte enables are always asserted during PCI transactions initiated by PCI Development platform so DS should never be seen in a master transaction.

The address associated with the AIU or AUI cycle of a master transaction is the target address of the operation, that is the remote device address.

8.5.2.1 MSTR/AUI

The delay to first data in transactions where PCI Development platform sources data in master transactions (MSTR/AUI) is controlled by the master delay field (bits 15..13 in the decode register at address 0x30). Performance is usually more important for master transactions. Since the PCI Development platform is initiating the transaction, the target address can be tracked internally to the user-area. Therefore, fewer cycles are needed in the user-area between when the address is known and when the first data can be emitted[5]. Prior to firmware v2.0 this delay was fixed at 3 cycles. In firmware v2.0 and above the delay can be set to any value in the range 0 to 7, although practical values start at 2.

The following sequence of states can be seen in a four DWORD MSTR/AUI (DMA write) sequence to address `0x00543210`, which writes the sequence of values `0xaaaaaaaa, 0xbbbbbbbb, 0xcccccc, 0xdddddd`, with the master delay field set to 3.

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	MSTR	Undefined	NO	NO
1	AUI	Undefined	NO	NO
2	DW	`0x00543210`	YES	NO
3	DW	Undefined	NO	NO
4	DW	`0xaaaaaaaa`	NO	YES
5	DW	`0xbbbbbbbb`	NO	YES
6	DW	`0x00cccccc`	NO	YES
7	DW	`0x00dddddd`	NO	YES
8	DV	Undefined	NO	YES
9	DV	Undefined	NO	YES
10	DV	Undefined	NO	YES
11	DV	Undefined	NO	YES
12	IDLE	Undefined	NO	YES
13	IDLE	Undefined	NO	NO

As previously discussed, in Transaction mode, v2.0 cannot sustain bursts after wait states when it sources data onto the PCI bus. As master a transaction cannot be terminated as abruptly as is possible for a target. In the face of a target wait state, v2.0 will emit up to two more data cycles before terminating the transaction.

8.5.2.2 MSTR/AIU

MSTR/AIU sequences are similar to AIU target transactions except that MSTR/AIU sequences usually suffer much longer delays between address and first data, particularly when the source of remote data is host memory.

[5] First data could be emitted immediately on seeing MSTR, without reference to the address, but this would render it impossible for the user-area to know the address at all since the Ebus would be busy transmitting data on the cycle when the address is available.

The following sequence of states might be seen in an eight DWORD MSTR/AIU (DMA read) sequence to address 0x00400500, with memory at 0x00400500 containing the values: 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, and 0x17. This sequence assumes a two-cycle target wait at the fourth data cycle.

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	MSTR	Undefined	NO	NO
1	AIU	Undefined	NO	NO
2	DW	0x00400500	YES	NO
3	DW	Undefined	YES	NO
4	DW	Undefined	YES	NO
5	DW	Undefined	YES	NO
6	DW	Undefined	YES	NO
7	DW	Undefined	YES	NO
8	DW	Undefined	YES	NO
9	DW	Undefined	YES	NO
10	DW	Undefined	YES	NO
11	DW	Undefined	YES	NO
12	DW	Undefined	YES	NO
13	DW	Undefined	YES	NO
14	DW	Undefined	YES	NO
15	DW	Undefined	YES	NO
16	DW	Undefined	YES	NO
17	DW	Undefined	YES	NO
18	DV	Undefined	YES	NO
19	DV	0x00000010	YES	NO
20	DV	0x00000011	YES	NO
21	DV	0x00000012	YES	NO
22	DW	0x00000013	YES	NO
23	DW	Undefined	YES	NO
24	DV	Undefined	YES	NO
25	DV	0x00000014	YES	NO
26	DV	0x00000015	YES	NO
27	DV	0x00000016	YES	NO
28	IDLE	0x00000017	NO	NO
29	IDLE	Undefined	NO	YES

8.5.3 Requests from User-area

The user request bits, EBus<35> and CnfgP_ld.din, are used to signal requests from the user-area to the PIF. They allow the user-area to read and write control registers in the PCI interface. In this way the user-area can determine the current state of the PIF, raise interrupts, initiate DMA, and other similar functions.

8.5.3.1 Bus Contention and AIF Cycles

The PCI interface will ignore control register accesses that are attempted when the EBus is not IDLE. The primary function of AIF is to allow the user-area to avoid contention for PCI interface internal resources when it is trying to access control registers. When these registers are written they use the same internal PIF paths that are used by write access from the user-area. The user-area needs to know when the registers are potentially being written from the PCI side, and hence when its own write requests are being rejected by the PIF. Of course some applications may additionally be able to use AIF to monitor driver access to the PCI interface.

Note that when the PIF is in static mode EBus<34:32> are disabled and pulled up. This corresponds to the state code AIF. Thus the state tracking machinery of an application expecting the PIF to be in Transaction Mode will see a sequence of AIF state codes. This choice is deliberate. In IDLE mode the user-area may legitimately drive EBus<31:0> which would conflict with EBus<15:0> in Static Mode.

8.5.3.2 Request Codes

Both request bits are *active low*, that is, they are asserted by a low logic value and deasserted by a high logic value. Therefore, a deconfigured user-area will pull-up the request bits thereby deasserting them. The request bits are mutually exclusive. If both are driven low, they are ignored, except that if either or both request bits are driven low the DMA engine is disabled from requesting a new transaction. Thus the request bits also provide coarse DMA flow control. Note that once PCI Development Platform has started to request mastership of the PCI bus it will complete it even if the request bits become active before the bus arbiter grants the request. Furthermore, activation of request bits does not cause any form of early termination of a currently active PCI transaction.

Read

CnfgP_ld.din requests that the contents of the link register be driven onto EBus<31:0>. There are two cycles of pipeline delay between the user-area asserting CnfgP_ld.din and the value being driven onto EBus<31:0>.The value is only valid when the state on EBus<34:32> is IDLE.

Write

EBus<35> requests that the value currently[6] being driven by the user-area onto EBus<31:0> be written into a PIF internal register. EBus<1:0> provides an address, and EBus<31:2> provides data. The possible target registers are:

Addr	PIF Register
0x0	DMA address register
0x1	DMA command register
0x2	Reserved for future use
0x3	Link register

The ability to write the DMA register from the user-area is disabled unless the appropriate security bit is set (see Chapter 9). Since EBus<1:0> provides an address, these bits cannot be written in the target registers.

[6] Same cycle as Ebus<35> is asserted.

Due to potential conflicts for shared resources, a write request signaled by EBus<35> is ignored if: on the same cycle in which EBus<35> is asserted, the state on EBus<34:32> is not IDLE in the current cycle or preceding cycle or the state is AIF in the following cycle. It is the user-area's responsibility to monitor the EBus state and track when requests have been ignored.

8.5.3.3 Link Register in Transaction Mode

In transaction mode, the link register takes on special meaning.

Bit Assignments for Link Register Reads

Among the low 16 bits, the even bits can be read and written from the PCI side at their usual offset, however, the odd bits represent internal state of the PIF. The currently defined bit assignments are:

Bit	Meaning
EBus<0>	Address bit must be set to 1
EBus<1>	Address bit must be set to 1
EBus<15>	DMA engine is active

Bit Assignments for Link Register Writes

The high 16 bits of the link register can always be read from the PCI side at their usual offset, however, they are updated only when EBus<35> is asserted (low logic value). The high 16 bits of the link register are reset when the PIF enters transaction mode. Some of the bits have special meaning in transaction mode. The currently defined bit assignments are:

Bit	Meaning
EBus<0>	Address bit must be set to 1
EBus<1>	Address bit must be set to 1
EBus<16>	Raise interrupt

8.5.4 64-bit Transaction mode

If the clock recovery circuit is configured to run at twice the PCI clock speed and PCI Development platform is placed in Transaction mode, it enters 64-bit Transaction mode.

64-bit Transaction mode is designed to resemble as much as possible the 32-bit Transaction mode described above. The main differences are:

Transaction prefixes (MSTR, AIU, AUI, and AIF) last two cycles.

In the cycle following the first address cycle (AIU, AUI, AIF), the address is not present on EBus<31:0>. The user-area circuit must wait until the cycle following the second address cycle.

The interpretation of the target and master delay fields is altered.

The conditions in which the Transaction mode write request is ignored are altered.

The interpretation of the target and master delay fields in 64-bit Transaction mode is that if the corresponding delay field is set to n, first data for that type of the transaction (be it target or master) should be on EBus<31:0> *2n+2* cycles after the address was driven onto EBus<31:0> or *2n+3* cycles after the second AUI cycle.

Requests to write PIF internal registers in Transaction mode through the use of EBus<35> are ignored if, on the same cycle in which EBus<35> is asserted: the state on EBus<34:32> is not IDLE in any of the preceding four cycles or the state is AIF on the current cycle. Additionally write requests in the second half period of the host PCI clock are ignored.

These differences are sufficiently minor that it is not difficult to design user-area circuits that function in either 32-bit or 64-bit Transaction mode without any need for the circuit to know which mode it is being used in. All that user software needs do is to load appropriate values of the target and master delay fields in accordance with the PLL operating mode.

