

BSP User Guide

**MIDAS M5000 Series
Single Board Computer
VxWorks 5.5**

Version 1.2-Release 1.5

Notice

The information in this document is subject to change without notice and should not be construed as a commitment by VMETRO. While reasonable precautions have been taken, VMETRO assumes no responsibility for any errors that may appear in this document.

Trademarks

Trademarked names appear throughout this document. Rather than list the names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, we hereby state that the names are used only for editorial purposes and to the benefit of the trademark owner with no intention of improperly using the trademark.

The mention of any trademarked name is not intended to imply that VMETRO products are affiliated, endorsed or sponsored by such trademark owner.

Software and Firmware Licensing

Any Software and Firmware code provided by VMETRO described herein is proprietary to VMETRO or its licensors. The use of this Software and Firmware is governed by a licensing agreement included on the media on which the Software and Firmware was supplied. Use of the Software or Firmware assumes that the user has agreed to the terms of the licensing agreement. VMETRO retains all rights to the Software and Firmware under the copyright laws of the United States of America and other countries. This Software or Firmware may not in contravention of the licensing agreement be furnished or disclosed to any third party and may not be copied or reproduced by any means, electronic, mechanical, or otherwise, in whole or in part, without specific authorization in writing from VMETRO.

Copyright © 2007 VMETRO

This document may not be furnished or disclosed to any third party and may not be copied or reproduced in any form, electronic, mechanical, or otherwise, in whole or in part, without the prior written consent of VMETRO.



Warranty

VMETRO products are warranted against defective materials and workmanship within the warranty period of 1 (one) year from date of invoice. Within the warranty period, VMETRO will, free of charge, repair or replace any defective unit covered by this warranty. A Return to Manufacturer Authorization (RMA) number should be obtained from VMETRO prior to return of any defective product. With any returned product, a written description of the nature of malfunction should be enclosed. The product must be shipped in its original shipping container or similar packaging with sufficient mechanical and electrical protection in order to maintain warranty. The product should be returned at the user's expense (including insurance for the full product value).

This warranty assumes normal use. Products subjected to unreasonably rough handling, negligence, abnormal voltages, abrasion, unauthorized parts replacement and repairs, or theft are not covered by this warranty and will if possible be repaired for time and material charges in effect at the time of repair. Any customer modification to VMETRO products, including conformal coating, voids the warranty unless agreed to in writing by VMETRO.

If boards that have been modified are returned for repair, this modification should be removed prior to the board being shipped back to VMETRO for the best possibility of repair. Boards received without the modification removed will be reviewed for reparability. If it is determined that the board is not repairable, the board will be returned to the customer. All review and repair time will be billed to the customer at the current time and materials rates for repair actions.

This product has been designed to operate with modules, carriers or compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage. VMETRO assumes no liability for any damages caused by such incompatibility. For products that have failed or malfunctioned due to abuse, miss-use or accident or for products that have failed or malfunctioned after the expiry of the warranty, the costs of repair or replacement will not be covered by VMETRO.

VMETRO specifically disclaims any implied warranty of merchantability and fitness for a particular purpose. The warranty provided herein for electronic equipment products is the user's sole and exclusive remedy. In no event shall VMETRO, or its distributors and agents, be liable for direct, indirect, special, incidental, or consequential damages (including but not limited to lost profits, penalties or damages payable to third parties) suffered or incurred, whether based on contract, tort or any other legal theory, even if VMETRO has been informed of the possibility of such damages. This limitation of liability may not be enforceable in certain jurisdictions; therefore the limitations may not apply. This warranty gives you specific rights. You may have other rights that vary from jurisdiction to jurisdiction.

VMETRO's warranty is limited to the repair or replacement policy described above and neither VMETRO nor its agent shall be responsible for consequential or special damages related to the use of their products.

Limited Liability

VMETRO does not assume any liability arising out of the application or use of any product described herein; neither does it convey any license under its patent rights nor the rights of others. VMETRO products are not designed, intended, or authorized for use as components in systems intended to support or sustain life, or for any application in which failure of the VMETRO product could create a situation where personal injury or death may occur. Should Buyer purchase or use VMETRO products for any such unintended or unauthorized application, Buyer shall indemnify and hold VMETRO and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that VMETRO was negligent regarding the design or manufacture of the part.

Contact Us

Worldwide HQ

VMETRO asa
Østensjøveien 32
0667 OSLO, Norway
Phone: +47 22 10 60 90
Fax: +47 22 10 62 02
info@vmetro.no

United Kingdom

VMETRO Ltd
Manor Courtyard
Hughenden Avenue
High Wycombe HP13 5RE
United Kingdom
Phone: +44 (0) 1494 476000
Fax: +44 (0) 1494 464472
sales@vmetro.co.uk

North American HQ

VMETRO, Inc.
1880 Dairy Ashford, Suite 400,
Houston TX 77077, U.S.A.
Phone: (281) 584-0728
Fax: (281) 584-9034
info@vmetro.com

VMETRO, Inc.

Suite 275
171 E. State St, Box 120
Ithaca, New York 14850
Phone: (607) 272 5494
Fax: (607) 272 5498
info@vmetro.com

Asia Pacific

VMETRO Pte Ltd
175A Bencoolen Street
#06-09 Burlington Square
Singapore 189650
Phone: +65 6238 6010
Fax: +65 6238 6020
info@vmetro.com.sg

Nordic & Baltic Countries

VSYSTEMS AB
Drottninggatan 104
SE-111 60 Stockholm
Contact: Bengt-Olof Larsson
Phone: +46 8 444 15 50
Fax: +46 8 444 15 60
info@vsystems.se

Germany

VSYSTEMS Electronic GmbH
Elisabethstrasse 30
80796 München
Contact: Ralf Streicher
Phone: +49 89 273 763 0
Fax: +49 89 273 763 10
info@vsystems.de

France

VSYSTEMS SAS
P.A. du Pas du Lac
5, rue Michaël Faraday
78180 Montigny-le-Bretonneux
Contact: Alain D'Aux
Phone: +33 1 30 07 00 60
Fax: +33 1 30 07 00 69
info@vsystems.fr

Italy

VSYSTEMS srl
via Cavour 123
10091 Alpignano (TO)
Contact: Luca Ravera
Phone: +39 11 9661319
Fax: +39 11 9662368
info@vsystems.it

Preface

Introduction

The VMETRO MIDAS M5000 is a single-board computer (SBC) built in a 6U VMEbus form factor based on the AMCC PPC440GX PowerPC processor. This document describes the VxWorks Board Support Package (BSP) for the PPC440GX processor on the VMETRO MIDAS 5000-series products.


This User's Guide provides important information on all aspects of the MIDAS M5xxx VxWorks Board Support Package (BSP). Answers to questions such as “How do I install the BSP? How do I boot the MIDAS board? How do I burn VxWorks boot code? How do I configure the MIDAS board to be VME bus master/slave at certain base addresses? How do I connect a PCI interrupt? How do I read/write the registers of a PCI device? Etc.” can be found in this document. The chapters are summarized below:

- Overview provides a brief description of the PPC440GX processor and its surroundings. Model Numbering describes the model-numbering approach used. Address Maps and Address Space Mapping describes the address space layout for both PPC440GX local and PCIbus perspectives.
- System Memory describes the SDRAM memory system used and describes how to access additional memory. Cache-safe Buffers provides a guideline to allocate a buffer that is noncacheable. BSP users with questions such as “What is a cache safe buffer? How do I get a buffer on the M5000 that can be used as shared memory between another board and the M5000?” can find the answers here.
- PCIbus Operations provides an overview of the PCI bus architecture of the M5xxx board and a description of the routines used to generate PCI bus cycles. BSP users with questions such as “What is a PMC slot? How do I read/write the configuration registers of a PMC in slot 1? How do I read/write PCI addresses? How do I convert a local address to a PCI address? Etc.” can find the answers here. This chapter assumes that the reader has some knowledge of PCI terminologies and bus specifications. PCI Interrupt Handling provides a guideline for connecting an Interrupt Service Routine (ISR) for a PCI device. BSP users with question such as “How do I connect an interrupt handler for my PMC board?” can find the answer here.
- Message Unit. Interrupt provides a guideline for using mailbox interrupt via the PPC440GX Message Unit. BSP users with questions such as “What is a Message Unit? Where is the MIDAS M5000's mailbox address? How can I interrupt the M5000 board from another board across PCI, VME and/or RACEway?” can find the answers here.
- VME Master & Slave Access Configuration provides a guideline to configure the M5xxx board to be master/slave at certain VME addresses. BSP users with question such as “How can I get the M5xxx board to read/write a VME slave at sextets address? How can a VME master read/write the M5xxx memory? Etc.” can find the answers here.
- RACEway-PCI Interface provides important information on how to handle M5000 board with the RACEway interface (“-R”) option. BSP users with question such as “How can I use a M5000-R board in a non-RACEway VME slot? I boot a M5xxx-R board with VxWorks and it hangs, what happens? Etc.” can find the answers here.

- Network Interfaces provides a guideline for using various network interfaces. BSP users with questions such as “How do I boot the M5000 board with sm? How do I boot the M5000 board with an ethernet interface?” can find the answers here.
- BSP Installation and Distribution provides information on BSP installation and software distribution.
- Burning VxWorks Boot Code provides a guide on how to burn VxWorks boot code on the M5000 board.
- DMA Drivers provides information on DMA drivers associated with the M5000
- The Midas File System provides information on the Midas File System implemented in FLASH memory of the M5000 board.
- Fibre Channel Information provides information about the Fibre Channel interface built into the M5000 board.
- Appendixes:
 - Troubleshooting
 - Deprecated Functions
 - BIST (Built In Self Test)
- Technical Support.

Style Conventions Used

- Code samples are `Courier` font and at least one size less than context.
- Text that represents user input is **Courier font**.
- Directory path names are *italicized*.
- File names are in **bold**.
- Absolute path file names are *italicized and in bold*.
- Pressing of individual keys will be indicated as **<key>**. For example:
 - **<Enter>** Press the key marked “Return” or “Enter”.
- Pressing a key-combination will be indicated as **Mod-n**, where “Mod” refers to any of the “Control” (**Cut**), “Alt” (**Alt**), or “Shift” (**Shift**) and “n” is any key. For example:
 - **Ctrl-z** Hold the Control key and press “z”.
 - **Alt-s** Hold the Alt key and press “s”.
- Simulated interaction with a computer will be shown in `Courier` type. Required keywords in computer interaction examples are shown in **bold Courier** type, and placeholders for items that vary or must be supplied by the user are indicated with *italic Courier* type. Output from the computer is shown in one of these three preceding styles. Input from the user is also displayed in one of these styles, but with the addition of underlining. Finally, comments that are not actually displayed or typed, but are provided in the text as aids to understanding, are shown in *italic Arial* type.

	<p>Warning! Indicates important information that can affect the operation of your M5xxx</p>
---	--

Note – This is information that will help you get the best performance.

IEC Prefixes for binary multiples

Symbol	Name	Origin	Derivation	Size
Ki	Kibi	Kilo binary	kilo	1024 bytes
Mi	Mebi	Mega binary	mega	1 048 576 bytes
Gi	Gibi	Gig binary	giga	1 073 741 824 bytes

Quality Assurance

VMETRO is dedicated to supplying our customers with products and services of the highest quality.

We therefore, continually review and assess our products and services with the aim to improve the processes involved in the development of our world-class products.

If you have any comments or feedback with respect to our products and services, please feel free to contact us through the support channels listed here, or email us at comments@vmetro.no

Technical Support

Please see the section Technical Support at the end of this guide.

Related Documentation

We recommend reading the documentation in the order shown.

- Release Notes
- M5000 User Guide (Hardware)

References Used in this document

[1] PPC440GX Embedded Processor User's Manual, AMCC.

[2] M5xxx User Guide, VMETRO, Inc.

[3] The PCI Specification v2.2, PCI Special Interest Group.

[4] VxWorks 5.5 Drivers API Reference Manual, Wind River Systems, Inc

[5] VMEbus Interface Components Manual, Tundra.

[6] The VMEbus Specification, VMEbus International Trade Association (VITA).

- [7] VxWorks 5.5 Programmer's Guide, Wind River Systems, Inc.
- [8] WindRiver Platforms, Getting Started, Wind River Systems, Inc
- [9] VxWorks 5.5 Network Programmer's Guide, Wind River Systems, Inc.
- [10] VxWorks 5.5 Reference Manual, Wind River Systems, Inc.
- [11] MIDAS Monitor User Guide, VMETRO, Inc.
- [12] MIDAS PXB DMA Driver Software Reference Manual, VMETRO, Inc.
- [13] VMFC Driver Software Reference Manual, VMETRO, Inc.

Contents

1	Overview	1
	1.1 Overview	2
	1.2 M5xxx Model Numbering	4
	1.3 M5xxx Address Maps and Address Space Mapping	5
2	System Memory	9
	2.1 System Memory	10
	2.2 Cache-safe Buffers	11
	2.3 Error Checking and Correction (ECC)	12
3	PCI Bus Operation	13
	3.1 PCI Bus Layout	14
	3.2 PCI Configuration Space Access	15
	3.3 PCI Memory and I/O Space Access	16
	sysBusToLocalAdrs	17
	sysLocalToBusAdrs	18
	3.4 PCI Interrupt Handling	19
	Interrupt Pin	19
	Interrupt Routing and Sharing	19
	Interrupt Connection and Enabling	20
	3.5 PCI Bus Operations	21
	Overview	21

PCI buses and MidasBusID 21
PMC slot 22
MidasBusIdFromPciBusNo 22
MidasBusIdToPciBusNo 24
MidasGetPmcBridgeBusNumbers 25
MidasPmcSlotInfoGet 26
MidasPciSlotInfoGet 27
3.6 PCI Optimizations 28

4 I2O Messaging Unit Support 29

4.1 Overview 30
4.2 Doorbell register support 31
 muOutDoorbellWrite 31
 muOutDoorbellRead 31
 31
4.3 Message Register support 32
 Common Definitions 32
 muMessageConnect 33
 muIsMessageConnected 34
 muMessageDisconnect 34
 muMessageEnable 35
 muMessageDisable 36
 muMessageWrite 37
 Example - Inbound Message Register 37
 Example - Remote Use of the Message Registers 40
4.4 Inbound Message Queue 41
 Common Definitions 41
 muCircularQueueInit 42
 muInPostQueueConnect 42
 muQueueConnect 43
 muInPostQueueDisconnect 43
 muCircularQueueFree 43
 muIsInPostQueueConnected 44
 muInPostQueueWrite 45
 Example - Inbound Message Queue 45
 Example - Remote Use of the Inbound Post Queue Register 48

5 VME Bus Operation 49

5.1 VME Master & Slave Access Configuration 50
 Overview 50
 VME Address Modifier (AM) Codes 56
 VME Master (PCI Slave) Access Windows 58
 VME Slave (PCI Master) Access Windows 59
 Functions 61
 uniPciSlaveImageSet 61
 uniVmeSlaveImageSet 63

uniVmeSlaveImageSetup	65
uniImageShow	67
5.2 Configuring PCI Slave Images in the Universe	68
<i>Procedure 68</i>	
<i>Viewing PCI Slave Image Configuration</i>	68
<i>Changing PCI Slave Image Configuration</i>	68
<i>Option 1 68</i>	
<i>Option 2 69</i>	
<i>Option 3 70</i>	
5.3 VME Interrupts	74
<i>VME Interrupt Handling</i>	74
<i>VME Interrupt Generation</i>	75
<i>VxWorks Target Shell Example : 76</i>	
5.4 Universe DMA Functionality	77
<i>Universe DMA Driver</i>	77
<i>Universe DMA Interface Functions</i>	77
uniDmaLibInit	77
uniDmaDirect	78
uniDmaChainCmdPktCreate	79
uniDmaChain	80
uniDmaChainStop	80
uniDmaNotifyFncSet	81

6 RACEway PCI Interface **83**

6.1 RACEway-PCI Interface	84
<i>Overview</i>	84
<i>PXB Initialization</i>	84
<i>PXB DMA Driver</i>	85

7 Network **87**

7.1 Ethernet (emac) Network Interface	88
<i>Configuring JUMBO packets 88</i>	
7.2 Shared Memory (sm) Backplane Network Interface	90
<i>Modifying the kernel configuration</i>	90
<i>Configuring The Development Host (UNIX)</i>	91
<i>Specifying IP Addresses And Host Names For VxWorks Nodes 91</i>	
<i>Specifying the Internet Gateway for VxWorks Nodes 91</i>	
<i>Configuring The SM Network</i>	92
<i>Configuring the SM Network Master 92</i>	
<i>BSP Configuration for the SM Network 92</i>	
<i>VxWorks Boot Parameters for the SM Network Master 92</i>	
<i>Configuring M5000 as an SM Network Participant 93</i>	
<i>Shared Memory Network Synchronization</i>	94
<i>Testing And Troubleshooting</i>	95
7.3 Gigabit Ethernet Throughput Performance	96

8 *BSP Installation* **99**

8.1 BSP Installation & Distribution 100
Installation 100
Files & Directories 100

9 *Burning VxWorks Boot Code* **103**

9.1 Burning VxWorks Boot Code from Rom Monitor (Serial) 104
9.2 Burning VxWorks Boot Code from VxWorks (Ethernet) 105
9.3 Burning VxWorks Boot Code from U-Boot (Ethernet) 106
Setting Network Parameters 106
 TFTP: 106
 NFS: 106
Flash VxWorks image 107

10 *DMA drivers* **109**

10.1 PPC440GX DMA Driver 110
 Setting up a DMA transaction 110
 Single DMA transactions 111
 Chained DMA transactions 112
 Common status structure 114
 Address translation functions 114
 Single blocking DMA transfer example 115
 Chained DMA transfer example 116

11 *MIDAS File System* **123**

11.1 The Midas File System (MFS) 124
 Overview 124
 The MFS Functions 124
 mfs_open 124
 mfs_close 125
 mfs_remove 125
 mfs_dir 126
 mfs_seek 126
 mfs_stat 127
 mfs_tell 127
 mfs_eof 128
 mfs_ftrunc 128
 mfs_gets 129
 mfs_read 130
 mfs_write 131
 mfs_pwd 131
 mfs_ini_gettext 132

mfs_ini_settext	133
mfs_ini_setlong	133
mfs_ini_setlongh	134
mfs_ini_getlong	134
mfs_usr_load_file	135
11.2 The vxbsp.ini File	136
<i>The RACEdrv Section</i>	136
<i>The VmeInterface Section</i>	136
11.3 The mmon.ini File	137
<i>The BoardInfo Section</i>	137
<i>The AutoStart Section</i>	137
12 Fibre Channel Support	139
12.1 Fibre Channel Information	140
<i>Overview</i>	140
APPENDIXES	141
A Troubleshooting	143
<i>Hello World Example Using WorkBench</i>	144
B Deprecated Functions	145
pciToLocalAdrs (replaced with sysBusToLocalAdrs)	146
pciLocalToPciAdrs (replaced with sysLocalToBusAdrs)	147
C Built In Self Test (BIST) API	149
Built In Self Test API Contents	150

Figures

Tables

TABLE 1-1.M5xxx switch settings affecting BSP	3
TABLE 1-2.M5xxx BSP 32-bit Effective (Virtual) Address Map with PCI Auto-config (default) 5	
TABLE 1-3.M5xxx Inbound PCibus-Memory-Space-Relative Address Map	6
TABLE 1-4.Effective virtual address space of PPC440GX to PCI I/O Space	7
TABLE 5-1.M5xxx Default Universe VME and PCI Slave Images.	51
TABLE 5-2.M5xxx Supported VME AM Codes	56
TABLE 6-1.PXB-related flags used by M5xxx BSP	85

1

Overview

This section briefly describes the features and architecture of the PPC440GX and its incorporation in the VMETRO M5xxx SBC. Complete details on the CPU itself may be found in the PPC440GX Embedded Processor User's Manual[1] available from AMCC. See “Related Documentation” on page vii.

