PEV1100 User's Manual

	Product Name	PCI Express to VME64x Interface
	Product Identifier	PEV1100
	Author	Jean-Francois Gilot
	Company	IOxOS Technologies
	Created	Apr 19, 2008
	Version	3.0
	Last modified	Apr 19, 2012
	Review	
	Approval	
	Distribution	Public

		Name	Date	Status
	Review			
	Approval			

PEV1100

PCI Express to VME64x Interface

User Manual

Reference : PEV1100_UM_081801
Version : 3.0
Date : Apr 19, 2012
Web site : www.ioxos.ch
Technical Support : support@ioxo.ch
Phone : +41 22 364 76 90
Fax : +41 22 364 76 90

Copyright Information

Disclaimer

IOxOS Technologies provides this documentation without warranty, term or condition of any kind, either express or implied, including, but not limited to, express and implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

The information in this document has been carefully checked and is believed to be entirely reliable. While all reasonable efforts to ensure accuracy have been taken in the preparation of this manual, IOxOS Technologies assumes no responsibility resulting from omissions or errors in this manual, or from the use of information contained herein.

In no event will IOxOS Technologies be liable for damages arising directly or indirectly from any use of or reliance upon the information contained in this document. IOxOS Technologies may make improvements or changes in the product(s) and/or the program(s) described in this documentation at any time.

IOxOS Technologies retains the right to make changes to this product at any time, without notice. Products may have minor variations to this publication, known as errata. IOxOS Technologies assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of IOxOS Technologies products.

Trademarks

IOxOS Technologies and IOxOS logo are registered trade marks of IOxOS Technologies SA

All other trademarks are the property of their respective owners

Warnings

Static electricity can damage integrated circuit components and cards. Make sure you use proper ESD handling procedures (refer to EIA-625, ESD Association Handbook or MIL-HDBK-263) while working with cards and components.

Warranty

For warranties and repair policies, refer to IOxOS Technologies General Conditions of Sale

Record History		
Version	**Date**	**Description of Changes**
0.01	May 1, 2008	File creation
0.99	11/27/08	Preliminary release
1.00	06/01/09	First release
2.00	08/20/09	Support for multicrate configurations and real time linux (Xenonai)
3.00	01/12/10	Short PCI IO space and DMA list mode

Table of Contents

Illustration Index

Index of Tables

1 Introduction

1.1 Document Identification

The PEV1100_UM_081801 is the user's manual of the PEV1100 PCI Express to VME64x Interface.

1.2 Intended Audience

This manual is intended for use by system designers, software developers and support personnel.

It is recommended that the reader has a reasonable background in PC architectures, including experience or knowledge of I/O buses and related protocols.

1.3 Companion Documents

PEV1100 Reference Manual

1.4 References

Reference	Title	Version	Date	Organization
	PCI Local Bus Specification	3.0	Aug 12,2002	PCI-SIG
	PCIe Base Specification	2.0	Jan 4, 2007	PCI-SIG
	PCIe External Cabling Specification	1.0	Dec 20,2006	PCI-SIG
VITA 1-1994 (R2002)	VME64	1	1994	VITA
VITA 1.1-1997(R2005)	VME64 Extensions	1.1	1997	VITA
	VME64 2eSST			VITA
	PMC			
	XMC			
	ExpressLane PEX8112-AA PCI Express to PCI Bridge	1.1	Nov 2007	PLX
	ExpressLane PEX8624-AA 24-lanes/6-ports PCI Express Gen 2 Switch	0.9	Jan 2008	PLX

Table 1: References

1.5 Document Organization

This manual is composed of the following sections

1. Introduction
2. PEV1100 Capabilities
3. Getting Started
4. Software references

Section 2 gives an overview of the PEV1100 architecture and describe the features it implements

Section 3 is a step by step procedure to install and run the PEV1100

Section 4 contains a description of the software delivered with the PEV1100.

1.6 Conventions and Notations

Examples, code references and screen copy are inserted in frames an use fixed fonts.

Bold typefaces are used to indicate characters to be typed in.

1.7 Acronyms and Abbreviations

Term	Definition
DMA	Direct Memory Access
IEEE	Institute of Electrical & Electronics Engineers
IO	Input/Output
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	PCI Express
VITA	VMEBUS International Trade Association
VME	Versa Module Eurocard (IEEE1014)

Table 2: Acronyms and Abbreviations

2 PEV1100 Capabilities

2.1 Overview

2.1.1 PEV1100 system components

The IOxOS Technologies PEV1100 has been designed to allow a host computer to access in transparent way IO resources hosted in a VME crate by extending the host PCI Express Bus over medium distances (up to 7 meters). The PEV1100 provides a low latency high throughput data path between the host and any VME chassis

The host computer, referred to as local host can be any PC, workstation or server equipped with at least one PCI express x4 connector. The extension kit consists of 3 elements as show in Figure 1.

- a PCI Express passive adapter card [ref: PEC3100]
- a PCI Express extension cable [ref: PEC1100]
- a VME 6U interface board [ref: PEV1100]

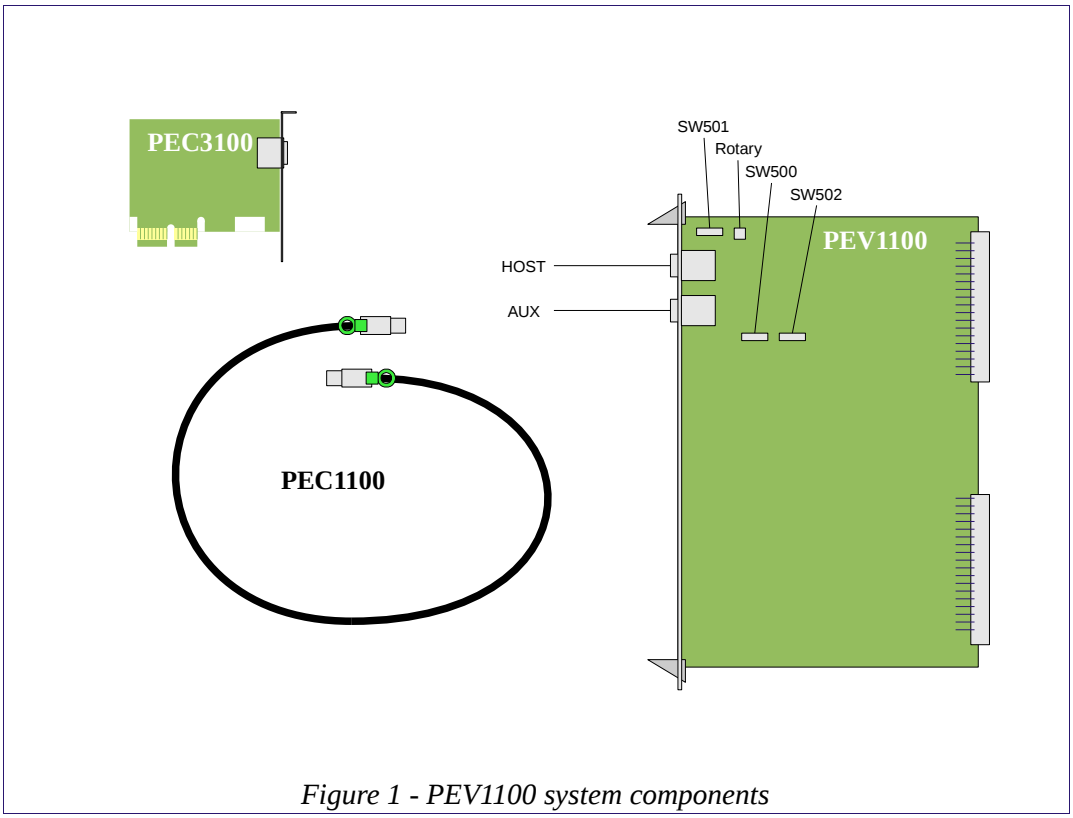


Figure 1 - PEV1100 system components

The PCI Express adapter card installs in the local host in a PCI Express slot supporting x4 operation. One end of the extension cable plugs into the adapter card.

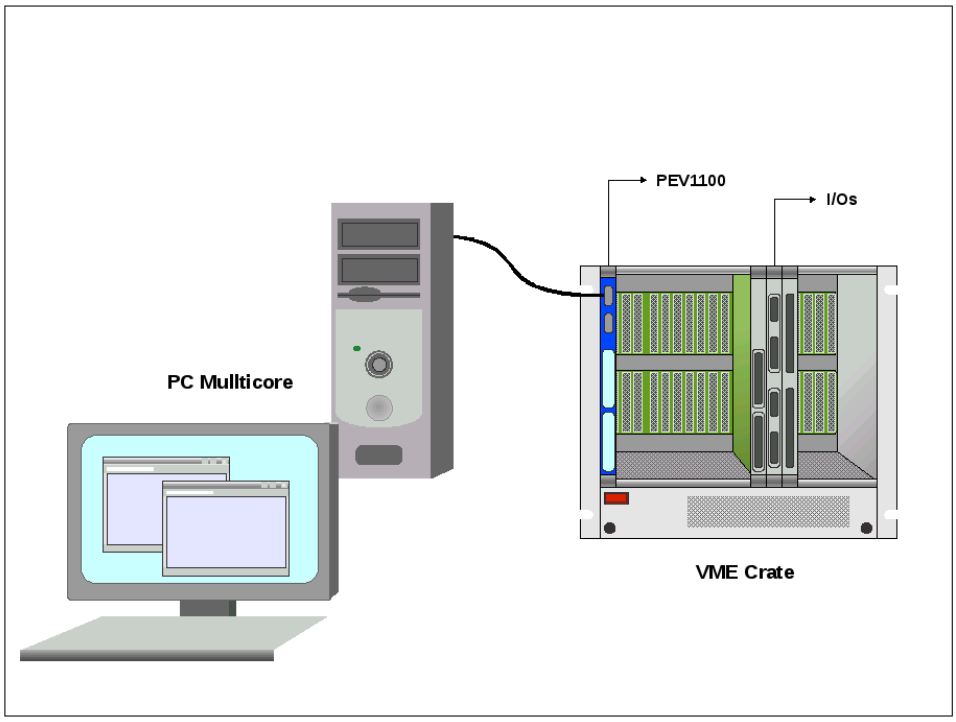
The PEV1100 6U VME boards installs in a VME chassis. If you intend to use it as VME system controller, set static options to enable slot 1 features. The other end of the extension cable plugs into the upper connector (HOST) of the PEV1100 VME board. The other connector (AUX) allows to chain another VME chassis (multi-crate configuration) or the connect another host computer referred as system host.

2.1.2 Typical application

Error: Reference source not found shows a PEV1100 typical application. The key components are:

- the PEV1100 6U VME interface
- a VME crate hosting IO boards
- a local host to control the IO boards

That configuration will be used along this manual to illustrate the PEV1100 features.



The **Local Host** is a high performance multi-core 1U server hosting two PCI express slots. The adapter card is place in one of the PCIe slot and a PCI Express external cable connects to the PEV1100 upper connector. It is dedicated to the real time part of the data acquisition system and run Linux with the Xenomai real time extensions

The **PEV1100** interface is installed in slot 1 of a VME crate. It is configured as VME system controller

2.2 PEV1100 Functional Diagram

Figure 2 shows a functional diagram of the PEV1100. The central element is a 6 ports PEX8624 PCI Express switch connecting all PEV1100 system elements through PCI Express links:

- the local host
- the system host or another PEV1100
- a PEX8112 PCIe/PCI bridge controlling two PMC slots
- two XMC slots
- an FPGA implementing
    - ✗ a PCI Express End Point
    - ✗ a VME64x master/slave interface
    - ✗ a Shared Memory controller with integrated DMA engines
    - ✗ user area

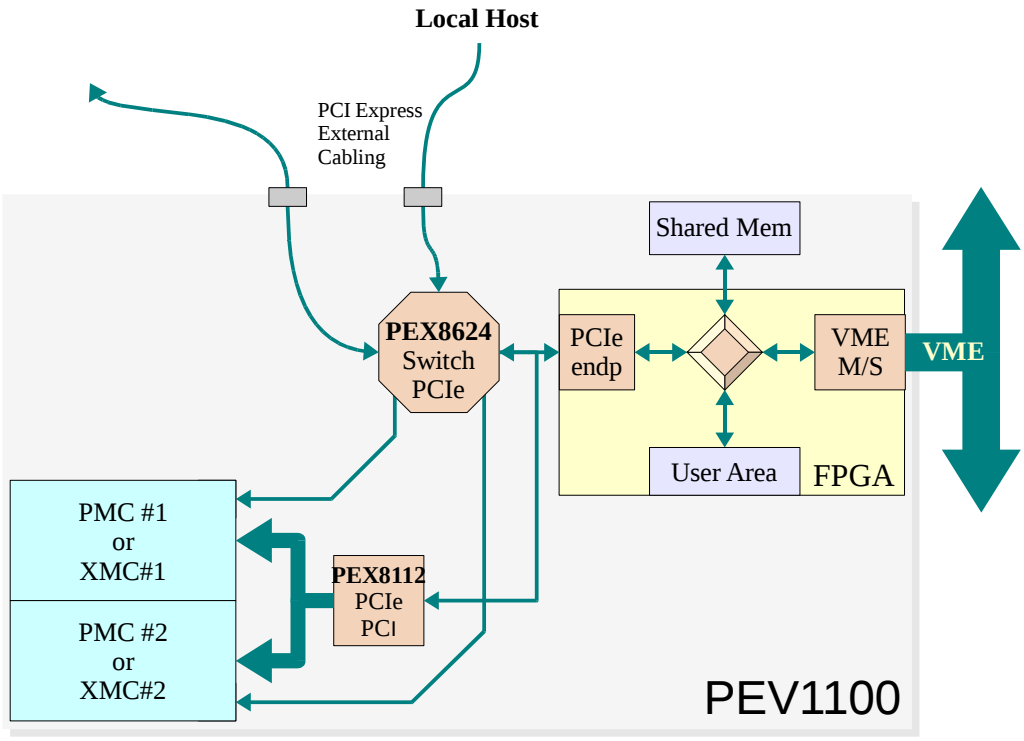


Figure 2 - PEV1100 Functional Diagram

2.3 PEX8624 PCI Express Switch

From a functional point of view, the PEX 8624 switch can be thought of as a hierarchy of PCI-to-PCI bridges (see Figure 3), with one upstream PCI-to-PCI bridge (UP#0) and up to five downstream PCI-to-PCI bridges (DP#i[1]) connected by an internal virtual PCI bus. Each bridges has a set of configuration registers compliant with the PCI and PCI Express system models.

Registers of the upstream PCI-PCI bridge are accessible by Type 0 Configuration requests targeting the upstream bus interface The local host shall connect one of its PCI express slot to that interface using the adapter card and a PCI Express external cable plugged in the PEV1100 upper connector.

The local host shall then use conventional PCI software and procedures to set up and identify all ports and links connected through the PEX8624. This is usually done automatically at boot time by the PCI enumeration process.

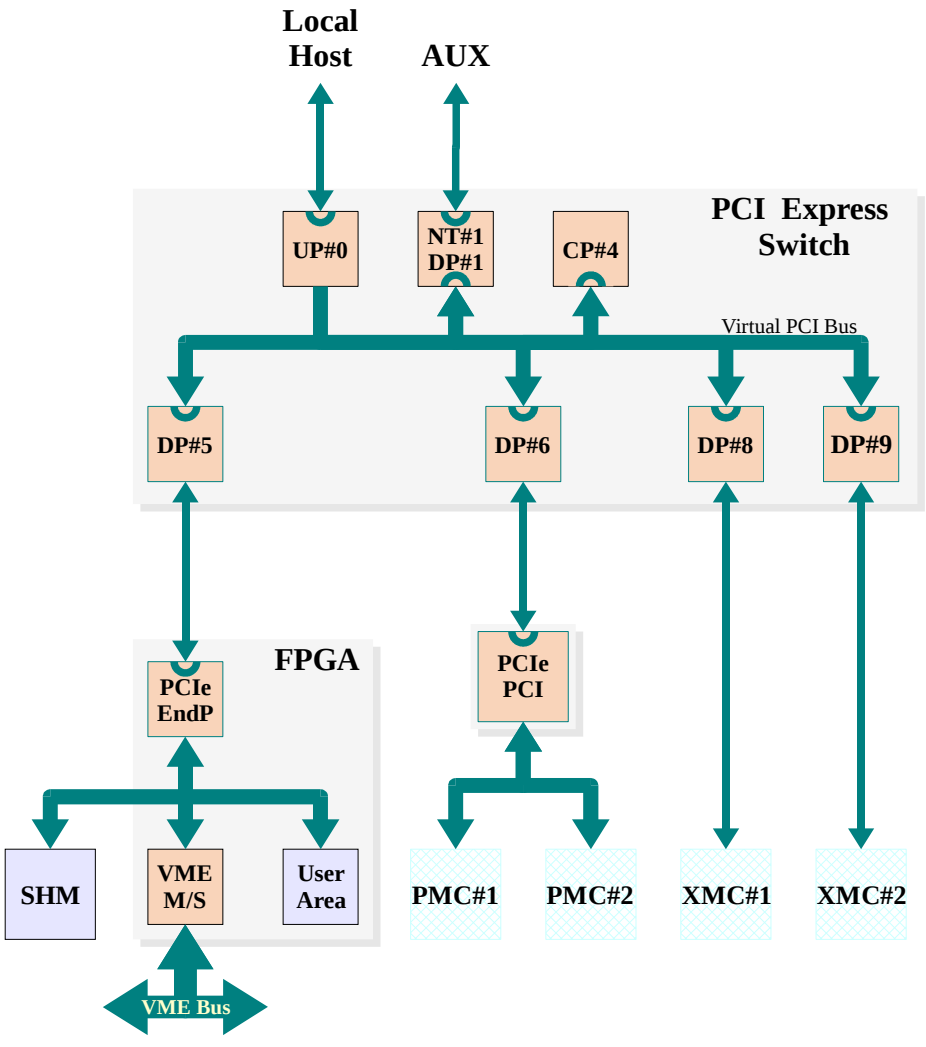


Figure 3: PEV1100 PCI Tree

After the PEX 8624 and its links are set up, data can be routed through the PEX 8624, from one Port to

1 port #1 can be either downstream (DP#1) for multicrate configuration or non transparent (NT#1) for connection with a system host

another. Responses and other communication are returned, by way of the same links, to the initiator. The PEX 8624 is transparent to these data transfers.

2.4 FPGA Port

The FPGA implements a real time non blocking switch supporting up to four simultaneous transactions among its communication ports:

- PCI Express End Point
- User Area
- VME64x master/slave interface
- DMA and Shared Memory

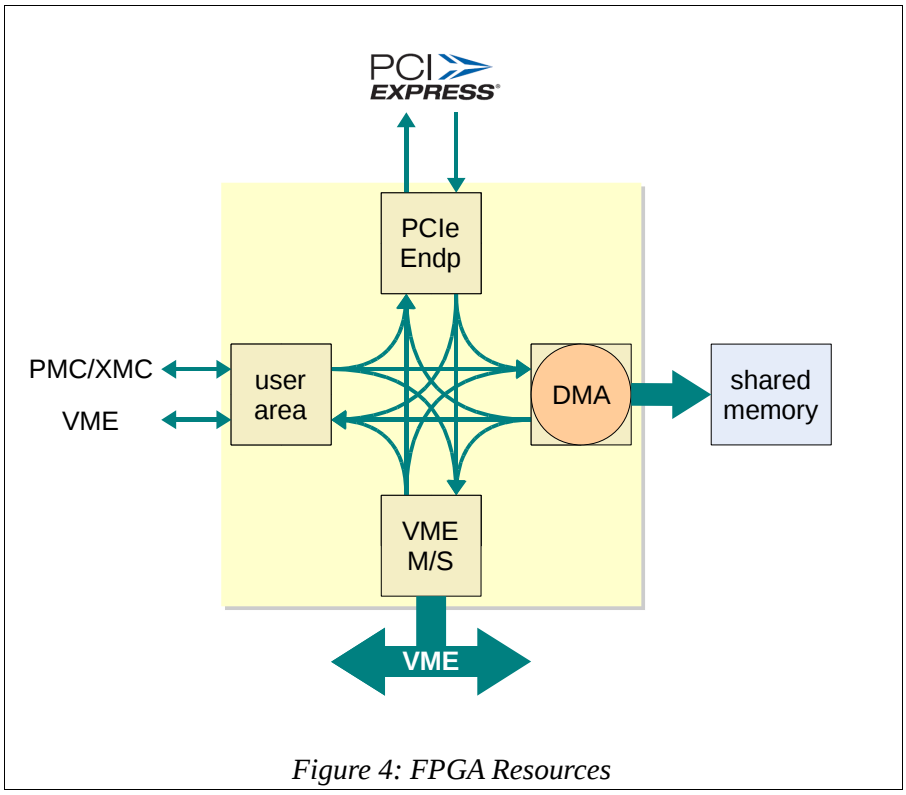


Figure 4: FPGA Resources

2.4.1 PCI Express End Point

The FPGA PCIe End Point is connected to port#5 of the PEX8624 PCI Express switch. It is the Local Host access point to :

- Configure VME64x interface
- Perform VME cycles
- Access FPGA user area
- Read/write the Shared Memory
- Drive the DMA controller

2.4.1.1 PCI Configuration Space

The PCIe End Point implement a PCI 2.3 Compatible Type 0 Header. It supports legacy and PCI Express extended capabilities.

Register Name				Offset
byte 3	byte 2	byte 1	byte 0	
Device ID		Vendor ID		0x00
Status		Command		0x04
Class Code			Revision ID	0x08
BIST	Header Type	Latency	Cache Line Size	0x0c
Base Address of PMEM window (64/32 bits, up to 4 GBytes)				0x10
				0x14
Base Address of MEM window (32 bits, up to512 MBytes)				0x18
0x00000000				0x1c
Base Address of IO window (32 bits, 4 KBytes)				0x20
0x00000000				0x24
Card Bus CIS Pointer				0x28
Subsystem ID		Subsystem Vendor ID		0x2c
Expansion ROM Base Address				0x30
Reserved			Capabilities Ptr	0x34
Reserved				0x38
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	0x3c

Table 3: PCIe-VME64x Configuration Header

The PEV1100 Device ID is 0x1100 and the IOxOS Technologies Vendor ID is 0x7357.