When programming the DMA Engine from software, if the low two bits of the DMA address register are set to 0x3 PCI Development platform will generally try to generate 64-bit PCI transactions, even if the PLL is configured at PCI clock speed. But for PCI Development platform to accept 64-bit PCI transactions and for the DMA Engine to generate them when programmed from user-area, three further requirements must be met. First, rather obviously, it must be in a 64-bit slot. Second, the PLL must have achieved lock (lock status can be read in *struct PamRegs*). Third, the 64-bit transaction mode security enable must be set (see Chapter 9).

When PCI Development platform is sourcing data in 64-bit Transaction mode, if the current PCI transaction is a 32-bit transaction it is treated in the same way as if the other party had introduced wait states. Target transactions are stopped after one data cycle, and master transactions are stopped after two data cycles.

Assuming a 4 DWORD write burst to the address `0x20000` in PCI Development platform address space with the values `0x0, 0x1, 0x2` and `0x3` and a 64-bit transaction, the sequence of states and values could be:

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	AIU	Undefined	NO	NO
1	AIU	Undefined	NO	NO
2	DW	`0xXX020000`	YES	NO
3	DW	Undefined	YES	NO
4	DV	Undefined	YES	NO
5	DV	0x0	YES	NO
6	DV	0x1	YES	NO
7	DV	0x2	YES	NO
8	IDLE	0x3	YES	NO
9	IDLE	Undefined	NO	NO

If the same sequence were transferred using a 32-bit transaction, the sequence of states and values might be:

Cycle	State	EBus<31:0>	PIF Drives	User-area Drives
0	AIU	Undefined	NO	NO
1	AIU	Undefined	NO	NO
2	DW	`0xXX020000`	YES	NO
3	DW	Undefined	YES	NO
4	DV	Undefined	YES	NO
5	DW	0x0	YES	NO
6	DV	Undefined	YES	NO
7	DW	0x1	YES	NO
8	DV	Undefined	YES	NO

9	DW	0x2	YES	NO
10	DV	Undefined	YES	NO
11	DW	0x3	YES	NO
12	IDLE	Undefined	YES	NO
13	IDLE	Undefined	NO	NO

8.5.5 Interrupts

The PCI Development Platform has six interrupt sources: an end of DMA interrupt, a user programmable interrupt through the link register, two user programmable interrupt activated by the ring bits, an interrupt that detects partial word writes and a loss of PLL lock interrupt. In transaction mode, control of these interrupts is performed through bit manipulations in the VCO and link registers within the PIF. Enabling of interrupts are done in the VCO register. The state of the interrupt sources is in the Clock register. The VCO and Clock registers are defined in `PamRegs.h`. The link register is used to raise the user programmable interrupts as initiated by the user-area FPGAs. The ring pins can also be used to raise an interrupt. The link register can be read from the user area or the PCI bus, but `Link<16>` may only be written from the user area. The notable bits in the VCO, Clock, and link registers are shown below.

VCO Register

Bit	Meaning
VCO<4>	Enable global interrupt
VCO<5>	Global interrupt active
VCO<8>	Enable Ring[0] interrupt
VCO<9>	Enable Ring[1] interrupt
VCO<10>	Enable DMA done interrupt
VCO<11>	Enable link<16> interrupt
VCO<14>	Enable PLL not locked interrupt
VCO<15>	Enable Partial word write interrupt

Clock Register

Bit	Meaning
VCO<8>	Ring[0] interrupt Active
VCO<9>	Ring[1] interrupt Active
VCO<10>	DMA done interrupt Active
VCO<11>	Link<16> interrupt Active
VCO<14>	PLL not locked interrupt Active
VCO<15>	Partial word write interrupt Active

Link Register

Bit	Meaning
Link<0>	Address bit must be set to 1
Link<1>	Address bit must be set to 1
Link<15>	DMA engine is active
Link<16>	Raise interrupt

8.5.5.1 End of DMA Interrupt

To use the end of DMA interrupt, both the global enable and the end of DMA enable bits should be set, as shown below:

```
/*
enable end of DMA interrupt bits
interrupt enables: global (bit 4) and end of DMA (bit 10)
*/
PAMREGS(pam)->vco = (1<<4) | (1<<10);
```

The end of DMA interrupt is automatically generated when the DMA engine's length counter rolls over. The driver disables the global enable in the VCO register. The application programmer's interrupt service routine should first clear the interrupt by writing a "1" to `Clock<10>`, then set both the global (`VCO<4>`) and DMA done (`VCO<10>`) enables. There is no chance of getting caught in an infinite loop of interrupts, since the hardware has already cleared the interrupt. To verify the interrupt source was the DMA engine, the clock register can be checked.

8.5.5.2 User Interrupt in the Link Register

The user interrupt can be set by accessing the link register from the user-area FPGAs (see Section 8.5.3). To enable the user programmable interrupt, the global enable and user interrupt enable both have to be set. This is shown below:

```
/*
enable user interrupt bits
interrupt enables: global (bit 4) and user enable (bit 11)
*/
PAMREGS(pam)->vco = (1<<4) | (1<<11);
```

To generate the actual interrupt, bit 16 of the link register has to be set. The driver will then disable the global interrupt enable before passing control to the user's interrupt service routine. Writing a one to bit 11 of the clock register clears the interrupt. This should be done before enable bits are reset. The following sequence sets the interrupt. To verify the interrupt source was the user programmable interrupt, the clock register can be read.

Cycle	Ebusstate	Value on EBUS<31:0>	EBUS<35>
0	IDLE	XXXXXXXX	1
1	IDLE	0x00010003	0
2	IDLE	XXXXXXXX	1

Note

`EBus <35>` requests a write to a register in the PIF and is active low. Refer to Section 8.5.3. The programmer should not write "1"s to unspecified VCO and Clock register bits (i.e. read-modify-write can place other components in an unknown state).

The following software sequence clears the interrupt and resets the appropriate enables.

```
/* Clear interrupt */
PAMREGS(pam)->clock = (1<<11);
/* Reset enables */
PAMREGS(pam)->vco = (1<<4) | (1<<11);
```

8.5.5.3 User Interrupt Using the Ring Bits

A user programmable interrupt set directly by the ring bits. The ring bits are a two bit wide bus that connects to all four user area FPGAs. These two bits can be used to control the interrupt from any FPGA.

In the PIF, bits 8 and 9 in two registers control the Rings. Bits 8 and 9 in the decode register at address 0x30 determine whether the PIF drives the corresponding Ring or not (logic one means the PIF drives). Bits 8 and 9 in the `dwnld1` register at address `0x10` are the Ring values. They are always readable, and are write-able if the PIF is driving.

There is a different enable for each of the ring bits. To enable one of the ring bits, the global enable and one of the ring enables both have to be set. This is shown below for `ring[0]`:

```
/*
enable user interrupt bits
interrupt enables: global (bit 4) and ring[0] enable (bit 8)
*/
PAMREGS(pam)->vco = (1<<4) | (1<<8);
```

To generate the actual interrupt, `ring[0]` would have to be driven high for at least one cycle. The driver will then disable the global interrupt enable before passing control to the user's interrupt service routine. Writing a one to bit 10 of the clock register clears the interrupt. This should be done before enable bits are reset and `ring[0]` should no longer be driven high.

The following software sequence clears the interrupt active bit and resets the appropriate enables.

```
/* Clear interrupt */
PAMREGS(pam)->clock = (1<<8);
/* Reset enables */
PAMREGS(pam)->vco = (1<<4) | (1<<8);
```

The same process can be used to enable and clear the `ring[1]` interrupt. The programmer should substitute bit 11 for bit 10 in both the Clock and VCO registers.

8.5.6 Hardware controlled stopping of Clkusr

Clkusr can be stopped by one of four software selectable hardware sources. The stop sources are enabled by loading a 1 into the corresponding enable in the PIF. Clkusr is stopped when any of the enabled stop sources is in a logic low state. Clkusr is guaranteed to stop glitch-free in a logic high state. Due to pipelining and resynchronization Clkusr may stop several cycles after the assertion of the corresponding stop signal. The higher the Clkusr frequency, the more cycles are required, however, even at 90MHz the stopping overhead is no more than 8 Clkusr cycles.

Name	Enable	Source
Ring<0>	bit 16, register 0x28	from PIF or any usrlca
Ring<1>	bit 17, register 0x28	from PIF or any usrlca
Cntlr l.U init<0>	bit 18, register 0x28	from usrlca0
Cntlr l.U init<1>	bit 19, register 0x28	from usrlca1

9
Security Considerations

9.1 Introduction

Many capabilities of PCI Development Platform could compromise system security and integrity if all users were granted free access to them.

- DMA with invalid addresses can provoke hardware exceptions.
- DMA into kernel data structures and code can cause the system to crash or bypass normal access checks.
- Deconfiguring the PIF can provoke hardware exceptions.
- Deconfiguring the PIF allows a new PIF configuration to be loaded, which could disable current security features.
- Promiscuous mode can allow a user to capture bus traffic and thereby gain read access to normally protected data.

To protect against these kinds of abuse, access to some PIF functions is only enabled when certain security enable bits in *struct PamRegs* are set. As described in Chapter 6, these bits can be read from any of the aliased copies of *struct PamRegs* but may only be written using addresses above the 64 KB offset. PCI Development Platform device drivers may restrict access to these addresses to privileged users. For non-privileged users attempts to map the 64 KB to 128 KB range to the PCI Development Platform address space are remapped to 0 KB to 64 KB. In this case, attempts to write the bits by non-privileged users will fail. Therefore, the usual algorithm to set a security bit is to try to write it and then read it to see if it was set. If it was not set, the user lacks sufficient privilege. Note that security bits are persistent: a privileged user may leave a security bit set for a non-privileged user's use.