Note – The BSP requires Tornado 2.2.1 with patch 90451 available from Wind River

In general, P2P bridges may be “enabled” or “disabled”. Note that all three onboard P2Ps must be enabled in order for the PPC440GX to have visibility of the Universe and other components on the Quaternary Bus. Typically, users will not need to worry about enabling or disabling the onboard P2Ps because the BSP properly configures the bridges through a process known as “PCI auto configuration”. In the rare cases in which the user wishes to perform manual configuration of P2P bridges, the functions to do so are available in the BSP. However, the user is cautioned that improperly changing the configuration of onboard P2Ps can have unexpected results, particularly if the P2P bridges are configured with inconsistent PCI bus numbers. See the section on PCIbus operations for more information.

There are many switches on the M5xxx that are fully documented in the M5xxxUser Guide[2]. Only a few of the switches affect the operation of the BSP, and these are shown in the table below. Default positions are noted in **bold**.

TABLE 1-1. M5xxx switch settings affecting BSP

Switch	Controls	Closed	Open	Default position
sw10-3	CPU FLASH write enable	Write disabled	Write enabled	Write enabled
sw10-4	CPU FLASH Monitor write enable	Write enabled	Write disabled	Write disabled
sw10-1	CPU Serial EEPROM write enable	Write enabled	Write disabled	Write enabled
sw3-4	Controls board Reset to VME Reset propagation. This affects the function <code>systemReset ()</code>	Enabled	Disabled	Enabled
sw6-2	ECC	Disabled	Enabled	Disabled

1.2 M5xxx Model Numbering

The M5xxx is offered with a number of optional components, leading to many different possible hardware configurations. The configuration of each M5xxx is encoded in the model number for the board. This section reviews the available M5xxx options and how the model number can be used to identify which options are present.

The general format for model numbers is:

M5ABCRP-XYZ#-rr where:

M is for Midas

5 represents this generation of Midas product

A is for the number of PMC positions (typically '2', or '5' for mezzanine)

B is for the number of PPCs ('0'=pure carrier, '1'=intelligent)

C is reserved for future options

R is used when RACE++/PXB++ is available

P denotes that there is a mounted P0 connector

XYZ gives front panel options present on the board, from top to bottom. XYZ can consist of:

- F is for Fibre channel SFF connector (optical)
- E is for Fast Ethernet (10/100) RJ45 connector (copper)
- G is for Gigabit Ethernet (10/100/1000) SFF (optical)
- J is for Gigabit Ethernet (10/100/1000) RJ45 (copper)

is for serial interface options, which can be:

- <no number> is for RS232 (2 ports)
- 4 is for RS232 (1 port) and RS422 (1 port)

rr is for the ruggedized version of the product

Additional restrictions apply to the usage of the three front panel. In particular:

“X” can be E (emac0), G (emac2), or J (emac2). “X” cannot be F.

“Y” can be E (emac1) or F

“Z” can be F, G (emac3), or J (emac3). “Z” can only be F if “Y” is also F.

The names in parentheses above refer to the name of each port within VxWorks. These names are used when specifying a device used to boot VxWorks.

1.3 M5xxx Address Maps and Address Space Mapping

The address map layouts (CPU and PCI) for the M5xxx BSP implementation are as follows. These maps are shown as supported with the default PCI auto-configuration. Manual PCI configuration is not currently supported by the M5xxx BSP. A detailed look at PCI address space assignment is given in the section of PCI bus layout.

TABLE 1-2. M5xxx BSP 32-bit Effective (Virtual) Address Map with PCI Auto-config (default)

Address range	Resource Mapped	Mapped by
0x00000000-0x0dffffff ^b	Cached System SDRAM access	MMU TLB Entry
0x0e000000-0x0fffffff ^{ab}	Non-cached System SDRAM access	MMU TLB Entry
0x10000000-0xbfffffff ^c	PCI outbound translation window	MMU TLB Entry with prefetch
0xc0000000-0xeffffffff ^c	PCI outbound translation window	MMU TLB Entry without prefetch
0xf0000000-0xf0ffffff	Internal CPU Peripherals	MMU TLB Entry
0xf1000000-0xf1ffffff	I2O	MMU TLB Entry
0xf2000000-0xf2ffffff	SRAM	MMU TLB Entry
0xf3000000-0xf4ffffff	FLASH memory (cached)	MMU TLB Entry
0xf5000000-0xf5ffffff	PLD	MMU TLB Entry
0xf8000000-0xfbffffff	PCI I/O outbound	MMU TLB Entry
0xfc000000-0xfdffffff	Not mapped--available	-
0xfd000000-0xfdffffff	PCI-X bridge	MMU TLB Entry
0xfe000000-0xffffffff	FLASH memory (non-cached)	MMU TLBEntry

- a. The cached and non-cached regions access the same physical SDRAM.
- b. User configurable through NONCACHEABLE_MEMORY_SIZE
- c. User configurable through PCI_MASTER_PREFETCHABLE_POOL_SIZE

Regions marked “Not mapped--available” can provide addressing to PCIbus resources. To enable access to these regions, the PPC440GX MMU must be initialized appropriately. This is done by adding entries to the sysStaticTlbDesc[] array found in sysLib.c. See the sysStaticTlbDesc[] array in "sysLib.c" for more details.

Figure 1-2 shows a graphic representation of Table 1-2.

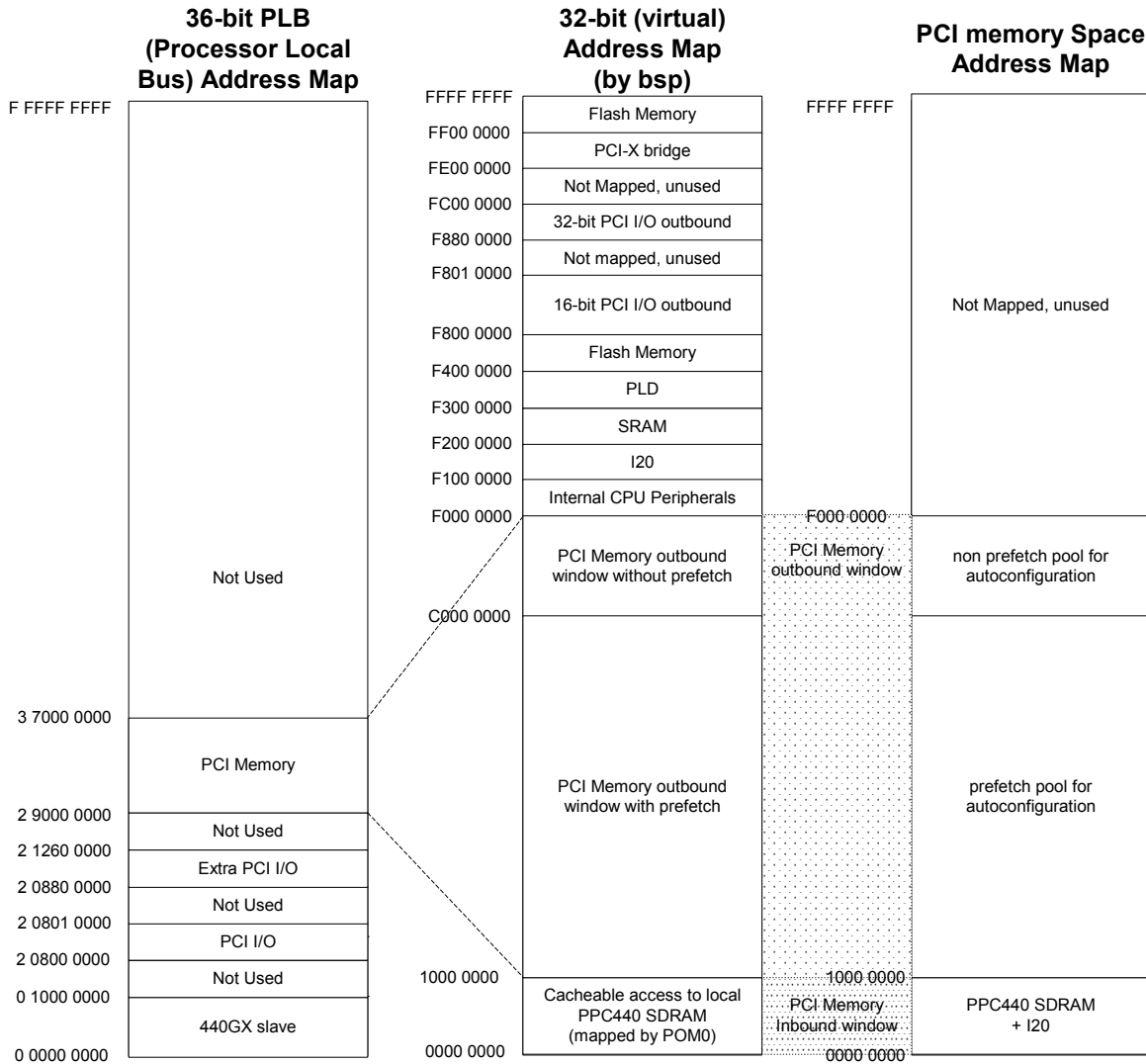


FIGURE 1-2. Outbound Address Mapping to PCI Memory Space with PCI Auto-Configuration

TABLE 1-3. M5xxx Inbound PCIbus-Memory-Space-Relative Address Map

PCI Address range	Resource Mapped	Mapped by
0x00000000-0x00000fff	PPC440GX I2O	PIM0/PIM1
0x00001000-0x0fffffff	SDRAM	PIM0

The “I2O” is the PPC440GX Message Unit which facilitates the transfer of messages between the PPC440GX and a device on PCI. Please see the PPC440GX Embedded Processor User's Manual[1] for more information on the I2O. Figure 1-3 shows how the inbound PCI memory maps to local PPC440GX memory.

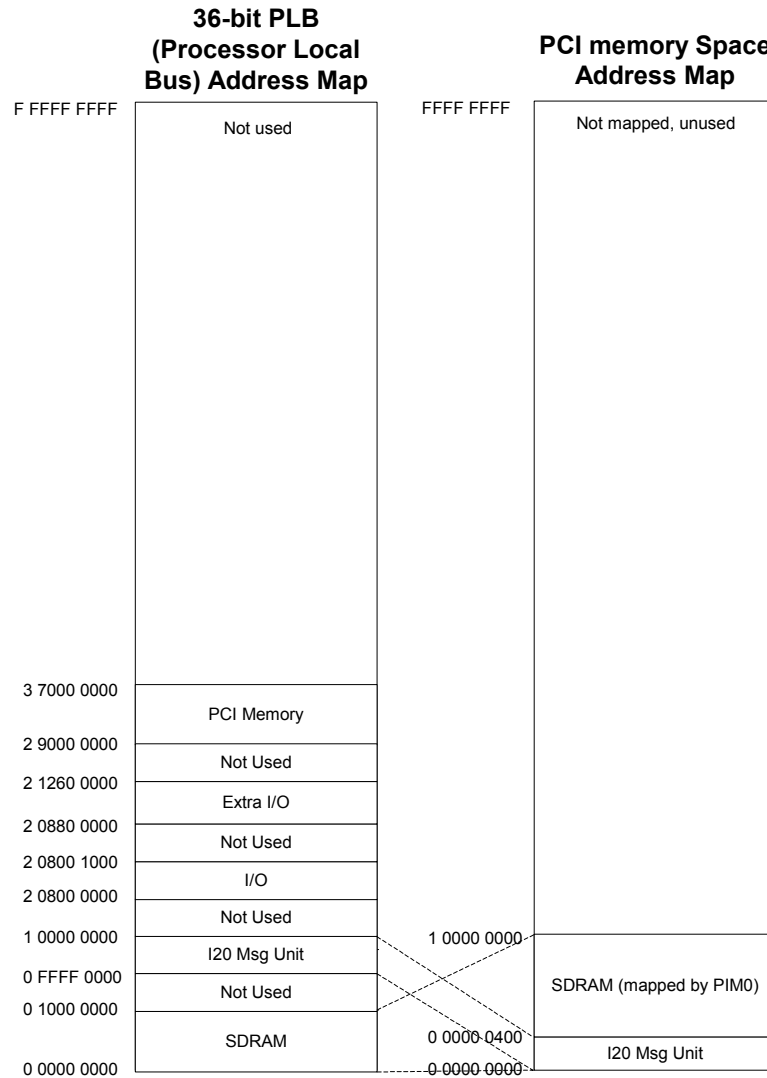


FIGURE 1-3. Inbound Address Mapping from PCI Memory Space with PCI Auto-Configuration

TABLE 1-4. Effective virtual address space of PPC440GX to PCI I/O Space

Local Memory Range	PCI I/O Address range	Resource Mapped	Mapped by
0xf8800000-0xfbffffff	0x800000-0x4000000	32-bit PCI I/O Space	MMU
0xf8000000-0xf800ffff	0x0-0x1000	16-bit PCI I/O Space	MMU

The figure below shows how the Effective virtual address space of PPC440GX memory maps to PCI I/O Space.

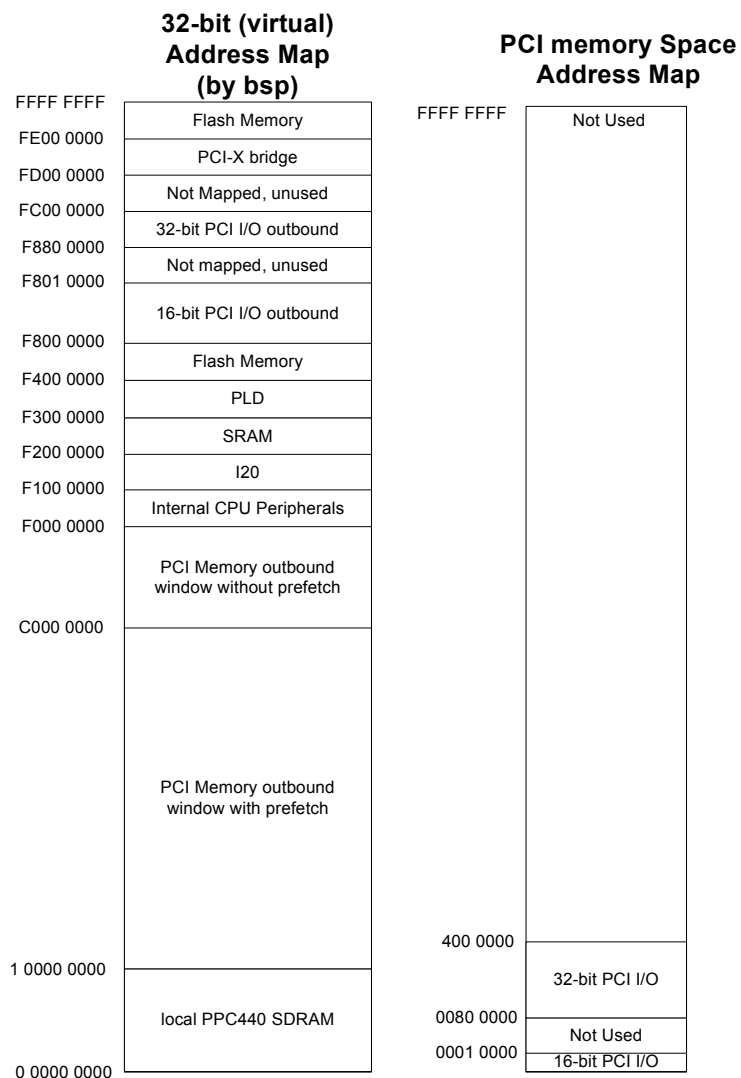


FIGURE 1-4. Address Mapping from Local Address Space to PCI I/O Space

2

System Memory

2.1 System Memory

The MIDAS M5xxx implementation includes 256 MBof 166 MHz Double Data Rate (DDR) SDRAM. Memory is mapped by MMU Translation Lookaside Buffer (TLB) entries, which permits the SDRAM memory to be accessed as either cacheable or non-cacheable. The `malloc()` or `calloc()` functions can be used to allocate cacheable memory, while `cachedmaMalloc()` can be used to allocate non-cacheable memory.

By default, the M5xxx BSP implementation makes only 32 MiB available to the operating system. This is because the PowerPC compiler uses the single instruction (branch) direct calls to subroutines (by default). Single branch instruction allow only 25 effective address displacement bits plus a signed bit resulting in possible jumps to subroutines within +/- 32 MiB offset from the current branch instruction (i.e., at the current program counter). Object code compiled with single branch instructions cannot call subroutines if they are loaded more than 32 MiB from the VxWorks libraries (object modules are loaded into the heap which starts at `sysMemTop()` and grows toward address 0).

If the compiler is instructed to use dual instruction indirect calls (the same as used when calling function pointers), the resulting object module can call subroutines anywhere within the 32-bit effective address space. This can be done by using the compiler option `-mlongcall`. However, dual instruction calls are slightly more expensive in both code space and execution time. Therefore, it is often better to keep `sysMemTop()` at 32 MiB in order to provide the best possible efficiency.

By default, the `sysMemTop()` function returns a value close to the 32 MiB effective address range. The `sysPhysMemTop()` function can be used to determine the total amount of memory present on the board. There are, at present, four ways an application can make use of the memory beyond the first 32 MiB.

1. The first method is to create a separate memory pool for the extra memory (i.e., not part of the system memory pool). Refer to `memPartLib` (`memPartCreate`) in the VxWorks Reference Manual.
2. The second method of getting around this restriction for downloadable applications is to use the `-mlongcall` compiler option in the GNU compiler. However, this option may introduce an unacceptable amount of performance penalty and extra code size for some applications. It is for this reason that the VxWorks kernel is not compiled using `-mlongcall`.
3. The third method uses the standard VxWorks distribution, but relies on loading all code modules first, while only 32 MiB of memory is available, then adding the extra memory to the system memory pool by a call to:

```
memAddToPool (LOCAL_MEM_LOCAL_ADRS + 0x02000000, size);
```
4. The fourth method involves simply addressing the extra memory directly, independent of VxWorks. In this case, the user is responsible for managing the extra memory since the VxWorks memory-management functions (such as `malloc`, `calloc`, `free`, etc.) will not work for this method.

2.2 Cache-safe Buffers

Any time there is asynchronous access to DRAM, there is a potential cache coherency problem (i.e., data in the cache is different from data in DRAM). The PPC440GX data cache may be write-through cache (i.e., data is always written to both cache and memory when the CPU performs a write) or copyback (i.e., data is flushed to main memory only when a pending read must reuse the previously-written cache line). By default, the data cache is copyback.

If write-through caching is used, there is no cache coherency problem whenever data is transferred from local memory to remote memory. Copyback caching (the default case) improves processor throughput, but the local memory does not immediately reflect the value written. Read access by an external agent (e.g., a DMA controller) may pick up the old (or uninitialized) value if the dirty cache line has not yet been flushed when the external read access commences.

Regardless of the cache mode used, when data is written by another master (typically a DMA controller) to local memory, the local buffer is now inconsistent with the cache. To handle cache coherency problem on the M5xxx board, the user has three options:

- use cache-safe buffers. Cache-safe buffer is the best approach since data integrity is assured on a per-buffer basis, and the performance penalty of calling `cacheInvalidate` is avoided. The M5xxx BSP supports the VxWorks routine `cacheDmaMalloc` whenever MMU support is included (the default case).
- call `cacheInvalidate` before reading from the local buffer and call `cacheFlush` after writing to the local buffer.
- disable the data cache. This is not recommended except during device driver debugging.