BAR0 and BAR1 hold the 64 bit PCI base address of the window allowing to map the PEV1100 resources (VME bus, shared memory, user space) in prefetchable mode.

BAR2 holds the 32 bit PCI base address of the window allowing to map the PEV1100 resources (VME bus, shared memory, user space) in non-prefetchable mode..

BAR4 holds the 32 bit PCI base address of the window allowing to map the FPGA local registers in the PCI IO space. The size of that window is 4 Kbytes.

2.4.1.2 PCI IO Space

Table 4 List the registers referred in this manual and visible through the PCI I/O window of the FPGA End Point. A full description of each register implemented in the FPGA is given in the PEV1100 Hardware Reference (see §1.3). The address offset of the register depend on the size of the PCI IO space (see §3.1.2.1.2).

Register	Offset		Description	Hardware Reference
	Legacy	short		
ILOC_STATIC	0x000	0x00	Static Option (switches) setting	§3.2.1
ILOC_PONFSM	0x00c	0xcc	Status of power-on sequence instruction execution	§3.2.4
ILOC_SPI	0x010	0xd0	SFLASH control and status	§3.2.5
ILOC_PCIE_SW	0x014	0xd4	PEX8624 status information	§3.2.6
ILOC_SIGN	0x018	0x04	FPGA bitstream signature	§3.2.7
ILOC_GENCTL	0x01c	0x08	PEV1100 general status	§3.2.8
PCIE_MMUADD	0x020	0x0c	PCI Express Ingress MMU address pointer	§3.2.9

PCIE_MMUDAT	0x024	0x10	PCI Express Ingress MMU data	§3.2.10
PCIE_EP_STA	0x028	0xd8	PCI Express End Point status	
PCIE_ITC_IACK	0x080	0x80	Interrupt Acknowledge register	
PVME_Slot_1	0x400	0x20	VME Slot 1 Control and status	§3.3.1
PVME_MASCSR	0x404	0x24	VME Master Interface Control and status	§3.3.2
PVME_SLVCSR	0x408	0x28	VME Slave Interface Control and status	§3.3.3
PVME_INTG	0x40c	0x2c	VME Interrupt Generator Control and status	§3.3.4
PVME_MMUADD	0x410	0x30	VME Ingress MMU address pointer	§3.3.5
PVME_MMUDAT	0x414	0x34	VME Ingress MMU data	§3.3.6
PVME_ADDERR	0x418	0xe0	VME Address Error	
PVME_STAERR	0x41c	0xe4	VME Status Error	

Table 4: PEV1100 Control and Status Registers

2.4.1.3 PCI Memory Space

Two decoding windows are provided in the PCI Memory space in order to map the resources controlled by the FPGA:

- VME bus
- Shared Memory
- FPGA User Area

The first window, controlled by BAR0 and BAR1, is prefetchable and can be set to decode either 32 bit or 64 bit addresses. Its size depends on the position of switches SW501-7:5 (see Table 12)and can be adjusted from 128 MBytes up to 4 Gbytes. If 32 bit address decoding is selected (SW501-8 off), BAR1 is wired to 0x00000000.

The second window, controlled by BAR2 , is non-prefetchable and decode 32 bit addresses. Its size depends on the position of switches SW501-4:3 (see Table 11)and can be adjusted from 64 to 512 Mbytes

For each window, access to the resources is controlled by an address translation table allowing to map a local PCI address to a remote address in the resource address space. The translation table holds an identifier of the targeted resource, the remote address upper bits (A32:20) and additional information about the access mode to be used (i.e. the VME address modifier, or the byte swapping policy). The remote address lower bits (A19:0) are directly propagated from the PCI local address (see Figure 5).

Register **PCIE_MMUADD** and **PCIE_MMUDAT** shall be used to program the address translation tables.

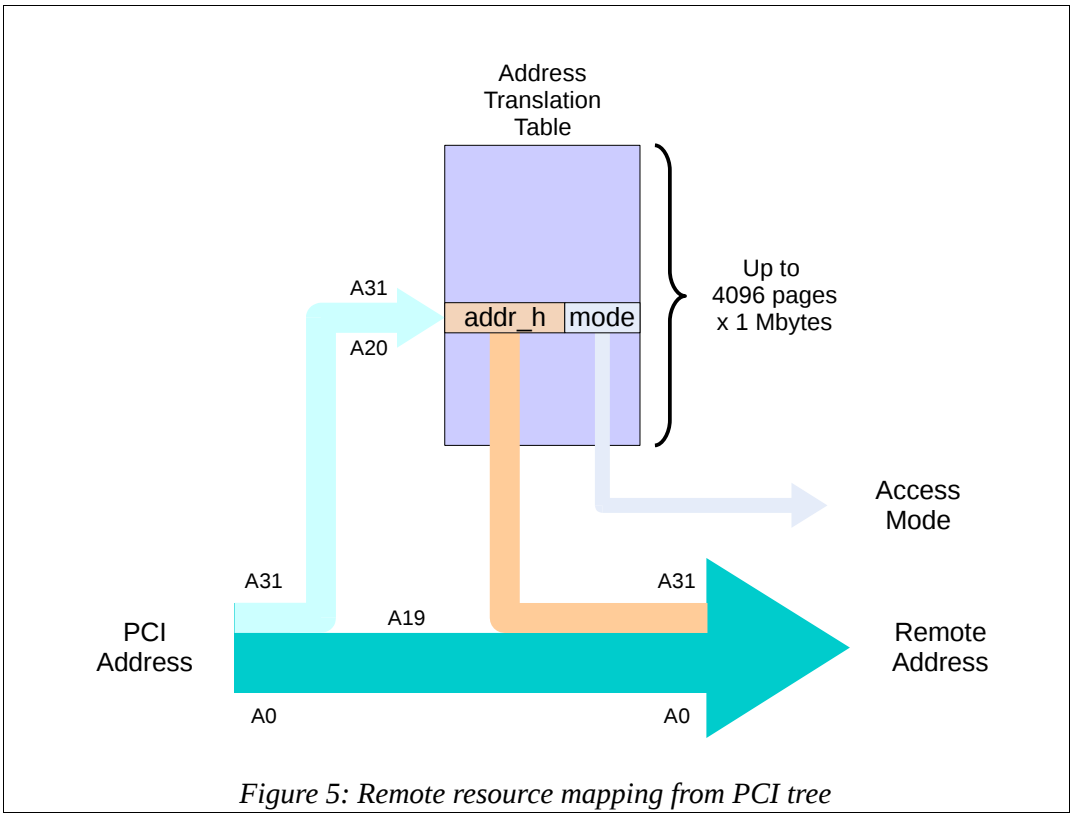


Figure 5: Remote resource mapping from PCI tree

2.4.2 VME64x Interface

2.4.2.1 Static Configuration

The VME64x interface is configured according to static options controlled by the switch SW501 and through a set of registers mapped in the PCI IO space from offset 0x400.

Static options allows the user to decide to work in VME64x mode or in legacy mode to support old backplanes.

2.4.2.2 System Controller

The PEV1100 has been designed to act as VME system controller. That capability is controlled by the **PVME_Slot_1** register accessible at offset 0x400 in the PCI I/O space of the PCIe to VME bridge.

This register allows to set the bus arbitration mode and bus timeout.

2.4.2.3 Master Interface

The PVME_MASCSR register allow to enable the VME mater interface and to set its operating mode:

- VME Request Mode
  - Release When Done
  - Release On Request
  - FAIR
  - No Release
- VME Request Level

A dedicated control bit in the PVME_MASCSR allows not to report over the PCIe tree the VME read cycles ended with bus error. In this case the occurrence of bus error can be linked to the generation of an interrupt or

can be checked in the PVME_STAERR register. The PVME_ADDERR register report the VME address of the cycle having generated the bus error.

VME address modifiers are generated by setting the proper access mode in the address translation page used to perform the VME access from the PCI tree.

2.4.2.4 Slave Interface

2.4.2.4.1 The CR/CSR space

The PEV1100 VME slave interface provide a 512 Kbytes window in the A24 address space compliant with the VME64x specification. If legacy mode is selected, switch SW501-8:4 allow to set the base address of that window.

In addition to the A24 decoding space, the slave interface offers in the A32 space an address mapping to the PCI Express tree, the shared memory or the FPGA user area.

2.4.2.4.2 PEV1100 Internal Resources

Decoding Window Base Address

The PEV1100 implement the Address Space Relocation mechanism in order to allow the user to set dynamically the base address of the decoding window.

The Function 0 ADER register in the CSR space (see VITA 1.1-1997(R2005) §10.2.2.2) is used for that purpose. That 32 bit register is seen as 4 bytes located from offset 0x7FF63 to 0x7FF6F (using D08(O) cycles). The user sets the the window base address by writing the compare bits.

The Dynamic Function Size Read mechanism shall be used to get the actual size of the address decoding window in order to know which of the compare bits are active. The DFS bit in the Function 0 Address Decoder Mask (located at offset 0x623 in the CR space) is set to indicate that the PEV1100 uses this mechanism.

When DFSR bit is set in Function 0 ADER, the PEV1100 latches its address mask bits in the register compare bits. By reading back the register, the user knows the window size. It is its responsibility to restore the register after that operation.

The Scatter/Gather Mechanism

The decoding window is divided in pages of 1 MBytes in size. The total number of pages is equal to the window size divide by the page size. For each page the user must define to which location it points by initializing a scatter/gather memory with the destination address and space. The **PVME_MMUADD** (offset 0x410) and **PVME_MMUDAT** (offset 0x414) shall be used to initialize the scatter/gather memory.

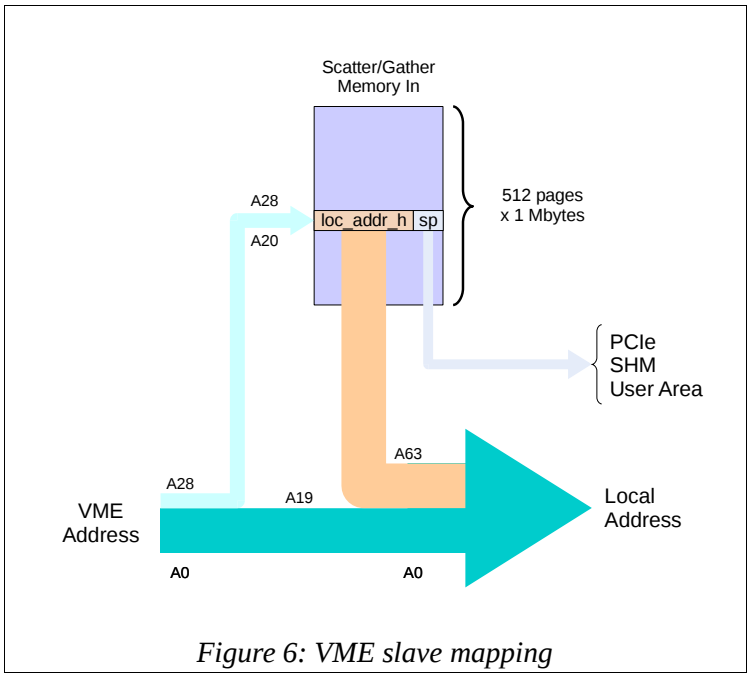


Figure 6: VME slave mapping

The vme base address by which one access a page is equal to the window base address plus the page number multiplied by the page size (VME address bits A20 to A28 are used to point in the scatter/gather memory)

2.4.3 Interrupt Handling

The following events can be programmed to generate an interrupt:

- VME IRQ1:7
- VME Error
- VME SYSFAIL
- VME ACFAIL
- End of DMA transfer(4 channels)
- occurrence of the local timer main tick

Each of these sources can be individually masked/unmasked. The interrupt handling logic scan permanently all unmasked interrupt sources.

When one of these events occurs, a Message Signaled Interrupt (MSI) is sent toward the local host. The MSI logic is initialized by the local host by configuring the MSI capability registers in the FPGA PCI end point (see PCIe Base Specification). These registers hold the data and destination address of the message to be generated. Upon reception of an MSI from the PEV1100, the host shall perform a read of the PCIE_ITC_IACK in order to identify the interrupt source.

It shall be noted that the interrupt scanning logic does not perform an automatic masking of the source having generated the interrupt and that it is the responsibility of the local host to mask it before restarting interrupt scanning by overwriting the PCIE_ITC_IACK register.

For VME IRQ interrupt, the interrupt scanning logic can be programmed to automatically generate the VME IACK cycle. In this case the PCIE_ITC_IACK contain the VME IRQ status information (ID and level).

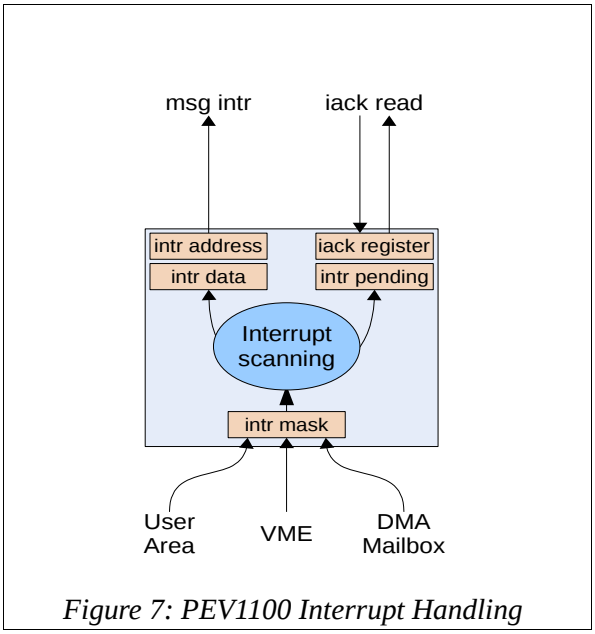


Figure 7: PEV1100 Interrupt Handling

2.4.4 Shared Memory

The shared memory can be directly mapped both in the PCI Express tree and on the VME bus.

From the PCI express, the shared memory can be made visible in the MEM and PMEM address windows using the corresponding ingress MMU lookup tables.

2.4.5 DMA Controller

The DMA controller allows to move data between the shared memory and either the VME bus, PCI Express tree or FPGA user area. The controller has 4 channels, two to import data in the shared memory and two to export data from the shared memory. The 4 channels can transfer data concurrently. In and out channels can be linked to move data autonomously between VME bus, PCI Express tree or FPGA user area.

2.4.6 User Area

The FPGA provides a User Area dedicated to customer specific applications. The User Area is connected to the PMC 1386.1 slots (Jn14 / Jn24) by two 32-bit direct links and has a low latency private data path to the Shared Memory.

The User Area is the perfect place to implement proprietary protocols and specific features, allowing the PEV1100 to really fit the needs of your system. IOxOS Technologies offers engineering and consulting services together with a comprehensive HDL based tool kit, .to ease the User Area application development.

2.5 PCI Express to PCI Bridge

Port#6 of the PEX8624 switch is connected to a PEX8112 PCI Express to PCI bridge controlling two IEEE1386.1 PMC slots. The PMC PCI bus is 32 bid wide and runs at 33/66 MHz.

Both PMC slots provide 32-bit User IO (Jn14 / Jn24) directly wired to the VME64x P2 connector, as defined by VITA 35.

2.6 XMC Slots

Port#8 and port#9 of the PEX8624 switch are connected to the XMC slots

3 Getting Started

3.1 System Set Up

3.1.1 Host Adapter Card

3.1.1.1 Installation

Insert the adapter card in one of the free PCI Express connector of your host (a x4 or higher is needed).

Secure the card front panel to the chassis frame using a screw or hold down mechanism. Beware the PCIe cable will pull on the card front panel.

3.1.2 PEV1100 VME64x Interface

3.1.2.1 Static Option

Before inserting the PEV1100 board in the VME crate, the static options shall be set. One set of jumpers (see 3.1.2.1.1)and three DIP switches (see 3.1.2.1.2) are implemented on the PEV1100 for that purpose. These options allow you to decide the following:

PEV1100 node identifier

Node identifier associated to the PEV1100 (see 3.1.2.1.3)

VME interface configuration

Do you work in VME64x backplane ?
- If yes
  - Enable VME64x (SW500-3 -> on)
- If not
  - Disable VME64x (SW500-3 -> off)
  - Are you going to put the board in slot 1 (system controller) ?
    - If yes
      - Enable VME Slot 1 (SW500-2 -> on)
    - if not
      - Disable VME Slot 1 (SW500-2 -> off)
  - you have to set the VME A24 slave base address (SW500-4:8)

Do you want to propagate VME SYSRST locally
- If yes
  - Enable VME SYSRST propagation (SW500-1 -> on)
- If not
  - Disable VME SYSRST propagation(SW500-1 -> off)

PCI Express switch configuration

You have first to decide from which port you are going to configure the PEX8624 switch. This port shall be configured as upstream port. Switches SW502-8:6 shall be used for that purpose.

If you connect to a local host through the upper connector (standard configuration), port#0 shall be the upstream port, and in this case you have to choose to use either the PEV1100 local clock or the clock signal coming from the host over the PCIe cable.

PCI Express End Point Configuration

You have to decide the size and addressing mode of the PCI PMEM and MEM address space decoded by the PEV1100. Switches SW501-8:3 shall be used for that purpose.

3.1.2.1.1 Jumpers

3.1.2.1.2 Switches

Three 8 positions micro DIP switches (see Table 5) are used to define static options. Their location on the board is showed in Figure 1

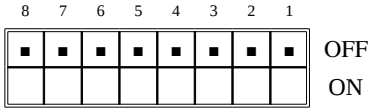


Table 5: Micro Switches Layout

The Micro switch labeled SW500 located versus the center of the PEV1100 board on the left side controls the setting off some VME options

Micro Switch SW500			
Position	**Factory**	**Name**	**Description**
1	***on***	**VME_RST_ENA**	allow VME SYSRESET to perform PEV1100 local reset [***on*** → enabled]
2	***off***	**VME_S1_ENA**	allow to enable VME system controller functionality [***on*** → enabled]
3	***off***	**VME64x_ENA**	allow to enable VME64x mode of operation [***on*** → enabled]
4	***off***	**VME_A24_19**	bit 19 of VME slave base address in A24 mode [***on*** → 1]
5	***off***	**VME_A24_20**	bit 20 of VME slave base address in A24 mode [***on*** → 1]
6	***off***	**VME_A24_21**	bit 21 of VME slave base address in A24 mode [***on*** → 1]
7	***off***	**VME_A24_22**	bit 22 of VME slave base address in A24 mode [***on*** → 1]
8	***off***	**VME_A24_23**	bit 23 of VME slave base address in A24 mode [***on*** → 1]

Table 6: VME Static Options

When **VME64x_ENA** is in the ***on*** position, the VME interface works according to the VME64x specification. In this mode, the base address of the VME A24 slave window is set according to the geographical address of the board.

The PEV1100 has been designed to behave as VME system controller. Setting **VME_S1_ENA** in the ***on*** position enables that feature. This setting is ignored if **VME64x_ENA** is set in the ***on*** position. In this case the board geographical position decides if the PEV1100 is system controller.

If the **VME_RST_ENA** switch is in the ***on*** position, the VME SYSRESET backplane signal is propagated inside the PEV1100 to generate a local reset.

VME_A24_19 to **VME_A24_23** are used to define the VME base address of the A24 slave window in legacy mode (VME64x disabled). Putting a switch in the ***on*** position sets the corresponding bit to 1. This setting is ignored if **VME64x_ENA** is set in the ***on*** position.

The Micro switch labeled SW501 located in the left upper corner of the PEV1100 board controls the setting of options related to the configuration of the FPGA and the PEX8624 PCI Express switch.

Micro Switch SW501			
Position	**Factory**	**Name**	**Description**
1	***off***	**SFLASH_PROG**	Allow to reprogram SFLASH through P7 connector [***on*** → P7]
2	***off***	**FPGA_BST_0**	define the index (from 0 to 3) of the bit stream file to be loaded in FPGA at boot time. If the load procedure fails, index 0 is then used for reload
3	***off***	**FPGA_BST_1**	
4	***off***		reserved
5	***off***	**PCIE_P4_NT**	Allow to set port#1 (P4 connector) as no transparent [***on*** → NT]
6	***off***	**PCIE_CFG_0**	Allow to configure the PEX8624 PCI Express switch Upstream Port
7	***off***	**PCIE_CFG_1**	
8	***off***	**PCIE_CFG_2**	

Table 7: FPGA and PEX8624 Static Options

The **SFLASH_PROG** switch shall be used for SFLASH reconfiguration using a specific tool connected to the P7 connector. In normal operation it shall be set in off position.

The **FPGA_BST_1:0** switches allows to choose which section of the SFLASH device will be used to load the PEV1100 FPGA.

FPGA_BST		**Description**
_1	**_0**	
off	***off***	Select SFLASH section #0 (factory default)
off	***on***	Select SFLASH section #1
on	***off***	Select SFLASH section #2
on	***on***	Select SFLASH section #3

Table 8: SFLASH Booting Configuration

The **PCIE_P4_NT** switch allows to configure the PCIe port connected to the front panel auxillary connector (lower position) in non transparent mode in order to allow another host to be connected to the PEV1100.

The **PCIE_CFG_2:0** switches shall be used to configure the PEX8624 PCI Express switch according to the following table:

PCIE_CFG			**Description**
_2	**_1**	**_0**	
off	***off***	***off***	PCI Host connected to upper connector, PEV1100 uses PCIe clock
off	***off***	***on***	PCI Host connected to upper connector, PEV1100 uses a local clock
off	***on***	***off***	XMC#1 is PCI host
off	***on***	***on***	XMC#2 is PCI host
on	***x***	***x***	reserved

Table 9: PEX8624 Port Configuration

The Micro switch labeled SW502 located in at the center of the PEV1100 board on the right side controls the setting of options related to the configuration of the base address registers of the FPGA PCIe end point..