Currently the security enables are bits 8 to 15 of the control field of *struct PamRegs*. The bit is enabled when it is logic high (1). Bits 14 and 15 are reserved for future use they should always be set to logic low (0).

The security bit functions are as follows:

Bit	Meaning
8	Enable PIF deconfiguration
9	Enable writes to DMA engine from user-area
10	Enable writes to DMA engine from PCI
11	Allow promiscuous mode
12	Disable address stepping on DMA
13	Allow 64-bit Transaction mode
14	Reserved to COMPAQ
15	Reserved to COMPAQ

The following statement shows how to set the all security bits active. On Windows NT systems, `PamControl` is used to change the security bits:

```
C:\pam\bin\PamControl security 3f
```

On TRU64 UNIX systems, the `PamControl` is run from the command line as shown below:

```
/usr/bin/PamControl security 3f
```

Note

The security bits are offset by 8 bits. For example, f sets all security bits and 8 would only set bit 11 (or promiscuous mode).

10
Software

10.1 Run-time Libraries

The following sections summarize the different functions that are included with the run-time libraries. The run-time libraries are broken up into three sections, according to the functions found in the different header files. The variable declarations and in-depth information about each function can be found within the header files source code.

10.1.1 PamRT.h

10.1.1.1 PamOpen

PamOpen opens a PAM board, allows the possibility for multiple PAM boards on a single PCI bus and allows the board to be brought up in a variety of states.

SYNTAX

```
#include <PamRT.h>

Pam PamOpen (device, mode)
const char *device;
enum PamOpenMode mode;
```

PARAMETERS

device	"/dev/pam0"
mode	PamWait PamWaitVerbose PamNoWait PamNoLock

EXAMPLE

```
int     fd;
if ((fd = PamOpen("/dev/pam", PamNoWait)) < 0) {
        perror ("PamOpen");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **PamOpen()** function returns the file descriptor, a nonnegative integer. Otherwise, a value of -1 is returned and errno is set to indicate the error.

10.1.1.2 PamClose

This function closes the PAM board and releases its lock.

SYNTAX

```
#include <PamRT.h>

int PamClose (fd);
int     fd;
```

PARAMETERS

fd A valid open file descriptor as returned by **PamOpen()**.

EXAMPLE

```
int     fd;

if (PamClose(fd)) {
        perror ("PamClose");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **PamClose()** function returns a value of 0 (zero). Otherwise, a value of -1 is returned and errno is set to indicate the error.

10.1.1.3 PamDownloadBitstream

PamDownloadBitstream resets the whole board, downloads a new bit stream to configure it, sets the clock period to the desired value, and establishes the standard initial state.

SYNTAX

```
#include <PamRT.h>

void PamDownloadBitstream (pam, bitstream, clockPeriod)
Pam pam;
const struct PamBitstream *bitstream;
double clockPeriod;
```

PARAMETERS

pam	The pointer to the current pam device.
bitstream	The pointer to the current application bitstream.
ClockPeriod	The clock speed that the FPGA can be loaded at.

EXAMPLE

```
Pam     pam;
extern struct PamBitstream          bitstream;
double clockPeriod = 30;

pam = PamOpen(device, PamWaitVerbose);
PamDownloadBitstream(pam, bitstream, clockPeriod );
```

RETURN VALUES

No return value.

10.1.1.4 PamDownloadFile

PamDownloadFile also downloads a bit stream but takes it from a file.

SYNTAX

#include <PamRT.h>

void PamDownloadFile(pam, *filename, clockPeriod)
Pam pam;
const char *filename;
double clockPeriod;

PARAMETERS

pam	The pointer to the current pam device.
filename	"/dev/pam0"
ClockPeriod	The clock speed that the FPGA can be loaded at.

EXAMPLE

Pam pam;
const char *filename;
double clockPeriod = 30;

pam = PamOpen(device, PamWaitVerbose);
PamDownloadFile (pam, filename, clockPeriod);

RETURN VALUES

No return value.

10.1.1.5 PamSetMode

PamSetMode gets/sets the mode of the interface between the bus interface FPGA and the user-area FPGAs. The action is determined by the *mode* parameter.

SYNTAX

#include <PamRT.h>

void PamSetMode(pam, mode)
Pam pam;
Enum PamInterfaceMode mode;

PARAMETERS

pam	The pointer to the current pam device.
mode	PamTransaction, PamPromiscuous, or PamStatic are valid modes

EXAMPLE

Pam pam;

pam = PamOpen(device, PamWaitVerbose);
PamSetMode(pam, PamPromiscuous);

RETURN VALUES

No return value.

10.1.1.6 PamSetModeAndDelay

PamSetModeAndDelay sets the mode of the interface between the bus interface FPGA and the user-area FPGAs. It also sets the delays for target and master transactions. The action is determined by the *mode* parameter.

SYNTAX

#include <PamRT.h>

void PamSetModeAndDelay(Pam pam, enum PamInterfaceMode mode, int target, int master);

Pam pam;
Enum PamInterfaceMode mode;

PARAMETERS

pam The pointer to the current pam device.

mode PamTransaction, PamPromiscuous, PamPromiscuousTransaction, or pamStatic are valid modes.

target the number of cycles the for a target transaction.

master the number of cycles the for a master transaction.

EXAMPLE

Pam pam;

pam = PamOpen(device, PamWaitVerbose);
PamSetModeAndDelay(pam, PamPromiscuous, 5, 5);

RETURN VALUES

No return value.

10.1.1.7 PamClockOn

PamClockOn starts the programmable user clock.

SYNTAX

#include <PamRT.h>

void PamClockOn(pam, clockIndex)
Pam pam;
int clockIndex;

PARAMETERS

pam The pointer to the current pam device.
clockIndex Sets the clock period to the desired value.

EXAMPLE

Pam pam;

pam = PamOpen(device, PamWaitVerbose);
PamClockOn(pam, clockIndex);

RETURN VALUES

No return value.

10.1.1.8 PamClockOff

PamClockOff stops the programmable user clock.

SYNTAX

#include <PamRT.h>

void PamClockOff(pam, clockIndex)
Pam pam;
int clockIndex;

PARAMETERS

pam	The pointer to the current pam device.
clockIndex	Sets the clock period to the desired value.

EXAMPLE

Pam pam;

pam = PamOpen(device, PamWaitVerbose);
PamClockOff(pam, clockIndex);

RETURN VALUES

No return value.

10.1.1.9 PamClockStep

PamClockStep starts the clock for <count> cycles, and waits for the completion of the burst.

SYNTAX

#include <PamRT.h>

void PamClockStep(pam, clockIndex, count)
Pam pam;
int clockIndex;
int count;

PARAMETERS

pam	The pointer to the current pam device.
clockIndex	Sets the clock period to the desired value.
count	The number of cycles.

EXAMPLE

Pam pam;
int count = 2;

pam = PamOpen(device, PamWaitVerbose);
PamClockStep(pam, clockIndex, count);

RETURN VALUES

No return value.

10.1.1.10 PamWriteWord

PamWriteWord this function performs single transactions to the board.

SYNTAX

#include <PamRT.h>

void PamWriteWord(pam, address, data)
Pam pam;

```
int address;
int data;
```

PARAMETERS

pam	The pointer to the current pam device.
address	This parameter is a word address within the board address space.
data	The data to be written to the pam device.

EXAMPLE

```
Pam pam;
Int address = 0x20000;  /* offset to the User Area */

pam = PamOpen(device, PamWaitVerbose);
PamWriteWord(pam, address, data);
```

RETURN VALUES

No return value.

10.1.1.11 PamReadWord

PamReadWord this function performs single transactions from the board.

SYNTAX

```
#include <PamRT.h>

void PamReadWord(pam, address)
Pam pam;
int address;
```

PARAMETERS

pam	The pointer to the current pam device.
address	This parameter is a word address within the board address space.

EXAMPLE

```
Pam pam;
int address = 0x20000;  /* offset to the User Area */

pam = PamOpen(device, PamWaitVerbose);
PamReadWord(pam, address);
```

RETURN VALUES

No return value.

10.1.1.12 PamFlush

PamFlush flushes the write buffer, ensuring that all pending writes have actually reached the board. It should be used between any two successive transactions to the board for which one cares about sequentially.

SYNTAX

```
#include <PamRT.h>

void PamFlush()
```

PARAMETERS

No parameters needed.

EXAMPLE

PamFlush();

RETURN VALUES

No return value.

Note

On some platforms it may be necessary to do a read from the PCI development platform after the writes to flush the write buffer. This is true on the TRU64 Personal Workstations.

10.1.2 PamState.h

10.1.2.1 PamLcaStateTable

PamLcaStateTable returns a handle to the mapping table needed to extract state information for a given LCA.

SYNTAX

#include<PamState.h>

const struct PamLcaStateTable *PamLcaGetStateTable(lca)
struct PamLcaType lca;

PARAMETERS

lca A pointer to a lca types structure.

EXAMPLE

const struct PamLcaStateTable *statetable;
extern struct PamLcaType lca; /* initialized in the design.c that contains the bitstream. */
statetable = PamLcaGetStateTable(lca);

RETURN VALUES

Returns a handle to the mapping table.

10.1.2.2 PamStateCLB

PamStateCLB returns the value of a given CLB bit. The coordinates of each CLB within the chip can be specified.