Default Memory Configuration

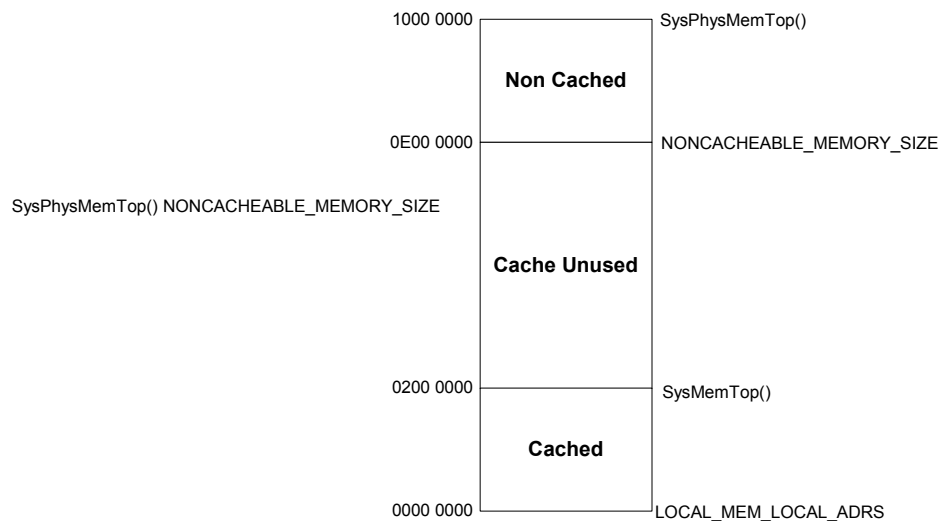


FIGURE 2-1. Default Memory Configuration

2.3 Error Checking and Correction (ECC)

The MIDAS M5xxx BSP supports Error Checking and Correction (ECC). Switch 6-2 on the MIDAS board is used to switch ECC on and off. When sw6-2 is ON, ECC is Enabled.

The ECC feature automatically corrects 1-bit errors. 2-bit and multiple bit errors are not corrected and will result in the suspension of the task that initiated the memory transaction.

3

PCI Bus Operation

This chapter covers the PCI bus operation associated with the M5xxx BSP. You should already be familiar with basic PCI bus operating principles. First the PCI bus layout of the M5xxx is discussed. Then, methods for accessing the PCI bus are presented. All of the BSP functions that provide access to the PCI bus are documented in this chapter.

3.1 PCI Bus Layout

PCI IDSEL numbers and configuration addresses are given in the M5xxx User Guide[2]. The device number to use in calls to PCI configuration functions (see section on PCIbus Operations below) is the IDSEL number minus 16. Therefore, PMC#1 with IDSEL pAD[16] is device number 0. Device numbers for devices behind P2P bridges on PMCs are set by the PMC hardware, according to the PCI Specification[3]. The standard VxWorks PCI-related query functions, `pciDeviceShow` and `pciHeaderShow` are very useful for reviewing device numbers, as well as a great deal of other information about each of the PCI devices in PCI configuration space.

The PCI auto-configuration process takes care of assigning all PCI Memory and I/O space resources. The regions of PCI Memory Space are summarized as follows:

0x10000000 - 0xBFFFFFFF (2.75 GB) for PCI prefetchable memory space
0xC0000000 - 0xEFFFFFFF (768 MB) for PCI non-prefetchable memory space

These should be sufficient for almost all real-world applications. In rare instances when more PCI memory space is needed, there are several possible approaches to obtaining larger amounts of PCI Memory Space.

If the VME outbound windows are not required, the non-prefetchable memory pool can be effectively made larger by disabling the VME outbound windows. See the section on VMEbus operation for more information on how to do this. If even larger amounts of PCI Memory Space are required, please contact Vmetro technical support for assistance.

The regions of PCI I/O Space are summarized as follows:

0xF8800000 - 0xFBFFFFFF (56 MiB) for 32-bit PCI I/O space
0xF8000000 - 0xF800FFFF (64 KB) for 16-bit PCI I/O space

These should be sufficient for almost all real-world applications. The sizes of the 16-bit and 32-bit PCI I/O space are as large as the PPC440GX allow them to be.

In order to gain a better understanding of what the PCI autoconfigurator is doing and to determine how much PCI Memory and I/O Space is being used by the PCI autoconfigurator, the `PCI_AUTO_DEBUG` constant found in `pciAutoConfigLib.c` can be `#define'd`. This constant is `#undef'ed` by default. By `#define'ing` this constant, a large amount of serial output will be generated by the PCI autoconfigurator. It is recommended that the `PCI_AUTO_DEBUG` variable be left `undef'ed` for distributable applications because the additional serial output causes the M5xxx to take longer to boot up.

On M55xx boards, the MEZZ500 daughter-board allows for three additional PCI-compatible PMC sites. The MEZZ500 has an on-board P2P that bridges between the Quaternary Bus and the three PMC sites on the MEZZ500. The PCI autoconfigurator configures the MEZZ500 P2P bridge and all devices installed on MEZZ500 PMC sites in the same way as all other P2P bridges and PCI/PCI-X devices in the system. Thus, the user need not take any special action to allocate or reserve PCI resources for the MEZZ500 or PMC devices mounted on the MEZZ500.

3.2 PCI Configuration Space Access

Access to M5xxx PCI configuration space is provided through functions defined in “target/h/drv/pci/pciConfigLib.h”, which is part of the standard VxWorks/Tornado installation. See the aforementioned file for details.

The following definitions in config.h can be used to access these resources:

```
M5000_PPC440GX_VENDOR  
M5000_PPC440GX_DEVICE  
M5000_PPC440GX_I2O_BAR  
M5000_PPC440GX_SDRAM_BAR.
```

3.3 PCI Memory and I/O Space Access

PCI Memory and I/O accesses are made through memory-mapped references to the regions designated in the CPU-Relative address map shown in the section on address maps.

Because PCIbus is inherently little-endian (least-significant byte resides at the lowest address) and the PPC440GX is inherently big-endian (most-significant byte at lowest address), care must be exercised when accessing device registers or other non-memory devices, so that data values are read and written properly.

An object and any pointer to it must agree in size, or the data read or written will be wrong or be in the wrong place. Data read from a multi-byte entity must be byte-swapped before being used or returned. Data must be byte-swapped before being written to a multi-byte entity.

The M5xxx BSP provides functions for accessing PCI-resident resources which take care of byte swapping and guaranteeing in-order access to system resources. They are:

```
IMPORT VOID    sysOutWord (ULONG address, UINT16 data); /* sysALib.s */
IMPORT VOID    sysOutLong (ULONG address, ULONG data); /* sysALib.s */
IMPORT VOID    sysOutByte (ULONG, UCHAR);           /* sysALib.s */
IMPORT USHORT  sysInWord  (ULONG address);          /* sysALib.s */
IMPORT ULONG   sysInLong  (ULONG address);          /* sysALib.s */
IMPORT UCHAR   sysInByte  (ULONG);                  /* sysALib.s */
```

PCI-to-CPU and CPU-to-PCI address translations are provided by the functions `sysBusToLocalAdrs` and `sysLocalToBusAdrs`, respectively.

sysBusToLocalAdrs

Synopsis

```
STATUS sysBusToLocalAdrs (
    int adrsSpace,
    char *busAdrs,
    char **pLocalAdrs
)
```

adrsSpace - Represents the bus address space in which busAdrs resides. The value can be one of the following: PCI_SPACE_IO_PRI (0x40) - 32-bit PCI I/O Space
 PCI_SPACE_MEMIO_PRI (0x41) - Non-cacheable PCI Memory Space
 PCI_SPACE_MEM_PRI (0x42) - Cacheable PCI Memory Space
 PCI_SPACE_IO16_PRI (0x43) - 16-bit PCI I/O Space

A supported VMEbus AM code (see section on VMEbus).

busAdrs - the bus address to be converted to a local address

pLocalAdrs - holds the returned local address equivalent of the busAdrs if it exists

Description This function converts a bus address to a local address. The function can be used with both PCI and VME address spaces. If the given bus address can be converted to a local address, the local address is placed in pLocalAdrs and the function returns OK. Otherwise, ERROR is returned. Note that an adrsSpace value of PCI_SPACE_CFG_PRI is not supported. In other words, sysBusToLocalAdrs() cannot be used to determine the local address space equivalent for PCI Config Space because there is no such direct address mapping between local and PCI configuration space.

Returns OK, or ERROR.

Example

```
/* This example finds the Universe UCSR (held in BAR0 of the Universe) in local
address space */
struct MIDAS_PCI_SLOT_INFO slot_info;
UINT32 PciBusNo, bar0, localAdrs;
/* Get slot_info for the Universe */
if (MidasPciSlotInfoGet (MPSLOT_UNIVERSE, &slot_info) == ERROR)
    return ERROR;
if (MidasBusIdToPciBusNo(slot_info.MidasBusId, &PciBusNo) == ERROR)
    return ERROR;
pciConfigInLong(PciBusNo, slot_info.PciDeviceNo, 0, 0x10, &bar0);
if (sysBusToLocal(PCI_SPACE_MEMIO_PRI, (char*)bar0, (char**)&localAdrs))
    return ERROR;
printf (The UCSR is at 0x%x in local address space\n", localAdrs);
```

sysLocalToBusAdrs

Synopsis

```
STATUS sysLocalToBusAdrs (
    int adrsSpace,
    char *localAdrs,
    char **pBusAdrs
)
```

`adrsSpace` - Represents the bus address space in which `pBusAdrs` resides. The value can be one of the following:
`PCI_SPACE_IO_PRI (0x40)` - 32-bit PCI I/O Space
`PCI_SPACE_MEMIO_PRI (0x41)` - Non-cacheable PCI Memory Space
`PCI_SPACE_MEM_PRI (0x42)` - Cacheable PCI Memory Space
`PCI_SPACE_IO16_PRI (0x43)` - 16-bit PCI I/O Space

A supported VMEbus AM code (see section on VMEbus).

`localAdrs` - the local address to be converted to a bus address

`pBusAdrs` - holds the returned bus address equivalent of `localAdrs` if it exists

Description This function converts a local address to a bus address. The bus address can be in either a PCI and VME address space. If the given local address can be converted to a bus address, the bus address is placed in `pBusAdrs` and the function returns OK. Otherwise, ERROR is returned. Note that an `adrsSpace` value of `PCI_SPACE_CFG_PRI` is not supported. In other words, `sysLocalToBusAdrs()` cannot be used to determine the PCI Config Space equivalent of local address space because there is no such direct address mapping between local and PCI configuration space.

Returns OK, or ERROR.

Example

```
/* This example allocates a cacheable buffer and determines the PCI address of
the buffer */
char *bufLocal, *bufPci;
bufLocal = (char *) malloc(1000);
if (sysLocalToBus(PCI_SPACE_MEM_PRI,bufLocal, &bufPci)
    return ERROR;
printf("The PCI address of the buffer is: 0x%x\n", bufPci);
```

3.4 PCI Interrupt Handling

This chapter documents interrupt routing and handling in the M5xxx implementation. The PCI autoconfigurator automatically configures the PCI configuration space registers known as “Interrupt Line” (offset 0x3c) and “Interrupt Pin” (offset 0x3d) for each PCI device. However, the PCI autoconfigurator does not install any Interrupt Service Routines (ISRs). Setting up the ISRs is typically performed by device drivers. In general, device drivers are specific to the operating system (i.e., VxWorks 5.5) as well as the type of CPU (i.e., PPC440GX). They are often supplied by the manufacturer of PCI devices. The M5xxx BSP includes the device drivers for all PCI devices mounted on the M5xxx itself, except for the Fibre Channel controllers and RACEway. These devices are supported through separate products available from VMETRO.

In PCI systems, interrupts may be shared by multiple PCI devices. In order to support interrupt sharing, each ISR must determine whether its associated device caused an interrupt and if so, it must clear the condition that caused the interrupt. The device driver/application programmer should use the `pciIntConnect()` function to install an ISR for a particular PCI device on the M5xxx board or mounted in a PMC slot. Using the VxWorks library function `intConnect()` is not guaranteed to work because `intConnect()` does not support interrupt chaining, whereas `pciIntConnect()` does. To disassociate an ISR from an interrupt, use the `pciIntDisconnect2()` function. The `pciIntConnect()` and `pciIntDisconnect2()` functions are documented in [4].

The application code needs to include the header files listed below:

- **intLib.h** - VxWorks interrupt-related function declarations
- **drv/pci/pciIntLib.h** - declarations for `pciIntConnect()`/`pciIntDisconnect()`
- **mdrv/include/MidasPciLib.h** - Midas PCI-related functions
- **mdrv/include/midasppc440.h** - PPC440GX primarily interrupt-related declarations

Interrupt Pin

Each PCI device may implement up to four interrupt pins (INTA, INTB, INTC, and INTD) on a physical package of a multi-function PCI device. If a package implements one pin, it must be INTA. If a package is single function, it must use INTA. The Interrupt Pin register can be read to determine which of the four pins each device uses (see the PCI Specification[3]).

Interrupt Routing and Sharing

The M5xxx implements sixteen interrupt inputs to the PPC440GX. See M5xxx User Guide[2] for a description of how interrupts from the various devices are routed to the Universal Interrupt Controller (UIC) of the PPC440GX. Some of the PPC440GX interrupt inputs are shared between PCI devices. Additionally, some PCI devices implement more than one interrupt output and may have multiple interrupt outputs routed to the same PPC440GX interrupt input.

For example, PMC#2's INTA and INTC are both connected to interrupt input #1 of the PPC440GX. If a multifunction device is installed on PMC#2 and two of its functions asserted INTA and INTC respectively, an interrupt on either of those lines would be sent to interrupt input #1 of the PPC440GX.

Interrupt Connection and Enabling

The `pciIntConnect()` routine will install an interrupt handler for any vector in the table, regardless of whether that vector is associated with an input to the UIC. The PowerPC architecture also defines the functions `intEnable()` and `intDisable()`. Following connection with `pciIntConnect()`, an interrupt must be enabled by calling `intEnable()`. In the M5xxx BSP implementation, these functions operate only on interrupt levels/vectors associated with interrupt inputs to the UIC. See `mdrv/include/midasppc440.h` for the interrupt levels and vectors defined for the M5xxx implementation.

The functions `sysIntEnable()` and `sysIntDisable()` in `sysLib.c` applies only to VMEbus interrupt-request levels. See the [7] for details.

3.5 PCI Bus Operations

Overview

The functions documented here are used to access the PCI buses on the M5xxx board. The application code needs to include the header files listed below (all BSP-provided header files can be found in the BSP directory “midasppc440” or in “midasppc440/mdrv/include”):

- MidasPciLib.h
- \$WIND_BASE/target/h/drv/pci/pciConfigLib.h

PCI buses and MidasBusID

All M5xxx boards have three (3) PCI-X buses which are referred to as the Primary, Secondary, and Tertiary PCI-X buses. All M5xxx boards also have one (1) PCI bus referred to as the Quaternary PCI bus. In addition, M55xx boards have a fifth PCI bus referred to as the MEZZ500 PCI bus. To differentiate between the five buses, some of the functions use a MidasBusId parameter. The MidasBusId is an identifier used to specify which of the physical PCI-X/PCI buses are being referenced. There is no MidasBusId associated with additional P2P bridges that may be present (for example, P2P bridges installed on PMCs). The MidasBusId can only be one of the following values:

- MIDAS_PRIMARY_BUS = 0 MidasBusId for primary PCI-X bus
- MIDAS_SECONDARY_BUS = 1 MidasBusId for secondary PCI-X bus
- MIDAS_TERTIARY_BUS = 2 MidasBusId for tertiary PCI-X bus
- MIDAS_QUATERNARY_BUS = 3 MidasBusId for quaternary PCI bus
- MIDAS_MEZZ500_BUS = 4 MidasBusId for MEZZ500 PCI bus

The MidasBusId is useful for identifying a PCI device (typically a PMC card) by its physical bus location. This need typically arises in applications where multiple PMC slots carry the same type of PMC card, and the user needs to associate each PMC card with different roles in the system. The function `MidasPmcSlotInfoGet()` is used to retrieve the physical parameter associated with a given PMC slot.

When there is only one PMC card of a certain type, the standard VxWorks library function `pciFindDevice` is preferred in order to retrieve PCI bus and device numbers for the PMC card.

Note that the MidasBusId is not necessarily the same value as the actual secondary bus numbers stored in the P2P bus number registers. The actual secondary bus number for a particular PCI device is referred to in the PCI Specification[3] documentation, the BSP itself, and this documentation as the “PCI bus number”. When PCI auto-configuration is enabled, PCI bus numbers are automatically determined by the PCI auto-configurator. The PCI bus numbers assigned by the PCI auto-configurator may change from one bootup to the next depending on the hardware and options being used. Factors that affect the PCI bus numbers chosen by the autoconfigurator include the number and configuration of all P2P bridges present in the system, whether the MEZZ500 is present, and whether the PXBs are being used. The functions `MidasBusNoToPciBusNo()` and `MidasBusNoFromPciBusNo()` are available for translating between the `MidasBusId` parameter and PCI bus number.

PMC slot

The M5xxx board can accommodate up to five (5) PCI Mezzanine Card (PMC) modules. Typically these PMCs provide I/O, memory, or even DSP functions. Typically, in order to access the registers of a PMC in PCI Configuration Space, the programmer has to provide information such as the device number, and `MidasBusId` which can be obtained with the function `MidasPmcSlotInfoGet`.

The function `MidasPmcSlotTblShow()` is useful for reviewing these attributes.

Following is a list of functions documented in this chapter: `MidasBusIdFromPciBusNo`, `MidasBusIdToPciBusNo`, `MidasGetPmcBridgeBusNumbers`, `MidasPciSlotInfoGet`, `MidasPmcSlotInfoGet sysBusToLocalAdrs`, and `sysLocalToBusAdrs`.

MidasBusIdFromPciBusNo

Synopsis

```
int MidasBusIdFromPciBusNo (
    UINT8 *MidasBusId,
    UINT8 PciBusNo
)
MidasBusId - pointer to the returned Midas Bus ID. Returned value is
either:
MIDAS_PRIMARY_BUS (0)
MIDAS_SECONDARY_BUS (1)
MIDAS_TERTIARY_BUS (2)
MIDAS_QUATERNARY_BUS (3)
or MIDAS_MEZZ500_BUS (4).
```

`PciBusNo` - The PCI bus number.

Description This function translates a PCI bus number to its corresponding Midas bus ID. The MIDAS bus ID is a software reference to one of the possible PCI-X/PCI buses on the M5xxx board. The PCI bus number is the actual PCI bus number stored in the bus number registers of the P2P bridges on a particular segment.

If any PMC cards (or any other interconnect bridge) implements a PCI bus which is physically off board the M5xxx, the corresponding PCI bus number cannot be mapped to a `MidasBusId` because only onboard busses have been assigned a `MidasBusId` value. `MidasBusIdFromPciBusNo()` will return ERROR for off board PCI bus numbers.

Returns OK, or ERROR.