Micro Switch SW502			
Position	**Factory**	**Name**	**Description**
1	***off***	**FPGA_PROG**	Allow to load FPGA bitstream through P7 connector [***on*** → P7]
2	***off***	**IO_SIZE**	Select PCI_IO size [***on*** → IO space 256 bytes]
3	***off***	**MEM_SIZE_0**	Define the size (from 64 to 512 Mbytes) of the non prefetchable memory window in the FPGA PCIe end point (BAR#2)
4	***off***	**MEM_SIZE_1**	
5	***on***	**PMEM_SIZE_0**	Define the size (from 0 to 4 Gbytes) of the prefetchable memory window in the FPGA PCIe end point (BAR#0)
6	***off***	**PMEM_SIZE_1**	
7	***off***	**PMEM_SIZE_2**	
8	***off***	**PMEM_A64**	Sets the addressing mode of the prefetchable memory window [***on*** → A64]

Table 10: FPGA PCIe End Point Configuration

The **FPGA_PROG** switch shall be used to load dynamically the FPGA bitstream using a specific tool connected to the P7 connector. In normal operation it shall be set in off position.

The **IO_SIZE** switch controls the size of the PCI IO space. In the ***off*** position, the size is 4 kbytes (legacy). In the ***on*** position, the size is 256 bytes (compressed IO space compatible with PCI 3.0)

The **MEM_SIZE_0:1** switch controls the size of the address window open in the PCI MEM space of the FPGA PCIe End Point. The size of that window

MEM_SIZE		Window Decoding Size
_1	**_0**	
off	***off***	64 MBytes
off	***on***	128 MBytes
on	***off***	256 MBytes
on	***on***	512 MBytes

Table 11: FPGA PCI MEM Window Configuration

The **PMEM_SIZE_0:2** switch controls the size of the address window open in the PCI PMEM space of the FPGA PCIe End Point.

PMEM_SIZE			Description
_2	**_1**	**_0**	
off	***off***	***off***	PCI PMEM and MEM decoding disabled (BAR#0 and BAR#2)
off	***off***	***on***	PCI PMEM decoding disabled (BAR#0)
off	***on***	***off***	128 MBytes
off	***on***	***on***	256 MBytes
on	***off***	***off***	512 MBytes
on	***off***	***on***	1 GBytes
on	***on***	***off***	2 GBytes
on	***on***	***on***	4 GBytes

Table 12: FPGA PCI PMEM Window Configuration

The **PMEM_A64** switch controls the address decoding mode of the PCI PMEM window. If in ***off*** position, address decoding is 32 bit (BAR#1 is wired to 0). Otherwise. BAR#0 and BAR#1 are used for 64 bit address decoding.

3.1.2.1.3 Rotary Encoder

A 16 positions rotary encoder allows to assign a node number to the PEV1100. The setting of that encoder has no effect on the behavior of the PEV1100. Its current position is made available to the software through the GPIO lines of the PEX8624 PCI Express switch. It is used by the PEV1100 device driver to identify the board in multi-crate configuration. In single crate configuration it should be set to 0 to match a default value allowing for compatibility with older version of the PEV1100 device driver.

3.1.2.2 Installation

Just insert the PEV1100 VME board in a VME Crate. If you choose the first slot, don't forget to select the system controller (slot 1) static option. Set the rotary encoder in order to define a node number.

3.1.3 PCI Express External Cable

Connect the PCI Express cable between the host adapter card and the PEV1100 host connector (upper connector).

3.2 Power-On Sequence

After having connected all elements together, first power-on the VME crate and wait 10 seconds to allow the PEV1100 to be in a stable state.

Then power-on the local host. You should see front panel LEDs blinking while the host is connecting with the PEV1100 through the PCI Express.

3.2.1 PCI Enumeration

The PCI enumeration process is executed by the BIOS at boot time. It is then mandatory that the PEV1100 must be connected and powered on when the PC boots unless the PC root complex and the operating system supports PCI Express hotplug (see § 3.2.2).

During the PCI enumeration process the BIOS executed by the processor scan the PCI tree in order to discover the various buses that exist and all PCI devices that reside on each of them. Starting with device 0 the BIOS performs PCI configuration cycles in order to read the Vendor ID from function 0 in each of the 32 possible devices on bus 0. If a valid vendor ID is returned, this indicates that a device is implemented. The Header Type field in the Header register indicates if the device implements a bridge function.

When a PCI/PCI bridge is discovered, the BIOS configures its bus number registers and performs a depth-first search before proceeding to discover additional devices on bus 0. At the end of the enumeration process, each bus in the PCI tree has been allocated a bus number, and the configuration registers of all devices are accessible from the host.

The host shall then initialize the configuration headers for each function of each device. In particular, it shall allocate the base address registers (BAR) and initialize accordingly, the IO, MEM and PMEN address windows in the PCI/PCI bridges.

BAR registers allows the host to access the PCI device internal resources by opening an address range in the PCI addressing space. In order to accomplish this, the host detects how many address ranges and the size of each are needed in order to assign mutually-exclusive ranges. This process is not always successful, because:

- it depends on the total address range allocated by the system to the PCI tree
- addressing modes supported by the system (32bit, 64bit)
- on the number and size of the address ranges to be allocated to each function in each device discovered in the PCI tree

Additional constrains are that all address ranges behind a PCI/PCI bridge must fall in the same addressing window and function base addresses must be assigned to a multiple of their size.

To summarize, the PCI address mapping will depend on:

- the chipset implementing the root complex
- the BIOS
- the devices present in the PCI tree

When a BIOS is unable to map all resources discovered in the PCI tree, it can either discard the biggest address ranges or it can simply hang, leaving a blank screen.

In order to cope with the different chipset and BIOS, the PEV1100 implement a set of switches (see §3.1.2.1.2)allowing the user to define:

- the size of each addressing windows (from 64 MBytes to 4 GBytes)
- the supported addressing modes (32 bit or 64 bit)

If you are not sure about the characteristics of your system, set the micro DIP switches SW502 (see Table 13) in order to disable the PEV1100 prefetchable address window (BAR#0) and set the non prefetchable address window (BAR#2) to 64 MBytes. This configuration should always work and let you go through this installation procedure.

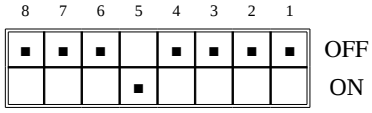


Table 13: SW502 Default Setting

3.2.2 Hot Plug Capabilities

PCI Express defines an hot plug and hot removal mechanism (see PCIe §6.7). A set of registers implemented in root and switch ports are defined for that purpose. If you want to benefit from hot plug capability you must choose a local host whose root complex implements that features. Usually laptops equipped with express card slots support hot plug. For servers and workstation you should check with the vendor if:

- the root complex support hot plug
- the slot in which you plan to install the adapter card implement the signals needed by the hot plug mechanism

PCI Express External cables and adaptor card implement the sideband signals needed to support hot plug.

The PEX8624 switch support hot plug on both upstream and downstream port.

If hot plug is supported by the local host hardware and software, you should then be able to power on and off the VME crate, or plug/unplug the cable without rebooting the local host.

However it shall be noted that the PCI Express hot plug mechanism is designed as a “no surprises”. In other words it doesn't permit to install or remove PCI Express card without first notifying the software.

IOxOS Technologies provides a deviceriver allowing to save/restore the PEV1100 configuration registers in order to implement manually hotplugging.

3.3 Linux Host

3.3.1 Checking for the PEV1100

3.3.1.1 Listing PCI devices

The ***lspci*** command (you must be superuser to execute that command) lists all PCI devices discovered during the enumeration process. The file “/usr/share/pci.ids” contains a list of all known PCI Ids.

```
linux-host:~>lspci
00:00.0 Memory controller: nVidia Corporation CK804 Memory Controller (rev a4)
00:01.0 ISA bridge: nVidia Corporation CK804 ISA Bridge (rev f1)
00:01.1 SMBus: nVidia Corporation CK804 SMBus (rev a2)
00:02.0 USB Controller: nVidia Corporation CK804 USB Controller (rev a2)
00:02.1 USB Controller: nVidia Corporation CK804 USB Controller (rev a4)
00:07.0 IDE interface: nVidia Corporation CK804 Serial ATA Controller (rev f3)
00:08.0 IDE interface: nVidia Corporation CK804 Serial ATA Controller (rev f3)
00:09.0 PCI bridge: nVidia Corporation CK804 PCI Bridge (rev f2)
00:0b.0 PCI bridge: nVidia Corporation CK804 PCIE Bridge (rev f3)
00:0c.0 PCI bridge: nVidia Corporation CK804 PCIE Bridge (rev f3)
00:0d.0 PCI bridge: nVidia Corporation CK804 PCIE Bridge (rev f3)
00:0e.0 PCI bridge: nVidia Corporation CK804 PCIE Bridge (rev a3)
00:18.0 Host bridge: Advanced Micro Devices [AMD] K8 [Athlon64/Opteron] HyperTransport
Technology Configuration
00:18.1 Host bridge: Advanced Micro Devices [AMD] K8 [Athlon64/Opteron] Address Map
00:18.2 Host bridge: Advanced Micro Devices [AMD] K8 [Athlon64/Opteron] DRAM Controller
00:18.3 Host bridge: Advanced Micro Devices [AMD] K8 [Athlon64/Opteron] Miscellaneous
Control
01:08.0 VGA compatible controller: ATI Technologies Inc ES1000 (rev 02)
02:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5722 Gigabit Ethernet PCI
Express
05:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:01.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:04.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:05.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:06.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:08.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:09.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
09:00.0 Bridge: IOxOS Technologies PEV1100 PCI Express VME Bridge (rev 01)
0a:00.0 PCI bridge: PLX Technology, Inc. PEX8112 x1 Lane PCI Express-to-PCI Bridge (rev
aa)
linux-host:~>
```

To obtain a “tree” view, just use ***lspci*** the “-***t***” option

```
linux-host:~>lspci -t
-[0000:00]-+-00.0
           +-01.0
           +-01.1
           +-02.0
           +-02.1
           +-07.0
           +-08.0
           +-09.0-[0000:01]----08.0
           +-0b.0-[0000:02]----00.0
           +-0c.0-[0000:03]--
           +-0d.0-[0000:04]--
           +-0e.0-[0000:05-0d]----00.0-[0000:06-0d]--+-01.0-[0000:07]--
           |                                         +-04.0-[0000:08]--
           |                                         +-05.0-[0000:09]----00.0
           |                                         +-06.0-[0000:0a-0b]----00.0-
[0000:0b]--
           |                                         +-08.0-[0000:0c]--
           |                                         \-09.0-[0000:0d]--
           +-18.0
           +-18.1
           +-18.2
           \-18.3
```

```
linux-host:~>
```

3.3.1.2 Hot Plug Support

Linux supports PCI Express hot plug through the ***pciehp*** module. By default this module is not loaded in the kernel. Before loading it you should make sure the root complex of the PC host support hot plugging.

3.3.1.3 Finding the PEV1100

To identify the PEV1100 in the PCI device list, just search for the PEX8624 PCI Express switch by looking for the “8624” string.

```
linux-host:~>lspci |grep 8624
05:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:01.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:04.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:05.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:06.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:08.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
06:09.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-port PCI Express Switch
(rev ab)
linux-host:~>
```

The first line in the list is the switch upstream port (the first discovered by the enumeration process). It has been located on bus 0xd in slot 0 [0d:00.0]

Then try to identify the FPGA PCI Express End point by looking for the “1100” string.

```
linux-host:~>lspci |grep 1100
09:00.0 Bridge: IOxOS Technologies PEV1100 PCI Express VME Bridge (rev 01)
linux-host:~>
```

It has been located on bus 0x11 in slot 0 [11:00.0]

Using again the ***lspci*** command we can do a byte dump of its PCI Configuration Header (Table 3).

Beware that 16 and 32 bit fields appear in reverse order because PCI is little endian.

```
linux-host:~>lspci -x -s 09:00.0
09:00.0 Bridge: IOxOS Technologies PEV1100 PCI Express VME Bridge (rev 01)
00: 57 73 00 11 07 05 10 00 01 00 80 06 10 00 00 00
10: 08 00 00 d0 00 00 00 00 00 00 00 b8 00 00 00 00
20: 01 40 00 00 00 00 00 00 00 00 00 00 08 20 50 50
30: 00 00 00 00 40 00 00 00 00 00 00 00 ff 00 00 00

linux-host:~>
```

A human readable display of that information is obtained with the **-v** option

```
linux-host:~>lspci -vv -s 09:00.0
09:00.0 Bridge: IOxOS Technologies PEV1100 PCI Express VME Bridge (rev 01)
        Subsystem: Device 2008:5050
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping-
```

```
SERR+ FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort-
>SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Interrupt: pin ? routed to IRQ 499
        Region 0: Memory at d0000000 (32-bit, prefetchable) [size=128M]
        Region 2: Memory at b8000000 (32-bit, non-prefetchable) [size=128M]
        Region 4: I/O ports at 4000 [size=4K]
        Capabilities: [40] Power Management version 3
                Flags: PMEClk- DSI+ D1- D2- AuxCurrent=0mA
PME(D0-,D1-,D2-,D3hot-,D3cold-)
                Status: D0 PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [48] Message Signalled Interrupts: Mask+ 64bit+ Count=1/4 Enable+
                Address: 00000000fee0300c  Data: 4142
                Masking: 0000000e  Pending: 00000000
        Capabilities: [60] Express (v1) Endpoint, MSI 00
                DevCap: MaxPayload 512 bytes, PhantFunc 1, Latency L0s <64ns, L1 <1us
                        ExtTag+ AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
                        RlxdOrd+ ExtTag- PhantFunc- AuxPwr- NoSnoop+
                        MaxPayload 128 bytes, MaxReadReq 512 bytes
                DevSta: CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr- TransPend-
                LnkCap: Port #0, Speed 2.5GT/s, Width x4, ASPM L0s L1, Latency L0
unlimited, L1 unlimited
                        ClockPM- Suprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk-
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed 2.5GT/s, Width x4, TrErr- Train- SlotClk- DLActive-
BWMgmt- ABWMgmt-
        Capabilities: [100] Virtual Channel <?>

linux-host:~>
```

We can observe the the FPGA PCI IO window has been allocated at address 0x4000 and the two PCI MEM windows at 0xd0000000 for the prefetchable one (size 128 MBytes) and at 0xb8000000 for the non prefetchable (size 128 MBytes)

3.3.2 Software installation

The PEV1100 software for Linux is provided as compressed archive file named “PEV1100_x.y.z.tgz” ready to be extracted anywhere in the file system tree

It is important that the **kernel sources package** is installed during the Linux installation. Without this, the kernel header files will not be available for building the PEV1100 device driver module. Typically, this package must be manually selected during the installation process. Please refer to the Linux installation documentation.

Due to the numerous flavors of Linux, IOxOS Technologies cannot support and test on all platforms. In order to provide customers with an easy to replicate platform, PEV1100 software is tested with easily-available Linux distributions. Other flavors of Linux should work ok or with a minimal porting effort. The PEV1100 Linux support has been built and tested on the OSes listed below

Operating System	Kernel
OpenSuse 10.3	2.6.24-*3 SMP x86_64*
Opensuse 11.1	2.6.27-7 SMP x86_64
Opensuse 11.1	2.6.28-*7 SMP x86_64 with Xenomai 2.4.7*

3.3.2.1 Extracting the sources

The PEV1100 software can be extracted anywhere in the Linux file system. The extraction process will

create a directory PEV1100/ containing all software sources provided by IOxOS Technologies to support the PEV1100 interface.

```
linux-host:ioxos> tar xzf PEV1100_2.0.0.tgz
```

The sources are organized as followed:

```
PEV1100/
   bin/
   cfg/
   doc/
   drivers/
   hotplug/
   include/
   lib/
   modules/
   scripts/
   src/
      examples/
      tools/
      XprsMon/
      XprsTst/
```

The file build.all located at the root of the tree is a bash script building all drivers, libraries and utilities for a “classical” linux kernel and, if found in the file system for a a real time linux kernel based on Xenomai 2.4.7

```
# check for Xenomai in the file system
XENODIR="/usr/xenomai"
ls $XENODIR
if [  $? -eq 0 ]; then
XENO=1
else
XENO=0
fi

# build PEV1100 device driver
cd drivers
./build.linux
if [  $XENO -eq 1 ]; then
./build.xeno
fi

# build hotplug device driver
cd ../hotplug
./build

# build libraries
cd ../lib
make
if [  $XENO -eq 1 ]; then
make xeno
fi

# build XprsMon
cd ../src/XprsMon
./build.linux
if [  $XENO -eq 1 ]; then
./build.xeno
fi

# build examples
cd ../examples
make
if [  $XENO -eq 1 ]; then
```

```
make xeno
fi

# build configuration tools
cd ../tools
make install

# build test suite
cd ../XprsTst
./build
```

3.3.2.2 Building Device Driver for the PEV1100

The first operation to performed by the build.all script, is to build the PEV1100 device driver. The directory ***PEV1100/drivers/*** contains all sources needed to build the loadable module ***pev.ko***. As already stated, the Makefile rely on the presence of the Linux kernel sources on the development machine.

The Makefile is written in such a way that the pev driver is expected to run on the machine on which it is compiled.

The *driver/build.linux* script generate a driver for a “classical” Linux and copies the kernel object in the *modules/* directoy under the name *pev-linux.ko*

```
rm -f *.o *.ko *.mod.c
make -C /lib/modules/2.6.28.7-xeno/build M=/home/ioxos/Release/PEV1100_2.0.0/drivers
modules
make[1]: Entering directory `/home/ioxos/kernel/linux-2.6.28.7'
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevdrvr.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevioctl.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevklib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/rdwrlib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/sflashlib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/maplib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/i2clib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/vmelib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/dmalib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/histolib.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.mod.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.ko
make[1]: Leaving directory `/home/ioxos/kernel/linux-2.6.28.7'
```

If Xenomai has been installed in the host filesystem, the *driver/build.xeno* script generate a driver for a real time Linux and copies the kernel object in the *modules/* directoy under the name *pev-xeno.ko*

```
rm -f *.o *.ko *.mod.c
make -C /lib/modules/2.6.28.7-xeno/build M=/home/ioxos/Release/PEV1100_2.0.0/drivers
modules
make[1]: Entering directory `/home/ioxos/kernel/linux-2.6.28.7'
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevdrvr.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevioctl.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pevklib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/rdwrlib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/sflashlib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/maplib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/i2clib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/vmelib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/dmalib.o
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/histolib.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.o
  Building modules, stage 2.
```

```
  MODPOST 1 modules
  CC      /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.mod.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/drivers/pev.ko
make[1]: Leaving directory `/home/ioxos/kernel/linux-2.6.28.7'
```

3.3.2.3 Building the PEV1100 hotplug driver

The *hotplug/* directory contains the sources of a device driver to allow the user to power-off/power-on the PEV1100 without rebooting the local host. The build script copies the generated kermel object in the *modules/* directory under the name *hppev.ko*

```
rm -f *.o *.ko *.mod.c
make -C /lib/modules/2.6.28.7-xeno/build M=/home/ioxos/Release/PEV1100_2.0.0/hotplug
modules
make[1]: Entering directory `/home/ioxos/kernel/linux-2.6.28.7'
  CC [M]  /home/ioxos/Release/PEV1100_2.0.0/hotplug/hpdrvr.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/hotplug/hppev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ioxos/Release/PEV1100_2.0.0/hotplug/hppev.mod.o
  LD [M]  /home/ioxos/Release/PEV1100_2.0.0/hotplug/hppev.ko
make[1]: Leaving directory `/home/ioxos/kernel/linux-2.6.28.7'
```

3.3.2.4 Building the PEV1100 users library

The *lib/* directory contains the sources of the user library interface with the driver. Executing a make in this directory builds the libpev.a library file.

```
gcc -I ../include  -c pevulib.c
gcc -I ../include  -c clilib.c
gcc -I ../include  -c tstlib.c
ar r libpev.a pevulib.o clilib.o tstlib.o
ar: creating libpev.a
ranlib libpev.a
```

If Xenomai has been installed in the host filesystem, the “make xeno” generate the library libpevrt.a to be used with the pev-xeno device driver in order to create real time linux applications.

```
gcc -I ../include  -I /usr/xenomai/include -c pevrtlib.c
ar r libpevrt.a pevrtlib.o
ar: creating libpevrt.a
ranlib libpevrt.a
cp /usr/xenomai/lib/librtdm.a .
cp /usr/xenomai/lib/libnative.a .
```

3.3.2.5 Building XprsMon

XprsMon is a command interpreter allowing a user to debug a VME system controlled by a PC host through the PEV1100 interface. The sources together with a Makefile shall be found in the *src/XprsMon/*. Executing buil.linux in that directory creates the ***XprsMon*** executable and copies it in the bin/ directory.

```
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c tty.c
gcc -g -DLITTLE_ENDIAN -DDEBUGrm -f *.o XprsMon
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c XprsMon.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c rdwr.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c conf.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c sflash.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c map.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c script.c  -I ../../include -c tst.c
```

```
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c timer.c
gcc -g -DLITTLE_ENDIAN -DDEBUG  -I ../../include -c vme.c
gcc -L ../../lib -o XprsMon XprsMon.o rdwr.o conf.o sflash.o map.o script.o tty.o tst.o
timer.o vme.o -lpev -lrt
```

If Xenomai has been installed in the host filesystem, the build.xeno script generates an **XprsMon** application linked with the real time library and copied in the *bin/* directory under the name **XprsMonRt**.

```
rm -f *.o XprsMon
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
XprsMon.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
rdwr.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
conf.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
sflash.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
map.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
script.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
tty.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
tst.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
timer.c
gcc -g -DLITTLE_ENDIAN -DDEBUG -DXENOMAI -I ../../include -I /usr/xenomai/include -c
vme.c
gcc -L ../../lib -o XprsMon XprsMon.o rdwr.o conf.o sflash.o map.o script.o tty.o tst.o
timer.o vme.o -lpev -lrt -lpevrt -lnative -lrtdm
```

3.3.2.6 Installing the PEV1100 examples

The directory src/examples contains few examples of linux applications driving the PEV1100 interface.

```
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -I /usr/xenomai/include -c BerrTst.c
gcc -L ../../lib -o BerrTst BerrTst.o  -lpev -lrt
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -I /usr/xenomai/include -c VmeTst.c
gcc -L ../../lib -o VmeTst VmeTst.o  -lpev -lrt
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -I /usr/xenomai/include -c DmaTst.c
gcc -L ../../lib -o DmaTst DmaTst.o  -lpev -lrt
```

```
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -I /usr/xenomai/include -DXENOMAI -c
DmaTst.c -o DmaTstRt.o
gcc -L ../../lib -o DmaTstRt DmaTstRt.o  -lpev -lrt -lpevrt -lnative -lrtdm
```

3.3.2.7 Installing the PEV1100 configuration tools

A set of configuration tools is provided to allow user's to generate binary objects to be loaded in the SFLASH device. These are useful for those developing new FPGA bitstreams.

```
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -c fsm2bin.c
gcc -L ../../lib -o fsm2bin fsm2bin.o
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -c mcs2bin.c
gcc -L ../../lib -o mcs2bin mcs2bin.o
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -c fpgabuild.c
gcc -L ../../lib -o fpgabuild fpgabuild.o
```

```
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -c fpgacheck.c
gcc -L ../../lib -o fpgacheck fpgacheck.o
gcc -g -DLITTLE_ENDIAN -DDEBUG -I ../../include -c hppev.c
gcc -L ../../lib -o hppev hppev.o
cp fsm2bin ../../bin
cp mcs2bin ../../bin
cp fpgabuild ../../bin
cp fpgacheck ../../bin
cp hppev ../../bin
```

3.3.2.8 Installing the PEV1100 test suite

The srcXprsTst/ directory contains a test suite to be controlled from XprsMon. The build script generate the test launcher XprsTst and a test file PevTst containing a set of the test executed for the validation of the PEV1100 hardware.

```
rm -f *.o XprsTst PevTst
gcc -g -DLITTLE_ENDIAN -DDEBUGno  -I ../../include -c XprsTst.c
gcc -g -DLITTLE_ENDIAN -DDEBUGno  -I ../../include -c tst_0x.c
gcc -L ../../lib -o XprsTst XprsTst.o -lpev -lrt
gcc -g -DLITTLE_ENDIAN -DDEBUGno  -I ../../include -c PevTst.c
gcc -L ../../lib -o PevTst PevTst.o  tst_0x.o -lpev -lrt
```

3.3.3 Loading the PEV1100 device driver

Before running any application using the PEV1100, it is mandatory to insert dynamically either the ***pev-linux.ko*** or the ***pev-xeno.ko*** device driver in the kernel. This is done by executing the ***insmod*** program. The driver initialization function allocates dynamically the device major number. This number is needed to create the nodes allowing applications to access the device and can be find in the /proc/devices file once the driver has been successfully installed.