SYNTAX

#include<PamState.h>

int PamStateCLB(x, y, bit, lca, *data, *table)
int x;
int y;
PamLcaStateBit bit;
int lca;
const unsigned char *data;
const struct PamLcaStateTable *table;

PARAMETERS

x, y are the coordinates of the CLB within the chip, (0,0) being the upper-left CLB.
bit is the bit to extract within the CLB.
lca is the lca number (0 to 3).

data is the readback data obtained from PamReadbackBitstream;
table is the mapping table obtained from PamLcaGetStateTable.

EXAMPLE

```
int clb_bit;
int x=0, y=0;
PamLcaStateBit bit = PAMLCA_I2;
int lca = 3;
const unsigned char *data;
const struct PamLcaStateTable *table;

clb_bit = PamStateCLB(x, y, bit, lca, data, table);
```

RETURN VALUES

Returns the value (0 or not 0) of a given CLB bit:

10.1.2.3 PamStateIOB

PamStateIOB returns the value of a given IOB bit. The pad number for the individual bit can be specified.

SYNTAX

```
#include<PamState.h>

int PamStateIOB(pad, bit, lca, *data, *table)
int pad;
PamLcaStateBit bit;
int lca;
const unsigned char *data;
const struct PamLcaStateTable *table;
```

PARAMETERS

pad is the pad number within the chip (counted clockwise, 1 being the leftmost pad of the upper side.
bit is the bit to extract within the IOB.
lca is the lca number (0 to 3).
data is the readback data obtained from PamReadbackBitstream.
table is the mapping table obtained from PamLcaStateTable.

EXAMPLE

```
int iob_bit;
int pad = 1;
PamLcaStateBit bit = PAMLCA_I2;
int lca = 3;
const unsigned char *data;
const struct PamLcaStateTable *table;

 iob_bit = PamStateIOB(pad, bit, lca, data, table);
```

RETURN VALUES

Returns the value (0 or 1) of a given IOB bit.

10.1.2.4 PamStateIsRam

PamStateIsRam returns the value 1 if the corresponding LUT to <bit> is configured as RAM, otherwise 0.

SYNTAX

#include<PamState.h>

```
int PamStateIsRAM( x, y, bit, lca, *data, *table);
int x;
int y;
PamLcaStateBit bit;
int lca;
const unsigned char *data;
const struct PamLcaStateTable *table;
```

PARAMETERS

x, y	are the coordinates of the CLB within the chip, (0,0) being the upper-left CLB;
bit	is the LUT WE bit to extract within the CLB. Valid values are PAMLCA_F or PAMLCA_G.
lca	is the lca number (0 to 3);
data	is the readback data obtained from PamReadbackBitstream;
table	is the mapping table obtained from PamLcaGetStateTable.

EXAMPLE

```
int lut_bit;
int x=0, y=0;
PamLcaStateBit bit = PAMLCA_I2;
int lca = 3;
const unsigned char *data;
const struct PamLcaStateTable *table;

lut_bit = PamStateIsRam(x, y, bit, lca, data, table);
```

RETURN VALUES

Returns the value 1 if the corresponding LUT to <bit> is configured as RAM, otherwise 0.

10.1.2.5 PamStateLUT

PamStateLUT returns the value of the LUT corresponding to <bit>.

SYNTAX

#include<PamState.h>

```
int PamStateLUT( x, y, bit, lca, *data, *table);
int x;
int y;
PamLcaStateBit bit;
int lca;
const unsigned char *data;
const struct PamLcaStateTable *table;
```

PARAMETERS

x, y	are the coordinates of the CLB within the chip, (0,0) being the upper-left CLB;
bit	is the LUT to extract within the CLB. Valid values are PAMLCA_F or PAMLCA_G.
lca	is the lca number (0 to 3).
data	is the readback data obtained from PamReadbackBitstream;
table	is the mapping table obtained from PamLcaGetStateTable.

EXAMPLE

```
int lut_bit;
int x=0, y=0;
PamLcaStateBit bit = PAMLCA_I2;
int lca = 3;
const unsigned char *data;
const struct PamLcaStateTable *table;

lut_bit = PamStateLUT(x, y, bit, lca, data, table);
```

RETURN VALUES

Returns the value of the LUT corresponding to <bit>.

10.1.3 PamFriend.h

10.1.3.1 PamFindBoard

This function returns a pointer to struct PamBoard that corresponds to a given user pointer.

SYNTAX

```
#include<PamFriend.h>

struct PamBoard *PamFindBoard(pam)
Pam pam;
```

PARAMETERS

pam the user pointer or 0 if this entry is free.

EXAMPLE

```
Pam pam = PamOpen(device, PamWaitVerbose);
struct PamBoard *board = PamFindBoard(pam);
```

RETURN VALUES

Returns a pointer to struct PamBoard.

10.1.3.2 PamReadInfo

PamReadInfo reads the “information” registers of the PAM board, checks them and fills the corresponding registers in the PamBoard structure.

SYNTAX

```
#include<PamFriend.h>

void PamReadInfo(Pam pam)
Pam pam;
```

PARAMETERS

pam the user pointer or 0 if this entry is free.

EXAMPLE

```
struct PamBoard *board = PamFindBoard(pam);
int rev;

PamReadInfo(pam);
rev = board->fwRevision;

printf (“ The firmware version of this board is %d.”, rev);
```

RETURN VALUES

No return value.

10.1.3.3 PamRegisterLayout

PamRegisterLayout returns an integer, which identifies the control register layout used by this board.

SYNTAX

```
#include<PamFriend.h>

int PamRegisterLayout(*board)
struct PamBoard *board;
```

PARAMETERS

board corresponds to a given user pointer.

EXAMPLE

```
struct PamBoard *board = PamFindBoard(pam);
int reg_layout;
int nM0_mask;

/* Check that the LCAs are configured */
/* Only applies to boards that do not support partial configuration */

switch (reg_layout = PamRegisterLayout(board)) {

   case 0:

     nM0_mask = TPAMCTRL_NM0;

     break;

   case 1:

     nM0_mask = PPAMCTRL_NM0;

     break;

   default:

     PamError(PamErrInfo, pam, "Unsupported register layout");
```

RETURN VALUES

Returns 0 if the board type is less than 2 or if the firmware version is 1 or if the firmware revision is less than 8. Otherwise it returns a 1.

10.1.3.4 PamResetAll

PamResetAll returns the complete board to its initial state, resets all the control registers, and deconfigures the user LCAs.

SYNTAX

```
#include<PamFriend.h>

void PamResetAll(pam)
Pam pam;
```

PARAMETERS

pam The pointer to the current pam device.

EXAMPLE

```
Pam pam = PamOpen(device, PamWaitVerbose);

PamResetAll(pam);
```

RETURN VALUES

No return value.

10.1.3.5 PamDownloadLcas

PamDownloadLcas downloads a bit stream to the board. If the bit stream chosen has some LCA positions empty, the corresponding LCAs on PCI Development Platform are left with their current configuration. This allows a form of hardware overlay. It is the programmer's responsibility to ensure that these configurations produced by this overlay are compatible.

SYNTAX

```
#include<PamFriend.h>

void PamDownloadLcas(pam, *bitstream, check);
Pam pam;
const struct PamBitstream *bitstream;
int check;
```

PARAMETERS

pam	The pointer to the current pam device.
bitstream	is a list of bitstreams that are searched in order.
check	a value of 1 = safe/slow version, and 0 = the fast unchecked version.

EXAMPLE

```
Pam pam = PamOpen(device, PamWaitVerbose);
extern struct PamBitstream bitstream;
double clockPeriod = 30;
int check = 1;

PamDownloadLcas(pam, *bitstream, check);
```

RETURN VALUES

No return value.

10.1.3.6 PamReadBackMinLength

PamReadBackMinLength returns the minimum size of the buffer holding a readback bit stream for a given board.

SYNTAX

```
#include<PamFriend.h>
unsigned PamReadbackMinLength(pam)
Pam pam;
```

PARAMETERS

pam	The pointer to the current pam device.

EXAMPLE

```
struct PamBoard *board = PamFindBoard(pam);
unsigned length = PamReadbackMinLength(pam);
```

RETURN VALUES
Returns the minimum size of the buffer.

10.1.3.7 PamReadBackBitstream

PamReadBackBitstream reads back the configuration bit stream and stores it in <bits>, which is of size <bits_length>. The bits buffer must be large enough to hold the readback bit stream of the largest chip currently installed on the board.

SYNTAX

#include<PamFriend.h>

void PamReadbackBitstream(pam, bits, length)
Pam pam;
unsigned char bitb[ ];
unsigned bits_length;

PARAMETERS

pam	The pointer to the current pam device.
bitb	will contain the n'th element raw readback bitstream as defined in the data book.
bits_length	size of the buffer returned by PamReadBackMinLength.

EXAMPLE

```
struct PamBoard *board = PamFindBoard(pam);
unsigned length = PamReadbackMinLength(pam);
unsigned char *bits = (unsigned char *)malloc(length);

if (bits == 0)
   PamError(PamErrNoMem, pam, 0);

PamReadbackBitstream(pam, bits, length);
```

RETURN VALUES

No return value.

10.1.3.8 PamSetClockSpeed

PamSetClockSpeed sets the VCO range and dividers to approximate the given period. It does not wait for the clock speed to stabilize. It takes the value (in MHz) of the reference frequency from the PamBoard structure (see section 10.1.3.9). The reference frequency in the PamBoard structure should be changed to the desired value if an external reference clock is used.