MidasBusIdFromPciBusNo (Continued)

```

Example      UINT8 MidasBusId;
                UINT8 PciBusNo;
                for (PciBusNo=0;PciBusNo<=255;PciBusNo++)
                {
                    if (MidasBusIdFromPciBusNo (&MidasBusId, PciBusNo) != ERROR)
                    {
                        switch (MidasBusId)
                        {
                            case MIDAS_PRIMARY_BUS:
                                printf("Midas primary PCI bus corresponds to PCI bus
                                number:%x\n", PciBusNo);
                                break;
                            case MIDAS_SECONDARY_BUS:
                                printf("Midas secondary PCI bus corresponds to PCI bus
                                number:%x\n", PciBusNo);
                                break;
                            case MIDAS_TERTIARY_BUS:
                                printf("Midas tertiary PCI bus corresponds to PCI bus
                                number:%x\n", PciBusNo);
                                break;
                            case MIDAS_QUATERNARY_BUS:
                                printf("Midas quaternary PCI bus corresponds to PCI bus
                                number:%x\n", PciBusNo);
                                break;
                            case MIDAS_MEZZ500_BUS:
                                printf("Midas MEZZ500 PCI bus corresponds to PCI bus
                                number:%x\n", PciBusNo);
                                }
                            }
                }

```

MidasBusIdToPciBusNo

Synopsis	<pre>int MidasBusIdToPciBusNo (UINT8 MidasBusId, UINT8 *PciBusNo)MidasBusId - The Midas Bus ID. Must be one of MIDAS_PRIMARY_BUS (0) MIDAS_SECONDARY_BUS (1) MIDAS_TERTIARY_BUS (2) MIDAS_QUATERNARY_BUS (3) MIDAS_MEZZ500_BUS (4).</pre> <p>PciBusNo - pointer to a UINT8 that holds the returned PCI bus number.</p>
Description	<p>This function translates a Midas bus ID to its corresponding PCI bus number. The MIDAS bus ID is a software reference to one of the buses on the M5xxx board. The PCI bus number is the actual PCI bus number stored in the bus number registers of the P2P bridges on a particular segment.</p>
Returns	<p>OK, or ERROR.</p>
Example	<pre>UINT8 PciBusNo; if (MidasBusIdToPciBusNo (MIDAS_PRIMARY_BUS, &PciBusNo) != ERROR) printf(Midas primary PCI bus corresponds to PCI bus number:%x\n", PciBusNo); if (MidasBusIdToPciBusNo (MIDAS_SECONDARY_BUS, &PciBusNo) != ERROR) printf(Midas secondary PCI bus corresponds to PCI bus number:%x\n", PciBusNo); if (MidasBusIdToPciBusNo (MIDAS_TERTIARY_BUS, &PciBusNo) != ERROR) printf(Midas tertiary PCI bus corresponds to PCI bus number:%x\n", PciBusNo); if (MidasBusIdToPciBusNo (MIDAS_QUATERNARY_BUS, &PciBusNo) != ERROR) printf(Midas quaternary PCI bus corresponds to PCI bus number:%x\n", PciBusNo); if (MidasBusIdToPciBusNo (MIDAS_MEZZ500_BUS, &PciBusNo) != ERROR) printf(Midas MEZZ500 PCI bus corresponds to PCI bus number:%x\n", PciBusNo);</pre>

MidasGetPmcBridgeBusNumbers

Synopsis

```
int MidasGetPmcBridgeBusNumbers (
    UINT8 PmcSlotNo,
    UINT8 *primary,
    UINT8 *secondary,
    UINT8 *subordinate
)
```

PmcSlotNo - The PMC slot number. Must be one of 1,2,3,4, or 5.

primary - pointer to the returned value of the primary PCI bus number associated with a P2P bridge mounted in a PMC slot

secondary - pointer to the returned value of the secondary PCI bus number associated with a P2P bridge mounted in a PMC slot

subordinate - pointer to the returned value of the subordinate PCI bus number associated with a P2P bridge mounted in a PMC slot

Description This function can be used to determine whether a given PMC has a P2P bridge mounted on it, and if so, which PCI bus number(s) has been allocated for it. If a P2P bridge is present on a PMC slot, the function returns OK. The primary, secondary, and subordinate PCI bus numbers associated with the P2P bridge are returned through parameters. If no P2P bridge is found in the given PMC slot, the function returns ERROR. The meaning of these PCI bus numbers are fully described in the PCI specification[3]. Basically, the primary PCI bus number is the bus number for the side of the bridge closest to the 440. The secondary PCI bus number is the bus number for the side of the bridge furthest from the 440. The subordinate PCI bus number is the highest numbered bus that exists behind the bridge.

Returns OK, or ERROR.

Example

```
UINT8 primary, secondary, subordinate;
INT8 i;
for (i=1;i<=5;i++)
if (MidasGetPmcBridgeBusNumbers(i, &primary, &secondary, &subordinate == OK)
    printf("PMC with P2P bridge found in PMC #%d, primary: %d,
secondary: %d, subordinate %d\n", i, primary, secondary, subordinate);
```

MidasPmcSlotInfoGet

Synopsis

```
int MidasPmcSlotInfoGet (
    int iPmcSlotNo,
    MIDAS_PCI_SLOT_INFO *pMidasPciSlot
)

iPmcSlotNo - The PMC slot number. Must be one of 1,2,3,4, or 5.
```

MidasPciSlot - pointer to structure defined as follows:

```
struct MIDAS_PCI_SLOT_INFO {
    char Name [16];
    UINT8 PciDeviceNo;
    UINT8 MidasBusId;
    UINT8 PciIntLine [PCI_INT_LINES];
};
```

Description This function can be used to get information such as the PCI device number, the MIDAS bus ID of a particular PMC slot, the name of the device, or the PCI interrupt line(s) associated with the slot. In combination with other PCI-related functions, this information can be used to read/write the configuration registers of the device. Note that when the PMC contains a P2P bridge, this function returns the PCI device number for the P2P bridge. The PCI device numbers for any devices behind the PMC's bridge are set by the PMC itself.

Returns OK, or ERROR.

Example

```
struct MIDAS_PCI_SLOT_INFO slot_info;
if (MidasPmcSlotInfoGet (1, &slot_info) == ERROR)
    return ERROR;

printf ("For the PMC in slot 1: \n");
printf ("The device name is: %s \n", slot_info.Name);
printf ("The device number is: %d \n", slot_info.PciDeviceNo);
switch (slot_info.MidasBusId)
{
    case PRIMARY_PCI_BUS:printf("PRIMARY PCI bus");break;
    case SECONDARY_PCI_BUS:printf("SECONDARY PCI bus");break;
    case TERTIARY_PCI_BUS:printf("TERTIARY PCI bus");break;
    case QUATERNARY_PCI_BUS:printf("QUATERNARY PCI bus");break;
    case MEZZ500_PCI_BUS:printf("MEZZ500 PCI bus");break;
    default:printf("Unknown PCI bus");break;
}
```

MidasPciSlotInfoGet

Synopsis

```
int MidasPciSlotInfoGet (
    int iPciSlotTblIdx,
    MIDAS_PCI_SLOT_INFO *pMidasPciSlot
)
```

iPciSlotTblIdx - The PCI slot table index. Must be one of the MPSLOT_<X> constants defined in MidasPciLib.h.

pMidasPciSlot - pointer to structure defined as follows:

```
struct MIDAS_PCI_SLOT_INFO {
    char Name [16];
    UINT8 PciDeviceNo;
    UINT8 MidasBusId;
    UINT8 PciIntLine [PCI_INT_LINES];
};
```

Description This function can be used to get information such as the PCI device number (see PCI spec.), the MIDAS bus ID, the name of the device, or the PCI interrupt line(s) associated with the PCI device. This information can then be used to read/write the configuration registers of the device.

Returns OK, or ERROR.

Example

```
struct MIDAS_PCI_SLOT_INFO slot_info;
if (MidasPciSlotInfoGet (MPSLOT_UNIVERSE, &slot_info) == ERROR)
    return ERROR;
printf ("Universe PCI device info: \n");
printf ("The device name is: %s \n", slot_info.Name);
printf ("The device number is: %d \n", slot_info.PciDeviceNo);
```

3.6 PCI Optimizations

During PCI autoconfiguration the M5000 BSP sets the Latency Timer register of all PCI devices (including any PMCs that may have mounted) to the maximum allowed value FF; which corresponds to 255 PCI cycles. The Latency Timer register specifies, in units of PCI bus clocks, the minimum guaranteed number of clocks allocated to the PCI master, after which it must surrender tenure as soon as possible after its GNT# is deasserted.

The reason for maximizing the latency timers in the M5000 BSP is because many MIDAS applications require a maximum throughput performance. If your application must minimize the latency of each PCI transaction, please refer to section 3.5.4 Arbitration Latency in the PCI Local Bus Specification revision 2.2, 2.3 or 3.0.

Note – Some PCI devices limit the maximum value of their Latency Timer to a value that is optimized for their maximum burst size. This is allowed according to the PCI Local Bus Specification and should not be mistaken as an error in the M5000 BSP.

4

I2O Messaging Unit Support

4.1 Overview

The M5xxx BSP uses the PPC440GX I2O Messaging Unit (IMU) to implement mailbox interrupts. The IMU allows external host processors and the PPC440GX to communicate via message passing and interrupt notification (see Chapter 21 of the PPC440GX User's Manual[1] for a highly detailed description of the IMU). On the M5xxx, the BSP makes the IMU visible on PCI, VME, and RACEway so that it is possible for another processor to communicate/interrupt the PPC440GX via those buses.

The IMU registers are accessible in local memory space at 0xF1FF0000. The IMU registers are located in PCI space at 0x00000000. . By default, this region is mapped to VME A32 (AM code 0x0d) at a location dependent on CPU number (see section on VME bus for more information). An external VME master can pass a message and interrupt the PPC440GX by writing to the appropriate VME A32 address where the Inbound Message Register 0 is mapped (see the file **mdrv/include/mu.h** for the address/byte offset of all the IMU's registers.) The IMU has three messaging mechanisms.

- 4 Message Registers: (two inbound and two outbound); writing a 32-bit value to one of the two Inbound Message Registers interrupts the PPC440GX.
- 2 Doorbell Registers.
- 2 Circular Queues.

Note – The M5xxx BSP only supports the Message Register mechanism. Doorbell Registers and Circular Queues are not currently supported by the BSP.

4.2 Message Register Support

This section documents the following functions: `muMessageConnect`, `muIsMessageConnected`, `muMessageDisconnect` and `muMessageEnable`. These functions let the user connect ISRs to the IMU, in particular the Inbound Message Registers. An additional function supported by the BSP related to the message unit is `sysMailboxConnect`, which internally calls `muMessageConnect`.

muMessageConnect

Synopsis `int muMessageConnect (`
 `int regNum,`
 `int (*isr) (int),`
 `int isrArg)`
 `regNum` - Inbound Message Register number, 0 or 1.
 `int (*isr) (int)` - routine called at each message interrupt.
 `isrArg` - one argument to be passed to the ISR.

Description This function connects an ISR to the Inbound Message register specified by the `regNum` argument.

Returns OK, or ERROR.

Example

```
int isr (int arg) {
    logMsg ("This is my ISR, arg = %d\n", arg);
    return (ERROR);
}

int isrInstall () {
    int arg = 4;

    /* Connect ISR to the Inbound Message Register 0 */
    if (muMessageConnect (0, isr, arg) == ERROR)
        return (ERROR);
}
```

muIsMessageConnected

Synopsis `int muIsMessageConnected (`
 `int regNum)`
 `regNum - Inbound Message Register 0 or 1.`

Description Test to see if there is an ISR connected to the Inbound Message Register specified by the `regNum` argument.

Returns `TRUE` if there is an installed ISR, or `FALSE`.

Example `If (muIsMessageConnected (1))`
 `printf ("There is an ISR installed for Inbound Message Register 1\n");`

muMessageDisconnect

Synopsis `int muMessageDisconnect (`
 `int regNum)`
 `regNum - which Inbound Message Register, 0 or 1.`

Description This function disconnects an ISR from the Inbound Message Register specified by the `regNum` argument.

Returns `OK`, or `ERROR`.

Example `if (muMessageDisconnect (1) == ERROR)`
 `return (ERROR);`

muMessageEnable

Synopsis `int muMessageEnable (`
 `int regNum)`
 `regNum - which Inbound Message Register, 0 or 1.`

Description This function enables the message interrupt.

Returns `OK`, or `ERROR`.

Example `if (muMessageEnable (1) == ERROR)`
 `printf ("Message Register 1 was successfully enabled. \n");`

5

VME Bus Operation

5.1 VME Master & Slave Access Configuration

Overview

This chapter defines terminology, and configuration macros that together provide a guideline for how VME master/slave windows can be configured in the M5xxx BSP. The standard frame of reference shall be the perceived view by the user and the local CPU (i.e. the PPC440GX). A master transaction is one where the M5xxx board takes control of the VME bus and initiates a VME transaction. A slave transaction is one where the M5xxx board responds as a VME bus slave device to a transaction initiated by some other board acting as VME bus master.

On the M5xxx, the VME interface is controlled with the Tundra Universe IID chip (hereafter called the Universe). The Universe databook[5] describes all the registers in the Universe and is a very useful resource to have handy when working with the Universe. In the terminology of the Universe, VME master windows are called "PCI slaves". Sometimes VME slaves are referred to as PCI masters. In other words, "VME slaves" are the same as "PCI masters" and "PCI slaves" are the same as "VME masters".

The Universe supports a maximum of 8 PCI slave images and 8 VME slave images. An "image" in Universe terminology is essentially just a "window" in address space through which one bus can access the other. There are functions available in the BSP to configure both PCI and VME slave images, and all of these functions are reviewed in this section. These functions are `uniPciSlaveImageSet()`, `uniVmeSlaveImageSet()`, and `uniVmeSlaveImageSetup()`. One useful function for reviewing Universe slave images is `uniImageShow()`. This function provides information about all of the currently defined Universe slave images (both PCI and VME) and their attributes. Other useful functions are `sysBusToLocalAdrs()` and `sysLocalToBusAdrs()` which are documented in the PCI section above.

By default, the PPC440GX BSP uses 3 PCI slave images and 1 VME slave image. The sections below review the default slave image configurations. Using parameters in the `vmbsp.ini` file, one additional PCI slave image and one additional VME slave image may be defined. Please see the section of the manual on the `vmbsp.ini` file for more details.

Note – The VME slave bases of the M5xxx board are set in software (see below), not by hardware jumpers. To change the bases from the default values, edit the VME Master/Slave Access Window Macros in `sysVme.h`.

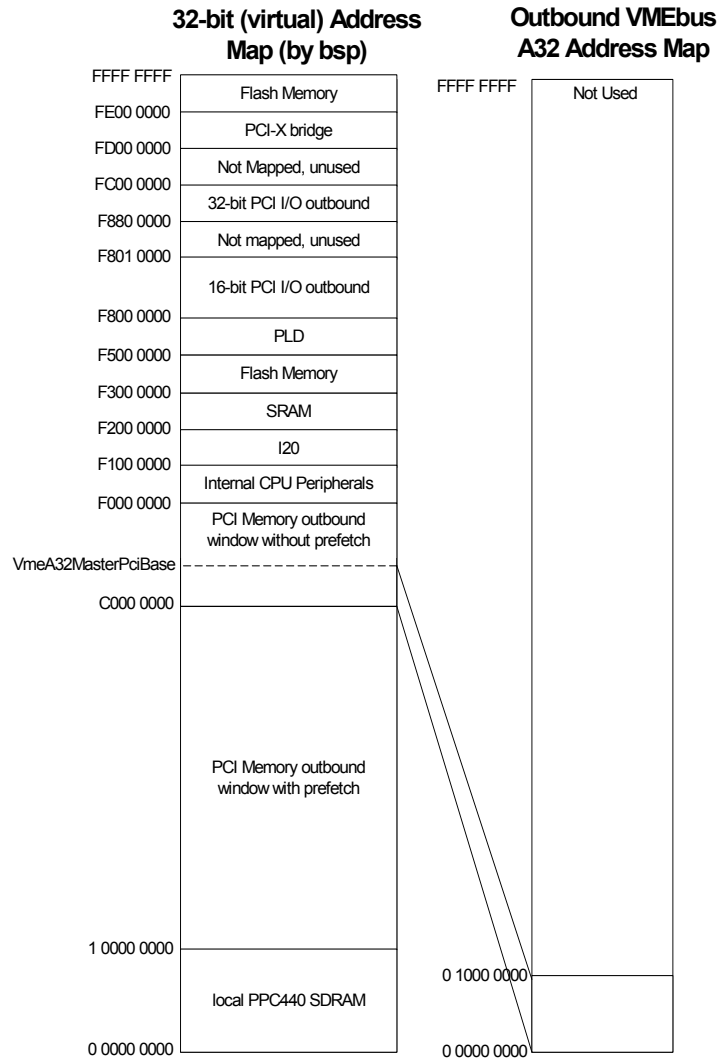


TABLE 5-1. M5xxx Default Universe VME and PCI Slave Images

Description	VME Base	PCI Base	Default Size	How to disable
A16 PCI slave	MIDAS_LOC_TO_VME_A16_VME_BASE (0x00000000)	Global variable: VmeA16MasterPciBase	VME_A16_MASTER_SIZE (64 KiB)	Set size macro to 0
A24 PCI slave	MIDAS_LOC_TO_VME_A24_VME_BASE (0x00000000)	Global variable: VmeA24MasterPciBase	VME_A24_MASTER_SIZE (16 MiB)	Set size macro to 0
A32 PCI slave	MIDAS_LOC_TO_VME_A32_VME_BASE (0x00000000)	Global variable: VmeA32MasterPciBase	VME_A32_MASTER_SIZE (256 MiB)	Set size macro to 0
Sec. A32 PCI slave	VmeA32Master2Base in vxbsp.ini	Global variable: VmeA32Master2PciBase	VME_A32_SEC_MASTER_SIZE (64 KiB)	Disabled by default; Add VmeA32Master2Base to vxbsp.ini to enable
A32 VME slave	VmeA32SlaveBase in vxbsp.ini OR If CPU=0, Base is 0 Else Base is 0x08000000 + (sysProcNum * 0x10000)	0x00000000 (points to PPC440GX I20)	VME_VMEA32_TO_LOC_SIZE If CPU=0, Size is 128 MiB. Else Size is 0x10000	In config.h, #define VME_A32_SLAVE_DISABLE OR Add VmeA32SlaveDisable in vxbsp.ini