The script file ***load*** located in the *modules/* directory performs all these operations. It loads a kernel module according to the argument given, retrieves the device number assigned by the kernel to the PEV1100 device and create the ***pev*** nodes in the */dev* directory. It shall be noted that you must have superuser privileges to run that script.

If the PEV1100 is not connected to the host, the device driver installation will fail. Before executing the ***load*** script, one should check with ***lspci*** that the PEV1100 has been discovered by the Linux.

To load the Linux “classical” driver, *pev-linux* shall be given has argument to the ***load*** script.

```
linux-host:PEV1100_2.0.0>su
Password:
linux-host:PEV1100_2.0.0>cd modules/
linux-host:modules>./load pev-linux
loading PEV1100 linux driver pev-linux.ko
linux-host:modules>
```

Using the ***lsmod*** command we can check if the ***pev*** module has been loaded

```
linux-host:modules>lsmod | grep pev
pev                    98672  0
linux-host:modules>
```

and get a list of the nodes created with read/write privileges for everybody.

```
linux-host:modules>ls -l /dev/pev*
```

```
crw-rw-rw- 1 root wheel 250,  0 2009-08-27 15:50 /dev/pev
crw-rw-rw- 1 root wheel 250,  0 2009-08-27 15:50 /dev/pev0
crw-rw-rw- 1 root wheel 250,  1 2009-08-27 15:50 /dev/pev1
crw-rw-rw- 1 root wheel 250, 10 2009-08-27 15:50 /dev/pev10
crw-rw-rw- 1 root wheel 250, 11 2009-08-27 15:50 /dev/pev11
crw-rw-rw- 1 root wheel 250, 12 2009-08-27 15:50 /dev/pev12
crw-rw-rw- 1 root wheel 250, 13 2009-08-27 15:50 /dev/pev13
crw-rw-rw- 1 root wheel 250, 14 2009-08-27 15:50 /dev/pev14
crw-rw-rw- 1 root wheel 250, 15 2009-08-27 15:50 /dev/pev15
crw-rw-rw- 1 root wheel 250,  2 2009-08-27 15:50 /dev/pev2
crw-rw-rw- 1 root wheel 250,  3 2009-08-27 15:50 /dev/pev3
crw-rw-rw- 1 root wheel 250,  4 2009-08-27 15:50 /dev/pev4
crw-rw-rw- 1 root wheel 250,  5 2009-08-27 15:50 /dev/pev5
crw-rw-rw- 1 root wheel 250,  6 2009-08-27 15:50 /dev/pev6
crw-rw-rw- 1 root wheel 250,  7 2009-08-27 15:50 /dev/pev7
crw-rw-rw- 1 root wheel 250,  8 2009-08-27 15:50 /dev/pev8
crw-rw-rw- 1 root wheel 250,  9 2009-08-27 15:50 /dev/pev9
linux-host:modules>
```

16 nodes have been created with minor device number going from 0 to 15. The minor device number allows to target a specific PEV1100 board in multi-crate configurations. When opening the device, a match between the position of the rotary encoder located on the board and the minor device number is required.

If compatibility with older version of the driver is needed, the rotary encoder shall be set to position 0.

If Xenomai has been installed in the Linux kernel, one can load the real time driver pev-xeno. It shall be noted that pev-linux and pev-xeno cannot be loaded together. They both offer the same capabilities. However, the pev-xeno version rely on xenomai interrupt service routines and synchronization mechanism in order to guarantee real time performances.

To load the Linux “real time” driver, *pev-xeno* shall be given has argument to the ***load*** script. If pev-linux is already installed, don't forget to remove it.

```
linux-host:modules>rmmod pev
linux-host:modules>./load pev-xeno
loading PEV1100 xenomai driver pev-xeno.ko
linux-host:modules>
```

When this has been done, we can run applications relying on this driver such ***XprsMon*** or ***XprsMonRt.***

3.3.4 Using XprsMon to access the PEV1100

The PEV1100 binaries are located in the PEV1100/bin directory. We first add it to the default search PATH:

```
linux-host:~>export PATH=~/PEV1100/bin:$PATH
```

and we execute the PEV1100 command interpreter ***XprsMon*** for node ***1***

```
linux-host:~>XprsMon 1

     +----------------------------------------+
     |    XprsMon - PEV1100 diagnostic tool   |
     |    IOxOS Technologies Copyright 2009   |
     +----------------------------------------+

initializing crate 1
Device driver: pev-linux
XprsMon#1>
```

Basically **XprsMon** implements a set of command (see §4.1.6) interpreted and executed when the user enters a command line on the keyboard. A command line is an ASCIIi string made of

- a command code
- a command extension (optional)

- zero or more command parameters

The ***help*** command displays a list of all supported command codes. "***?***" is an alias for ***help***.

```
XprsMon#1>help
conf      dm        dma       dp
ds        dv        fm        fp
fs        fv        help      i2c
ls        lv        map       pc
pio       pm        pp        pr
ps        pv        px        sflash
ts        tv        timer     tinit
tkill     tlist     tset      tstart
tstop     tty       vme
XprsMon#1>
```

help accept one parameter which is any supported command code in order to display the syntax of the command.

```
XprsMon#1>? pio
read/write data from/to PEV1100 PCI IO space
pio.<ds> <addr> <data>   where <ds> = b,s,w, (data size 1,2,4)
         <addr> = address offset in hexadecimal
         <data> = data in hexadecimal [write cyle]
XprsMon#1>
```

The command extension is separated from the command code by a comma. In the example above, the ***pio*** command extension <ds> is used to define the data size to be used for the read or write cycle. The command parameters are the cycle target address and the data for a write operation.

It shall be noted that, unless otherwise stated, ***XprsMon*** integer parameters are interpreted and displayed as hexadecimal values.

3.3.5 The PEV1100 Configuration

The **conf** command displays a summary of the PEV1100 hardware configuration.

```
XprsMon#1>conf
   Static Options [0x00262ad3]
      VME Interface
         A24 Base Address  : d00000
         System Controller : 64x- Slot1+ SysRstEna+
      PLX8624 Switch
         Port0 [P3]        : Upstream : Local Clock
         Port1 [P4]        : Downstream
         Port5 [FPGA]      : Downstream
         Port6 [PCI]       : Downstream
         Port8 [XMC#1]     : Downstream
         Port9 [XMC#2]     : Downstream
      FPGA
         Bit Stream        : 1
         PON FSM           : Disabled
         MEM size          : 128 MBytes
         PMEM size         : 128 MBytes
         PMEM mode         : A32
   VME Interface
      System Controller    : Enabled
         Arbtration mode   : PRI not pipelined
         Bus Timeout       : 16 usec
      Master               : Enabled
         Request Mode      : Release On Request
         Request Level     : 0
      Slave                : Enabled
```

```
         A24 base address  : 0xd00000
         A32 base address  : 0x10000000
         A32 window size   : 0x10000000
         CR/CSR            : Berr- SlvEna+ SysFail- SysFailEna- Reset-
      Interrupt Generator
         Vector            : 00
         Level             : 0
         Mode              : Register
         Status            : Cleared
   FPGA System Monitor
      Temperature          : 45.28 [48.23 - 30.02]
      VCCint               : 1.00 [1.01 - 0.99]
      VCCaux               : 2.51 [2.52 - 2.51]
      VCC1.8-INT           : 1.78
      VCC3.3-INT           : 3.23
      VCC5.0-VME           : 5.02
      VCC3.3-VME           : 3.33
XprsMon#1>
```

3.3.6 Script files

To avoid the burden of typing many times the same set of command, XprsMon is able to execute simple script files. Script files can include comments starting with the '#' character. When '#' is encountered by the line interpreter, all following characters are discarded.

If a line start with the '$' character, the following string of character is interpreted as a Linux command. The following commands are supported:

- sleep
- usleep

It shall be noted that script files can contain references to other script files.

3.3.7 Reloading an FPGA bitstream

After a reset, the FPGA bitstream file is automatically loaded from the SFLASH device according to the identification number selected in the static options SW501 (see Table 7). IOxOS Technologies provides binary files with a ***.sfl*** extention ready to be loaded in SFLASH. These files contains the FPGA bitstream, a register initialization code and a signature.

The XprsMon command ***sflash*** allows the user to perform a load operation in order to store a .sfl binary file in the PEV1100 SFLASH device.

```
linux-host:sfl>XprsMon

     +----------------------------------------+
     |    XprsMon - PEV1100 diagnostic tool   |
     |    IOxOS Technologies Copyright 2008   |
     +----------------------------------------+

XprsMon>sflash load fpga#1 pev_041108.sfl
Loading SFLASH from file pev_101208a.sfl at offset 0x200000 [size 0x200000]

!! Programming the SFLASH device is done one bit at a time
!! It requires millions of physical accesses loading the CPU at 100%
!! During that process the system will be hanging for periods of 10 seconds
!! This is the time needed to program one SFLASH sector
-> Just relax and sit back...

Writing device will take about 90 seconds.....00400000  -> done
Verifying device will take about 108 seconds...00400000 -> OK
XprsMon>
```

That operation takes two arguments

- an fpga identifier ***fpga#i*** (where i goes from 1 to 4)
- the file name of the ***.sfl*** file to be loaded

The sign operation displays the file signature

```
XprsMon>sflash sign fpga#1
FPGA#1 Signature at offset 0x3f0000

   + company:IOxOS Technologies
   + board:PEV1100
   + filename:pev_101208a.sfl
   + creation:Wed Dec 10 14:22:35 2008

   + mcs_file:pev1100_sw4_101208a.mcs
   + mcs_devname:XC5VLX30T
   + mcs_devid:0x02a6e093
   + mcs_offset:0x000000
   + mcs_size:0x11dfc0
   + mcs_checksum:0x339e1638
   + mcs_creation:Wed Dec 10 14:19:02 2008

   + fsm_file:pev_init.fsm
   + fsm_offset:0x1c0000
   + fsm_size:0x000278
   + fsm_checksum:0xcc952770
   + fsm_creation:Wed Sep 17 15:43:36 2008

XprsMon>
```

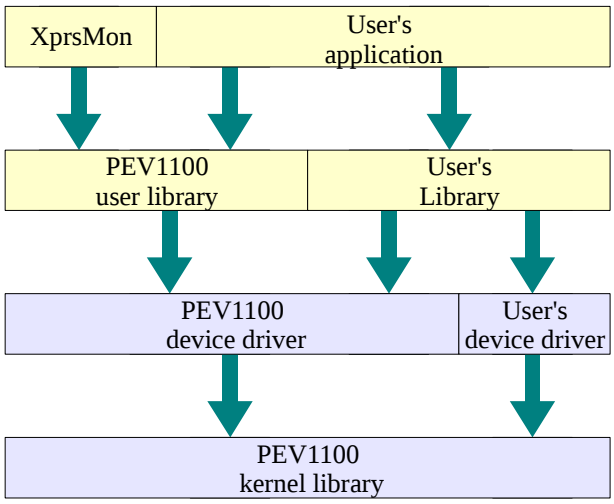
4 Software Reference

4.1 Linux Host

4.1.1 Software Organization

The PEV1100 software for Linux is delivered on an Iso9660 CDROM labeled ***PEV1100_x.y.z*** together with an installation notice. The CDROM contains a README file and a compressed archive file labeled ***PEV1100_xxyyzz.tgz*** containing all delivered files including the documentation.

The ***PEV1100_xxyyzz.tgz*** can be extracted anywhere in the Linux file system and produce a directory tree labeled *PEV1100/* (see §3.3.2.1). All source code is written in C language and has been compiled with the gnu compiler (gcc version 4.2.1).



The core of the software organization is the PEV1100 device driver (see §4.1.2). This is a Linux loadable module located in the *drivers/* directory of the *PEV1100/* directory tree.

A user library (see §4.1.4)interfacing applications with the driver is provided in the lib/ directory.

The code has been written and tested for a 64 bit little endian machine. Data size are declared as followed:

```
8 bit signed    → char,
8 bit unsigned  → uchar, u8
16 bit signed   → short,
16 bit unsigned → ushort, u16
32 bit signed   → int,
32 bit unsigned → uint, u32
64 bit signed   → long,
64 bit unsigned → ulong, u64
```

4.1.2 PEV1100 Device Driver

The PEV1100 device driver is a Linux module acting as interface between an application and the PEV1100 interface. It relies on the PEV1100 kernel library. It implements the following functions:

```
pev_init()
pev_exit()
pev_probe()
pev_remove()
pev_open()
pev_release()
pev_read()
pev_write()
pev_llseek()
pev_ioctl()
pev_mmap()
pev_irq()
```

All global data used by the driver and the kernel library are stored in the ***pev_dev*** control structure defined in the ***pevdrvr.h*** file.

4.1.2.1 Install/Remove

When the ***pev*** driver is installed in the kernel, the function ***pev_init()*** is automatically executed. That function performs the following operations:

- dynamically allocate a major device number
- scan the PCI device table to find all PLX8624 upstream ports present in the PCI tree
- register the device if at least one PLX8624 has been found.

For each PLX8624 PCI Express switch found it allocated a control structure in order to build a minor device whose number is equal to the node number selected by the rotary switch on the PEV1100 board. If it fails, it unregister the device and returns an error. Otherwise, it maps a 128 Kbytes address window in the PCI MEM space to access the PEC8624configuration registers.

It scan again the PCI device table behind each PLX8624 in order to find the FPGA PCIe end point and the PEX8112 PCI Express to PCI bridge. If found, both devices are mapped in kernel space and the FPGA Control and Status Registers are initialized.

4.1.2.2 Open/Close

open() select the PEV1100 control structure according to the minor device number. Return an error of minor number doesn't match any PEV1100 node number.

olose() release the PEV1100 control structure

4.1.2.3 Read/Write

read() and write() functions are not implemented in the pev driver.

4.1.2.4 Ioctl

The file ***pevioctl.h*** contains all declarations and definition to be used with the ***ioctl()*** system call. The file ***pevioctl.c*** implement all ioctl operations.

4.1.2.4.1 PEV1100 read/write operations

The **PEV_IOCTL_RDWR** command is a generic way to perform read and write access on any PEV1100 resource. The control structure has the following format:

```
struct pev_ioctl_rdwr
{
  void *buf;            /* data buffer pointer in user space */
  uint offset;          /* address offset in remote space    */
  uint len;             /* data transfer size in byte        */
  struct pev_rdwr_mode
```

```
  {
    char ds;            /* data size                        */
    char swap;          /* swap mode                        */
    char as;            /* address size                     */
    char dir;           /* transfer direction               */
    char crate;         /* remote crate identifier          */
    char am;            /* address modifier                 */
    char space;         /* remote space identifier          */
    char rsv3;
  } mode;
  void *k_addr;
};
```

The ***buf*** field is a pointer in the user's space to the data to be read or written.

The ***offset*** field is an address offset within the address space of the resource to be accessed. The resource is identified by the ***space*** fields in the ***pev_rdwr_mode*** data structure.

The field ***len*** defines the number of bytes to be transferred. For single access operations this field shall be set to zero.

The ***pev_rdwr_mode*** data structure holds the parameters defining the access type.

The ***ds*** field defines the data size to be used for each access. It can be 1 (byte), 2 (short), 4 (int) or 8 (long).

The ***as*** fields define the address size, It is used for VME accesses and can be 16, 24, 32, 48 or 64.

The ***dir*** field defines the transfer direction. It shall be 0 for a read and 1 for a write.

The ***crate*** field shall be set to the node number of the PEV1100 holding the targeted resource.

The ***space*** field shall be an identifier of the targeted resources. The following identifiers are valid:

```
RDWR_PCIE_CFG  → PEV1100 configuration register
RDWR_PCIE_IO   → PEV1100 PCI IO window (Control and status registers)
RDWR_PCIE_PMEM → PEV1100 Prefetchable memory window
RDWR_PCIE_MEM  → PEV1100 Non Prefetchable memory window
RDWR_PEX_MEM   → PEX8624 Non Prefetchable memory window
RDWR_KMEM      → Host kernel buffer
```

The ***k_addr*** field is used when **RDWR_KMEM** space is selected and shall hold the kernel base address of a data buffer allocated with **PEV_IOCTL_DMAC_ALLOC** (see §4.1.2.4.5).

4.1.2.4.2 PEV1100 address mapping operations

Some of the PEV1100 resources such the VME bus or the shared memory are accesses through an address translation table allowing to remap a local address to a remote address. Per example, the VME bus is seen by the CPU as a PCI memory resource. The address translation table defines the relationship between the PCI address used by the CPU and the VME cycle actually generated.

The **PEV_IOCTL_MAP_ALLOC** command allows to created an address translation window by allocating pages in the scatter/gather memory.

The **PEV_IOCTL_MAP_FREE** command allows to free previously allocated pages.

The **PEV_IOCTL_MAP_MODIFY** command allows to modify the mapping parameters of previously allocated pages.

All mapping operation are performed using the ***pev_ioctl_map_pg*** data structure as argument.

```
struct pev_ioctl_map_pg
{
  uint size;                          /* mapping size required by user          */
  char flag; char sg_id; ushort mode;   /* mapping mode                           */
```

```
  ulong rem_addr;                       /* remote address to be mapped              */
  ulong loc_addr;                       /* local address returned by mapper         */
  uint offset;                          /* offset of page containing local address  */
  uint win_size;                        /* size actually mapped                     */
  ulong rem_base;                       /* remote address of window actually mapped */
  ulong loc_base;                       /* local address of window actually mapped  */
  void *usr_addr;                       /* user address pointing to local address   */
};
```

The ***rem_addr*** and ***size*** parameters are used by the **PEV_IOCTL_MAP_ALLOC** command to define the window base address and size in bytes to be mapped in the destination space.

The ***sg_id*** parameter defines which address translation table shall be used for the mapping operation. Three tables are currently implemented in the PEV1100:

```
MAP_PCIE_MEM  → FGA End Point PCI MEM mapper
MAP_PCIE_PMEM → FGA End Point PCI PMEM mapper
MAP_VME_SLAVE → VME slave mapper
```

The ***mode*** parameter defines the addressing mode and the destination space Its value shall be the logical or of the space,

```
MAP_SPACE_PCIE → destination space is PCI tree
MAP_SPACE_VME  → destination space is VME bus
MAP_SPACE_SHM  → destination space is shared memory
```

swapping policy,

```
MAP_SWAP_NO    → no swapping
MAP_SWAP_AUTO  → auto swapping big → little endian
```

and address mode identifiers.

```
MAP_VME_SP      → VME supervisor mode
MAP_VME_A16     → VME A16 (AM = 0x29 or 0x2d)
MAP_VME_A24     → VME A24 (AM = 0x39 or 0x3d)
MAP_VME_A32     → VME A32 (AM = 0x09 or 0x0d)
MAP_VME_BLT     → VME A32 (AM = 0x0b or 0x0f)
MAP_VME_MBLT    → VME A32 (AM = 0x0c)
MAP_VME_IACK    → VME IACK
```

The ***flag*** parameter holds information to allow to share address mapping among several applications.