SYNTAX

#include<PamFriend.h>

void PamSetClockSpeed(pam, period)
Pam pam;
double period;

PARAMETERS

pam	The pointer to the current pam device.
period	Sets the clock to the desired period.

EXAMPLE

```
Pam pam = PamOpen(device, PamWaitVerbose);
double speed =  30;
double period = 1000.0/speed;

PamSetClockSpeed(pam, period);
```

RETURN VALUES

No return value.

10.1.3.9 PamClockPeriod

PamClockPeriod returns the exact clock period in nanoseconds as would be set by a call to PamSetClockSpeed with the same `<period>` parameter. It does no physical access to the board.

SYNTAX

#include<PamFriend.h>

double PamClockPeriod(Pam pam, double period);
Pam pam;
Double period;

PARAMETERS

pam	The pointer to the current pam device.
period	Sets the clock to the desired period.

EXAMPLE

PamSetClockSpeed(pam, period);

(void)printf("Clock period set to %.5fns (%.5fMHz)\n",

PamClockPeriod(pam, period),

1000.0 / PamClockPeriod(pam, period));

RETURN VALUES

Returns the exact clock period in nanoseconds.

10.1.3.10 PamWaitClock

PamWaitClock waits until the clock stepper/stopper has finished executing its last command.

SYNTAX

#include<PamFriend.h>

void PamWaitClock(Pam pam)
Pam pam;

PARAMETERS

pam	The pointer to the current pam device.

EXAMPLE

Pam pam = PamOpen(device, PamWaitVerbose);

PamWaitClock(Pam pam);

RETURN VALUES

No return value.

10.2 PAM Register Declaration

The register layout within the PCI interface can be found in the PamRegs.h header file. Control registers have a naturally sparse layout to simplify operation on early Alpha systems that have a normal read granularity of 64-bits. It is recommended that the programmer check register compatibility between different revisions of the software kit and firmware.

11
Command Line Controls

11.1 Introduction

The following sections describe the command line tools for controlling the PCI Development Platform module and other utilities in the software package. The name of each section is the name of the executable for Windows NT. If the TRU64 UNIX name is different, it is shown in parenthesis to the right.

11.2 PamTest

PamTest is a command line utility that tests the functionality of Development Platform. PamTest is not a design verification tool. It tests the basic operation of the board. For example, PamTest verifies the connections between FPGAs, the clock speed, and ability to download designs. To see all the current commands in PamTest, run PamTest -help. The man page for PamTest is also shown below. Not all of the commands listed are used for the PCI Development Platform module. The PamTest utility has been used for multiple versions of Development Platform. The PamTest option PamTest -C 0 will verify the correct operation of the Development Platform module. Most of the other tests are used during manufacturing and initial testing of the board and require test fixtures. The test fixtures needed for these options are not provide with the module. There are also useful options to create log files for errors and to exclude particular tests.

Note

Any daughter card should be disconnected when running PamTest.

```
PamTest - test of Pamette board

SYNTAX

  PamTest [testname ...] [options]

DESCRIPTION

  Testnames for PCI Pamette are (default is all):

       download
       clock
       connect
       readback
Testnames for TURBOchannel Pamette are (default is all):

       init

       prom
```

```
        download
        clock
        connect
        readback
        io
        dma
  Options are:
  -dev device
        (default is /dev/pam0)
  -C level
        connect level for PCI Pamette v1:
        level = 0
          internal connections only
        level = 1
without loop-back connectors
        level = 2
          with type 1 loopback connector
        level = 3
          with type 1 loopback connector and DRAM
  connect level for TURBOchannel Pamette:
        level = 0
          internal connections only
        level = 1
          without loop-back connectors
        level = 2
          with loop-back connectors on P2 P3 P4 P5 P6
        level = 3
          with loop-back connector on P7 or on P5 P6
  -e testname
        exclude testname
  -forceprom
force testprom even if EEPROM initialised
  -log filename
        append error messages in filename
-nostatus
        do not print status messages
  -nostop
        do not stop on error
```

```
-pass nb_pass
     (default is 1)
-print nb_err
     (default is 10)
-debug Code
     reserved to test debug
```

Note:

The PamTest application is currently designed for PCI buses that run at 33.3 MHz. Many PCI buses do not meet this requirement (the PCI specification only requires a PCI bus to operate at less than or equal to 33.3 MHz). This would cause the Pamtest application to fail during the clock test. If PamTest reports a clock frequency less than 33.3 MHz, your module may be functioning correctly. To check the functionality of the board run the following PamTest option, which excludes the clock test.

On Windows NT systems, run the following command from the MS-DOS prompt:

```
C:\pam\bin\PamTest –C 0 –e clock
```

On TRU64 UNIX systems, the appropriate command is shown below:

```
/usr/bin/PamTest –C 0 –e clock
```

The appropriate result for TRU64 UNIX and Windows NT is shown below.

```
-- PamTest of Aug 18 19971 15:18:11        --
Board : 2.1      Firmware : 2.0      Serial Number : 0
Config : 4020E   4020E  4020E  4020E
Download  OK
Connect   OK
Readback  OK
Sram      OK
```

11.3 Mergebit

The Mergebit application concatenates the four bit streams used to program the four individual FPGAs in the user-area, directions for use of the application follow:

```
Usage: mergebit [-v] [-c <name>] [-o <outfile>]
       <design>[.mergebit] ...
```

```
This program reads one or more description files "design.mergebit"
containing the names of the 4 rawbit (.rbt) files to merge, and
produces a binary file "<outfile>.pam" ready to be downloaded into a
PAM board. Multiple description files specify alternate bit streams.
The download process examines each possible bit stream until it finds
one which is compatible with the target board. The description file
must contain the file names in order, separated by whitespace. If the
extension ".rbt" is not present on a filename, it is added. If a
filename is "-", it will be substituted by a null bit stream (stream of
1s). The description file may also contain comments, extending from a
semicolon character ';' to the end of the line. LCA order is as
follows:

             LCA00 LCA01
             LCA02 LCA03
```

```
The -v flag selects the "verbose" mode (file names are printed).

If the -c flag is present, the output will be a C file suitable for
compilation and linking with the driver program of the application.
In this case, the default output filename is "design_pam.c". "name"
will be the C name of the structure declared.

The -o flag changes the name of the output file.  If the -o flag is
not present, the outfile file name is derived from the name of
first "design.mergebit" file.
```

11.4 Prom (ppam_prom for TRU64 UNIX)

The prom application is used for reading and writing to both the SROM (Xilinx) and the EEPROM (Amtel). The program can be used to read the current configuration or to write a new configuration to the EEPROM.

```
Usage C:\PAM\BIN\PROM.EXE read|write|copy [-x|a] [ -llenght] [-dev
        device] [filename]

   -x is the Xilinx prom (default)

   -a is the Amtel eprom

   length is in bytes (default 32 kB)

   input file can be either a RBT file a raw hex file

   output file is a raw hex file
```

11.5 PamControl

PamControl is a command line interpretive mode that allows direct access to Development Platform controls. This program allows the user to configure the board appropriately to their application by setting the mode of the board and the security bits. To see all the current commands in PamControl, run PamControl and type help on the command line. The man page for PamControl is shown below.

```
Usage:  PamControl [-dev dev] [-nolock|-wait] [command]

DESCRIPTION

This command-based program can perform several control and debug
operations on a Pamette board. If a command argument is given, it
will execute it and exit, otherwise it will enter interactive mode.
The help command lists the currently available commands. Expect
this list to change frequently as this program is used as a debug
tool and new functions are added to it as needed.

OPTIONS

  -dev dev  Open pam device dev instead of the default /dev/pam0.

-nolock   Open the pam device without taking the lock.

  -wait     Wait for the pam device to be unlocked.
```