FIGURE 5-1. *Default Usage of Outbound VME A32 Address Space*

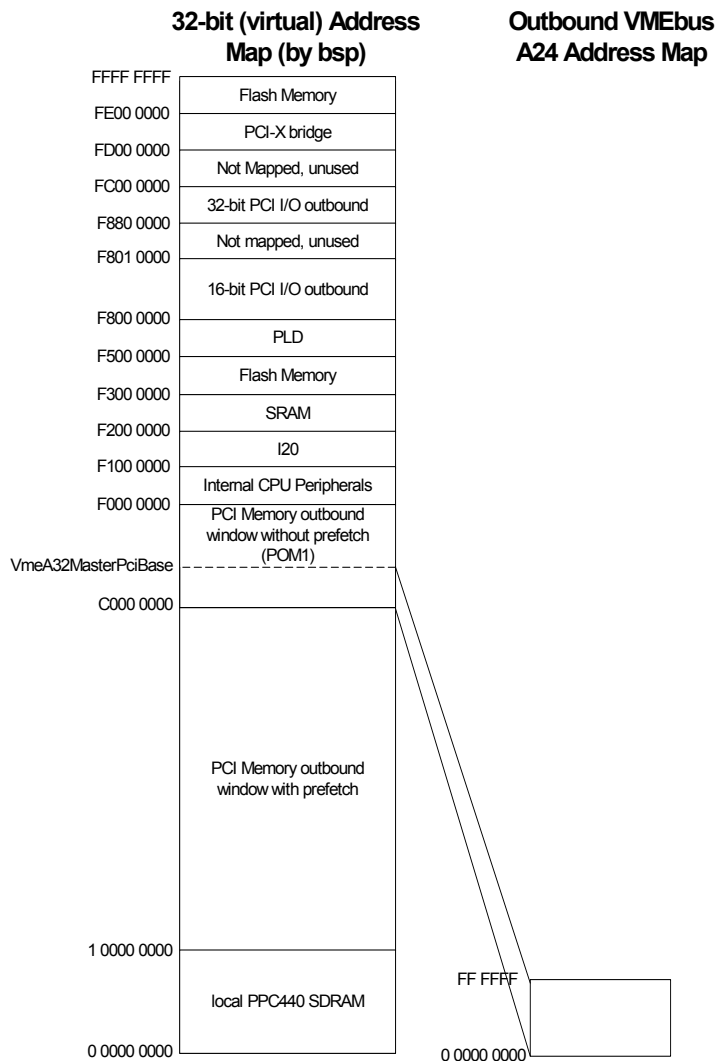


FIGURE 5-2. Default Usage of Outbound VME A24 Address Space

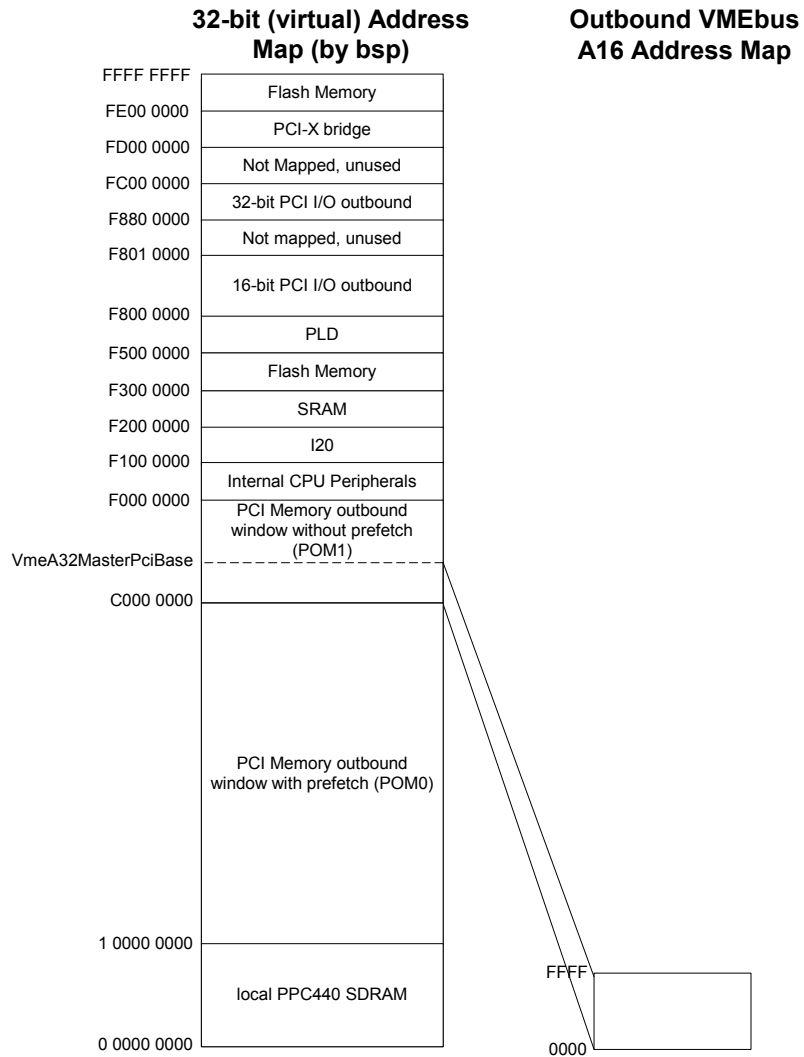


FIGURE 5-3. Default Usage of Outbound VME A16 Address Space

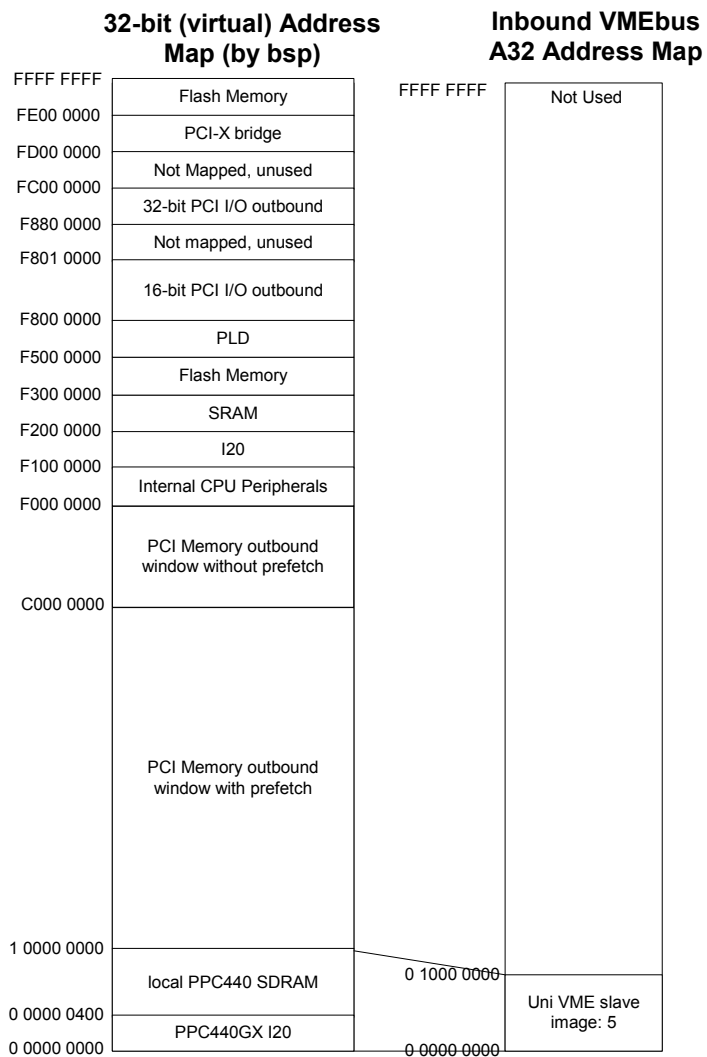


FIGURE 5-4. Default Usage of Inbound VME A32 Address Space

VME Address Modifier (AM) Codes

VME Address Modifier (AM) codes are a standard part of the VMEbus Specification[6]. In this section, the VME AM codes will be reviewed, along with an indication of whether each is supported by the address translation functions; `sysBusToLocalAdrs()` and `sysLocalToBusAdrs` in the BSP. The Universe driver can be setup to support all AM codes except A64 codes.

TABLE 5-2. M5xxx Supported VME AM Codes

HEX	Binary	Description	Supported
0x00	00 0000	A64 64-bit block transfer	Yes
0x01	00 0001	A64 32-bit single transfer	Yes
0x02	00 0010	Reserved	No
0x03	00 0011	A64 32-bit block transfer	Yes
0x04	00 0100	A64 LOCK command	No
0x05	00 0101	A32 LOCK command	No
0x06	00 0110	Reserved	No
0x07	00 0111	Reserved	No
0x08	00 1000	A32 non-priv. 64-bit block transfer	Yes
0x09	00 1001	A32 non-priv. 32-bit data access	Yes
0x0a	00 1010	A32 non-priv. 32-bit program access	Yes
0x0b	00 1011	A32 non-priv. 32-bit block transfer	Yes
0x0c	00 1100	A32 supervisory 64-bit block transfer	Yes
0x0d	00 1101	A32 supervisory 32-bit data access	Yes
0x0e	00 1110	A32 supervisory 32-bit program access	Yes
0x0f	00 1111	A32 supervisory 32-bit block transfer	Yes
0x10	01 0000	User-defined	No
0x11	01 0001	User-defined	No
0x12	01 0010	User-defined	No
0x13	01 0011	User-defined	No
0x14	01 0100	User-defined	No
0x15	01 0101	User-defined	No
0x16	01 0110	User-defined	No

TABLE 5-2. M5xxx Supported VME AM Codes (Continued)

0x17	01 0111	User-defined	No
0x18	01 1000	User-defined	No
0x19	01 1001	User-defined	No
0x1a	01 1010	User-defined	No
0x1b	01 1011	User-defined	No
0x1c	01 1100	User-defined	No
0x1d	01 1101	User-defined	No
0x1e	01 1110	User-defined	No
0x1f	01 1111	User-defined	No
0x20	10 0000	Reserved	No
0x21	10 0001	Reserved	No
0x22	10 0010	Reserved	No
0x23	10 0011	Reserved	No
0x24	10 0100	Reserved	No
0x25	10 0101	Reserved	No
0x26	10 0110	Reserved	No
0x27	10 0111	Reserved	No
0x28	10 1000	Reserved	No
0x29	10 1001	A16 non-priv. 32-bit access	Yes
0x2a	10 1010	Reserved	No
0x2b	10 1011	Reserved	No
0x2c	10 1100	A16 LOCK command	No
0x2d	10 1101	A16 supervisory 32-bit access	Yes
0x2e	10 1110	Reserved	No
0x2f	10 1111	Config. ROM/Control/Status Register	No
0x30	11 0000	Reserved	No
0x31	11 0001	Reserved	No
0x32	11 0010	A24 LOCK command	No

TABLE 5-2. M5xxx Supported VME AM Codes (Continued)

0x33	11 0011	Reserved	No
0x34	11 0100	A40 access	No
0x35	11 0101	A40 LOCK command	No
0x36	11 0110	Reserved	No
0x37	11 0111	A40 block transfer	No
0x38	11 1000	A24 non-priv. 64-bit block transfer	Yes
0x39	11 1001	A24 non-priv. 32-bit data transfer	Yes
0x3a	11 1010	A24 non-priv. 32-bit program access	Yes
0x3b	11 1011	A24 non-priv. 32-bit block	Yes
0x3c	11 1100	A24 supervisory 64-bit block transfer	Yes
0x3d	11 1101	A24 supervisory 32-bit data transfer	Yes
0x3e	11 1110	A24 supervisory 32-bit program access	Yes
0x3f	11 1111	A24 supervisory 32-bit block	Yes

VME Master (PCI Slave) Access Windows

A VME master access window is a window from the M5xxx onto the VME bus. A master access window enables the M5xxx board to become a VME bus master. The PPC440GX access to a local address within the window generates a VME bus transaction. This is what referred to as address translation mapping. The M5xxx BSP defines three VME master access windows (VME A32, A24, and A16), and optionally one additional A32 window (referred to as the secondary VME master access window). Each window is defined by a set of three macros: the PCI base, VME bus base, and window size. See Table 4 for a summary of the default windows and their sizes.

- The following macro defines a window onto VME A16 address space from the PPC440GX:
 1. PCI base address of the VME A16 window is configured automatically. The base address is stored in the global variable `VmeA16MasterPciBase`
 2. `VME_A16_MASTER_BASE` - VME A16 base of window. M5xxx BSP default is 0x0.
 3. `VME_A16_MASTER_SIZE` - size of the window into A16 space. M5xxx BSP default value is 64KB.
- The following macro defines a window onto VME A24 address space from the PPC440GX:
 1. PCI base address of the VME A24 window is configured automatically. The base address is stored in the global variable `VmeA24MasterPciBase`
 2. `VME_A24_MASTER_BASE` - VME A24 base of window. M5xxx BSP default is 0x0.
 3. `VME_A24_MASTER_SIZE` - size of the window into A24 space. M5xxx BSP default value is 16MB.
- The following macros define a window onto VME A32 address space from the PPC440GX:

1. PCI base address of the VME A32 window is configured automatically. The base address is stored in the global variable `VmeA32MasterPciBase`.
 2. `VME_A32_MASTER_BASE` - VME A32 base of window. M5xxx BSP default is 0x0.
 3. `VME_A32_MASTER_SIZE` - size of the window into A32 space. M5xxx BSP default value is 256MB.
- The following macros define an optional window (disabled by default) onto VME A32 address space from the PPC440GX:
 1. PCI base address of the optional VME A32 window is configured automatically. The base address is stored in the global variable `VmeA32Master2PciBase`
 2. `VME_A32_MASTER_2_BASE` - VME A32 base of the optional window. M5xxx BSP default is 0x0.
 3. `VME_A32_MASTER_2_SIZE` - size of the optional window. M5xxx BSP default is 64KB.

The PCI addresses (and hence the local addresses) corresponding to the VME windows are determined as part of the PCI autoconfiguration process. In order to access the VME windows from local address space, the local address must be determined. The function `sysBusToLocalAdrs` can be used to translate a VME address to a local address. The `uniImageShow()` function can also be used to view the configuration of Universe windows.

VME Slave (PCI Master) Access Windows

A VME slave access window is a window on the VME bus that allows other VME bus masters to access the M5xxx board as a VME bus slave device. A VME slave window usually makes main memory (RAM) available. This is frequently called Dual Porting Memory. Any VME bus access that addresses the VME slave window generates an access into the PPC440 local bus. Each window is defined by a set of three macros: the local base, VME bus base, and window size. Please see Table 4 for the default VME windows and sizes defined by the BSP.

- The following macros define a window from VME A16 address space into M5xxx memory.

Note – The M5xxx board uses A16 slave window to support BusNet. These macros are not defined by default if `#define INCLUDE_BUSNET` is not in `config.h`

1. `VME_A16_SLV_SIZE`
window size in A16 space. BSP default value is 4KB.
 2. `VME_A16_SLV_BUS`
VME (A16) bus base. BSP default value is 0x0.
 3. `VME_A16_SLV_LOCAL`
Not applicable.
- The following macros define a window from VME A32 address space into M5xxx memory:
 1. `VME_A32_SLV_SIZE`
size of window in A32 space. Window size depends on whether the entire DRAM is dual ported. By default only the sm master (processor number 0) dual ports its DRAM. For an M5xxx board which is not sm master, only 64KB is made visible (not the entire DRAM) in order to limit VME A32 bus space consumed. The user can easily dual port the entire DRAM by enabling the `#define MIDAS_MAP_DRAM_TO_VME` statement in `sysVme.h`.

2. **VME_A32_SLV_BUS**
VME A32 base of window. Each M5xxx board is mapped to a unique VME_A32_SLV_BUS using VME_A32_SLV_SIZE and the processor number.
3. **VME_A32_SLV_LOCAL**
Local bus base of window. Do not change this value!

Functions

uniPciSlaveImageSet

Synopsis STATUS uniPciSlaveImageSet

```
(  
    int     image,  
    UINT32  pciBase,  
    UINT32  vmeBase,  
    UINT32  size,  
    UINT32  pciAddrSpace,  
    UINT32  vmeAmCode,  
    UINT32  vmeDataWidth,  
    BOOL    postedWrites  
)
```

`image` - the Universe PCI slave image number, from 0 - 7

`pciBase` - the PCI base address of the window

`vmeBase` - the VME base address of the window

`size` - the size of the window in bytes

`pciAddrSpace` - the PCI address space. The value can be either `UNI_PCI_MEMORY_SPACE (0)`, `UNI_PCI_IO_SPACE (1)`, or `UNI_PCI_CFG_SPACE (2)`.

`vmeAmCode` - the VME AM code; specifying a "block" type AM code also implies that the similar AM code corresponding to "single" cycles will also be supported by the window.

`vmeDataWidth` - the data width supported by the window. The value can be either `UNI_VMEBUS_DATAWIDTH_8 (0)`, `UNI_VMEBUS_DATAWIDTH_16 (1)`, `UNI_VMEBUS_DATAWIDTH_32 (2)`, or `UNI_VMEBUS_DATAWIDTH_64 (3)`

`postedWrites` - whether the window allows posted (cached) writes. The value should be either `TRUE (1)` or `FALSE (0)`.

Description This function is used to configure a Universe PCI slave image. This allows the M5xxx to act as a VME master and read/write to other VMEbus devices configured as VME slaves.

uniPciSlaveImageSet (Continued)

Returns OK, or ERROR.

Example /* This example sets up a PCI slave image at PCI address 0xb0000000 and VME address 0x50000000. The VME address space is A24, the size of the window is 0x400000 (4 MB). The AM code is 0x3c (A24 supervisory 64-bit block transfer). Using this AM code implies that AM code 0x3d (A24 supervisory data access) is also supported. Also note that even though the AM code specifies "64-bit", only 32-bit data width will be supported because UNI_VMEBUS_DATAWIDTH_32 is specified. */

```
uniPciSlaveImageSet(
    7,
    0xb0000000,
    0x50000000,
    0x400000,
    UNI_PCI_MEMORY_SPACE,
    0x3c,
    UNI_VMEBUS_DATAWIDTH_32,
    TRUE);
```

uniVmeSlaveImageSet

```

Synopsis    STATUS uniVmeSlaveImageSet
              (
                int    image,
                UINT32 vmeBase,
                UINT32 pciBase,
                UINT32 size,
                UINT32 pciAddrSpace,
                UINT32 vmeAmCode,
                BOOL   postedWrites,
                BOOL   prefetchReads,
                BOOL   pci64,
                BOOL   pciLockOnRMWs
              )
  
```

`image` - the Universe VME slave image number, from 0 - 7

`vmeBase` - the VME base address of the window

`pciBase` - the PCI base address of the window

`size` - the size of the window in bytes

`pciAddrSpace` - the PCI address space. The value can be either `UNI_PCI_MEMORY_SPACE` (0), `UNI_PCI_IO_SPACE` (1), or `UNI_PCI_CFG_SPACE` (2).

`vmeAmCode` - the VME AM code; specifying a "block" type AM code also implies that the similar AM code corresponding to "single" cycles will also be supported by the window.

`postedWrites` - specifies whether the window will support posted write operations. With posted write, write operations may return before the data has been written to its final destination on the PCI bus.

`prefetchReads` - specifies whether the window will support prefetch read operations. With prefetch read, read operations may actually read more than the amount of data requested so that surrounding data may more quickly be returned on the next operation.

`pci64` - specifies whether the window will support 64-bit operation. If this is enabled, the window supports a data width of up to 64-bits.

`postedWrites` - whether the window allows posted (cached) writes. The value should be either `TRUE` (1) or `FALSE` (0).

`pciLockOnRMWs` - tells whether the window will use PCI Lock signal on Read-Modiy-Write cycles from VME to PCI bus.

uniVmeSlaveImageSet (Continued)

Description This function is used to configure a Universe VME slave image. This allows the M5xxx to act as a VME slave and support reading/writing to the local PCI bus from other VMEbus devices configured as VME masters. Note that the AM code parameter specifies the size of VMEbus address space (i.e., A16, A24, or A32), whether the window will support either "Data" or "Program" transactions (not both), and whether the window will support either "Supervisor" or "Non-privileged" operation (not both). The Universe VME slave image itself can simultaneously support both "Data" and "Program" transactions and both "Supervisor" and "Non-privileged" operation. In order to open a window with support for such combinations, use the uniVmeSlaveImageSetup() function instead. Also note that AM code does not specify whether VME slave images always support single and/or block operation. The Universe VME slave images always support both single and block operation (the VME master controls whether transfers are single cycle or block transfers).

Returns OK, or ERROR.

Example /* This example sets up a VME slave image at PCI address 0xb0000000 and VME address 0x50000000. The AM code is 0x3d (therefore, VME address space is A24, and 32-bit "Supervisory" "Data" transactions are supported), the size of the window is 0x400000 (4 MB). */

```
uniVmeSlaveImageSet(
    7,
    0x50000000,
    0xb0000000,
    0x400000,
    UNI_PCI_MEMORY_SPACE,
    0x3d,
    TRUE,
    TRUE,
    TRUE,
    TRUE);
```

uniVmeSlaveImageSetup

```

Synopsis      STATUS uniVmeSlaveImageSetup
                (
                int      image,
                UINT32   vmeBase,
                UINT32   pciBase,
                UINT32   size,
                UINT32   pciAddrSpace,
                BOOL     amCodeAdrsSpace,
                BOOL     amCodeUser,
                BOOL     amCodeSuper,
                BOOL     amCodeData,
                BOOL     amCodeProgram,
                BOOL     postedWrites,
                BOOL     prefetchReads,
                BOOL     pci64
                BOOL     pciLockOnRMWs
                )
  
```

`image` - the Universe VME slave image number, from 0 - 7

`vmeBase` - the VME base address of the window

`pciBase` - the PCI base address of the window

`size` - the size of the window in bytes

`pciAddrSpace` - the PCI address space. The value can be either `UNI_PCI_MEMORY_SPACE (0)`, `UNI_PCI_IO_SPACE (1)`, or `UNI_PCI_CFG_SPACE (2)`.

`amCodeAdrsSpace` - specifies the VME address portion of the AM codes supported by the window. It should be one of `UNI_AMCODE_A16(0)`, `UNI_AMCODE_A24(1)`, or `UNI_AMCODE_A32(2)`.

`amCodeUser` - specifies whether the window will respond to VME transactions with AM codes indicating "Non-privileged" access.

`amCodeSuper` - specifies whether the window will respond to VME transactions with AM codes indicating "Supervisory" access.

`amCodeData` - specifies whether the window will respond to VME transactions with AM codes indicating "Data" access.