The ***loc_addr*** parameter is filled by the driver after a successful allocation and contains the local address to be used to target the remote address That address is an address offset within the mapping window and can be used as offset parameter by the PEV_IOCTL_RDWR command or by the mmap() function.

The other parameters are used internally by the driver mapping functions and should be left untouched.

4.1.2.4.3 PEV1100 SFLASH operations

The **PEV_IOCTL_SFLASH_ID** command returns the SFLASH hardware identifier.

SFLASH read/write operations are performed using the ***pev_ioctl_sflash_rw*** data structure as argument.

```
struct pev_ioctl_sfash_rw
{
  void *buf;
  uint offset;
  uint len;
```

```
};
```

The ***buf*** field is a pointer in the user's space to the data to be read or written.

The ***offset*** field is the SFLASH offset where data have to be read or written

The **len** field defines the number of bytes to be transferred.

The **PEV_IOCTL_SFLASH_RD** command shall be used to read an SFLASH address range and copy its content in the buffer provided by the application.

The **PEV_IOCTL_SFLASH_RW** command shall be used to copy a data buffer provided by the application in the SFLASH device. SFLASH sector boundaries are automatically handled by the driver, by first reading the entire sector, updating it and writing it back. SFLASH programming is done one bit at a time, thus overloading the CPU. The system seems to be hanging when that operation is performed.

4.1.2.4.4 PEV1100 local timer operation

All timer operation are performed using the ***pev_ioctl_timer*** data structure as argument.

```
struct pev_ioctl_timer
{
  uint operation;   /* operation to perform   */
  uint time;        /* tick counter (msec)    */
  uint utime;       /* usec counter           */
  uint mode;        /* operating mode         */
};
```

The **PEV_IOCTL_TIMER_START** command starts the PEV1100 local timer. The ***mode*** field of the ***pev_ioctl_timer*** data structure is used to define in which mode the timer shall be started.

```
TIMER_1MHZ         → timer frequency 1 MHz
TIMER_5MHZ         → timer frequency 5 MHz
TIMER_25MHZ        → timer frequency 25 MHz
TIMER_100MHZ       → timer frequency 100 MHz
TIMER_BASE_1000    → timer period 1000 usec (1 msec)
TIMER_BASE_1024    → timer period 1024 usec
TIMER_SYNC_LOC     → timer synchronization local
TIMER_SYNC_USR1    → timer synchronization user signal #1
TIMER_SYNC_USR2    → timer synchronization user signal #2
TIMER_SYNC_SYSFAIL → timer synchronization VME sysfail
TIMER_SYNC_IRQ1    → timer synchronization VME IRQ#1
TIMER_SYNC_IRQ2    → timer synchronization VME IRQ#2
TIMER_SYNC_ENA     → timer synchronization enable
TIMER_OUT_SYSFAIL  → issue sync signal on VME sysfail
TIMER_OUT_IRQ1     → issue sync signal on VME IRQ#1
TIMER_OUT_IRQ2     → issue sync signal on VME IRQ#2
```

It shall be set to the bitwise OR of **TIMER_FREQ_xxx**, **TIMER_BASE_xxx**, **TIMER_SYNC_xxx** or **TIMER_OUT_xxx**.

The **PEV_IOCTL_TIMER_STOP** command stops the PEV1100 local timer

The **PEV_IOCTL_TIMER_READ** command reads the PEV1100 local timer and update the ***time*** and ***utime*** fields of the ***pev_ioctl_timer*** data structure with the current value of the main counter and the micro timer.

4.1.2.4.5 PEV1100 DMA operation

The **PEV_IOCTL_DMA_MOVE** request code allows to move data between a source and a destination

space using the PEV1100 embedded DMA engines.

Data transfer operations using the DMA engines are performed using the ***pev_ioctl_dma_req*** data structure as argument.

```
struct pev_ioctl_dma_req
{
  ulong src_addr;
  ulong des_addr;
  uint size;
  uchar src_space; uchar src_mode; uchar des_space; uchar des_mode;
  uchar start_mode; uchar end_mode; uchar intr_mode; uchar wait_mode;
  uint dma_status;
};
```

The ***src_addr*** shall be initialized by the calling application with the bus address of the source buffer.

The ***des_addr*** shall be initialized by the calling application with the bus address of the destination buffer.

The ***size*** field shall be initialized by the calling application with the size of the requested transfer. For a block transfer (see ***start_mode***), the size maximum is 0xff800. For a list transfer, the maximum number of element in the list is 63.

```
DMA_SIZE_MAX_BLK   → maximum block size
DMA_SIZE_MAX_LIST  → maximum list size
```

The ***src_space*** and ***des_space*** shall be initialized by the calling application with the source and destination spaces. The following values are accepted:

```
DMA_SPACE_PCIE                    → PCI Express
DMA_SPACE_SHM                     → shared memory
DMA_SPACE_VME | DMA_VME_A16       → VME single transfer A16
DMA_SPACE_VME | DMA_VME_A24       → VME single transfer A24
DMA_SPACE_VME | DMA_VME_A32       → VME single transfer A32
DMA_SPACE_VME | DMA_VME_SGL       → VME single transfer A32
DMA_SPACE_VME | DMA_VME_BLT       → VME block transfer A32
DMA_SPACE_VME | DMA_VME_MBLT      → VME MBLT
DMA_SPACE_VME | DMA_VME_2eVME     → VME 2eVME
DMA_SPACE_VME | DMA_VME_2eFAST    → VME 2eFAST
DMA_SPACE_VME | DMA_VME_2e160     → VME 2eSST160
DMA_SPACE_VME | DMA_VME_2e233     → VME 2eSST233
DMA_SPACE_VME | DMA_VME_2e320     → VME 2eSST320
```

The ***src_mode*** and ***des_mode*** shall be used by the application to pass to the driver additional parameters associated to the source and destination spaces. For **DMA_PCIE_SPACE**, these fields allows to specify the traffic class and the number of outstanding read request:

```
DMA_PCIE_TC0      → Traffic Class 0
DMA_PCIE_TC1      → Traffic Class 1
DMA_PCIE_TC2      → Traffic Class 2
DMA_PCIE_TC3      → Traffic Class 3
DMA_PCIE_TC4      → Traffic Class 4
DMA_PCIE_TC5      → Traffic Class 5
DMA_PCIE_TC6      → Traffic Class 6
DMA_PCIE_TC7      → Traffic Class 7
DMA_PCIE_RR1      → 1 outstanding read request
DMA_PCIE_RR2      → 2 outstanding read request
DMA_PCIE_RR3      → 3 outstanding read request
```

The ***start_mode*** shall be set by the application to define the mode to be used for the data transfer. The following modes are supported:

```
DMA_MODE_BLOCK    → move one block from src to des
DMA_MODE_LIST_RD  → move a list of buffer pointed by src_addr to a destination buffer
```

If **DMA_MODE_LIST_RD** is selected, the ***src_addr*** parameter is expected to point on a list of ***pev_ioctl_dma_list*** data structure holding the transfer parameters:

```
struct pev_ioctl_dma_list
{
  ulong addr;
  uint size;
  uint mode;
};
```

In this case, the ***size*** parameter of the ***pev_ioctl_dma_req*** data structure shall be interpreted as the number of element in the transfer list (maximum 63). The exact meaning of the fields in the ***pev_ioctl_dma_list*** data structure depend on the value of the ***src_space*** parameter. For **DMA_VME_SPACE**, ***addr*** shall be the VME address, ***size*** the buffer size in byte and ***mode***, the VME transfer mode (Address Modifier).

The ***end_mode*** shall be initialized to 0 by the calling application (reserved for future use)

The ***intr_mode*** shall be set to **DMA_INTR_ENA** if DMA interrupts must be enabled, else it shall be set to 0.

The ***wait_mode*** shall be set to **DMA_WAIT_INTR** if the system call must return when the DMA engine has stopped (at the end of transfer or in case of error).If set to 0, the driver return to the application after having started the DMA.

The ***dma_status*** field is updated by the driver before returning to the application**.**

The **PEV_IOCTL_BUF_ALLOC** request code allocates a data buffer in kernel space suitable for DMA access. It takes a ***pev_ioctl_buf*** data structure as argument.

```
struct pev_ioctl_buf
{
  int kmem_fd;
  uint size;
  void *b_addr;
  void *u_addr;
  void *k_addr;
};
```

The ***size*** field shall be initialized by the calling application with the size in byte of the requested buffer.

The driver updates the ***b_addr*** and ***k_addr*** fields with the bus and kernel addresses of the buffer actually allocated.

The bus address is the address used by the DMA engines to access the buffer.

The kernel address is used by the CPU while executing in kernel mode to access the buffer. That address shall be given in the ***pev_ioctl_rdrw*** data structure when using PEV_IOCTL_RDWR command to perform read/write access to that buffer.

The fields ***kmem_fd*** and ***u_addr*** are not used by the kernel but are provided to allow the application to perform a subsequent call to the mmap() function of the /dev/kmem device in order to map the buffer in user's space.

4.1.2.5 Mmap

The mmap function allows to map the PEV1100 PCI MEM and PMEM address space in the application space.

4.1.2.6 Interrupt handling

The PEV1100 uses PCI Express Message Signaled Interrupts (MSI) to notify the host of the occurrence of asynchronous events.

4.1.3 PEV1100 Kernel Library

The PEV1100 kernel library implements a set of low levels functions used to handle the PEV1100. That library is part of the PEV1100 linux module. Each function takes as first argument a pointer to the pev_dev data structure.

```
int pev_rdwr(struct pev_dev *, struct pev_ioctl_rdwr *);
void pev_sflash_id(struct pev_dev *, unsigned char *);
int pev_sflash_read(struct pev_dev *, struct pev_ioctl_rdwr *);
int pev_sflash_write(struct pev_dev *, struct pev_ioctl_rdwr *);
int pev_map_init(struct pev_dev *, struct pev_ioctl_map_ctl *);
int pev_map_alloc(struct pev_dev *,  struct pev_ioctl_map_pg *);
int pev_map_free(struct pev_dev *,  struct pev_ioctl_map_pg *);
int pev_map_modify(struct pev_dev *,  struct pev_ioctl_map_pg *);
int pev_map_find(struct pev_dev *,  struct pev_ioctl_map_pg *);
int pev_map_read(struct pev_dev *,  struct pev_ioctl_map_ctl *);
int pev_map_clear(struct pev_dev *,  struct pev_ioctl_map_ctl *);
int pev_map_set_sg(struct pev_dev *, struct pev_ioctl_map_ctl *, uint);
int pev_map_clear_sg(struct pev_dev *, struct pev_ioctl_map_ctl *, uint);
void pev_sg_master_32_set(struct pev_dev *, uint, ulong,  uint);
void pev_sg_master_64_set(struct pev_dev *, uint, ulong,  uint);
void pev_sg_slave_vme_set(struct pev_dev *, uint, ulong,  uint);
void pev_i2c_pex_read(struct pev_dev *, struct pev_ioctl_i2c *);
void pev_i2c_pex_write(struct pev_dev *, struct pev_ioctl_i2c *);
void pev_vme_conf_read(struct pev_dev *, struct pev_ioctl_vme_conf *);
void pev_vme_conf_write(struct pev_dev *, struct pev_ioctl_vme_conf *);
uint pev_vme_crcsr(struct pev_dev *, struct pev_ioctl_vme_crcsr *);
uint pev_vme_crcsr(struct pev_dev *, struct pev_ioctl_vme_crcsr *);
void pev_timer_read( struct pev_dev *, struct pev_ioctl_timer *);
int pev_timer_start( struct pev_dev *, struct pev_ioctl_timer *);
void pev_timer_restart( struct pev_dev *);
void pev_timer_stop( struct pev_dev *);
```

4.1.4 PEV1100 User's Library

The user library interfaces applications with the PEV device driver. It expects the device ***/dev/pevX*** (where X is the crate number) to be existing with with read/write access for the user executing the application linked to the library. Compatibility with previous release of the library (single crate version) is obtained by setting the crate number to 0.

A multicrate interface is also provided in which function takes the crate number as first parameter and are labeled ***pevx_xxx_yyy(crate, …)***. When the multicrate version of the function is invoked, it checks if the initialization function for that crate has been called and returns an error if not.

Basically the library functions re-package user's parameters in a data structure ready to be passed to the driver ***ioctl()*** functions. If not otherwise stated, library functions returns the value returned by the corresponding ***ioctl()*** call.

4.1.4.1 Initialization

To be allowed to call functions defined in pevulib.a, the application shall first call an initialization function performing the open of the driver and initializing the library private data structure.

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   struct pev_node
   *pev_init( int crate)
or
   *pevx_init( int crate)
```

Description

That function must be called by the application before invoking any other function belonging to the PEV1100 user's library. Among other things, it performs an open of the */dev/pevX (where X is the crate number)* device and return the file descriptor in the *pev_node* data structure. This file descriptor can be used by the application for direct call to the pev device driver functions. When installing the driver, the script ***load*** in the modules directory creates automatically the /dev/pevX (X going from 0 to 15) device nodes.
Compatibility with older version of that library is obtained by choosing 0 as crate number.

Parameters

crate	crate number as set by the rotary switch located on the PEV1100

Return

pev	pointer to a pev_node data structure
	pev->fd pev file descriptor (returned by the open call)
NULL	Cannot open PEV1100 device driver

In multicrate configuration, the ***pev_init(crate)*** or ***pevx_init(crate)*** function shall be called for each crate to be accessed. If the single crate version of the library is used, the function ***pev_set_crate(crate)*** shall be called to make sure subsequent function calls will be targeted to the PEV1100 having ***crate*** as node number.

```
Prototype
   struct pev_node
   *pev_set_crate( int crate)

Description
   Set the default node number used by library function calls to crate.

Parameters
   crate       crate number as set by the rotary switch located on the PEV1100

Return
   pointer to the pev_node data structure if pev_init() has been successfully
   called previously.
   NULL in case of error (non initialized node).
```

4.1.4.2 Register access

A set of functions allows applications to perform read and write access on the PEV1100 control and status registers mapped in the PCI IO space.

4.1.4.2.1 CSR Read

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>

   int
   pev_csr_rd( int idx)
or
   pevx_csr_rd( int crate, int idx)

Description
   Perform a 32 bit read cycle in the PEV1100 PCI IO space in order to return
   the current value of the control and status register (see §Table 4) whose
   offset is equal to the idx parameter.

Parameters
   idx    register index (address offset in PCI IO window)

Return
   register current value
```

4.1.4.2.2 CSR Write

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>
```

```
   void
   pev_csr_wr( int idx,
               int data);
or
   pevx_csr_wr( int crate
                int idx,
                int data);

Description
   Perform a 32 bit write cycle in the PEV1100 PCI IO space in order to set
   the current value of the control and status register (see §Table 4) whose
   offset is equal to the idx parameter..

Parameters
   idx    register index (address offset in PCI IO window)
   data   32 bit value to write in register

Return
   None
```

4.1.4.2.3 CSR Set

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_csr_set( int idx,
                int data);
or
   pevx_csr_set( int crate,
                 int idx,
                 int data);

Description
   Perform a 32 bit read cycle followed by a 32 bit write cycle in the PEV1100
   PCI IO space in order to set the value  of the control and status register
   (see §Table 4) whose offset is equal to the idx parameter. The value
   written in the register is the bitwise OR between the result of the read
   cycle and the data parameter. This is equivalent to perform the following
   operation: pev_csr_wr( idx, data | pev_csr_rd( idx)).
   That function is useful to reset status bit in registers where they have to
   be overwritten to be cleared.

Parameters
   idx    register index (address offset in PCI IO window)
   data   bit field (32 bit) holding the bit to be set in the register

Return
   none
```

4.1.4.3 Generic read/write

The generic read/write function allows to move data between the application address space and the physical resources located on the PEV1100.

Prototype

```
   #include <pevulib.h>
   #include <pevxulib.h>
or
   #include <pevioctl.h>

   int
   pev_rdwr( struct pev_ioctl_rdwr *rdwr_p)
or
   pevx_rdwr( struct pev_ioctl_rdwr *rdwr_p)
```

Description

That function allows to move data between a buffer allocated in the application space and a PEV1100 physical resource. Basically it performs an ***ioctl()*** call with the **PEV_IOCTL_RDWR** request code (see §4.1.2.4.1). The argument shall be a pointer to an ***pev_ioctl_rdwr*** data structure initialized with the transfer parameters.

Parameters

rdwr_p pointer to a ***pev_ioctl_rdwr*** data structure initialized with the transfer parameters.

- *rdwr_p->buf* pointer to the data buffer in user's space
- *rdwr_p->offset* offset in the PEV1100 resource address space
- *rdwr_p->len* size in byte of the data transfer
- *rdwr_p->mode.ds* data size
  - ***RDWR_BYTE*** → 8 bit access (byte)
  - ***RDWR_SHORT*** → 16 bit access (short)
  - ***RDWR_INT*** → 32 bit access (int)
  - ***RDWR_LONG*** → 64 bit access (long)
- *rdwr_p->mode.swap* swapping mode
  - ***RDWR_NOSWAP*** → no swapping
  - ***RDWR_SWAP*** → swapping
- *rdwr_p->mode.dir* transfer direction
  - ***RDWR_READ*** → from PEV1100 to local memory (read)
  - ***RDWR_WRITE*** → from PEV1100 to local memory (write)
- *rdwr_p->mode.space* PEV1100 address space
  - ***RDWR_PCIE_CFG*** → FPGA End point PCI configuration space
  - ***RDWR_PCIE_IO*** → FPGA End point PCI IO window (BAR#4)
  - ***RDWR_PCIE_MEM*** → FPGA End point PCI MEM window (BAR#2)
  - ***RDWR_PCIE_PMEM*** → FPGA End point PCI PMEM window (BAR#0)
  - ***RDWR_PEX_MEM*** → PEX8624 PCI MEM window (BAR#2)
  - ***RDWR_KMEM*** → HOST kernel space (DMA buffer)
- *rdwr_p->k_addr* Kernel address of DMA buffer (if ***RDWR_KMEM***)

Return

```
   0       Data transfer successful
   < 0     See ioctl() error codes
```

4.1.4.4 PEV1100 Resource Mapping

Basically the PEV1100 acts as a bridge between the following address space

- PCI tree
- VME bus
- Shared Memory
- FPGA user's area

Going from one space to another implies the initialization of address translation tables in order to map source addresses to destination addresses. A set of functions allows application to dynamically create, modify, and destroy these address translation windows.

The PEV1100 hardware offers three address translation tables. The first two allows the local host to perform data cycles from the PCI MEM and PMEM spaces targeted to the VME bus, shared memory and FPGA user's space. The third one allows a VME master to perform data cycle from the VME bus (through the PEV1100 A32 slave port) targeted to the PCI tree, shared memory and FPGA user's area. Each table can contain multiple translation windows. The address allocator in the pev device driver uses a search algorithm designed to maximize the number of windows one can create and allows to share the same window among different applications.

4.1.4.4.1 Allocating a new address translation window

```
Prototype


   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   void
   pev_map_alloc( struct pev_ioctl_map_pg *map_p)
or
   pevx_map_alloc( int crate, struct pev_ioctl_map_pg *map_p)

Description
   That function allows to create an address window in a local space pointing
   to a remote area in a destination space specified in the mode field of the
   mapping data structure given as parameter.

Parameters

   map_p   pointer to a  pev_ioctl_map_pg data structure initialized with the
           mapping parameters.
           map_p->rem_addr      base address of remote area to be mapped
           map_p->size          Size (in bytes)  of remote area to be mapped
           map_p->sg_id         Identifier of mapping table to be used
                 MAP_MASTER_32 → mapper for PCI MEM window in FPGA End Point
                 MAP_MASTER_64 → mapper for PCI PMEM window in FPGA End Point
                 MAP_SLAVE_VME → mapper for VME slave A32 window
```

map_p->mode Mapping mode (16 bit)

- ***MAP_SPACE_PCIE*** → PCI Express tree is the destination space
- ***MAP_SPACE_VME*** → VME bus is the destination space
- ***MAP_SPACE_SHM*** → Shared Memory is the destination space
- ***MAP_ENABLE*** → Enable address translation
- ***MAP_ENABLE_WR*** → Enable write accesses
- ***MAP_VME_USR*** → if VME space perform user access
- ***MAP_VME_SP*** → if VME space perform supervisor access
- ***MAP_VME_A16*** → if VME space perform A16 address cycles
- ***MAP_VME_A24*** → if VME space perform A24 address cycles
- ***MAP_VME_A32*** → if VME space perform A32 address cycles
- ***MAP_VME_BLT*** → if VME space perform BLT address cycles
- ***MAP_VME_MBLT*** → if VME space perform MBLT address cycles
- ***MAP_VME_2eSST*** → if VME space perform A16 address cycles

Return

0 Mapping successful
pev_ioctl_map_pg data structure has been updated with the mapping parameters.

map_p->loc_addr Address offset in local space mapping rem_addr

map_p->offset Page offset in translation table used to create the address mapping

map_p->win_size Actual window size (in bytes) allocated for the mapping

map_p->rem_base Remote address of window base

map_p->loc_base Local address of window base

-1 *Mapping not successful*

4.1.4.4.2 Freeing a previously allocated address translation window

Prototype

```
#include <pevulib.h>
or
#include <pevxulib.h>
#include <pevioctl.h>

void
pev_map_free( struct pev_ioctl_map_pg *map_p)
or
pevx_map_free( int crate, struct pev_ioctl_map_pg *map_p)
```

Description

That function free a mapping window previously allocated by a call to ***pev_map_alloc()***. The field *sg_id* and *offset* of the ***pev_ioctl_map_pg*** data structure are used by the pev driver to identify the window to be freed.