11.6 Pciperf (ppam_pciperf for TRU64 UNIX)

The Pciperf (ppam_pciperf) tool is used to exercise and spy on the PCI bus. The tool can be used to measure throughput and traffic on the PCI bus. The Pciperf command has the ability to exchange 64 bits on the PCI bus, if your system supports the 64-bit extended PCI. The other tests only test 32 bits, allowing them to be used on all PCI busses; therefore, this function can be used to test the full PCI extension. The following man page explains the function and set up of the Pciperf tool.

```
This application turns the PCI Pamette into a PCI exerciser and
spy. The information in this README regarding register layout has
been superceded by the interface.ps or interface.pdf document in
the PCI Pamette documentation archive.

NB: As of v1.8 of the PCI Pamette firmware initiating DMA and using
the spying mode (a.k.a. promiscuous mode) requires the setting of
security bits (as documented in interface.ps)

This is most easily done with the command:

        PamControl security C

which sets security bits 2 - to allow DMA - and 3 to allow
promiscuous mode.

-------------------------------------------------------------

If . is the directory into which you unpacked the archive and from
which you are reading this README, then the application is in
./runtime/pciperf

Running pciperf with no arguments prints the following cryptic
usage message:

Usage: runtime/pciperf

       [ -dev device ]

       [ -verbose 0-6 ] (-v also accepted)

       [ -timeout n ] (default 100)

       [ -rep repetions ] (default 1)

       [ -flush ]

       [ -dirty ]

       <cmd> [<flags>]


 cmd is small digit for PIO tests or coded DMA engine command for
DMA

  see README for more details
```

```
Meaning of flags

-verbose 0   [default] prints a single number which is the MB/s
achieved.

-verbose 1   prefixes the number with "Average rate:  "

-verbose 2   also prints the rate for each pass (meaningful only
with -rep n>1)

-verbose 3   prints a summary of the burst types seen.

-verbose 4   prefixes the summary with a compact transaction log

-verbose 5   prints a detailed log of every bus cycle recorded.

-verbose 6   prints uninterpreted trace data for debugging the
program itself

-timeout n   specifies the number of idle bus cycles after which
the trace analysis stops. When spying on other traffic (cmd code 10
below) you may wish to set a large value, say 100000.
```

```
-rep n   number of times to repeat the test

-flush   tries to ensure that DMA memory region is not in cache

-dirty   tries to ensure that DMA memory region is in cache

The cmd determines how the PCI bus is exercised.


PIO tests


PIO tests read/write a contiguous 2 kB block.

0 - 32-bit PCI reads.

1 - 32-bit PCI writes.

2 - like 0, but accepts 64-bit PCI reads.

3 - like 1, but accepts 64-bit PCI writes.

10 - generates no traffic on its own but may spy other traffic.


Codes 4-F are TRU64 UNIX specific, not all are currently
implemented

4 - used with a special tweaked version of the PCI Pamette PCI
interface which responds more quickly in a certain address region.
On the standard PCI interface 4 behaves like 0

6 - like 4, but accepts 64-bit transactions

8 - on ev5 it seems by using a non-linear load sequence we can
persuade the processor to issue 32 byte reads.  The specific
sequence within a 32 byte block is [0..7] [24..31] [8..15] [16..23]

A - like 8, but accepts 64-bit transactions

C - like A, but loop unrolled once

DMA tests and command encoding

------------------------------

cmd may also encode a DMA transaction request. On TRU64 UNIX, user
must be root to generate DMA transactions, because we need the
mlock syscall.

The DMA transaction are encoded in a arcane format that exactly
mirrors the value to be loaded into the 32-bit CSR that controls
DMA requests. See the interface.ps or interface.pdf documents for a
definitive reference on DMA command encoding.

By default DMA requests are 32-bit linear increment. The optional
flags permits the specification of:

1 - intel cacheline wrap mode (now deprecated)

2 - cacheline wrap with linear increment within cacheline

3 - 64 bit
```

```
The fields in the DMA control CSR are as follows.

        31..28  : PCI command 3..0

        15..12  : delay to next request

        11.. 2  : 0x400 - burst length

         1      : must be 0

0      : must be 1

For example ppam_pciperf -v 3 7c000fc1 on an AlphaStation 200 4/166
might report:

# ppam_pciperf -v 3 7c000fc1

Total DMA length = 4096 bytes

DMA Engine preferred burst length = 64 bytes

67.3% A [mmwr 1*decode 15*data / 1*data]

13.7% B [mmwr 1*decode 8*data 1*disc / 1*disc]

11.5% C [mmwr 1*decode 7*data / 1*data]

 0.6% D [mmwr 1*decode 1*retry / 1*retry]

 6.9% idle

Pass average rate: 103.2 MB/s

Average rate: 103.2 MB/s


Burst lengths of 2 or 3 are treated as burst length 1 due to
internal pipelining restrictions.

Some care should be exercised in the choice of "length" and overrun
due to the pipelined control that was necessary in this part of the
PCI interface. The DMA engine will start a new burst whenever the
specified "length" has not been achieved.

If the total "length" is not a multiple of the "burst length", the
DMA engine may issue extra DMA cycles up to the next multiple of
"burst length" beyond the specified length.

The interface will start requesting DMA when bit 27 is set. Bits
27..16 are a counter which will count up by one for each longword
transferred -- when bit 27 transitions to zero DMA stops. Be
careful at the page boundary, it may be possible for a burst
started just before it to spill into the next page -- it depends on
the host bridge.

The length field (11..2) is the desired burst length in long words.
If the burst is disconnected the next request will try to complete
the DMA up to the burst boundary then stop and start a new request
for the next burst. Top bit must be one in length counter otherwise
burst length is limited to one. Hence maximum burst size is 0x200
or 2048 bytes.

The "delay to next request" (bits 15..12) specifies a delay between
bursts. This let's us throttle back the rate, it also let's us
optimize throughput on host bridges that introduce their own wait
states.
```

When the remaining "length" is less than twice the "burst length" and "delay to next request" is set to 0 or 1, the interface increases "delay to next request" to 2 or 3. If this is not done the DMA engine can issues an extra burst (beyond the rules stated above) because the decrement of "length" is delayed by a couple of cycles from the cycle when the actual data is transferred.

Notes on "-verbose 3" and "-verbose 4" output

The burst type summary produced by the flag "-verbose 3" should be read as follows: each distinct transaction is given an upper case alphabetic classification. The percentage is the percentage of total bus cycles that the bus was busy with this type of transaction. Next is a breakout of the sequence of cycles within a given transaction type, the "/" indicates when frame went high, dataSkip means a data phase but with byte masks disabled, other cycle type names should I hope be self explanatory, repeated cycles are indicated by count*type, e.g. 3*data64 means three 64 bit data cycles.

The observed transaction log over time using the upper case alphabetic transaction type classifications. The log consists of a sequence of pairs, a count of number of cycles the PCI was idle, and the transaction type that was observed after those idle cycles. If the transaction type is identical to the immediately previous transaction type a "." is substituted for the upper case alphabetic.

For instance in the following example we see a sequence of 16 dword memory writes each separated by one idle cycle lasting in total about 1300 bytes, followed by a mixture 16 dword memory writes and 8 dword memory writes caused by host disconnects.

```
# ppam_pciperf -verbose 4 7e010fc1

Total DMA length = 2044 bytes

DMA Engine preferred burst length = 64 bytes

 6 A  1 .  1 .  1 .  1 .  1 .  1 .  1 .

 1 .  1 .  1 .  1 .  1 .  1 .  1 .  1 .

 1 .  1 .  1 .  1 .  1 B  1 A  1 .  1 C

 1 D  1 A  1 .  1 .  1 .  1 C  1 D  1 A


DMA Engine preferred burst length = 64 bytes

 6 A  1 .  1 .  1 .  1 .  1 .  1 .  1 .

 1 .  1 .  1 .  1 .  1 .  1 .  1 .  1 .

 1 .  1 .  1 .  1 .  1 B  1 A  1 .  1 C

1 D  1 A  1 .  1 .  1 .  1 C  1 D  1 A

85.4% A count=30   len=18  bytes=64  avg-idle=1.3

 3.8% C count=2    len=12  bytes=32  avg-idle=1.0

 3.2% D count=2    len=10  bytes=32  avg-idle=1.0

 0.6% B count=1    len=4   bytes=0   avg-idle=1.0
```

```
Total observed data = 2048 bytes
-------
85.4% A [mmwr 1*decode 15*data / 1*data]
 3.8% C [mmwr 1*decode 8*data 1*disc / 1*disc]
 3.2% D [mmwr 1*decode 7*data / 1*data]
 0.6% B [mmwr 1*decode 1*retry / 1*retry]
 7.0% idle
Pass average rate: 106.9 MB/s
```

12

Restoring Original PCI Interface Design or Upgrading Firmware

12.1 Introduction

As discussed in Section 2.1.2, the PCI development platform has the ability to be programmed with a customized PCI interface. The EEPROM is used to store the current PCI interface design. The user has the ability to overwrite the information in this EEPROM with their own design. The original design is permanently stored in the SROM and this design can be restored in a few simple steps.

Power down the system and place the fail safe jumper in the ON position. The failsafe jumper is located on the front of the board in the upper left corner (see **Error! Reference source not found.**). Reboot the system and run `PamControl`. PamControl is explained in detail in Section 11.5. At the PamControl prompt, run the `SetConfig` option. The user is prompted for the serial number. The serial number can be found on the yellow label, which is located on the back of the board. Enter only the last six digits of the serial number. The program then prompts for the LCA numbers. The LCA numbers identify the version of the board. The boards are classified according to the FPGAs mounted in the user area. The list below explains the value which should be assigned to each LCA according to the model number of the board.

For 2T-PAMP1-PM, 2T-PAMP1-AA, and 2T-PAMP1-BA

```
LCA0?     4010E
LCA1?     4010E
LCA2?     4010E
LCA3?     4010E
```

For 2T-PAMP1-CA and 2T-PAMP1-DA

```
LCA0?     4020E
LCA1?     4020E
LCA2?     4020E
LCA3?     4020E
```

For 2T-PAMP1-EA

```
LCA0?     4028EX
LCA1?     4028EX
LCA2?     4028EX
LCA3?     4028EX
```

For 2T-PAMP2-FA

```
LCA0?     4044XLA
LCA1?     4044XLA
LCA2?     4044XLA
LCA3?     4044XLA
```

For 2T-PAMP2-GA

```
LCA0?     4085XLA
LCA1?     4085XLA
LCA2?     4085XLA
LCA3?     4085XLA
```

If the model number is not known, examine the FPGAs and identify the size of the chips in the user area. There are five FPGAs mounted on the board. Four of them should have the same size. The size is located underneath the word XILINX. The first two letters are always XC and the size follows. For example, if the FPGA says XC4010E, the size of the chip is 4010E. The above list identifies the current available options.