`amCodeProgram` - specifies whether the window will respond to VME transactions with AM codes indicating "Program" access.

uniVmeSlaveImageSetup (Continued)

Synopsis (continued) `postedWrites` - specifies whether the window will support posted write operations. With posted write, write operations may return before the data has been written to its final destination on the PCI bus.

`prefetchReads` - specifies whether the window will support prefetch read operations. With prefetch read, read operations may actually read more than the amount of data requested so that surrounding data may more quickly be returned on the next operation.

`pci64` - specifies whether the window will support 64-bit operation. If this is enabled, the window supports a data width of up to 64-bits.

`postedWrites` - whether the window allows posted (cached) writes. Either TRUE (1) or FALSE (0).

`pciLockOnRMWs` - tells whether the window will use PCI Lock signal on Read-Modify-Write cycles from VME to PCI bus.

Description This function is used to configure a Universe VME slave image. This allows the M5xxx to act as a VME slave and support reading/writing to the local PCI bus from other VMEbus devices configured as VME masters. The difference between this function and `uniVmeSlaveImageSet` is that `uniVmeSlaveImageSetup` allows the window to simultaneously support multiple AM codes while `uniVmeSlaveImageSet` only allows a single AM code to be specified. Note that the Universe VME slave images always support both single and block operation (the VME master controls whether transfers are single cycle or block transfers).

Returns OK, or ERROR.

Example

```
/* This example sets up a VME slave image at PCI address 0xb0000000 and VME
address 0x50000000. VME address space is A24, both "Supervisory" and "Non-
privileged" transactions are supported, as well as both "Data" and "Program"
operations. The size of the window is 0x400000 (4 MB). */
```

```
uniVmeSlaveImageSetup(
    7,
    0x50000000,
    0xb0000000,
    0x400000,
    UNI_PCI_MEMORY_SPACE,
    UNI_AMCODE_A24,
    TRUE,
    TRUE,
    TRUE,
    TRUE,
    TRUE,
    TRUE,
    TRUE,
    TRUE,
    TRUE);
```

uniImageShow

Synopsis void uniImageShow
 (
)

Description This function is used to obtain a summary of the configuration of the Universe windows. It is very useful for determining the current status of VME connectivity. The information reported by this function is extracted directly from the UCSR in PCI Configuration Space.

Returns Nothing is returned, other than the textual output printed by the function.

Example The following text is typical of the output of uniImageShow:

```
-> uniImageShow
Universe PCI slave images (VME master windows):
Image Type Local   PCI Base VME Base Size      VMEAM PWEN VDW
-----
0      MEM  d2000000 d2000000 00000000 00010000 2d   Y   32
1      MEM  d0000000 d0000000 00000000 01000000 3d   Y   32
2      MEM  c0000000 c0000000 00000000 10000000 0d   Y   32

Universe VME slave images (PCI master windows):
Image Type Local   PCI      VME      Size      VMEAM codes PWEN PREN LD64 LRMW
-----
5      MEM  00000000 00000000 00000000 11000000 09 0a 0d 0e Y   Y   N   N
value = 0 = 0x0
```


5.2 Configuring PCI Slave Images in the Universe

Procedure

PCI slave images are used whenever the M5xxx needs to access external VME resources. The Universe supports up to 8 PCI slave images. By default, the M5xxx BSP uses 3 of the 8 PCI slave images. These PCI slave images support (256 MiB + 16 MiB) in VME A32 space starting at 0x00000000, 16 MiB in VME A24 space starting at 0x00000000, and 64 KiB in VME A16 space starting at 0x0000. A fourth PCI slave image optionally used by the BSP is disabled by default. It can be enabled by using the VmeA32Master2Base flag in the VmeInterface section of the vxbsp.ini file. If this parameter is present, the fourth PCI slave image will support 64 KiB in VME A32 space starting at the value of the VmeA32Master2Base parameter.

Viewing PCI Slave Image Configuration

A convenient way to view how the Universe PCI (and VME) slave images are configured is to use the uniImageShow function. In the default BSP configuration, the uniImageShow function gives the following information:

```
-> uniImageShow
Universe PCI slave images (VME master windows):
Image Type Local      PCI Base VME Base Size      VMEAM PWEN VDW
-----
0      MEM  d2000000 d2000000 00000000 00010000 2d      Y   32
1      MEM  d1000000 d1000000 00000000 01000000 3d      Y   32
2      MEM  c0000000 c0000000 00000000 11000000 0d      Y   32

Universe VME slave images (PCI master windows):
Image Type Local      PCI      VME      Size      VMEAM codes PWEN PREN LD64 LRMW
-----
5      MEM  00000000 00000000 00000000 10000000 09 0a 0d 0e Y      Y   N   N
```

This default configuration covers the VMEbus interface requirements of many applications, but not all.

Changing PCI Slave Image Configuration

There are several ways to change the PCI slave image configuration to customize to application requirements.

Option 1

The simplest approach is to modify the constants in the BSP that determine the VME base addresses, sizes, and AM codes of the default PCI slave images. The BSP can then be recompiled and reloaded into the M5xxx. This is appropriate when only one or two VME regions are required, which may be the case when interfacing to one or two external VME resources.

For example, suppose the M5xxx needs to interface with a VME-based digitizer board that uses 16 MiB of VME A32 space (specifically AM code 0x09) starting at 0x60000000 in VME A32 space. If there are no other VME interface requirements, the existing A32 window can be modified. This is done by modifying the code in `sysVme.c` that sets up the PCI slave image. The original code:

```
/* Set Universe window to VME A32 space */
uniPciSlaveImageSet (MIDAS_UNI_A32_WIN_NUM,
                    VmeA32MasterPciBase,
                    MIDAS_LOC_TO_VME_A32_VME_BASE,
                    VME_A32_MASTER_SIZE,
                    UNI_PCI_MEMORY_SPACE,
                    0x0d,
                    UNI_VMEBUS_DATAWIDTH_32,
                    TRUE);
```

may be modified to:

```
/* Set Universe window to VME A32 space */
uniPciSlaveImageSet (MIDAS_UNI_A32_WIN_NUM,
                    VmeA32MasterPciBase,
                    0x60000000,
                    0x01000000,
                    UNI_PCI_MEMORY_SPACE,
                    0x09,
                    UNI_VMEBUS_DATAWIDTH_32,
                    TRUE);
```

If the single region is larger than `VME_A32_MASTER_SIZE`, then the value of `VME_A32_MASTER_SIZE` would need to be changed.

Option 2

Another approach to configuring the PCI slave image in this example is to modify the existing VME slave images in the application prior to usage (instead of modifying the BSP). Because the default value of `VME_A32_MASTER_SIZE` is 256 MiB, this approach will only work if the region size needed is less than or equal to 256 MiB. In the case for the example above, simply call the following from the application prior to using the PCI slave image:

```

/* Set Universe window to VME A32 space */
uniPciSlaveImageSet (MIDAS_UNI_A32_WIN_NUM,
                    VmeA32MasterPciBase,
                    0x60000000,
                    0x01000000,
                    UNI_PCI_MEMORY_SPACE,
                    0x09,
                    UNI_VMEBUS_DATAWIDTH_32,
                    TRUE);

```

This second approach has the disadvantage of keeping 240 MiB (256 MiB - 16 MiB) of PCI space MMU-mapped, reserved, and unused, although this shouldn't be a problem in most applications since there is usually more than enough PCI memory space available to the PCI autoconfigurator.

Option 3

The most flexible approach to configuring PCI slave images is a little more sophisticated. In this approach, the total amount of extra space required for all additional VME interface regions is reserved by the PCI autoconfigurator. Then, the application can use `uniPciSlaveImageSet` to reconfigure existing PCI slave images and to configure up to four additional PCI slave images.

This sequence is necessary because the PCI autoconfigurator must consistently configure the three P2P bridges between the CPU and the Universe chip to handle all of the PCI slave images that will be used.

For example, suppose the M5xxx needs to interface with four VMEbus digitizer boards with the following specifications:

Board	VME base	Size	AM code
1	0x18000000	0x20000000	0x0d
2	0x40000000	0x10000000	0x0d
3	0x60000000	0x10000000	0x0d
4	0x80000000	0x10000000	0x0d

Also assume that these four boards are the only VME interfacing requirement for the M5xxx.

The total amount of VME space required to interface to these boards is 1 GiB + 256 MiB. Due to the large amount of space needed, a default setting in the PCI autoconfigurator must be changed to make this possible.

The PCI autoconfigurator reserves all PCI memory space associated with VME spaces from non-prefetchable memory space. By default, the M5xxx BSP has a maximum of 768 MiB available for all non-prefetchable PCI resources. This can be increased by decreasing the value of `PCI_MASTER_PREFETCH_POOL_SIZE` in **config.h**. For this example, the constant can be set in **config.h** as follows:

```
#define PCI_MASTER_PREFETCH_POOL_SIZE 0x80000000
```

Since a total of 0xE0000000 memory space is available for both prefetchable and non-prefetchable resources, the above line increases the amount of non-prefetchable space used by the PCI autoconfigurator to 0x60000000 (0xE0000000 - 0x80000000).

In this example, the default A32 PCI slave image may be used for the first board since there are no additional VME interface requirements. The first slave image is configured as follows (by modifying the **sysVme.c** file in the location described above):

```
/* Set Universe window to VME A32 space */
uniPciSlaveImageSet(MIDAS_UNI_A32_WIN_NUM,
                    VmeA32MasterPciBase,
                    0x18000000,
                    VME_A32_MASTER_SIZE,
                    UNI_PCI_MEMORY_SPACE,
                    0x0d,
                    UNI_VMEBUS_DATAWIDTH_32,
                    TRUE);
```

For this example, the value of `VME_A32_MASTER_SIZE` should be changed in **sysVme.h** to 0x20000000 to match the size of the requirement for Board 1:

```
#define VME_A32_MASTER_SIZE 0x20000000
```

Three more PCI slave images must still be configured to meet the interface requirements. In this example, the optional fourth PCI slave image may be used for the one of these three PCI slave images.

To properly configure the P2P bridges, change the value of `VME_A32_MASTER_2_SIZE` in **sysVme.h** to 0x30000000. The value of `VmeA32Master2Base` should also be set in **vxbsp.ini**, for example to 0x40000000. This can be done from the Midas monitor with the line:

```
#miset VmeInterface VmeA32Master2Base 0x40000000 vxbsp.ini
```

After rebooting, uniImageShow shows:

```
-> uniImageShow
Universe PCI slave images (VME master windows):
Image Type Local   PCI Base VME Base Size      VMEAM PWEN VDW
-----
0    MEM  e1000000 e1000000 00000000 00010000 2d   Y   32
1    MEM  b0000000 b0000000 00000000 01000000 3d   Y   32
2    MEM  90000000 90000000 10000000 20000000 0d   Y   32
3    MEM  b1000000 b1000000 40000000 30000000 0d   Y   32

Universe VME slave images (PCI master windows):
Image Type Local   PCI      VME      Size      VMEAM codes PWEN PREN LD64 LRMW
-----
5    MEM  00000000 00000000 00000000 10000000 09 0a 0d 0e Y   Y   N   N.
```

The following command reconfigures the optional PCI slave image window:

```
uniPciSlaveImageSet (3,
                      VmeA32Master2PciBase,
                      0x40000000,
                      0x10000000,
                      UNI_PCI_MEMORY_SPACE,
                      0x0d,
                      UNI_VMEBUS_DATAWIDTH_32,
                      TRUE);
```

The following two commands configure the additional PCI slave image windows:

```
uniPciSlaveImageSet (4,
                      VmeA32Master2PciBase + 0x10000000,
                      0x60000000,
                      0x10000000,
                      UNI_PCI_MEMORY_SPACE,
                      0x0d,
                      UNI_VMEBUS_DATAWIDTH_32,
                      TRUE);
```

```
uniPciSlaveImageSet (5,
                    VmeA32Master2PciBase + 0x20000000,
                    0x80000000,
                    0x10000000,
                    UNI_PCI_MEMORY_SPACE,
                    0x0d,
                    UNI_VMEBUS_DATAWIDTH_32,
                    TRUE);
```

At this point, all VME interface requirements are met. This is confirmed with uniImageShow:

```
-> uniImageShow
Universe PCI slave images (VME master windows):
```

Image	Type	Local	PCI Base	VME Base	Size	VMEAM	PWEN	VDW
0	MEM	e1000000	e1000000	00000000	00010000	2d	Y	32
1	MEM	b0000000	b0000000	00000000	01000000	3d	Y	32
2	MEM	90000000	90000000	18000000	20000000	0d	Y	32
3	MEM	b1000000	b1000000	40000000	10000000	0d	Y	32
4	MEM	c1000000	c1000000	60000000	10000000	0d	Y	32
5	MEM	d1000000	d1000000	80000000	10000000	0d	Y	32

```
Universe VME slave images (PCI master windows):
```

Image	Type	Local	PCI	VME	Size	VMEAM codes	PWEN	PREN	LD64	LRMW
5	MEM	00000000	00000000	00000000	10000000	09 0a 0d 0e	Y	Y	N	N

5.3 VME Interrupts

The Universe chip provides the means to:

- Handle VME interrupts on a selected set of levels (i.e. act as interrupt controller) and notify the PPC440GX processor with a PCI interrupt and the vector from the interrupt source.
- Generate VME interrupts on all levels and automatically communicate a user specified interrupt vector to the interrupt controller.

The Universe driver in the M5xxx BSP implements functions that support both VME interrupt handling and generation. “VME Interrupt Handling” on page 74 describes how to setup and handle VME interrupts. “VME Interrupt Generation” on page 75 describes how to generate VME interrupts.

VME Interrupt Handling

All boards may generate interrupts on all lines - in contrast to that there may only be one interrupt controller (destination) for a given interrupt line. (I.e. there may be several interrupt sources and only one interrupt destination for a given VME interrupt line.) Therefore, as board configuration and selection of interrupt controllers is user system specific, VME interrupt handling is not enabled by default in the M5xxx BSP. You should determine which boards should handle which interrupt levels and activate the hardware (Universe) on each board accordingly.

The `uniPciIntEnable()` function is used to activate and route interrupts from a user-specified VME interrupt line to a user specified PCI interrupt PIN. This function is declared as follows:

```
STATUS uniPciIntEnable
(
    int lint,          /* PCI interrupt pin */
    int source        /* Universe interrupt source */
);
```

<source> is one of the following:

```
UNI_INT_ACFAIL - AC fail signal
UNI_INT_SYSFAIL - Sys Fail signal
UNI_INT_SW_INT - Software interrupt
UNI_INT_SW_IACK - Software interrupt Acknowledged
UNI_INT_VERR - VME bus error
UNI_INT_LERR - PCI bus error
UNI_INT_DMA - DMA controller interrupt
UNI_INT_VIRQ7 - VME interrupt level 7
UNI_INT_VIRQ6 - VME interrupt level 6
UNI_INT_VIRQ5 - VME interrupt level 5
UNI_INT_VIRQ4 - VME interrupt level 4
UNI_INT_VIRQ3 - VME interrupt level 3
UNI_INT_VIRQ2 - VME interrupt level 2
UNI_INT_VIRQ1 - VME interrupt level 1
UNI_INT_VOWN - VME ownership interrupt
```

<lint> is one of the following:

```

UNI_INT_LINT7 - PCI interrupt pin 7
UNI_INT_LINT6 - PCI interrupt pin 6
UNI_INT_LINT5 - PCI interrupt pin 5
UNI_INT_LINT4 - PCI interrupt pin 4
UNI_INT_LINT3 - PCI interrupt pin 3
UNI_INT_LINT2 - PCI interrupt pin 2
UNI_INT_LINT1 - PCI interrupt pin 1
UNI_INT_LINT0 - PCI interrupt pin 0

```

It is also required to implement and register a function that can act as the interrupt service routine (ISR) for a given interrupt source. When generating the VME interrupt, the interrupt source communicates a vector that will be used by the main interrupt service routine of the Universe driver (implemented internally in the M5xxx BSP) in order to lookup the ISR registered for the interrupt source.

The ISR must be registered with the VxWorks function called `intConnect` which is declared in `$(WIND_BASE)/target/h/intLib.h` as follows :

```

STATUS intConnect
(
    VOIDFUNCPTR* vector,
    VOIDFUNCPTR routine,
    int parameter
);

```

<vector> is the vector communicated by the interrupt source.

<routine> is a function pointer to the interrupt service routine for the interrupt source/device.

<parameter> is a user-defined value that will be sent as the one and only parameter to the ISR when called.

VME Interrupt Generation

All boards may generate interrupts on all lines - in contrast to that there may only be one interrupt controller (destination) for a given interrupt line. (I.e. there may be several interrupt sources and only one interrupt destination for a given VME interrupt line.)

In order to generate a VME interrupt the `uniVmeIntGenerate` function is called as follows:

```

STATUS uniVmeIntGenerate
(
    int level,
    int vector /* interrupt vector to return (0-255) */
)

```

Note – The Universe II only supports even vector numbers. The least significant bit is always 0.

<level> is VME interrupt level to generate.
 <vector> is a value in the range 0-255 that will be communicated to the interrupt controller in order for the interrupt controller to identify the interrupt source and call the correct interrupt service routine.

VxWorks Target Shell Example :

You need two boards for this example :

- BOARD1 - board that is to act as VME interrupt controller (handler)
- BOARD2 - board that is to act as VME interrupt source (device)

1. Register an interrupt service routine on BOARD1:

```
-> intConnect 0x30,logMsg, "\n\nhello\n\ngoodbye\n\n"
```

2. Tell the Universe to handle VME interrupt level 3, and signal to the 440GX on PCI interrupt line 0:

```
-> uniPciIntEnable 0,3
```

3. Generate the interrupt on BOARD2:

```
-> uniVmeIntGenerate 3,0x30
```

4. The following messages should appear on the console of BOARD1 :

```
-> interrupt:

hello

goodbye
```

5.4 Universe DMA Functionality

The Universe includes a built-in DMA controller that enables high-speed block transfers between PCI and VME. This enables high-throughput block transfers without the involvement of the CPU. The BSP includes a module, called **uniDmaLib**, which provides functions for interfacing with the Universe DMA controller. Both direct DMA transfers and chained DMA transfers are supported by this module.

Universe DMA Driver

In order to use the Universe DMA controller module of the BSP, include the following line at the top of the application source file:

```
#include "uniDmaLib.h"
```

The object module `<bsp directory>/mdrv/lib/uniDmaLib.o` must be loaded prior to loading the application. The **uniDmaLib.o** module may also be directly linked with the application.

The Universe DMA controller supports 2 modes of operation:

1. Direct mode transfers a single block of data between the PCI bus and the VME bus. Use `uniDmaDirect` function to initiate a direct DMA transfer.
2. Linked list (chained) mode transfers one or multiple blocks of data between the PCI bus and the VME bus. The DMA engine uses DMA command packets to describe how to transfer each block of data. Use `uniDmaChainCmdPkCreate` to create DMA command packets. Use the `uniDmaChain` function to initiate a chained DMA transfer. Use `uniDmaChainStop` to stop a chained DMA transfer.

Use the `uniDmaNotifyFncSet` to specify a function to be called when the DMA engine is done or halts due to an error situation.

Universe DMA Interface Functions

uniDmaLibInit

Synopsis	STATUS uniDmaLibInit (int show_release)
Description	This function initializes the Universe DMA library. If the parameter show_release is TRUE, the driver release info is printed to the console.
Returns	OK or ERROR
Notes	none

uniDmaDirect

```
STATUS uniDmaDirect
(
  UINT32 vmeAdrs,
  UINT32 pciAdrs,
  UINT32 byteCount,
  UINT32 vmeAmCode,
  BOOL pci64,
  int direction
)
```

Synopsis

This function commands the Universe DMA engine to transfer a single DMA block.

<vmeAdrs> is the VME bus address of the DMA block.

<pciAdrs> is the PCI bus address of the DMA block.

<byteCount> is the number of bytes in the DMA block.

<vmeAmCode> is the VME Address Modifier code to be used when transferring the DMA block.