Parameters

map_p pointer to a ***pev_ioctl_map_pg*** data structure initialized with the mapping parameters.

	map_p->offset	Offset (as returned by ***pev_map_alloc()***) of mapping window to be freed
	map_p->sg_id	Identifier of mapping table to be used

Return

0	Mapping window successfully freed
-1	Invalid mapping window parameters

4.1.4.4.3 Modifying a previously allocated address translation window

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   void
   pev_map_modify( struct pev_ioctl_map_pg *map_p)
or
   pevx_map_modify( int crate, struct pev_ioctl_map_pg *map_p)
```

Description

That function modifies a mapping window previously allocated by a call to ***pev_map_alloc()***. The field *sg_id* and *offset* of the ***pev_ioctl_map_pg*** data structure are used by the pev driver to identify the window to be modified.

Parameters

map_p	pointer to a ***pev_ioctl_map_pg*** data structure initialized with the mapping parameters.	
	map_p->offset	Offset (as returned by ***pev_map_alloc()***) of mapping window to be modified
	map_p->sg_id	Identifier of mapping table to be modified
	map_p->rem_addr *map_p->size* *map_p->mode*	Parameters to be modified (see ***pev_map_alloc()***). Beware that the rem_addr and size must be compatible with the window size previously allocated.

Return

0	Mapping window successfully freed
-1	Invalid mapping window parameters

4.1.4.4.4 Getting a list of all allocated address translation windows

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   int
   pev_map_read( struct pev_ioctl_map_ctl *ctl_p)
or
   pevx_map_read( int crate, struct pev_ioctl_map_ctl *ctl_p)

Description
   That function updates the pev_ioctl_map_ctl data structure with the
   control parameters of the translation table identified by the sg_id field.

Parameters
   ctl_p   pointer to a  pev_ioctl_map_ctl data structure.

           ctl_p->sg_id    Identifier of mapping table

           ctl_p->map      Pointer to a pev_map_blk[] data structure big enough
                           to hold a full image of the address translation
                           table. The size of that table shall be equal to
                           ctl_p->pg_num * ctl_p->pg_size
                           If that pointer is set to NULL, only control fields
                           in the  pev_ioctl_map_ctl data structure are
                           updated.


Return

   0       pev_ioctl_map_pg data structure pointed by ctl_p parameter has been
           updated with the table control parameters.

           ctl_p->pg_num   Number of pages in the address translation table

           ctl_p->pg_size  Page size

           ctl_p->loc_addr Local address of the first address translation
                           window in the table.

   -1      If table identifier is invalid
```

4.1.4.4.5 Clearing all allocated address translation windows

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   int
   pev_map_clear( struct pev_ioctl_map_ctl *ctl_p);
or
   pevx_map_clear( int crate, struct pev_ioctl_map_ctl *ctl_p);

Description
```

That function clears all mapping window previously allocated by a call to ***pev_map_alloc()***. The field *sg_id* is used by the pev driver to identify the table to be cleared.

Parameters

ctl_p	pointer to a ***pev_ioctl_map_ctl***.	
	ctl_p->sg_id	Identifier of mapping table

Return

0	If successful
-1	If table identifier unknown

4.1.4.4.6 Mapping an allocated address translation window in user's space

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   void
   *pev_mmap( struct pev_ioctl_map_pg *map_p)
or
   *pevx_mmap( int crate, struct pev_ioctl_map_pg *map_p)
```

Description

That function maps in user's space an address translation window previously allocated by a call to ***pev_map_alloc()***. The ***mmap()*** sytem call is performed with *map_p->rem_addr as offset parameter and map_p->size as length parameter. The prot and flags parameter are set to* ***PROT_READ|PROT_WRITE*** *and* ***MAP_SHARED***.

Parameters

map_p	pointer to a ***pev_ioctl_map_pg*** data structure initialized with the mapping parameters.	
	map_p->sg_id	Identifier of mapping table to be modified
	map_p->loc_addr	Passed as offset parameter to ***mmap()***
	map_p->size	Passed as length parameter to ***mmap()***

Return

addr	Address returned by ***mmap()*** system call (see Linux man pages)

4.1.4.5 DMA operations

Buffer suitable for DMA access shall be allocated using special kernel function to make sure they are locked

in memory and visible from the PCI Express. This operation is done by the device driver using the PEV_IOCTL_BUF_ALLOC ioctl() command.

Once a buffer has been allocated, its kernel address can be used to map it in user's space using the /dev/kmem device. That operation can only be done in superuser mode.

4.1.4.5.1 Allocating a buffer in system memory suitable for DMA

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   int
   pev_buf_alloc( struct pev_ioctl_buf *buf_p)
or
   pevx_buf_alloc( int crate, struct pev_ioctl_buf *buf_p)
```

Description

That function updates the ***pev_ioctl_buf*** data structure with the parameters of the buffer allocated by the kernel in system memory.

Parameters

buf_p pointer to a ***pev_ioctl_buf*** data structure.

buf_p->size Size of the buffer to be allocated

Return

0 ***pev_ioctl_buf*** data structure pointed by buf_p parameter has been updated with the buffer parameters.

buf_p->k_addr Buffer kernel address (suitable for subsequent call to mmap() function using /dev/kmem device)

buf_p->b_addr Buffer bus address (suitable for DMA parameters)

-1 Buffer was not allocated

4.1.4.5.2 Freeing an allocated buffer

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   void
   pev_buf_free( struct pev_ioctl_buf *buf_p)
or
   pevx_buf_free( int crate, struct pev_ioctl_buf *buf_p)
```

Description

That function free a buffer previously allocated by a call to ***pev_buf_alloc()***.

Parameters

buf_p pointer to a ***pev_ioctl_buf*** data structure initialized with the buffer parameters.

buf_p->k_addr Kernel address (as returned by ***pev_buf_alloc()***) of the buffer to be freed

buf_p->b_addr Bus address (as returned by ***pev_buf_alloc()***) of the buffer to be freed

buf_p->size Size (as returned by ***pev_buf_alloc()***) of the buffer to be freed

Return

0 Buffer successfully freed

-1 Invalid buffer parameters

4.1.4.5.3 Performing a DMA transfer

Prototype

```
#include <pevulib.h>
or
#include <pevxulib.h>
#include <pevioctl.h>

void
pev_dma_move( struct pev_ioctl_dma_req *req_p)
or
pevx_dma_move( int crate, struct pev_ioctl_dma_req *req_p)
```

Description

This is the generic function to *perform DMA transfers. Transfer parameters shall be loaded in the* ***pev_ioctl_dma_req*** *data structure*.

Parameters

req_p pointer to a ***pev_ioctl_dma_req*** data structure initialized with the transfer parameters.

req_p->des_addr DMA transfer destination address

req_p->des_space DMA transfer destination space

```
DMA_SPACE_PCIE → PCI Express
DMA_SPACE_VME  → VME bus
DMA_SPACE_SHM  → shared memory
DMA_VME_A16    → VME A32 single transfer
DMA_VME_A24    → VME A32 single transfer
DMA_VME_A32    → VME A32 single transfer
DMA_VME_BLT    → VME block transfer
DMA_VME_MBLT   → VME MBLT
DMA_VME_2eVME  → VME 2eVME
```

```
                DMA_VME_2eFAST → VME 2eVME fast
                DMA_VME_2e160  → VME 2eSST 160 MBytes/sec
                DMA_VME_2e233  → VME 2eSST 233 MBytes/sec
                DMA_VME_2e320  → VME 2eSST 320 MBytes/sec

          req_p->src_addr   DMA transfer source address

          req_p->src_space  DMA transfer source space

          req_p->size       DMA transfer size (in bytes)

          req_p->start_mode DMA starting mode

                DMA_MODE_BLOCK  → block transfer
                DMA_MODE_PIPE   → read/write are pipelined (only for VME)

          req_p->intr_mode  Interrupt mode

                DMA_INTR_ENA    → enable interrupt

          req_p->wait_mode  Wait mode

                 DMA_WAIT_INTR  → wait for interrupt


Return
   0       DMA transfer ended

   -1      Invalid transfer parameters
```

4.1.4.5.4 Reading a list of buffer from VME using DMA

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>
   #include <pevioctl.h>

   void
   pev_dma_vme_list_rd( void *uaddr,
                        struct pev_ioctl_dma_list *list_p,
                        int size)
or
   pevx_dma_vme_list_rd( int crate,
                         void *uaddr,
                         struct pev_ioctl_dma_list *list_p,
                         int list_size)

Description
   That function takes as input a list of VME read transfer to be executed by
   the DMA engine. Data collected are pushed sequentially  in a buffer
   provided by the application.
   Any VME address boundary and size are supported in the list parameters.
   However, physical transfers on the VME are always performed with 64 bits (8
   bytes) alignment. Unrequested data are then trowed away before filling the
   user's buffer.
   The function returns when all requested data have been copied in the user's
   buffer.

Parameters
```

uaddr Pointer (void *) to the destination address (buffer address in user's space)

list_p pointer to a list of ***pev_ioctl_dma_list*** data structures initialized with the VME transfer parameters.

list_p->addr DMA transfer source address (VME address)

list_p->size DMA transfer size (in bytes)

list_p->mode DMA transfer mode (VME Address Modifier)

DMA_VME_A16 → *VME A32 single transfer*
DMA_VME_A24 → *VME A32 single transfer*
DMA_VME_A32 → *VME A32 single transfer*
DMA_VME_BLT → *VME block transfer*
DMA_VME_MBLT → *VME MBLT*
DMA_VME_2eVME → *VME 2eVME*
DMA_VME_2eFAST → *VME 2eVME fast*
DMA_VME_2e160 → *VME 2eSST 160 MBytes/sec*
DMA_VME_2e233 → *VME 2eSST 233 MBytes/sec*
DMA_VME_2e320 → *VME 2eSST 320 Mbytes/sec*

list_size *VME list size (number of elements in the list pointed by **list_p**)*

Return

0 DMA transfer ended

-1 Invalid transfer parameters

4.1.4.6 VME Interface Configuration

4.1.4.6.1 Reading the VME current configuration

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_vme_conf_read( struct pev_ioctl_vme_conf *vc_p)
or
   pevx_vme_conf_read( int crate, struct pev_ioctl_vme_conf *vc_p)
```

Description

Read the PEV1100 VME interface current configuration.
All hardware registers related to the VME interface configuration are read and the data structure pointed by ***vc_p*** is updated accordingly.

Parameters

vc_p Pointer to a pev_ioctl_vme_conf data structure

Return

0 The ***pev_ioctl_vme_conf*** data structure pointed by parameter vc_p has

```
          been updated with the current status of vme interface
          vc_p->a24_base    VME base address of A24 slave window (CR/CSR)
          vc_p->a24_size    Size of A24 slave window (CR/CSR)
          vc_p->a32_base    VME base address of A32 slave window
          vc_p->a32_size    Size of A32 slave window
          vc_p->x64         Set to 1 if VME64x enabled, else 0
          vc_p->slot1       Set to 1 if System Controller enabled, else 0
          vc_p->sysrst      Set to 1 if SYSRST transmission enabled, else 0
          vc_p->rto         Retry time out in usec (512 usec or disabled)
          vc_p->arb         VME arbitration mode (valid if Slot 1)
          vc_p->bto         VME bus time out in usec (valid if Slot 1)
          vc_p->req         VME request mode
          vc_p->level       VME request level
          vc_p->mas_ena     Set to 1 if VME master enabled, else 0
          vc_p->slv_ena     Set to 1 if VME slave enabled, else 0
          vc_p->slv_retry   If set to 1, do not use VME Slave Retry
          vc_p->burst       Write postin burst size (VME slave)
```

4.1.4.6.2 Setting a new VME Configuration

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_vme_conf_read( struct pev_ioctl_vme_conf *vc_p)
or
   pevx_vme_conf_read( int crate, struct pev_ioctl_vme_conf *vc_p)
```

Description

Sets the hardware registers controlling the VME interface according to the parameters found in the ***pev_ioctl_vme_conf*** data structure. To change only a subset of the configuration parameters, first call ***pev_vme_conf_read()*** to fill the data structure with the current configuration, modify it and call ***pev_vme_conf_write()***.
To map the VME slave window with a 1 Mbytes granularity, bit 3 of the ***slv_ena*** field shall be set to 1.

Parameters

```
   vc_p    vc_p shall point to a pev_ioctl_vme_conf data structure used to
           update the hardware registers controlling the VME interface
           vc_p->a32_base    VME base address of A32 slave window
           vc_p->a32_size    Size of A32 slave window
```

```
            vc_p->rto          Retry time out in usec (512 usec or disabled)
            vc_p->arb          VME arbitration mode (valid if Slot 1)
            vc_p->bto          VME bus time out in usec (valid if Slot 1)
            vc_p->req          VME request mode
            vc_p->level        VME request level
            vc_p->mas_ena      Set to 1 if VME master enabled, else 0
            vc_p->slv_ena      Set bit 0 to 1 to enable  VME slave, else set to 0
                               Set bit 3 to 1 to enable 1 Mbyte granularity
            vc_p->slv_retry    If set to 1, do not use VME Slave Retry
            vc_p->burst        Write posting burst size (VME slave)



Return
   0
```

4.1.4.7 PEV1100 SFLASH Access

The ***pev_sflash_xxx()*** set of functions allows to read/write from/to the SFLASH device. Be aware that the SFLASH device is accessed bit per bit from the CPU through a serial link. Operations are then very slow and consume all CPU resources.This means that during SFLASH operation the linux system will be almost frozen. Programing one sector (256 kbytes) takes about 15 seconds.

4.1.4.7.1 SFLASH identification

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>

   int
   pev_sflash_id( char *id)
or
   pevx_sflash_id( int crate, char *id)

Description
   Fill the string pointed by id with the SFLASH 3 bytes hardware identifier

Parameters
   id     pointer to a 4 bytes string

Return
   0  if OK
   -1 if error
```

4.1.4.7.2 SFLASH read

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_sflash_read( uint offset,
                    void *addr,
                    uint len)
or
   pevx_sflash_read( int crate,
                     uint offset,
                     void *addr,
                     uint len)
```

Description

Copy ***len*** byte from a data buffer pointed by ***addr*** in SFLASH at offset ***offset***.

Parameters

offset	offset in SFLASH where the data are to be read from
addr	address of the buffer where the data are to be copied
len	number of bytes to by copied

Return

0	in case of success
-1	in case of error

4.1.4.7.3 SFLASH write

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_sflash_write( uint offset,
                     void *addr,
                     uint len)
or
   pevx_sflash_write( int crate,
                      uint offset,
                      void *addr,
                     uint len)
```

Description

Copy ***len*** byte from a data buffer pointed by ***addr*** in SFLASH at offset ***offset***.

Parameters

offset	offset in SFLASH where the data are to be copied
addr	address of the buffer containing the data to be copied
len	number of bytes to by copied

Return

```
   0      in case of success
   -1     in case of error
```

4.1.4.8 PEV1100 timer functions

The PEV1100 FPGA implements a local timer driven by one of the following synchronization sources:

- a 1 KHz local clock
- VME SYSFAIL
- VME IRQ1
- VME IRQ2

The main counter is incremented by the selected synchronization source giving the main tick. The occurrence of the tick can generate an interrupt to the local host.

A free running micro timer driven by a local clock (programmable frequency from 1 MHz to 125 MHz) is reset to 0 at each occurrence of the the main tick .

When the main counter is read, the current value of the micro timer is catch in order to provide an accurate (up to 8 nsec resolution) timing measurement.

4.1.4.8.1 Starting the timer

Prototype

```
   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_timer_start( int mode,
                    int msec)
or
   pevx_timer_start( int crate,
                     int mode,
                     int msec)
```

Description

Start the PEV1100 local timer in an operating mode defined by the *mode* parameter. The main counter is preloaded with the value of *msec* parameter.

Parameters

mode Timer mode of operation. Allow to set the synchronization source, frequency and output mode.

TMR_FREQ_1MHZ	→ micro timer frequency set to 1 MHz
TMR_FREQ_5MHZ	→ micro timer frequency set to 5 MHz
TMR_FREQ_25MHZ	→ micro timer frequency set to 25 MHz
TMR_FREQ_100MHZ	→ micro timer frequency set to 100 MHz
TMR_SRC_LOC	→ main timer sync source is 1 KHz local clock
TMR_SRC_SYSFAIL	→ main timer sync source is VME SYSFAIL
TMR_SRC_IRQ1	→ main timer sync source is VME IRQ1
TMR_SRC_IRQ2	→ main timer sync source is VME IRQ2
TMR_OUT_SYSFAIL	→ timer output drives VME SYSFAIL
TMR_OUT_IRQ1	→ timer output drives VME IRQ1
TMR_OUT_IRQ2	→ timer output drives VME IRQ2

```
   msec    Main timer initial value


Return
   None
```

4.1.4.8.2 Stopping the timer

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_timer_stop( void)
or
   pevx_timer_stop( int crate)

Description
   Stop the PEV1100 local timer.

Parameters
   None

Return
   None
```

4.1.4.8.3 Reading the timer

```
Prototype

   #include <pevulib.h>
or
   #include <pevxulib.h>

   void
   pev_timer_read( struct pev_time *tm_p)
or
   pevx_timer_read( int crate, struct pev_time *tm_p)

Description
   Read the PEV1100 local timer current value. The current value of the two
   counters (msec and usec) are returned in the time and utime fields of the
   pev_time data structure.

Parameters
   tm_p    Pointer to a pev_time data structure


Return
```

```
   time    The pev_time data structure pointed by parameter tm_p has been
           updated with the current value of timer counters. The current value
           of the main counter is returned.

           tm_p->time        Current value of the main counter

           tm_p->utime       Current value of the micro counter
```

4.1.5 Examples

4.1.5.1 Accessing a VME device from the host

The following examples performs a kind of VME loopback test by accessing the PEV1100 shared memory from its VME master port through its VME slave port. All physical accesses to the shared memory are performed by the application in user's mode.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/time.h>

typedef unsigned int u32;
#include <pevioctl.h>
#include <pevulib.h>
```

pevulib.h contains the function declaration of the PEV1100 user's library and pevioctl.h the declaration of the data structure used to perform the address mapping operations.

```
static float tst_read( void *);
static float tst_write( void *);
```

Declare two local functions to perform VME read and write test

```
struct pev_node *pev;
struct pev_ioctl_map_pg shm_mas_map;
struct pev_ioctl_map_pg vme_mas_map;
struct pev_ioctl_map_pg vme_slv_map;
struct pev_ioctl_vme_conf vme_conf;
struct timeval ti, to;
struct timezone tz;
```

The ***pev*** pointer holds the return value of the ***pev_init()*** call.

Three data structure are defined to hold address mapping information

- ***shm_mas_map*** → direct mapping to the Shared Memory
- ***vme_slv_map*** → VME slave mapping to the Shared Memory
- ***vme_mas_map*** → master mapping to the VME slave address pointing the Shared Memory

The ***vme_conf*** data structure holds the PEV1100 VME interface current configuration.

```
main( int argc,
      void *argv[])
{
```

```
void *shm_loc_addr, *shm_vme_addr;
int i, data, *p;
long vme_addr;
long dt, dt1, dt2;
float usec;
uint crate;

printf("Entering VME test program\n");

/* call PEV1100 user library initialization function */
crate = 0;
pev = pev_init( crate);
if( !pev)
{
  printf("Cannot allocate data structures to control PEV1100\n");
  exit( -1);
}
/* verify if the PEV1100 is accessible */
if( pev->fd < 0)
{
  printf("Cannot find PEV1100 interface\n");
  exit( -1);
}
```

The first call to be performed is ***pev_init()*** in order to allow the user's library to access the PEV1100 device driver.

```
/* get the current VME configuration */
pev_vme_conf_read( &vme_conf);
printf("VME A32 base address = 0x%08x [0x%x]", vme_conf.a32_base, vme_conf.a32_size);
if( vme_conf.mas_ena)
{
  printf(" -> enabled\n");
}
else
{
  printf(" -> disabled\n");
}
```

```
/* create an address translation window in the VME slave port */
/* pointing to the PEV1100 Shared Memory                      */

vme_slv_map.rem_addr = 0x000000; /* shared memory base address */
vme_slv_map.mode = MAP_ENABLE|MAP_ENABLE_WR|MAP_SPACE_SHM;
vme_slv_map.flag = 0x0;
vme_slv_map.sg_id = MAP_SLAVE_VME;
vme_slv_map.size = 0x100000;
pev_map_alloc( &vme_slv_map);

/* calculate the VME base address at which the Shared Memory has been mapped */
vme_addr = vme_conf.a32_base + vme_slv_map.loc_addr;
printf("shared Memory is visible at VME A32 address 0x%08x\n", vme_addr);
```

```
/* create an address translation window in the PCIe End Point */
/* pointing to the VME address at which the Shared Memory has been mapped  */
vme_mas_map.rem_addr = vme_addr;
vme_mas_map.mode = MAP_ENABLE|MAP_ENABLE_WR|MAP_SPACE_VME|MAP_VME_A32;
vme_mas_map.flag = 0x0;
vme_mas_map.sg_id = MAP_MASTER_32;
vme_mas_map.size = 0x100000;
pev_map_alloc( &vme_mas_map);
```

```
  printf("offset in PCI MEM window to access SHM throug VME : %p\n",
vme_mas_map.loc_addr);

  printf("perform the mapping in user's space");
  shm_vme_addr = pev_mmap( &vme_mas_map);
  printf("%p\n", shm_vme_addr);
  if( shm_vme_addr == MAP_FAILED)
  {
    printf("Failed\n");
    goto VmeTst_exit;
  }
  printf("Done\n");
```

```
  /* create an address translation window in the PCIe End Point */
  /* pointing to the PEV1100 local address of the Shared Memory */
  shm_mas_map.rem_addr = 0x000000; /* shared memory base address */
  shm_mas_map.mode = MAP_ENABLE|MAP_ENABLE_WR|MAP_SPACE_SHM;
  shm_mas_map.flag = 0x0;
  shm_mas_map.sg_id = MAP_MASTER_32;
  shm_mas_map.size = 0x100000;
  pev_map_alloc( &shm_mas_map);

  printf("local address = %p\n", shm_mas_map.loc_addr);
  printf("offset in PCI MEM window to access SHM locally : %p\n", shm_mas_map.loc_addr);


  printf("perform the mapping in user's space : ");
  shm_loc_addr = pev_mmap( &shm_mas_map);
  printf("%p", shm_loc_addr);
  if( shm_loc_addr == MAP_FAILED)
  {
    printf(" ->Failed\n");
    goto VmeTst_exit;
  }
  printf(" -> Done\n");
```

```
  usec = tst_read( shm_vme_addr);
  printf("VME read cycle %4f usec\n", usec);
  usec = tst_write( shm_vme_addr);
  printf("VME write cycle %4f usec\n", usec);
  usec = tst_read( shm_loc_addr);
  printf("SHM read cycle %4f usec\n", usec);
  usec = tst_write( shm_loc_addr);
  printf("SHM write cycle %4f usec\n", usec);
```

```
VmeTst_exit:
  pev_munmap( &shm_mas_map);
  pev_map_free( &shm_mas_map);
  pev_munmap( &vme_mas_map);
  pev_map_free( &vme_mas_map);
  pev_map_free( &vme_slv_map);

  pev_exit( pev);

  exit(0);
}
```

```
float
tst_read( void *addr)
{
  int i, data;
  int *s, *d;
  long dt1, dt2;

  gettimeofday( &ti, &tz);
  s = (int *)addr;
  for( i = 0; i < 0x10000; i++)
  {
    data = *s++;
  }
  gettimeofday( &to, &tz);
  dt1 = ((to.tv_sec -  ti.tv_sec) * 1000000) + ( to.tv_usec - ti.tv_usec);
  gettimeofday( &ti, &tz);
  s = (int *)addr;
  for( i = 0; i < 0x20000; i++)
  {
    data = *s++;
  }
  gettimeofday( &to, &tz);
  dt2 = ((to.tv_sec -  ti.tv_sec) * 1000000) + ( to.tv_usec - ti.tv_usec);
  return( (float)(dt2-dt1)/0x10000);
}
```

```
float
tst_write( void *addr)
{
  int i, data;
  int *s, *d;
  long dt1, dt2;

  data = 0xa5a5a5a5;
  gettimeofday( &ti, &tz);
  d = (int *)addr;
  for( i = 0; i < 0x10000; i++)
  {
    *d++ = data;
  }
  gettimeofday( &to, &tz);
  dt1 = ((to.tv_sec -  ti.tv_sec) * 1000000) + ( to.tv_usec - ti.tv_usec);
  gettimeofday( &ti, &tz);
  d = (int *)addr;
  for( i = 0; i < 0x20000; i++)
  {
    *d++ = data;
  }
  gettimeofday( &to, &tz);
  dt2 = ((to.tv_sec -  ti.tv_sec) * 1000000) + ( to.tv_usec - ti.tv_usec);
  return( (float)(dt2-dt1)/0x10000);
}
```

4.1.6 PEV1100 Command Interpreter

4.1.6.1 Introduction

XprsMon is a Linux application linked with the PEV1100 user's library. It allows the user to perform interactively a set of command to operate the PEV1100 interface. Once the ***pev*** driver has been inserted in the kernel, ***XprsMon*** is very useful to setup and debug a VME configuration.