After finishing SetConfig and exiting PamControl, the EEPROM should be written with the configuration information just entered. On Windows NT systems, the EEPROM is written from the MS-DOS prompt as shown below:

```
C:\pam\bin\prom write  c:\pam\lib\pif.rbt
```

On TRU64 UNIX systems, the SRAM Test is run from the command line as shown below:

```
/usr/bin/ppam_prom write /usr/lib/Pam/pif.rbt
```

The original configuration is now restored in the PCI interface. The system should be powered down and the failsafe jumper placed in the OFF position. When the system is rebooted, the original configuration is restored.

A
Major User Buses/Pins

The following table lists the major user signal buses along with pin connections present and their use.

Table -1 Major PCI Development Platform User Buses/Pins

User Signal	pif	Lca0	Lca1	Lca2	Lca3	PMC/ CMC	Use
Ebus[31:0]	X	X	X				TM Datapath PIF to user-area
Ebus[34:32]	X	X	X				TM Ebus State
Ebus[35]	X	X	X				TM Request Code
cnfgP_ld.Din	X	X	X				TM Request Code
Sbus0.x[1:0][2:0] (6 pins)		X	X				User-defined Sbus0.x[0][1] on lca0 output ONLY Sbus0.x[1][1] on lca1 output ONLY
Sbus0.d[31:0]		X	X				User-defined
Sbus1.x[1:0][1:0] (4 pins)				X	X		User-defined Sbus1.x[0][1] on lca2 output ONLY Sbus1.x[1][1] on lca3 output ONLY
Sbus1.d[15:0]				X	X		User-defined
Wbus0[35:0]		X		X			User-defined
Wbus1[35:0]			X		X		User-defined
Clksys	X	X	X	X	X	X	Copy of PCI or External Clock
ClkUsr		X	X	X	X		User-programmable Clk
Srbus0.addr[14:0]		X					Lca0 SRAM Address
Srbus0.data[15:0]		X					Lca0 SRAM Data
Srbus0.write		X					Lca0 SRAM Write
Srbus0.oe		X					Lca0 SRAM Output Enable
Srbus0.bank[1:0]		X					Lca0 SRAM Bank Select
Srbus1.addr[14:0]			X				Lca1 SRAM Address
Srbus1.data[15:0]			X				Lca1 SRAM Data
Srbus1.write			X				Lca1 SRAM Write
Srbus1.oe			X				Lca1 SRAM Output Enable
Srbus1.bank[1:0]			X				Lca1 SRAM Bank Select
X = pin connections present							

Table A-1 (Cont.) Major PCI Development Platform User Buses/Pins

User Signal	pif	Lca0	Lca1	Lca2	Lca3	PMC/ CMC	Use
Spci (95 pins)				X		X	User-defined I/O -or- Connections support secondary 64-bit PCI bus Lca2 Pins 5,6,47,197,198 Output ONLY Lca2 Pins 9, 76 Input ONLY
Spci.reserved[3:0]					X	X	User-defined Input ONLY
Spci.interrupts (3 pins)					X	X	User-defined I/O -or- Can be connected directly to Primary PCI bus interrupts
Drbus (92 pins) .addr, data, ras, cas, write					X	X	64 User-defined I/O -or- DRAM Interface Ras, cas, write NOT connected to PMC
Busmode[3:0]			X			X	User-defined I/O
Clkext					X	X	External Clock input
Ring[1:0]	X	X	X	X	X		User-defined Connected to Global buffers within FPGAs
X = pin connections present							

B
Special Purpose/Restricted Use Pins

The following table lists the special purpose/restricted use signals along with the pin connections present, their special use, and restrictions for user application.

Table B-1 PCI Development Platform Special Purpose/Restricted Use Pins

Signal	pif	Lca 0	Lca 1	Lca 2	Lca3	Special Use	Restrictions for User Application
Cntlr_l_0 .u_prog, .rb, .u_din, .u_init	X	X				User-area Download / Readback	DO NOT USE Pins 108, 48, 77 Lca0 Pin 151 (.u_din) is connected to lca1 Pin 99 and may be used as user I/O after configuration
Cntlr_l_1 .u_prog, .rb, .u_din, .u_init	X		X			User-area Download / Readback	DO NOT USE Pins 108, 48, 77 Lca1 Pin 151 (.u_din) is connected to lca1 Pin 99 and may be used as user I/O after configuration
Cntlr_l_2 .u_prog, .rb, .u_din, .u_init	X			X		User-area Download / Readback	DO NOT USE Pins 108, 48, 77 Lca2 Pin 151 (.u_din) is connected to lca3 Pin 99 and may be used as user I/O after configuration
Cntlr_l_3 .u_prog, .rb, .u_din, .u_init	X				X	Configuration / Readback	DO NOT USE Pins 108, 48, 77 Lca3 Pin 151 (.u_din) is connected to lca2 Pin 99 and may be used as user I/O after configuration
Cntlr_g .u_done, .u_cclk, .u_rbtrig	X	X	X	X	X	User-area Download / Readback	DO NOT USE Pins 103, 153, 50
CnfgP.ld0 .cclk, .prog, .done, .promce, .ser_en[1:0]		X				Used to allow lca0/1 to source the configuration bitstream.	DO NOT USE lca0 Pins 204, 46, 47, 203, 76, 59 for normal user application.
X = pin connections present							

Table B-1 (Cont.) PCI Development Platform Special Purpose/Restricted Use Pins

Signal	pif	Lca 0	Lca 1	Lca 2	Lca3	Special Use	Restrictions for User Application
CnfgP.ld1 .u_reset, .otp_ce, .ce_mux			X			Used to allow lca0/1 to source the configuration bitstream.	DO NOT USE lca1 Pins 204, 203, 76 for normal user application.
CnfgP.ld.din	X	X	X			Used to allow lca0/1 to source the configuration bitstream to pif.	DO NOT USE Pin 5 for normal user application.
X = pin connections present							

C
PAM Driver Interfaces for TRU64 UNIX

PAM Driver Interfaces

The following sections describe the PAM driver interfaces.

open()

The **open()** function establishes a connection between the PAM device named by the path parameter and a file descriptor. The opened file descriptor is used by subsequent I/O functions, such as **read()**, **write()**, and **ioctl().**

SYNTAX

```
#include <sys/fcntl.h>

int open (path, oflag, mode );
const char      *path;
int             oflag;
mode_t  mode;
```

PARAMETERS

path	"/dev/pam”
oflag	O_RDWR
mode	This parameter is not used.

EXAMPLE

```
int     fd;
if ((fd = open("/dev/pam", O_RDWR, 0)) < 0) {
        perror (“open”);
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **open()** function returns the file descriptor, a nonnegative integer. Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **open()** man page for a description of the possible values for errno.

close()

This system call is used by an application to release exclusive control of the PAM device.

SYNTAX

```
int close (fd);
int     fd;
```

PARAMETERS

fd A valid open file descriptor as returned by **open()**.

EXAMPLE

```
int     fd;

if (close(fd)) {
        perror ("close");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **close()** function returns a value of 0 (zero). Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **close()** man page for a description of the possible values for errno.

ioctl()

The **ioctl()** system call performs device specific actions on or for a device. The action is determined by the *request* parameter. The following sections will describe each of the requests and provide examples of their usage.

Each **ioctl()** system call will have the same general syntax. The differences of the specific requests will be discussed in the following sections.

SYNTAX

```
#include <sys/ioctl.h>
#include "PamRegs.h"

int ioctl (fd, request, arg)
int             fd;
unsigned long   request;
void            *arg;
```

PARAMETERS

fd A valid open file descriptor as returned by **open()**.
request The ioctl command to be performed on the device.

PAMIOUVTOPHY
PAMIOGETOWNER
PAMIOGETTCINFO
PAMIOGETRECVR
PAMIOSETRECVR
PAMIOSETOWNER
PAMIODISABLEINTR
PAMIOENABLEINTR
PAMIOGETINTRTIME
PAMIORESTORECONFIG
PAMIOUVTOBUS
PAMIOGETBUSCLKPSPERIOD
PAMIOGETDEVMEMSIZE

arg The parameters for this request.

EXAMPLE

```
if (ioctl (fd, PAMIOSETRCVR, 0)) {
        perror ("PAMIOSETRCVR");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **ioctl()** function returns a value of 0 (zero). Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **ioctl()** man page for a description of the possible values for errno.

PAMIOUVTOPHY

Using the PAMIOUVTOPHYas the request parameter converts the current virtual address to a physical address.

This should be used in conjunction with mlock(2) system call (implies effective uid==0). If virtual address is not locked the translation may change at any time.

This ioctl was used to used for DMA on TURBOchannel. It must not be used with PCI devices. Use PAMIOUVTOBUS instead.

PAMIOGETOWNER

Using the PAMIOGETOWNER as the request parameter returns the process id of a current owner process.

The owner process provides advisory information to help identify a current process holding lock.

PAMIOGETRECVR

Using the PAMIOGETRECVR as the request parameter returns the process id of the current interrupt receiver process.

PAMIOSETRECVR

Using the PAMIOSETRECVR as the request parameter sets the interrupt receiver to the current process.

PAMIOSETOWNER

Using the PAMIOSETOWNER as the request parameter sets the process id of current owner process.

PAMIODISABLEINTR or PAMIOENABLEINTR

Using PAMIODISABLEINTR or PAMIOENABLEINTR as the request parameter enables or disables interrupts.

These routines do not control hardware posting of interrupts. They just control the drivers ability to access the board in the face of an interrupt. If an interrupt arrives from the board while interrupts are disabled the system will hang. The purpose of these routines is to unwire the interrupt handler while the interface is being reconfigured so that board accesses are not generated by interrupts from other devices which share the same interrupt line.

Your process must have superuser privilege to implement these calls.

PAMIOGETINTRTIME

Using PAMIOGETINTRTIME as the request parameter gets the rpcc recorded at the time of the last interrupt.

PAMIORESTORECONFIG

Using the PAMIORESTORECONFIG as the request parameter restores the PCI configuration registers to values recorded at boot time.

PAMIOUVTOBUS

Using the PAMIOUVTOBUS as the request parameter converts the user virtual address to an I/O bus address. This call should be used in conjunction with the mlock(2) system call (implies effective uid==0 -- superuser).

PAMIOGETBUSCLKPSPERIOD

Using the PAMIOGETBUSCLKPSPERIOD as the request parameter will get the bus clock period (in picoseconds).

PAMIOGETDEVMEMSIZE

Using the PAMIOGETDEVMEMSIZE as the request parameter gets the device memory space size.

Protecting Resources in an SMP Environment

The Pam driver currently contains no mechanisms for protecting resources in an SMP environment. If the user requires synchronized access to kernel data when executing multiple threads, modify the stanza.static file to funnel a device driver onto a single CPU. You can do this by modifying the following from:

```
Device_Char_Funnel = DEV_FUNNEL_NULL
```

to

```
Device_Char_Funnel = DEV_FUNNEL
```

D
PAM Driver Interfaces for Windows NT

Please refer to the source code and samples that are installed with the device driver.

E
Register Definitions

Control Register (0x8)

Bit	Meaning
Control<0>	Program pin LCA0
Control <1>	Program pin LCA1
Control <2>	Program pin LCA2
Control <3>	Program pin LCA3
Control <6>	Init pin for the PIF
Control <7>	Enable Init pin for the PIF
Control <8>	Enable PIF to be deconfigured
Control <9>	Enable DMA engine access from the user area
Control <10>	Enable DMA engine access from the host
Control <11>	Enable Promiscuous mode
Control <12>	Enable the ability to turn off address stepping
Control <13>	Enable 64 bit transactions

Download1 Register (0x10)

Bit	Meaning
Dwnld1<0>	DIN/INIT pins LCA0
Dwnld1<1>	DIN/INIT pins LCA1
Dwnld1<2>	DIN/INIT pins LCA2
Dwnld1<3>	DIN/INIT pins LCA3
Dwnld1<4>	Cclk Busy
Dwnld1<6>	DONE pins for LCAs
Dwnld1<8>	Ring[0] data
Dwnld1<9>	Ring[1] data
Dwnld1<13>	Enable 64 bit transactions

Download2 Register (0x18)

Bit	Meaning
Dwnld2<0>	Data for LCA0
Dwnld2<1>	Data for LCA1
Dwnld2<2>	Data for LCA2
Dwnld2<3>	Data for LCA3
Dwnld2<8>	Data for LCA0
Dwnld2<9>	Data for LCA1
Dwnld2<10>	Data for LCA2
Dwnld2<11>	Data for LCA3

Clock Register (0x20)

Bit	Meaning
Clock <0>	Clock stepping bit 0
Clock <1>	Clock stepping bit 1
Clock <5>	User Clock busy
Clock <6>	PLL lock
Clock <8>	Ring[0] interrupt
Clock <9>	Ring[1] interrupt
Clock <10>	DMA done interrupt
Clock <11>	link<16> interrupt
Clock <14>	PLL not locked interrupt
Clock <15>	Partial word write interrupt

VCO Register (0x28)

Bit	Meaning
VCO<0>	VCO system clock
VCO<1>	VCO system data
VCO<2>	PLL bypass
VCO<3>	User Clock source
VCO<4>	Enable global interrupt
VCO<5>	Global interrupt active
VCO<8>	Enable Ring[0] interrupt
VCO<9>	Enable Ring[1] interrupt
VCO<10>	Enable DMA done interrupt
VCO<11>	Enable link<16> interrupt
VCO<14>	Enable PLL not locked interrupt
VCO<15>	Enable Partial word write interrupt
VCO<16>	User Clock with stop ring[0]
VCO<17>	User Clock with stop ring[0]
VCO<18>	User Clock with stop init[0]
VCO<19>	User Clock with stop init[0]

Decode Register

Bit	Meaning
Decode<0>	PIF mode bit[0]
Decode<1>	PIF mode bit[1]
Decode<8>	Enable ring[0]
Decode<9>	Enable ring[1]
Decode<10>	Target delay bit
Decode<11>	Target delay bit
Decode<12>	Target delay bit
Decode<13>	Master delay bit
Decode<14>	Master delay bit
Decode<15>	Partial word write interrupt Active

Link Register (from user area)

Bit	Meaning
Link<0>	Address bit must be set to 1
Link<1>	Address bit must be set to 1
Link<15>	DMA engine is active
Link<16>	Raise interrupt

Glossary

Bit stream

A collection of bits that is used to program the configuration of the Xilinx FPGAs.

bus

A collection of many transmission lines or wires. The bus interconnects computer system components, providing a communications path for addresses, data, and control information or external terminals and systems in a communications network.

CD-ROM

Compact disc read-only memory. The optical removable media used in a compact disc reader.

CLB

Configurable logic block. The internal build block of the Xilinx FPGA.

Clkext

An external clock that can be used to control the board. The clock can be brought in through pin 1 of the mezzanine card connector JN4.

ClkSys

The default clock that is used to control the board. It is a reconstructed PCI clock that can run at single or double the PCI clock speed.

Clkusr

A programmable PLL that can be used as a secondary clock source. The Clkusr does not have definitive phase relation to ClkSys.

CMC

Common mezzanine card. The IEEE 1386 standard defines the common mezzanine card in terms of physical properties.

Deconfiguration

The ability to deactivate a current configuration of an FPGA. The bit stream must be reloaded to recover the configuration.

DRAM

Dynamic random-access memory.

Ebus

The bus that connects the PCI interface chip to usrlca0 and usrlca1.

Ebusstate

The three state bits which define the current state of Ebus during transactions mode. he state bits are used for flow control and the avoidance of contention on Ebus.

EEPROM

Electrically Erasable Programmable Read Only Memory. The EEPROM is used to store a version of PCI interface. There is also an SROM that stores PCI interface design, which cannot be erased.

FPGA

Field Programmable Gate Array. The FPGA is the foundation of the PCI development platform. Each FPGA contains a number of CLBs that interconnected by routing channels. The configuration of the CLBs and the interconnection of the CLBs is programmable.

IOB

Input Output Buffer. The IOB is the driver that is located on the pad within the FPGA. Typically, the IOB can be programmed as a TTL driver, CMOS driver, or Tri-stateThis allows the FPGA to function properly in multiple environments.

LCA

Logic Cell Array. The LCA is the collection of CLBs with in the Xilinx 4000 series FPGAs.

LUT

Look up table. Each CLB contains three LUTs for function generation. Two of the LUTs have four inputs and the third has just three inputs. All of the LUTs have a single output.

NOP

No operation.

PCI

Peripheral component interconnect. An industry-standard expansion I/O bus that is the preferred bus for high-performance I/O options. PCI is available in a 32-bit and 64-bit version.

PIF

PCI interface chip. The PIF is a proprietary PCI design programmed in a Xilinx 4010E FPGA. The PIF has three modes of operation: transaction, promiscuous, and static.

PMC

PCI mezzanine card. The PMC is the same mechanically as the CMC. The difference is the PMC has a PCI interface device on it and negotiates according to the PCI standard. See IEEE Draft Standard 1386.1

Promiscuous Mode

A mode of the PIF in which the PCI development platform constantly snoops the PCI bus. The PIF runs at twice the speed of the PCI bus. On half of a PCI clock period, the PIF registers the PCI control signals and on the other half of the period, the PIF reads the data or address present on the PCI bus.

Reconfiguration

Changing the configuration of the programmable device. The FPGAs on the PCI development platform are infinitely reconfigurable. Some programmable devices can only be configured once, an SROM, and therefore, are not reconfigurable.

Rings

A two bit wide bus that is connected to all for User Area FPGAs. The Ring bits can be used to generate interrupts.

SBUS

South bus. There are two separate south buses on the PCI development platform. One SBUS connects Usrlca 0 to Usrlca 1 and the other connects Usrlca 2 to Usrlca 3.

Side Effect on Read

A side effect on read is when the act of reading a register changes the state of the register.

SMP

Symmetric multiprocessing.

SRAM

Static random-access memory.

Static Mode

A mode of the PIF in which the PIF has statically defined data paths between the PIF and Usrlca 0 and Usrlca 1. Ebus is split into two 16-bit wide paths one for reading data and the other for writing data.

Transaction Mode

The mode of the PIF that allows 32-bit or 64-bit data to be passed in either direction between the user area and the PIF. In this mode the PIF has interrupts and DMA capabilities. This mode models the PCI target and initiator. Ebusstate identifies the current state of ongoing transactions.

User Area

The four FPGAs that are logically placed between the PCI interface chip and the CMC connectors.

UsrLCA

User logic cell array. This is another term used to refer to one of the FPGAs in the user area.

WBUS

West bus. There are two separate west buses on the PCI development platform. One WBUS connects Usrlca 0 to Usrlca 2 and the other connects Usrlca 1 to Usrlca 3.