If <pci64> is TRUE, then PCI dual address cycles are enabled.

<direction> is either UNI_DMA_V2L (0) meaning VME to PCI, or UNI_DMA_L2V (1) meaning PCI to VME.

Description

Returns

OK or ERROR

The `vmeAdrs` and `pciAdrs` parameters are not required to have any particular byte-alignment in memory. For example, a PCI or VME address of 0x00200001 can be used. However, the `vmeAdrs` and `pciAdrs` must be aligned to an 8-byte boundary with each other. For example, if a PCI address of 0x00200001 is used, some valid VME addresses are 0x10001001, 0x10001009, 0x10001011, etc. If `vmeAdrs` and `pciAdrs` are not aligned to an 8-byte boundary with each other, no DMA transfer is performed, even though `uniDmaDirect` returns with no error.

The DMA transfer will only succeed if the VME target device accepts the specified AM code. If no VME target accepts the AM code, a VME bus error will occur. There must be only one VME target device in the system accepting the specified AM code and VME address. Otherwise the results of this function are unpredictable.

The size of the DMA transfer can be as small as one byte and as large as 16 MB. If a "block" AM code is specified (such as 0x8, 0xb, 0xc, or 0xf), and the size of the transfer is less than 8 bytes, the Universe generates single-cycle accesses (i.e. using a "non-block" data AM code such as 0x9 or 0xd). The Universe also generates single-cycle accesses when `vmeAdrs` and `pciAdrs` are not aligned to 8-byte boundaries. See the Universe Reference Manual from Tundra Semiconductor Corp. for more information.

Notes

uniDmaChainCmdPktCreate

```

UNI_DMA_CHAIN_CMDPKT_NODE * uniDmaChainCmdPktCreate
(
  UINT32 vmeAdrs,
  UINT32 pciAdrs,
  UINT32 byteCount,
  UINT32 vmeAmCode,
  BOOL pci64,
  int direction,
  UNI_DMA_CHAIN_CMDPKT_NODE *prev,
  UNI_DMA_CHAIN_CMDPKT_NODE *next
)

```

Synopsis

<uniDmaChainCmdPktCreate> allocates a new DMA chain command packet and initializes it according to the arguments given.

<vmeAdrs> is the VME bus address of the DMA block.

<pciAdrs> is the PCI bus address of the DMA block.

<byteCount> is the number of bytes in the DMA block.

<vmeAmCode> is the VME Address Modifier code to be used when transferring the DMA block.

If <pci64> is TRUE then PCI Dual Address Cycles are enabled.

<direction> is either UNI_DMA_V2L (0) meaning VME to PCI, or UNI_DMA_L2V (1) meaning PCI to VME.

If the newly created command packet is to be part of an already existing chain of DMA command packets, <prev> should point to the <UNI_DMA_CHAIN_CMDPKT_NODE> structure representing the packet in front of the new one, and <next> should point to the <UNI_DMA_CHAIN_CMDPKT_NODE> structure representing the next packet in the chain. Both <prev> and <next> may point to NULL.

Description

Returns

Pointer to newly created <UNI_DMA_CHAIN_CMDPKT_NODE> structure on success or NULL on failure

The chain of DMA command packets is stored internally as a singly linked list. Please see the **uniDmaLib.h** file for the complete structure definition.

Internally, this function calls `cacheDmaMalloc` to allocate memory for the new node. Therefore, the memory for the node can be freed by calling `cacheDmaFree` with the returned value of `uniDmaChainCmdPktCreate` as the parameter. Note that freeing a node in this manner invalidates the chain of DMA command packets unless the user manually re-attaches the previous and next links of the chain.

Please see the Notes section of the `uniDmaDirect` function in regard to `vmeAdrs`, `pciAdrs`, `byteCount`, and `vmeAmCode`. These notes apply to the parameters of the `uniDmaChainCmdPktCreate` function when the created command packet is used in a chained DMA transfer.

Notes

uniDmaChain

Synopsis

```
STATUS uniDmaChain
(
  UNI_DMA_CHAIN_CMDPKT_NODE *cpp
)
```

Description This function commands the Universe DMA engine to run a chained DMA block transfer. <cpp> points to the first command packet node in the chain of DMA command packets to be executed.

Returns OK or ERROR

Notes none

uniDmaChainStop

Synopsis

```
STATUS unidDmaChainStop
(
  UNI_UCSR *ucsr
)
```

Description This function commands the Universe DMA engine to stop after the current DMA command packet node transfer is complete. If the DMA engine is not active, an error is returned. This function does not return until after the current command packet node DMA transfer is complete.

Returns OK or ERROR

Notes none

uniDmaNotifyFncSet

	<pre>STATUS uniDmaNotifyFncSet (FUNCPTR fnc, int arg)</pre>
Synopsis	<p>This function may be used to specify a function to be called when a DMA event occurs. A DMA event may be one of</p> <ul style="list-style-type: none">UNI_DMA_EVENT_DONE - DMA completion eventUNI_DMA_EVENT_LERR - DMA engine caused PCI bus errorUNI_DMA_EVENT_VERR - DMA engine caused VME bus errorUNI_DMA_EVENT_P_ERR - DMA protocol error event <p><fnc> points to the entry point of the function to be called on DMA event</p>
Description	<p><arg> is the first argument to be passed to <fnc></p>
Returns	<p>Always OK</p>
	<p>The function declaration for <fnc> is of the form:</p> <pre>void uniDmaNotifyFnc (int arg, int event);</pre>
	<p>The second argument passed to <fnc> is the DMA event that caused the call to <fnc>. This argument is one of</p> <ul style="list-style-type: none">UNI_DMA_EVENT_DONE - DMA completion eventUNI_DMA_EVENT_LERR - DMA engine caused PCI bus errorUNI_DMA_EVENT_VERR - DMA engine caused VME bus errorUNI_DMA_EVENT_P_ERR - DMA protocol error event
Notes	

6

RACEway PCI Interface

6.1 RACEway-PCI Interface

Overview

An M5xxx board with the “R” option (M5XXX-R) has an on-board RACEway-PCI interface ASIC chip, the PXB. The MIDAS BSP initializes the PXB (if present) to a known non-conflicting state in order to guarantee proper operation.

See also chapter 9 for information about the bundled PXB DMA Driver.

PXB Initialization

The PXB interface chip operates in either “bridge” mode or “endpoint” mode. In bridge mode, the PXB chip operates like a PCI P2P bridge. In endpoint mode, the PXB chip operates like a PCI device. The M5xxx BSP only supports the PXB operating in bridge mode. When an M5xxx-R board boots VxWorks, the BSP will attempt to initialize the PXB, and this includes placing the PXB in bridge mode. However, if the M5xxx-R is not installed in a RACEway VME slot (i.e., the slot's P2 is not overlaid with an ILK device), the PXB initialization process will cause the board to hang. The M5xxx monitor allows the user to switch the PXB initialization on or off, thus enabling an M5xxx-R board to be used in either a RACEway or non-RACEway VME slot.

- To enable PXB initialization, enter the following command at the VxWorks target shell prompt:
-> mfs_ini_settext("vxbsp.ini", "RACEdrv", "PxbInit", "TRUE")
- To disable PXB initialization, enter the following command at the VxWorks target shell prompt:
-> mfs_ini_settext("vxbsp.ini", "RACEdrv", "PxbInit", "FALSE")

Or omit this item altogether.

However, the ability to inhibit PXB initialization is potentially quite dangerous, especially in boards deployed in production system, because if the PXB chip is not initialized, RACEway-PCI data transfer will work with undefined result or not at all. In other words, RACEway-PCI communication requires PXB initialization. To insure that the BSP always initializes the PXB, see the function `sys_pxb_init` in `sysLib.c` and follow the instructions there.

When using PCI auto-configuration and PXB initialization is enabled, the auto-configurator reserves a secondary bus number to each PXB device, but it does not scan for PCI devices behind the PXB device, as it would normally do for other P2P bridges. The BSP provides a special mechanism that allows the PCI auto-configurator to reserve PCI memory and I/O address space for PXB-related applications. This mechanism is available through a set of parameters in the "RACEdrv" section of the `vxbsp.ini` file. The PXB-related parameters are shown in Table 6-1

TABLE 6-1. PXB-related flags used by M5xxx BSP

Parameter name	Meaning	Values	Default
PxbInit	Initialize PXB	TRUE, FALSE	FALSE
PxbPrefMemSize	Size of prefetch memory pool for PXB	Power of 2, 1MiB - 512MiB	256 MiB
PxbMemIoSize	Size of non-prefetch memory pool for PXB	Power of 2, 1 MiB - 64 MiB	0 (disabled)
PxbIoSize	Size of PCI I/O pool for PXB	Power of 2, 16 bytes - 256 KiB	0 (disabled)

Note – If a M5xxx-R board hangs during VxWorks boot, first check to see if the VME slot is a RACEway slot. If the slot is a non-RACEway slot, reset the board, enter the M5xxx monitor, disable PXB initialization then reset the board and allow it to boot VxWorks.

PXB DMA Driver

The PXB DMA driver is sold as part of a separate product called "RACE driver for VxWorks" (RACE-DRV-VXWORKS) and is documented in the PXB DMA Driver Software Reference Manual.

7

Network

This chapter documents various network interfaces supported by the M5xxx BSP. The conditional `#define INCLUDE_NETWORK` in **config.h** controls the inclusion of network protocol and device driver support by the M5xxx BSP.

7.1 Ethernet (emac) Network Interface

The PPC440GX CPU includes four Ethernet interfaces, generally referred to as “emac”. The M5xxx BSP includes support for the emac interface using the Wind River Enhanced Network Device (END) driver model which is enabled when INCLUDE_END is defined in **config.h**.

The M5xxx may support up to 3 simultaneous Ethernet connections, depending on the specific configuration of the board. In all configurations, at least one Ethernet connection will always be available. Either emac0 or emac2 will be available in all configurations. The other ports that may be available are emac1 or emac3.

- emac0 is the onboard fast Ethernet port (10/100 Mbps)
- emac1 is an IOSpacer fast Ethernet port (10/100 Mbps)
- emac2 is an IOSpacer gigabit Ethernet port (10/100/1000 Mbps)
- emac3 is an IOSpacer gigabit Ethernet port (10/100/1000 Mbps)

Configuring the boot parameters for the emac interface may be done using the 'c' command in the VxWorks bootrom, or through the 'bootChange()' command of a running VxWorks system. An example of boot parameters for the emac interface is shown in Figure 7-1

```
boot device           : emac0
processor number      : 0
host name             :
file name             : /usr/wind/target/config/bspname/vxWorks
inet on ethernet (e) : 192.168.100.10
inet on backplane (b) :
host inet (h)         : 192.168.100.1
gateway inet (g)      :
user (u)              : fred
ftp password (pw) (blank=use rsh) :
flags (f)             : 0
```

FIGURE 7-1. Example Boot parameters for the emac interface

Configuring JUMBO packets

The Ethernet Network Interface driver supports JUMBO packets for both the fast Ethernet and Gigabit Ethernet channels. JUMBO packets is a feature of gigabit Ethernet controllers, which basically increases the Ethernet frame size (also referred to as MTU - Maximal Transmission Unit) from 1500 bytes (typically) to several kilobytes. The maximum Ethernet frame size depends on the Ethernet controller.

Ethernet communication with JUMBO packet MTU sizes gives better performance than smaller packets because the number of interrupts and CPU overhead is reduced per byte transmitted. This is of benefit where it is necessary to maximize Gigabit Ethernet throughput potential as much as possible.

The use of JUMBO packets is not needed in order to maximize the throughput potential of Fast Ethernet. By default the fast Ethernet channels are configured to use 1500 bytes Ethernet frame sizes (MTU) while the Gigabit Ethernet channels are configured to use 8000 bytes Ethernet frame sizes.

When the Ethernet Network Interface communicates with other Ethernet network devices, the network devices will negotiate and use the smallest of the maximum frame sizes of both of the devices. The configuration of MTU sizes should therefore not break communication between network devices.

The PPC440GX supports MTU sizes up to 9000 bytes. The default MTU size is set to 8000, this is because 8000 is better aligned with the MAL buffer sizes than the maximum MTU (9000), and therefore gives higher performance.

MTU size can be configured in **config.h**

```

/*
 * EMAC MTU Size.
 *
 * MTU Size is the maximum packet size the EMAC is configured to use when
 * communicating with other Ethernet devices.
 * Maximum MTU size for the PPC440GX is 9000.
 *
 * The Fast Ethernet EMACS have a default value of 1500.
 * The Gigabit Ethernet EMACS have a default value of 8000.
 * The default value of 8000 for the Gigabit Ethernet EMACS gives an optimal
 * performance due to alignment with the MAL buffer size.
 */

#define EMAC0_MTU      1500 /* Fast Ethernet Channel */
#define EMAC1_MTU      1500 /* Fast Ethernet Channel */
#define EMAC2_MTU      8000 /* Gigabit Ethernet Channel */
#define EMAC3_MTU      8000 /* Gigabit Ethernet Channel */

```

For most users the default configuration should work fine.

7.2 Shared Memory (sm) Backplane Network Interface

1. M5xxxboards. If the M5xxx boards are attached, then the problem is more likely to be with the gateway or with the host system configuration.
2. You can use host system utilities, such as arp, netstat, etherfind, and ping, to study the state of the network from the host side.

7.3 Gigabit Ethernet Throughput Performance

Most Ethernet transfer protocols use the IP stack when transferring data across Ethernet. The IP stack is processor demanding and therefore it is important to configure and use the stack optimally in order to get high throughput.

In order to improve Gigabit Ethernet throughput the BSP has implemented support for JUMBO packets. This feature enhances the Ethernet throughput performance significantly.

The maximum size of the JUMBO packets for the PPC440GX is 9000 bytes. By default the size is set to 8000 since this gives better alignment with the internal buffer sizes in the PPC440GX.

It is important that Gigabit Ethernet switches that are used together with JUMBO packets have support for JUMBO packets and that this feature is enabled. Otherwise communication will fail.

It is also important that both of the devices that communicate with Gigabit Ethernet have support for JUMBO packets and that this feature is enabled. Otherwise the standard MTU size of 1500 will be used, which will slow down the throughput significantly.

In order to get maximum throughput performance on Gigabit Ethernet, the JUMBO packets should be enabled and set to size 8000 as they are by default (look for `EMAC2_MTU` and `EMAC3_MTU` in `config.h`). Optimizations of the IP stack are also configured by default in `config.h`.

It is important to increase the TCP and/or UDP socket buffer sizes in order to get the maximum throughput. This can be done for test purposes by setting global variables from the VxWorks shell prompt as follows:

```
-> tcp_sendspace=0x38000
-> tcp_recvspace=0x38000
-> udp_sendspace=0x38000
-> udp_recvspace=0x38000
```

A better approach however, is to use the command `setsockopt(...)` to set the buffer sizes to the same values as above on socket level. This is to avoid that all TCP and UDP sockets use such large buffer sizes.

```
/*
 * Set SO_SNDBUF and SO_RCVBUF for TCP and UDP.
 */
{
    int soSndBufSize = 0x38000;
    int soRcvBufSize = 0x38000;

    if (setsockopt(fd, SOL_SOCKET, SO_SNDBUF, (char *) &soSndBufSize,
        sizeof(soSndBufSize)) < 0)
        printf("ERROR:setsockopt failed for SO_SNDBUF.\n");
    if (setsockopt(fd, SOL_SOCKET, SO_RCVBUF, (char *) &soRcvBufSize,
        sizeof(soRcvBufSize)) < 0)
        printf("ERROR:setsockopt failed for SO_RCVBUF.\n");
}
```


8

BSP Installation

8.1 BSP Installation & Distribution

Installation

The M5xxx BSP is distributed on CD-ROM media. Refer to the CD-ROM for installation procedure. This BSP is only compatible with Tornado 2.2.1 (which includes VxWorks 5.5.1) with Patch 90451 applied. The patch can be obtained from Wind River Systems technical support.

Files & Directories

The following is a summary of M5xxx PPC440GX BSP software distribution. This distribution contains all the BSP specific files that allows VxWorks to run on the M5xxx.

doc - this directory contains all of the documentation for the BSP

mdrv - this directory contains all M5xxx-specific drivers and associated header files.

mdrv/include - this directory contains all the BSP user-include header files.

mdrv/include/Common.h - numerous common definitions used by the BSP

mdrv/include/flash.h - flash memory driver header file

mdrv/include/flashlib.h - flash storage device interface library header file

mdrv/include/i2o_mu.h - I2O message unit header file

mdrv/include/iospacer.h - header file supporting io spacer

mdrv/include/mfs.h - header file supporting the Midas File System in Flash

mdrv/include/mfs_ini.h - .ini file support for Midas File System

mdrv/include/mfs_usr.h - header file for support of Midas File System (mfs_user functions)

mdrv/include/midas.h - header file containing many M5xxx board-specific addresses and other hardware defines.

mdrv/include/MidasIntLib.h - header file with MIDAS interrupt routing support

mdrv/include/MidasPciLib.h - M5xxx-specific PCI usage header file

mdrv/include/midasppc440.h - header file with PPC440GX-specific I/O addresses and constants

mdrv/include/model.h - header file associated with identifying model number of Midas products

mdrv/include/mu.h - message unit definition header file

mdrv/include/p2pLib.h - header file for PCI-to-PCI (P2P) bridge library

mdrv/include/pcicfg.h - PCI configuration header file

mdrv/include/pciLib.h - PCI library header file

mdrv/include/ppc440DmaLib.h - header file for the PPC440 DMA

mdrv/include/pxb.h - header file with PXB driver definitions

mdrv/include/pxb_bsp_midas.h - header file with Midas BSP-specific PXB driver definitions

mdrv/include/sprom.h - header file supporting sprom.c

mdrv/include/sprom_ppc440.h - PPC440 specific SPROM definitions

mdrv/include/sysLibMidas.h - header file for extra sysLib functionality added to the M5xxx BSP

mdrv/include/uniDmaLib.h - header file for Universe DMA engine

mdrv/include/uniLib.h - header file for Universe library

mdrv/lib - contains all the BSP archive libraries.

mdrv/lib/libmdrvs.a - this is an archive library that contains all the BSP drivers. The object modules in this library are linked in on demand as part of VxWorks.

440gxBusErr.s - PPC440GX-dependent bus error reporting

440gxBusErrOriginShow.c - used for decoding the bus error value from `sysBusErrRegsGet`

config.h - this file contains configuration parameters for VxWorks. This file may be edited directly. Usually, BSP development from the command prompt is the desired approach. However, the BSP may be developed from within Tornado itself, in which case, the configuration parameters may be changed from within Tornado.

configNet.h - Platform-independent network configuration

cprDcr.h - IBM chip clocking and PDR DCR definitions

dmaDcr.h - PPC440GP DMA controller DCR access assembly routines

ebcDcr.h - IBM external bus controller DCR definitions

emacEnd.c/h - EMAC Ethernet driver with patch 90835

L2Lib.c - Level 2 cache controller support

Makefile - this file performs the compilation and linking steps to build various VxWorks images as requested by the user.

maDcr.h - IBM Memory Access Layer (MAL) DCR access assembly routines

midas_ppc440.h - MIDAS PPC440GX board (M5xxx) header file

pciAutoConfigLib.c - modified version of the VxWorks PCI autoconfigurator that supports the M5xxx

phyLib.c/.h - MII/physical network configuration code and header

ppc440gx.h - IBM PPC440GX specific header file

ppc440Timer.c - PPC440GX timer function library

romInit.s - Assembly source code supporting power up reset

sdramDcr.h - IBM SDRAM controller DCR access assembly routines

sdrDcr.h - IBM system DCR register definitions

sysALib.s - this is the assembly source code supporting sysLib.c

sysBusPci.c - Support routines for PCI auto-configuration

sysCpcr.c/.h - IBM PPC440GX clocking & power and system device access header

sysDcr.h/.s - DCR access header and assembly routines

sysLib.c - this is the primary BSP source file

sysNet.c - Ethernet hardware initialization

sysSerial.c - Serial hardware initialization

sysVme.c/.h - Functions for VMEbus support

uicDcr.h - IBM Universal Interrupt Controller (UIC) DCR access definitions and assembly routines

uicIntr.c - PPC440GX IBM Universal Interrupt Controller (UIC) library

usrExtra.c - VxWorks support for optionally included modules

zmiiLib.h - ZMII macro header

9

Burning VxWorks Boot Code

There are two methods of burning VxWorks boot code:

- **Ethernet** - This is the preferred method and should be used where Ethernet connection is available.
- **Serial Port** - If no Ethernet access is available or VxWorks does not boot, then the Rom Monitor (serial load) must be used to burn VxWorks boot code.