XprsMon command syntax is as follow:

cmd.ext para#1:m#1 para#2:m#2 …

where

- **cmd** is the command name
- **ext** is a command extension (optional)
- **para#i** are the command parameters
- **m#i** are the parameter modifiers (optional)

The ***help*** command displays a list of all allowed command names.

XprsMon parameters can be strings, integers, ranges or data set. By default integers are interpreted and displayed as hexadecimal numbers.

Some commands takes address range and data set parameters. Ranges shall be in the form of

start..end

or

start..end:space

where

- **start** and **end** are integers
- **space** is an address space identifier

Data set parameter shall be in the form of

c:data..para

where

- **c** is a ascii character identifying the set
- **data** and **para** are integers

XprsMon is started by invoking the **XprsMon** binary file from the Linux shell with the node number as parameter.

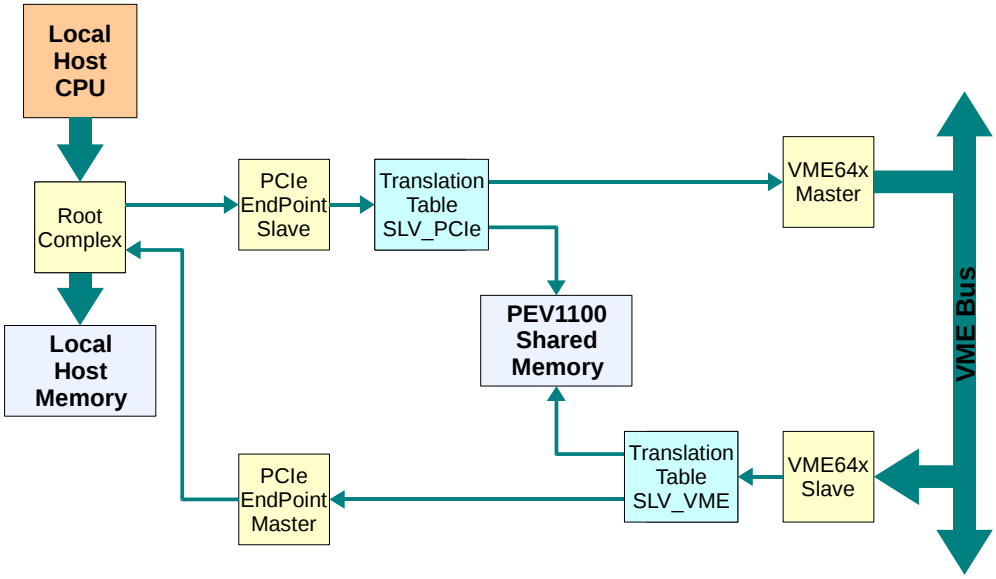
```
linux-host:->XprsMon 2

     +----------------------------------------+
     |    XprsMon - PEV1100 diagnostic tool   |
     |    IOxOS Technologies Copyright 2009   |
     +----------------------------------------+

initializing crate 2
Device driver: pev-linux
XprsMon#2>
```

During its initialization phase, **XprsMon** tries to opens the device ***/dev/pev*** in order to access the PEV1100. If the open system call fails, **XprsMon** exits.

In case of success, the following operations are performed:

- allocate a 1 Mbyte window in the PCI MEM space to access the PEV1100 Shared Memory
- allocate a 1 Mbyte window in the PCI MEM space to access the VME bus
- allocate a 1 Mbyte data buffer in kernel space for DMA access to the Local Host Memory
- allocate a 1 Mbyte window in the VME A32 slave to access the PEV1100 shared memory
- allocate a 1 Mbyte window in the VME A32 slave to access the Local Host Memory (data buffer

XprsMon provides a set of command to perform CPU read/write accesses from/to these spaces


Local Host CPU
Root Complex
Local Host Memory
PCIe EndPoint Slave
Translation Table SLV_PCIe
PEV1100 Shared Memory
VME64x Master
VME Bus
Translation Table SLV_VME
VME64x Slave
PCIe EndPoint Master

4.1.6.2 Command List

Command	Description	Reference
conf	show PEV1100 configuration	$4.1.6.2.1
dm	PEV1100 address mapping operations	$4.1.6.2.2
dma	data transfer using DMA	$4.1.6.2.3
dp	display data buffer from PEV1100 PCI MEM window	$4.1.6.2.4
ds	display data buffer from PEV1100 shared memory	$4.1.6.2.5
dv	display data buffer from VME address space	$4.1.6.2.6
fm	fill data buffer in system memory	$4.1.6.2.7
fp	fill data buffer in PEV1100 PCI MEM space	$4.1.6.2.8
fs	fill data buffer in PEV1100 shared memory	$4.1.6.2.9
fv	fill data buffer in VME address space	$4.1.6.2.10
help	display list of commands or syntax of command <cmd>	$
i2c	"perform i2c command	$
ls	read/write loop from/to PEV1100 shared memory	$
lv	read/write loop from/to VME address space	$
map	PEV1100 address mapping operations	$4.1.6.2.11
pc	read/write data from/to PEV1100 PCI configuration space	$4.1.6.2.12
pio	read/write data from/to PEV1100 PCI IO space	$4.1.6.2.13
pm	read/write data from/to PEV1100 system memory	$
pp	read/write data from/to PEV1100 PCI MEM space	$4.1.6.2.14
pr	read/write data from/to PEV1100 register	$
ps	read/write data from/to PEV1100 shared memory	$
pv	read/write single data from/to VME address space	$4.1.6.2.15
px	read/write data from/to PEX86XX registers	$4.1.6.2.16
lql		$
sflash	sflash operation	$
ts	perform read/write test on shared memory	$
tv	perform read/write test on VME bus	$
timer	perform operation on PEV1100 internal timer	$4.1.6.2.18
tinit	launch test suite	$4.1.6.2.19
tkill	kill test suite	$
tlist	display a list of existing test	$
tset	set test control parameter	$
tstart	start execution of a test or a chain of tests	$
tstop	stop current test execution	$
tty	send string to ttyUSB0	$4.1.6.2.21
vme	configure vme interface	$4.1.6.2.20

Table 14: XprsMon Command List

4.1.6.2.1 Show board configuration

```
Command
   conf →  display board hardware configuration

Synopsis
   conf <operation> <resource>

      where <operation> operation to be perform
               show  → display resource current configuration
            <resource> string identifying the PEV1100 internal resource
               all     → all avialable resources
               fpga    → FPGA
               shm     → on board shared memory
               smon    → FPGA system monitoring
               switch  → Front End PCIe switch (PEX8624)
               static  → static options (switches and jumpers)
               vme     → VME64x interface

Description
   The conf command allows to display the current configuration of all
   hardware resources available on the PEV1100 board.

Examples
   XprsMon#2>conf show all
   PEV1100 Configuration
      Static Options [0x002612d2]
         VME Interface
            A24 Base Address  : d00000
            System Controller : 64x- Slot1+ SysRstEna-
         PLX8624 Switch
            Port0 [P3]        : Upstream : External Clock
            Port1 [P4]        : Non Transparent
            Port5 [FPGA]      : Downstream
            Port6 [PCI]       : Downstream
            Port8 [XMC#1]     : Downstream
            Port9 [XMC#2]     : Downstream
         FPGA
            Bit Stream        : 1
            PON FSM           : Disabled
            MEM size          : 128 MBytes
            PMEM size         : 128 MBytes
            PMEM mode         : A32
      PCIe SWITCH Status
         Identifier           : 0x862410b5
      FPGA Status
         Identifier           : 0x06050901
         Bit stream loaded    : 1
      Shared Memory
         Size                 : 512 MBytes
      VME Interface
         System Controller    : Enabled
            Arbtration mode   : PRI not pipelined
            Bus Timeout       : 16 usec
         Master               : Enabled
            Request Mode      : Release On Request
            Request Level     : 0
         Slave                : Enabled
            A24 base address  : 0xd00000
            A32 base address  : 0x20000000
            A32 window size   : 0x10000000
            CR/CSR            : Berr- SlvEna+ SysFail- SysFailEna- Reset-
         Interrupt Generator
```

```
            Vector            : 00
            Level             : 0
            Mode              : Register
            Status            : Cleared
      FPGA System Monitor
         Temperature          : 50.20 [50.20 - 28.05]
         VCCint               : 1.00 [1.00 - 1.00]
         VCCaux               : 2.52 [2.52 - 2.51]
         VCC1.8-INT           : 1.79
         VCC3.3-INT           : 3.26
         VCC5.0-VME           : 5.02
         VCC3.3-VME           : 3.36
XprsMon#2>
```

4.1.6.2.2 Display address range from system memory

Command

dm → display content of System Memory

Synopsis

dm.<ds>[<swap>] <start>[..<end>]

where <ds> = b,s,w, (data size 1,2,4)
<swap> = s if swapping is reqired
<start> = start address in hexadecimal
<end> = end address in hexadecimal

Description

The **dm** command shall be used to display the content of the kernel buffer allocated in system memory during the XprsMon initialization phase. This buffer is provided to perform test with the VME slave interface and to offer DMA engines an access to the system memory.
The command extensions is made of 1 or 2 characters. The first character can be:

- **b** → for byte access
- **c** → for 16 bit access
- **w** → for 32 bit access
- **l** → for 64 bit access

The second character (optional) shall be **s** if the data are to be displayed byte swapped (meaningful only for 16, 32 and 64 bits accesses).
The first parameter shall be a range of address offsets whose content are to be displayed according to the data size chosen. Address offsets are offset related to the base address of the allocated buffer.

Examples

```
XprsMon#2>dm.b 0
0x00000000 : 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ........
0x00000010 : 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f  ........
0x00000020 : 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f   !"#$%&'
0x00000030 : 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f  01234567
XprsMon#2>
```

4.1.6.2.3 Perform DMA operation

Command

dma → perform a data transfer using DMA engines

Synopsis

```
dma start <des_addr>:<des_space>  <src_addr>:<src_space> <size>

   where <des_addr>  = address offset in the destinatation space
         <des_space> = destination space identifier
         <src_addr>  = address offset in the source space
         <src_space> = source space identifier
         <size>      = transfer size in byte
```

Description

The **dma** command allows to move data from a source address space identified by <src_space> to a destination address space identified by <des_space>. Allowed address space identifiers are:

- 0 → PCI memory space
- 2 → Shared Memory
- 3 → FPGA user's space
- 31 → VME A32
- 41 → VME BLT
- 51 → VME MBLT
- 61 → 2eVME
- 71 → 2eVME Fast
- 81 → 2eSST 160
- 91 → 2eSST 233
- a1 → 2eSST 320

The <src_addr> and <des_addr> parameters are the address offsets in the source and destination spaces.

Examples

Move 64 Kbytes from the VME address 0x80000000 to the kernel buffer allocated by XprsMon using VME MBLT transfer mode.

```
XprsMon>dma start 0:0 80000000:51 10000
```

4.1.6.2.4 Display PCI MEM address range

Command

dp → display content of PCI addresses (MEM space)

Synopsis

```
dp.<ds>[<swap>] <start>[..<end>]

   where <ds> = b,s,w, (data size 1,2,4)
         <swap> = s if swapping is required
         <start> = start offset in hexadecimal
         <end> = end offset in hexadecimal
```

Description

The **dp** command shall be used to perform address cycle through the PCI MEM space window of the PEV1100 FPGA PCIe End Point. It takes an address offset range as parameter. The actual location from where the data will be read depends on the initialization of the address translation table associated to the FPGA PCI MEM window. That command is useful to display the content of address ranges in resources mapped by other application. The **map** command (see §4.1.6.2.11) shall be used to check the actual address mapping.
The command extensions is made of 1 or 2 characters. The first character can be:

- **b** → for byte access
- **c** → for 16 bit access
- **w** → for 32 bit access

```
    l → for 64 bit access
  The second character (optional) shall be s if the data are to be displayed
  byte swapped (meaningful only for 16, 32 and 64 bits accesses).
  The first parameter shall be a range of address offsets whose content shall
  be displayed. Address offsets are offset related to the PCI base address of
  the FPGA PCI MEM window. Physical cycles are performed according to the
  data size chosen.
  Basically that command allows to display content of remote addresses
  already mapped (by other application) through the mas_32 mapping table. The
  map show command shall be used to display existing mapping and address
  offsets to be used.


Examples
  XprsMon#2>dp.w 100000
  0x00100000 : 00000000 00000004 00000008 0000000c  ................
  0x00100010 : 00000010 00000014 00000018 0000001c  ................
  0x00100020 : 00000020 00000024 00000028 0000002c   ...$...(...,...
  0x00100030 : 00000030 00000034 00000038 0000003c  0...4...8...<...
  XprsMon#2>
```

4.1.6.2.5 Display Shared Memory address range

```
Command
  ds →  display content of Shared Memory

Synopsis
  ds.<ds>[<swap>] <start>[..<end>]

     where <ds> = b,s,w, (data size 1,2,4)
           <swap> = s if swapping is required
           <start> = start address in hexadecimal
           <end> = end address in hexadecimal

Description
  The ds command shall be used to display the content of shared memory
  address ranges.
  The command extensions is made of 1 or 2 characters. The first character
  can be:
    b → for byte access
    c → for 16 bit access
    w → for 32 bit access
    l → for 64 bit access
  The second character (optional) shall be s if the data are to be displayed
  byte swapped (meaningful only for 16, 32 and 64 bits accesses).
  The first parameter shall be a range of address offsets whose content shall
  be displayed according to the data size chosen. Address offsets are offset
  related to the base address of the PEV1100 on board shared memory.

Examples
  XprsMon#2>ds.b 0
  0x00000000 : 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ........
  0x00000010 : 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f  ........
  0x00000020 : 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f   !"#$%&'
  0x00000030 : 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f  01234567
  XprsMon#2>
```

4.1.6.2.6 Display VME address range

```
Command
   dv →  display content of VME addresses

Synopsis
   dv.<ds>[<swap>] <start>[..<end>] m:<mode>

      where <ds> = b,s,w, (data size 1,2,4)
            <swap> = s if swapping is required
            <start> = start address in hexadecimal
            <end> = end address in hexadecimal
            <mode> = address mode

Description
   The dv command shall be used to perform PCI configuration cycles targeted
   to the PEV1100 FPGA PCIe End Point. It takes an address range as parameter.
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
     l → for 64 bit access
   The second character is optional and shall be s if byte swapping is
   required (meaningful only for 16, 32 and 64 bits accesses).
   The first parameter shall be a range of VME address whose content shall be
   displayed according to the data size chosen. XprsMon creates dynamically
   the address mapping in order to perform the requested VME cycles.
   The second parameter shall be a string defining the VME access mode:
     crcsr  → generate address modifier for VME64x CR/CSR (0x2f)
     a16    → generate A16 data access address modifier (0x29/0x2d)
     a24    → generate A24 data access address modifier (0x39/0x3d)
     a32    → generate A32 data access address modifier (0x09/0x0d)
     blt    → generate address modifier A32/D32 block transfer (0x0b/0x0f)
     mblt   → generate address modifier A32/D64 block transfer (0x08/0x0c)
     ao24   → generate A24 address only cycle
     ao32   → generate A32 address only cycle
     iack   → generate VME IACK cycle
   If the access mode parameter is missing the one used in the last cycle is
   kept. When XprsMon is launched, a32 is selected by default.
   If the address range parameter is missing, the display continues from the
   the address used in the last VME cycle executed.

Examples
   XprsMon#2>dv.s 40000000..40000020
   0x40000000 : 0100 0302 0504 0706 0908 0b0a 0d0c 0f0e  ................
   0x40000010 : 1110 1312 1514 1716 1918 1b1a 1d1c 1f1e  ................
   XprsMon#2>dv.ss 40000000..40000020
   0x40000000 : 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f  ................
   0x40000010 : 1011 1213 1415 1617 1819 1a1b 1c1d 1e1f  ................
   XprsMon#2>dv
   0x40000020 : 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f  ! #"%$'&)(+*-,/.
   0x40000030 : 3031 3233 3435 3637 3839 3a3b 3c3d 3e3f  1032547698;:=<?>
   XprsMon#2>
```

4.1.6.2.7 Fill System Memory address range

```
Command
   fs →  fill an address range in shared memory with a data set
```

Synopsis

fs.<ds>[<swap>] <start>[..<end>] data

where <ds> = b,s,w, (data size 1,2,4)
<swap> = s if swapping is required
<start> = start offset in hexadecimal
<end> = end offset in hexadecimal
<data> = data set

Description

The **fs** command shall be used to fill a range of address in shared memory with a data set.
The command extensions is made of 1 or 2 characters. The first character can be:

b → for byte access
c → for 16 bit access
w → for 32 bit access
l → for 64 bit access

The second character is optional and shall be **s** if byte swapping is required (meaningful only for 16, 32 and 64 bits accesses).
The first parameter shall be a range of address offsets (relative to the shared memory base address) whose content shall be initialized according to the data size chosen. XprsMon creates dynamically the address mapping in order to access the shared memory.
The second parameters defines the data set to be used to fill the address range. The syntax is made of a set identifier followed by a semi colon and one or more arguments. If the set identifier is missing, the address range is filled with a constant data.

<data> → fill data set with a constant data

else it is filled with a data set

r:<seed> → fill with pseudo random data
s:<start>..<inc> → fill with a ramp
w:<data>..<mask> → fill with a walking pattern

For random data set, the Linux rand() function is used with <seed> parameter as seed for the random generator. The first data of the set is the <seed> parameter
The ramp data set starts with the <start> parameter and <inc> parameter is used as increment to generate the next data in the set.
The walking pattern data set is calculated according to the following formula:

<data> XOR <mask>

To calculate the next data in the set, the mask is left rotated by 1 over the data size. This allows to generate a 32 bit walking 1 with

w:ffffffff..fffffffe

and a walking 0 with

w:ffffffff..1

Examples

```
XprsMon#2>fs.w 0..10000 w:ffffffff..1
XprsMon#2>ds.w 0
0x00000000 : fffffffe fffffffd fffffffb fffffff7  ................
0x00000010 : ffffffef ffffffdf ffffffbf ffffff7f  ................
0x00000020 : ffffffeff fffffdff fffffbff fffff7ff  ................
0x00000030 : fffffefff ffffdfff fffbffff fff7ffff  ................
XprsMon#2>
```

4.1.6.2.8 Fill PCI address range

Command

fp → fill an address range in FPGA PCI MEM space with a data set

Synopsis

fp.<ds>[<swap>] <start>[..<end>] data

where <ds> = b,s,w, (data size 1,2,4)
<swap> = s if swapping is required
<start> = start offset in hexadecimal
<end> = end offset in hexadecimal
<data> = data set

Description

The **fp** command shall be used to fill a range of address in in the FPGA PCI MEM window with a data set. The actual location where the data will be written depends on the initialization of the address translation table associated to the FPGA PCI MEM window. That command is useful to initialize address ranges in resources mapped by other application. The **map** command (see §4.1.6.2.11) shall be used to check the actual address mapping.
The command extensions is made of 1 or 2 characters. The first character can be:

- **b** → for byte access
- **c** → for 16 bit access
- **w** → for 32 bit access
- **l** → for 64 bit access

The second character is optional and shall be **s** if byte swapping is required (meaningful only for 16, 32 and 64 bits accesses).
The first parameter shall be a range of address offsets (relative to the FPGA PCI MEM window base address) whose content shall be initialized according to the data size chosen.
The second parameters defines the data set to be used to fill the address range. The syntax is made of a set identifier followed by a semi colon and one or more arguments. If the set identifier is missing, the address range is filled with a constant data.

<data> → fill data set with a constant data

else it is filled with a data set

- **r:**<seed> → fill with pseudo random data
- **s:**<start>..<inc> → fill with a ramp
- **w:**<data>..<mask> → fill with a walking pattern

For random data set, the Linux rand() function is used with <seed> parameter as seed for the random generator. The first data of the set is the <seed> parameter
The ramp data set starts with the <start> parameter and <inc> parameter is used as increment to generate the next data in the set.
The walking pattern data set is calculated according to the following formula:

<data> XOR <mask>

To calculate the next data in the set, the mask is left rotated by 1 over the data size. This allows to generate a 32 bit walking 1 with

w:ffffffff..fffffffe

and a walking 0 with

w:ffffffff..1

Examples

```
XprsMon#2>fp.w 100000..110000 r:12345678
XprsMon#2>dp.w 100000
0x00100000 : 12345678 c1da14f0 623d80e6 f0cba6c2  xV4.......=b....
0x00100010 : fe38c547 7bf64f5d 1c6df870 3473d32f  G.8.]O.{p.m./.s4
0x00100020 : 397915d9 a23accf0 fc887b38 7abe7811  ..y9..:.8{...x.z
0x00100030 : 314a695b 8e0f2fdc 5e104039 f0307802  [iJ1./..9@.^.x0.
```

```
XprsMon#2>
```

4.1.6.2.9 Fill address range in shared memory

```
Command
   fs →  fill an address range in shared memory with a data set

Synopsis
   fs.<ds>[<swap>] <start>[..<end>] data

      where <ds>    = b,s,w, (data size 1,2,4)
            <swap>  = s if swapping is required
            <start> = start offset in hexadecimal
            <end>   = end offset in hexadecimal
            <data>  = data set

Description
   The fs command shall be used to fill a range of address in shared memory
   with a data set.
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
     l → for 64 bit access
   The second character is optional and shall be s if byte swapping is
   required (meaningful only for 16, 32 and 64 bits accesses).
   The first parameter shall be a range of address offsets (relative to the
   shared memory base address) whose content shall be initialized according to
   the data size chosen. XprsMon creates dynamically the address mapping in
   order to access the shared memory.
   The second parameters defines the data set to be used to fill the address
   range. The syntax is made of a set identifier followed by a semi colon and
   one or more arguments. If the set identifier is missing, the address range
   is filled with a constant data.
     <data>            → fill data set with a constant data
   else it is filled with a data set
     r:<seed>          → fill with pseudo random data
     s:<start>..<inc>  → fill with a ramp
     w:<data>..<mask>  → fill with a walking pattern
   For random data set, the Linux rand() function is used with <seed>
   parameter as seed for the random generator. The first data of the set is
   the <seed> parameter
   The ramp data set starts with the <start> parameter and <inc> parameter is
   used as increment to generate the next data in the set.
   The walking pattern data set is calculated according to the following
   formula:
       <data> XOR <mask>
   To calculate the next data in the set, the mask is left rotated by 1 over
   the data size. This allows to generate a 32 bit walking 1 with
     w:ffffffff..fffffffe
   and a walking 0 with
     w:ffffffff..1


Examples
   XprsMon#2>fs.s 0..10000 s:0..2
   XprsMon#2>ds.s 0
```

```
0x00000000 : 0000 0002 0004 0006 0008 000a 000c 000e  ................
0x00000010 : 0010 0012 0014 0016 0018 001a 001c 001e  ................
0x00000020 : 0020 0022 0024 0026 0028 002a 002c 002e   ."."$.&.(.*.,...
0x00000030 : 0030 0032 0034 0036 0038 003a 003c 003e  0.2.4.6.8.:.<.>.
XprsMon#2>
```

4.1.6.2.10 Fill VME address range

Command

fv → fill a VME address range with a data set

Synopsis

fv.<ds>[<swap>] <start>[..<end>] data [**m:**<mode>]

```
   where <ds>    = b,s,w, (data size 1,2,4)
         <swap>  = s if swapping is required
         <start> = start address in hexadecimal
         <end>   = end address in hexadecimal
         <data>  = data set
         <mode>  = address mode
```

Description

The **fv** command shall be used to fill a range of VME address with a data set.
The command extensions is made of 1 or 2 characters. The first character can be:

- **b** → for byte access
- **c** → for 16 bit access
- **w** → for 32 bit access
- **l** → for 64 bit access

The second character is optional and shall be **s** if byte swapping is required (meaningful only for 16, 32 and 64 bits accesses).
The first parameter shall be a range of VME address whose content shall be initialized according to the data size chosen. XprsMon creates dynamically the address mapping in order to perform the requested VME cycles.
The second parameters defines the data set to be used to fill the address range. The syntax is made of a set identifier followed by a semi colon and one or more arguments. If the set identifier is missing, the address range is filled with a constant data.

- <data> → fill data set with a constant data

else it is filled with a data set

- **r:**<seed> → fill with pseudo random data
- **s:**<start>..<inc> → fill with a ramp
- **w:**<data>..<mask> → fill with a walking pattern

For random data set, the Linux rand() function is used with <seed> parameter as seed for the random generator. The first data of the set is the <seed> parameter
The ramp data set starts with the <start> parameter and <inc> parameter is used as increment to generate the next data in the set.
The walking pattern data set is calculated according to the following formula:

<data> XOR <mask>

To calculate the next data in the set, the mask is left rotated by 1 over the data size. This allows to generate a 32 bit walking 1 with

w:ffffffff..fffffffe

and a walking 0 with

w:ffffffff..1

The third parameter shall be a string defining the VME access mode:

```
     crcsr  → generate address modifier for VME64x CR/CSR (0x2f)
     a16    → generate A16 data access address modifier (0x29/0x2d)
     a24    → generate A24 data access address modifier (0x39/0x3d)
     a32    → generate A32 data access address modifier (0x09/0x0d)
     blt    → generate address modifier A32/D32 block transfer (0x0b/0x0f)
     mblt   → generate address modifier A32/D64 block transfer (0x08/0x0c)
     ao24   → generate A24 address only cycle
     ao32   → generate A32 address only cycle
     iack   → generate VME IACK cycle
   If the access mode parameter is missing the one used in the last VME cycle
   is kept. When XprsMon is launched, a32 is selected by default.

Examples
   XprsMon#2>fv.w 40000000..40010000 w:ffffffff..fffffffe
   XprsMon#2>dv.w 40000000
   0x40000000 : 00000001 00000002 00000004 00000008  ................
   0x40000010 : 00000010 00000020 00000040 00000080  .... ...@.......
   0x40000020 : 00000100 00000200 00000400 00000800  ................
   0x40000030 : 00001000 00002000 00004000 00008000  ..... ...@......
   XprsMon#2>
```

4.1.6.2.11 Managing address mapping

```
Command
   map →  handle address mapping

Synopsis
   map <operation> <map identifier>

      where <operation> operation to be perform
            <map identifier> string identifying the mapping

Description
   The map command shall be used to handle the PEV1100 address mapping tables.
   The show operation displays the PEV1100 current mapping information for the
   table identified by the <map identifier> parameter:
     mas_32  → mapping in PCI MEM space
     mas_64  → mapping in PCI PMEM space
     slv_vme → mapping in VME slave interface
   If no map identifier is given, all maps are shown.
   The clear operation resets all pages in the translation table identified by
   the <map identifier> parameter. This destroy all mapping previously
   performed by any application and should be used only as a last resort
   operation.
Examples
   XprsMon#2>map show

   +=========================================================+
   + Map Name : mas_32
   +----------+----+----+----------+------------------+------+
   |  offset  |flag| usr|   size   |   remote address | mode |
   +----------+----+----+----------+------------------+------+
   | 00000000 | 01 | 01 | 00100000 | 0000000000000000 | 2003 | SHM
   | 00100000 | 01 | 01 | 00100000 | 0000000000000000 | 1303 | VME A32

   +=========================================================+
   + Map Name : mas_64
   +----------+----+----+----------+------------------+------+
   |  offset  |flag| usr|   size   |   remote address | mode |
   +----------+----+----+----------+------------------+------+
```

```
+=========================================================+
+ Map Name : slv_vme
+----------+----+----+----------+------------------+------+
|  offset  |flag| usr|   size   |   remote address | mode |
+----------+----+----+----------+------------------+------+
| 00000000 | 01 | 01 | 00100000 | 0000000000000000 | 2003 | SHM
| 00100000 | 01 | 01 | 00100000 | 0000000002d00000 | 0003 | PCIe

XprsMon#2>map show mas_32

+=========================================================+
+ Map Name : mas_32
+----------+----+----+----------+------------------+------+
|  offset  |flag| usr|   size   |   remote address | mode |
+----------+----+----+----------+------------------+------+
| 00000000 | 01 | 01 | 00100000 | 0000000000000000 | 2003 | SHM
| 00100000 | 01 | 01 | 00100000 | 0000000000000000 | 1303 | VME A32

XprsMon#2>
```

4.1.6.2.12 Patch PCI configuration register

Command

pc → read/write data from/to PEV1100 PCI configuration space

Synopsis

pc.<ds>[<swap>] <reg> [<data>]

where <ds> = b,s,w, (data size 1,2,4)
<swap> = s if swapping is required
<reg> = register offset in hexadecimal
<data> = data in hexadecimal [write cycle]

Description

The **pc** command shall be used to perform PCI configuration cycles targeted to the PEV1100 FPGA PCIe End Point. It takes up to 2 parameters.
The command extensions is made of 1 or 2 characters. The first character can be:

- **b** → for byte access
- **c** → for 16 bit access
- **w** → for 32 bit access

The second character (optional) shall be **s** if the data are to be byte swapped before being written or displayed (meaningful only for 16, 32 and 64 bits accesses).
The first parameter shall be the address offset of the configuration register to be accessed.
The second parameter shall be the data to write in the register. If it is missing, XprsMon performs a read cycle on the register, displays the data and prompt for a new data to overwrite the register. To return to the command interpreter enter the “.” character. If you enter CR (Carriage Return), XprsMon will read the register at <reg>+<ds> and display it current value.

Examples

```
XprsMon#2>pc.s 0
0x00000000 : 7357 ->
0x00000002 : 1100 -> .
XprsMon#2>
```

4.1.6.2.13 Patch PCI address in the FPGA End Point IO space

```
Command
   pio →  read/write data from/to PEV1100 PCI IO space

Synopsis
   pio.<ds>[<swap>] <offset> [<data>]

      where <ds> = b,s,w, (data size 1,2,4)
            <swap> = s if swapping is required
            <offset> = address offset in hexadecimal
            <data> = data in hexadecimal [write cycle]

Description
   The pio command shall be used to perform PCI IO cycles targeted to the
   PEV1100 FPGA PCIe End Point. It takes up to 2 parameters.
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
   The second character (optional) shall be s if the data are to be byte
   swapped before being written or displayed (meaningful only for 16, 32 and
   64 bits accesses).
   The second character is optional and shall be s if byte swapping is
   required (meaningful only for 16 and 32 bits accesses).
   The first parameter shall be the address offset in PCI MEM space to be
   accessed.
   The second parameter shall be the data to write in the address offset. If
   it is missing, XprsMon performs a read cycle on the register, displays the
   data and prompt for a new data to overwrite the register. To return to the
   command interpreter enter the "." character. If you enter CR (Carriage
   Return), XprsMon will read the register at <reg>+<ds> and display it
   current value.

Examples

   XprsMon>
```

4.1.6.2.14 Patch PCI address in the FPGA End Point MEM space (BAR#2)

```
Command
   pp →  read/write data from/to PEV1100 PCI MEM space

Synopsis
   pp.<ds>[<swap>] <offset> [<data>]

      where <ds> = b,s,w, (data size 1,2,4)
            <swap> = s if swapping is required
            <offset> = address offset in hexadecimal
            <data> = data in hexadecimal [write cycle]

Description
   The pp command shall be used to perform PCI MEM cycles targeted to the
   PEV1100 FPGA PCIe End Point. It takes up to 2 parameters.
```

```
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
   The second character (optional) shall be s if the data are to be byte
   swapped before being written or displayed (meaningful only for 16, 32 and
   64 bits accesses).
   The second character is optional and shall be s if byte swapping is
   required (meaningful only for 16 and 32 bits accesses).
   The first parameter shall be the address offset in PCI MEM space to be
   accessed.
   The second parameter shall be the data to write in the address offset. If
   it is missing, XprsMon performs a read cycle on the register, displays the
   data and prompt for a new data to overwrite the register. To return to the
   command interpreter enter the "." character. If you enter CR (Carriage
   Return), XprsMon will read the register at <reg>+<ds> and display it
   current value.

Examples
    XprsMon#2>map show mas_32

   +=========================================================+
   + Map Name : mas_32
   +----------+----+----+----------+------------------+------+
   |  offset  |flag| usr|   size   |   remote address | mode |
   +----------+----+----+----------+------------------+------+
   | 00000000 | 01 | 01 | 00100000 | 0000000000000000 | 2003 | SHM
   | 00100000 | 01 | 01 | 00100000 | 0000000040000000 | 1303 | VME A32

   XprsMon#2>pp.w 100000
   0x00100000 : 00000001 -> 12345678.
   0x00100004 : 00000002 -> .
   XprsMon#2>pp.w 100000
   0x00100000 : 12345678 ->
   0x00100004 : 00000002 -> .
   XprsMon#2>
```

4.1.6.2.15 Patch an address on the VME bus

```
Command
   pv →  read/write data from/to VME bus

Synopsis
   pv.<ds>[<swap>] <vme_addr> [<data>] [m:<mode>]

      where <ds> = b,s,w, (data size 1,2,4)
            <swap> = s if swapping is required
            <vme_addr> = VME address in hexadecimal
            <data> = data in hexadecimal [write cycle]
            <mode> = VME access mode

Description
   The pv command shall be used to perform VME cycles through the PEV1100. It
   takes up to 3 parameters.
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
```

```
   The second character (optional) shall be s if the data are to be byte
   swapped before being written or displayed (meaningful only for 16, 32 and
   64 bits accesses).
   The second character is optional and shall be s if byte swapping is
   required (meaningful only for 16 and 32 bits accesses).
   The first parameter shall be the VME address.
   The second parameter shall be the data to write in the VME address. If it
   is missing or equal to '?', XprsMon performs a read cycle on the register,
   displays the data and prompt for a new data to overwrite the register. To
   return to the command interpreter enter the “.” character. If you enter CR
   (Carriage Return), XprsMon will read the register at <reg>+<ds> and display
   it current value.
   The third parameter shall be a string defining the VME access mode:
     crcsr  → generate address modifier for VME64x CR/CSR (0x2f)
     a16    → generate A16 data access address modifier (0x29/0x2d)
     a24    → generate A24 data access address modifier (0x39/0x3d)
     a32    → generate A32 data access address modifier (0x09/0x0d)
     blt    → generate address modifier A32/D32 block transfer (0x0b/0x0f)
     mblt   → generate address modifier A32/D64 block transfer (0x08/0x0c)
     ao24   → generate A24 address only cycle
     ao32   → generate A32 address only cycle
     iack   → generate VME IACK cycle
   If the access mode parameter is missing the one used the last time a VME
   cycle was executed is kept. When XprsMon is launched, a32 is selected by
   default.

Examples
   XprsMon#2>pv.w 40000000 ? m:a32
   0x40000000 : 12345678 -> 0
   0x40000004 : 00000002 -> .
   XprsMon#2>pv.w 40000000
   0x40000000 : 00000000 ->
   0x40000004 : 00000002 -> .
   XprsMon#2>
```

4.1.6.2.16 Patch PEX8624 registers

```
Command
   px →  read/write data from/to PEV1100 PCI MEM space

Synopsis
   px.<ds>[<swap>] <offset> [<data>]

      where <ds> = b,s,w, (data size 1,2,4)
            <swap> = s if swapping is required
            <offset> = address offset in hexadecimal
            <data> = data in hexadecimal [write cycle]

Description
   The px command allow to access the PEX8624 internal registers by performing
   PCI MEM cycles in its address space. It takes up to 2 parameters.
   The command extensions is made of 1 or 2 characters. The first character
   can be:
     b → for byte access
     c → for 16 bit access
     w → for 32 bit access
   The second character (optional) shall be s if the data are to be byte
   swapped before being written or displayed (meaningful only for 16, 32 and
   64 bits accesses).
   The second character is optional and shall be s if byte swapping is
```

```
   required (meaningful only for 16 and 32 bits accesses).
   The first parameter shall be the address offset in PCI MEM space to be
   accessed.
   The second parameter shall be the data to write at the address offset. If
   it is missing, XprsMon performs a read cycle on the register, displays the
   data and prompt for a new data to overwrite the register. To return to the
   command interpreter enter the "." character. If you enter CR (Carriage
   Return), XprsMon will read the register at <reg>+<ds> and display it
   current value.

Examples

   XprsMon#2>px.s 0
   0x00000000 : 10b5 ->
   0x00000002 : 8624 -> .
   XprsMon#2>
```

4.1.6.2.17 SFLASH operations

```
Command
   sflash →  perform SFLASH operation

Synopsis
   sflash <operation> [<para#1>][<para#2>][<para#3>]

      where <operation> is the operation to be performed
              id    → shows the SFLASH identifier
              load  → loads an FPGA bitstream file in SFLASH
              sign  → shows signature of FPGA loaded in SFLASH
              read  → read SFLASH content

Description
   The sflash command allows to manage the PEV1100 SFLASH.
   The id operation displays the SFLASH device hardware identifier
   The load operation reads a file from the file system and writes it in the
   SFLASH device according to the following syntax:
   sflash load offset filename
   The read operation displays the timer current value in msec amd usec.

Examples
   XprsMon#2>sflash id
   SFLASH identifier 20:20:18
   XprsMon#2>sflash sign fpga#1
   FPGA#1 Signature at offset 0x3f0000

      + company:IOxOS Technologies
      + board:PEV1100
      + filename:pev1100_070509a.sfl
      + creation:Fri Aug 28 15:07:37 2009

      + mcs_file:pev1100_070509a.mcs
      + mcs_devname:XC5VLX30T
      + mcs_devid:0x02a6e093
      + mcs_offset:0x000000
      + mcs_size:0x11dfc0
      + mcs_checksum:0xcbde562a
      + mcs_creation:Fri Aug 28 15:06:01 2009
```

```
      + fsm_file:pev_init.fsm
      + fsm_offset:0x1c0000
      + fsm_size:0x000278
      + fsm_checksum:0xcc952770
      + fsm_creation:Wed Sep 17 15:43:36 2008

   XprsMon#2>
```

4.1.6.2.18 Local Timer operations

```
Command
   timer →  perform timer operation

Synopsis
   timer <operation>]

      where <operation> is the operation to be performed
              start → starts the timer
              stop  → stops the timer
              read  → read the timer current value

Description
   The timer command allows to manage the PEV1100 local timer.
   The start operation starts the timer.
   The stop operation stops the timer.
   The read operation displays the timer current value in msec amd usec.

Examples
   XprsMon#2>timer start
   XprsMon#2>timer read
   current timer value : 3520.834500 msec
   XprsMon#2>timer read
   current timer value : 5648.904840 msec
   XprsMon#2>timer stop
   XprsMon#2>timer read
   current timer value : 8352.000000 msec
   XprsMon#2>timer read
   current timer value : 8352.000000 msec
   XprsMon#2>
```

4.1.6.2.19 Test Control

```
Command
   tinit →  launch test program

Synopsis
   tinit [<testfile>]

      where <testfile> is the test program to be launched

Description
   The tinit command launches the test program testfile in a xterm window. If
```

```
   no testfile is given, the "./PevTst" is used as test program..
   When the test program is launched, a bidirectional pipe is created by
   XprsMon to establish a communication channel. This channel is used by
   XprsMon to pass the commands tlist, tset, tstart and tkill to the test
   program.


Examples
   XprsMon#2>tinit ./PevTst
   XprsTst->Launching:./PevTst XprsTst.cfg 5 8->Done:./PevTst 8594
   XprsMon#2>
```

4.1.6.2.20 VME interface configuration

```
Command
   vme → handle VME interface

Synopsis
   vme <operation>]

      where <operation> is the operation to be performed
              conf → configure interface parameters

Description
   The conf operation allows to configure the following parameters on the VME
   interface:
     a32 base address
     a32 window size
   If bit 0 of a32_size is set, the slave interface is enabled with a 1 Mbytes
   granularity. Otherwise a 16 Mbytes granularity is used.That operation
   updates the hardware registers controlling the VME slave interface.
   If granularity mode is changed, the init command shall be used to re-
   initialize the VME slave interface according to the current setting of the
   hardware registers. That command destroys all existing address mapping.
Examples
   XprsMon#2>vme conf
   VME configuration
   a32_base [20000000] ->
   a32_size [10000000] -> 8000000
   XprsMon#2>vme conf
   VME configuration
   a32_base [20000000] ->
   a32_size [08000000] ->
   XprsMon#1>vme init
   VME INIT...
   XprsMon#2>
```

4.1.6.2.21 TTY operations

```
Command
   tty → allow to send a string of characters on /dev/ttyUSB0

Synopsis
   tty <operation> [<string>]

      where <operation> is the operation to be performed
```

```
            <string> is a string of character
Description
   The tty command allows to synchronize external devices with XprsMon using a
   USB to serial line converter. This can be very useful to perform automated
   test involving CPUs in the VME crate
   The open operation opens the /dev/ttyUSB0 device
   The close operation closes the /dev/ttyUSB0 device
   The send operation sends a character string <string> to the /dev/ttyUSB0
   device

Examples
   XprsMon>
```

Back Cover