9.1 Burning VxWorks Boot Code from Rom Monitor (Serial)

On the M5xxx board, VxWorks boot code is stored in non-removable FLASH. The board is always first booted by the M5xxx low-level monitor program which will then boot VxWorks automatically. The M5xxx monitor is used to download VxWorks boot code into FLASH via the serial port and file transfer with the KERMIT protocol. The M5xxx monitor prompt is the symbol #. Follow the steps below to burn VxWorks boot code into FLASH:

1. Make the VxWorks boot file if one does not exist. This is typically “bootrom.bin”, but other targets that produce binary may be used, as appropriate for the application.
2. Use the `set-baudrate` command to configure the baudrate for the download operation. Baud rates up to 115200 bps are supported.
3. Connect to the M5xxx board's serial port with any terminal emulator program (such as HyperTerminal in Windows) using the configured baudrate. The terminal emulator program must be able to send files with the KERMIT protocol.
4. At the M5xxx monitor prompt, download the boot file. (This procedure automatically erases the appropriate region of FLASH, so it is not necessary to explicitly erase the FLASH.)

```
#serial-load 0xf4000000 [0x<filesize>]
```

Where the optional parameter <filesize> is the length of the file in Bytes.

- Escape back to the host and send the boot file with the KERMIT protocol.
- Wait for file transfer to finish. There will be a number of progress messages displayed as the M5xxx monitor writes the bootcode to FLASH.

Note – after the serial-load command is given, the board waits to receive the boot file, accepting no further input until file transfer is done or time-out exception.

9.2 Burning VxWorks Boot Code from VxWorks (Ethernet)

In order to program a boot image from VxWorks, Ethernet must be configured properly. Details about how this is done are described in “Ethernet (emac) Network Interface” on page 88.

Burning boot code into Flash from VxWorks is done as follows:

1. Make sure that the Ethernet connection is working. For example, use the command “ping” or “ls”.
2. Program the boot image through: `flashLoad "<path>/<filename>", <address>`

Example:

```
flashLoad "/tornado-2.2.1.90451-ppc/target/config/midas-ppc440-bsp1.2-
r1.3.1/bootrom.bin", 0xf4000000
```

The address 0xf4000000 is the default address to program boot images. This address has to match ROM_TEXT_ADRS in **config.h** and **Makefile**. ROM_TEXT_ADRS is possible to change, but it must be within the Flash address area.

If the Flash is locked by switch settings, the `flashLoad` command will inform the you about this. The M5000 User Guide explain how the locking/unlocking of Flash through DIP switches is performed. The default address 0xf4000000 is not locked by fabric switch settings.

9.3 Burning VxWorks Boot Code from U-Boot (Ethernet)

The M5000 board is shipped with a U-Boot image programmed in Flash memory. U-Boot is a bootloader with an ethernet driver which is used to load and boot Linux on the M5000 board.

Additionally, U-Boot can load and flash VxWorks images across Ethernet - which is the preferred method. Serial loading with the MIDAS Monitor is the only alternative.

In order to download a VxWorks binary image to the M5000, U-Boot supports two communication protocols: TFTP (client) and NFS (client). The VxWorks image must be available on a networked TFTP server or NFS server in order to load the image. The U-Boot command sequence in order to download a VxWorks image is described below:

Setting Network Parameters

The following command sequence is required to setup the network parameters for both TFTP and NFS download methods:

1. Specify which ethernet port to use (ppc_440x_eth0, ppc_440x_eth1, ppc_440x_eth2 or ppc_440x_eth3):
=> setenv ethact ppc_440x_eth0
2. Set the IP address of the M5000 board:
=> setenv ipaddr 192.168.168.175
3. Set the hostname of the M5000 board:
=> setenv hostname c5000
4. Set the IP address of the NFS or TFTP server:
=> setenv serverip 192.168.168.4

TFTP:

1. Put the vxWorks image in the */tftpboot* directory on the TFTP server
2. Download the VxWorks image (**bootrom.bin** or **vxWorks.st_rom.bin**) into SDRAM at address 01000000:
=> tftpboot 1000000 bootrom.bin

NFS:

1. Put the vxWorks image in the exported directory on the NFS server (for instance */export/images/bootrom.bin*).
2. Download the VxWorks image (**bootrom.bin** or **vxWorks.st_rom.bin**) into SDRAM at address 01000000:
=> nfs 1000000 /export/images/bootrom.bin

Flash VxWorks image

The following command sequence is required in order to flash the vxWorks image for both TFTP and NFS download methods:

1. 1. Unlock the flash blocks where the VxWorks image should be written
=> `protect off f4000000 f41ffffff`
2. 2. Erase the flash blocks where the VxWorks image should be written
=> `erase f4000000 f41ffffff`
3. 3. Copy the VxWorks image from SDRAM to flash (the filesize parameter is updated automatically by U-boot when the file is downloaded) :
=> `cp.b 1000000 f4000000 $(filesize)`

10 *DMA drivers*

The M5xxx has several on-board DMA controllers including the PPC440GX, Universe, and PXB, which move data between DRAM, PCI memory, VME, and RACEway. DMA drivers are included in the M5xxx BSP for the PPC440GX.

10.1 PPC440GX DMA Driver

The PPC440GX DMA Driver is capable of moving data to/from any memory reachable from the PPC440GX. However for performance reasons this driver is mostly applicable for data transfers involving system DRAM. Once the DMA transfer is initiated data is moved without any need for processor intervention. The driver supports one channel transfers, where the transfers can be single DMA transactions or chained (scatter/gather) DMA transactions. To use the PPC440GX DMA driver include the **ppc440DmaLib.h** header file from the BSP distribution.

Setting up a DMA transaction

A DMA transaction is set up by using a DMA transaction structure. The structure is defined as follows:

```
typedef struct ppc440DmaTransactionInfo
{
    int Channel; /* Always 0 */
    int Alignment;
    uint32_t TransferByteCount;
    uint32_t SourceAddressLow;
    uint32_t SourceAddressHigh;
    uint32_t DestinationAddressLow;
    uint32_t DestinationAddressHigh;
    int Status;
    int DetailedStatus;
    /* Pointer to callback routine */
    int (*Callback)(int Status,
        int DetailedStatus,
        void *UserContext);
    /* User applied callback parameter */
    void *UserContext;
} ppc440DmaTransaction_t;
```

The fields are used as follows:

Channel: DMA channel to use. Must be set to 0.

Alignment: Alignment of user buffers. The alignment affects the transfer line width used by the DMA engine to transfer data, and thus the performance. The user application can either force this field to an alignment or leave it up to the driver to decide. If forced then the driver will return with an error if the user buffers are not correctly aligned. Values used for this field are as follows:

DMA_ALIGNED_SELECT (0)	Alignment selected by driver.
DMA_ALIGNED_1 (1)	1 byte aligned.
DMA_ALIGNED_2 (2)	2 bytes aligned.
DMA_ALIGNED_4 (4)	4 bytes aligned.
DMA_ALIGNED_8 (8)	8 bytes aligned.
DMA_ALIGNED_16 (16)	16 bytes aligned.

TransferByteCount: Size of transaction in bytes. The DMA engine supports transferring 1024k transfer lines. The transfer line width is dependent on the alignment, and thus the maximum value of this field is 1024k times the alignment.

Buffer addresses: Addresses of the source and destination buffers as seen from the processor local bus (PLB). Translation functions from local and PCI addresses are provided by the driver. See “Address translation functions” on page 114 for a description. The buffers must be aligned as implied by the alignment field.

Status: Status of operation where applicable. Always set OK or ERROR.

DetailedStatus: Detailed status of operation where applicable. See header file for values.

Callback: Pointer to callback function. If this field is set to NULL then blocking mode is implied.

UserContext: Pointer to user context used in non-blocking mode. The callback function will be called with this field as a parameter.

Some fields in the structure are only applicable to the given user mode. For single transactions the driver supports both blocking and non-blocking modes. Chained transactions only support non-blocking mode.

Single DMA transactions

Transfers involving only one transaction structure are most efficiently executed through the `ppc440DmaXfer` function. This function supports both blocking and non-blocking mode. Blocking mode is chosen by setting the callback field in the transaction structure to NULL. The `ppc440DmaXfer` function is defined as follows:

```
int ppc440DmaXfer
(
    ppc440DmaTransaction_t *trans,
    int timeout,
    ppc440DmaStatus_t *dmaStatus
);
```

The fields are used as follows:

trans: Pointer to a DMA transaction structure. See “Setting up a DMA transaction” on page 110 for a description.

timeout: Driver timeout for execution. The PPC440GX DMA engine does not support timeouts. Thus this timeout is only used in software when waiting for control over the engine.

dmaStatus: Pointer to a status structure. See “Common status structure” on page 114 for a description. This structure is only used on ERROR.

The return value for the `ppc440DmaXfer` functions only reflects whether the driver was able to initiate the transfer or not. To find the status of the actual transfer the relevant fields in the DMA transaction structure must be checked.

Chained DMA transactions

Transfers involving one or more DMA transactions are executed through the `ppc440DmaChainXfer` function. DMA chains can only be executed in a non-blocking mode. Before a chain can be executed a DMA descriptor must be created through the `ppc440DmaChainDescCreate` function. This function is defined as follows:

```
ppc440DmaChainDesc_t *ppc440DmaChainDescCreate
(
    ppc440DmaTransaction_t *trans,
    ppc440DmaChainDesc_t *prev,
    ppc440DmaChainDesc_t *next,
    ppc440DmaStatus_t *dmaStatus
);
```

The fields are used as follows:

trans: Pointer to a DMA transaction structure. See “Setting up a DMA transaction” on page 110 for a description. Since DMA chains can only be executed in non-blocking mode, the user application must fill in the callback fields. The user application may choose not to have a callback on every transaction in the chain. This is however not advised since there is no way to find out if the given transaction failed or not.

prev: Pointer to a DMA descriptor. If this field is not NULL, then the new descriptor is added in the chain after the previous descriptor.

next: Pointer to a DMA descriptor. If this field is not NULL, then the new descriptor is added in the chain before the next descriptor.

dmaStatus: Pointer to a status structure. See “Common status structure” on page 114 for a description of this structure. This structure is only used if the create function for some reason failed (i.e. returned NULL).

If the `ppc440DmaChainDescCreate` function succeeded in creating a new descriptor then a pointer to this descriptor is returned. If not the return value is NULL.

To remove a descriptor the `ppc440DmaChainDescRemove` function is used. This function is defined as follows:

```
int ppc440DmaChainDescRemove
(
    ppc440DmaChainDesc_t *desc,
    ppc440DmaStatus_t *dmaStatus
);
```

The fields are used as follows:

desc: Pointer to the descriptor to be removed from the chain.

dmaStatus: Pointer to a status structure. See “Common status structure” on page 114 for a description. This field is only used on ERROR. This field may be set to NULL if status is not wanted.

This function will return OK or ERROR depending on if the remove succeeded or not. Further status may be retrieved through the status structure. When a descriptor is removed the rest of the chain is modified to keep the chain consistent.

The DMA chain is executed through the `ppc440DmaChainXfer` function. This function is defined as follows:

```
int ppc440DmaChainXfer
(
    ppc440DmaChainDesc_t *desc,
    int timeout,
    ppc440DmaStatus_t *dmaStatus
);
```

The fields are used as follows:

desc: Pointer to a DMA descriptor. The user application may choose which descriptor in the chain to start at. If the chain is to be started from the beginning then the user application must make sure that this field is a pointer to the first descriptor.

timeout: Driver timeout for execution. The PPC440GX DMA engine does not support timeouts. Thus this timeout is only used in software when waiting for control over the engine.

dmaStatus: Pointer to a status structure. See “Common status structure” on page 114 for a description of this structure. This structure is only used on ERROR.

Common status structure

Several of the function calls in the PPC440GX DMA driver interface use a common status structure. This structure is defined as follows:

```
typedef struct ppc440DmaStatus
{
    int status_code;
    char status_text[256];
} ppc440DmaStatus_t;
```

The fields are used as follows:

status_code: Integer value representing the status.

status_text: Symbolic string describing the status.

Usually this structure is used to describe an error condition. See the relevant function call for use.

Address translation functions

The PPC440GX DMA driver expects processor local bus (PLB) addresses as input. The driver provides functions to translate both local addresses and PCI addresses to PLB addresses. These are:

```
int ppc440LocalAdrsToPlbAdrs
(
    uint32_t localAdrs,
    uint32_t *plbAdrsLo,
    uint32_t *plbAdrsHi
);
```

```
int ppc440PciAdrsToPlbAdrs
(
    int adrsSpace,
    uint32_t pciAdrsLo,
    uint32_t pciAdrsHi,
    uint32_t *plbAdrsLo,
    uint32_t *plbAdrsHi
);
```

Each function will return OK or ERROR depending on if the translation succeeded or not. The `ppc440PciAdrsToPlbAdrs` function currently supports PCI MEMORY address space and 32 bits addresses.

Single blocking DMA transfer example

The following source code is an example of a single blocking DMA transfer. The code starts in the `doSingle` function.

```
#include "vxWorks.h"
#include "ppc440DmaLib.h"

extern int logMsg(char *fmt, ...);

/* ppc440DmaXferBlocking is a simple interface to the ppc440DmaLib BSP
module.
 * THE ALIGNMENT WILL BE DETERMINED BY THE DMA DRIVER. MAX TRANSFER SIZE
IS
 * DEPENDENT ON THE ALIGNMENT.
 * sourceBuffer is the local address of the data to be DMA'ed
 * destinationBuffer is the local address of where to place the data
 * bufferSizeBytes is the size of the buffer to transfer
 */

int ppc440DmaXferBlocking
(
    uint32_t sourceBuffer,
    uint32_t destinationBuffer,
    int     bufferSizeBytes
)
{
    int status = OK;
    ppc440DmaTransaction_t trans;
    ppc440DmaStatus_t dmaStatus;
    char funcName[] = "ppc440DmaXferBlocking";

    /* Set up the transfer data structure */
    trans.Alignment = DMA_ALIGNED_SELECT; /* Driver selected alignment */
    trans.Callback = NULL;
    trans.UserContext = NULL;
    trans.Channel = 0;
    trans.Status = 1234;
    trans.DetailedStatus = 4321;
    trans.TransferByteCount = bufferSizeBytes;
}
```

```

ppc440LocalAdrsToPlbAdrs (sourceBuffer,
                          &(trans.SourceAddressLow),
                          &(trans.SourceAddressHigh));

ppc440LocalAdrsToPlbAdrs (destinationBuffer,
                          &(trans.DestinationAddressLow),
                          &(trans.DestinationAddressHigh));

status = ppc440DmaXfer(&trans, -1, &dmaStatus);

if ( status != OK)
{
    logMsg("%s: xfer FAILED. Status = 0x%x, Text = %s\n",
          funcName, dmaStatus.status_code, dmaStatus.status_text,
          0,0,0);
}
else
{
    if (trans.Status != OK)
    {
        logMsg("%s: transaction FAILED. Status = %d, DetailedStatus =
0x%x\n",
              funcName, trans.Status, trans.DetailedStatus,0,0,0);
        status = ERROR;
    }
}

return(status);
}

```

Chained DMA transfer example

The following source code is an example of a chained DMA transfer. The code starts in the doChain function.

```

#include "vxWorks.h"
#include "stdio.h"
#include "cacheLib.h"
#include "taskLib.h"
#include "ppc440DmaLib.h"
void setPattern(uint32_t *buf, int size)

```

```

    {
        for (; size > 0; size--, buf++)
        {
            *buf = size;
        }
    }

int checkPattern(uint32_t *buf, int size)
{
    for (; size > 0; size--, buf++)
    {
        if (*buf != size)
        {
            return (ERROR);
        }
    }
    return (OK);
}

void erasePattern(uint32_t *buf, int size)
{
    for (; size > 0; size--, buf++)
    {
        *buf = 0xffffffff;
    }
}

static int doXferChained_totalIntr;
static int doXferChained_status;

int doXferChainedIsr
(
    int Status,
    int DetailedStatus,
    void *UserContext
)
{
    doXferChained_totalIntr++;
    if (doXferChained_status)
    {
        /* Already set donit overwrite */
    }
}

```

```
        return (OK);
    }

    if (Status)
    {
        doXferChained_status = DetailedStatus;
    }

    if (UserContext != NULL)
    {
        doXferChained_status = ERROR;
    }
    return (OK);
}

static int doXferChained
(
    int width,
    int bufferSize,
    uint32_t sourceBuffer,
    uint32_t destinationBuffer,
    int descs
)
{
    int status = OK;
    int cnt;
    ppc440DmaTransaction_t trans;
    ppc440DmaStatus_t dmaStatus;

    ppc440DmaChainDesc_t *desc[10] = {0,0,0,0,0,0,0,0,0,0};
    ppc440DmaChainDesc_t *prev = NULL;

    if (descs > 10)
    {
        printf("Number of descs larger than 10\n");
        return (ERROR);
    }

    setPattern( (uint32_t *)sourceBuffer, bufferSize/4);
    erasePattern( (uint32_t *)destinationBuffer, bufferSize/4);
```

```

trans.Alignment = width;
trans.Callback = doXferChainedIsr;
trans.UserContext = NULL;
trans.Channel = 0;

ppc440LocalAdrsToPlbAdrs (sourceBuffer,
                          &(trans.SourceAddressLow),
                          &(trans.SourceAddressHigh));

ppc440LocalAdrsToPlbAdrs (destinationBuffer,
                          &(trans.DestinationAddressLow),
                          &(trans.DestinationAddressHigh));

trans.TransferByteCount = bufferSize;
trans.Status = 1234;
trans.DetailedStatus = 4321;

for (cnt = 0; cnt < desc; cnt++)
{
    desc[cnt] = ppc440DmaChainDescCreate(&trans,
                                        prev,
                                        NULL,
                                        &dmaStatus);

    if ( !(desc[cnt]) )
    {
        status = ERROR;
        for (; cnt > 0; cnt--)
        {
            ppc440DmaChainDescRemove (desc[cnt-1], NULL);
        }
        break;
    }
    prev = desc[cnt];
}

if (status == OK)
{
    doXferChained_totalIntr = 0;
    doXferChained_status = OK;
}

```

```
status = ppc440DmaChainXfer(desc[0],
                            sysClkRateGet()*30,
                            &dmaStatus);

if (status != OK)
{
    printf("ppc440DmaChainXfer FAILED."
          " Status = 0x%x, text = %s\n",
          dmaStatus.status_code, dmaStatus.status_text);
}
else
{
    /* Wait for all descs */
    while (doXferChained_totalIntr < descs)
    {
        taskDelay(sysClkRateGet());
    }
    status = doXferChained_status;
    if (status == OK)
    {
        /* Check buffer */
        if (checkPattern( (uint32_t *)destinationBuffer,
                        bufferSize/4) != OK)
        {
            printf("checkPattern FAILED\n");
            status = ERROR;
        }
    }
}

for (cnt = 0; cnt < descs; cnt++)
{
    if (ppc440DmaChainDescRemove(desc[cnt], &dmaStatus) != OK)
    {
        status = ERROR;
    }
}

return (status);
}

int doChain(void)
```

```

{
    int width = 16;
    int bufferSize = 1024;
    int descs = 2;
    int status = OK;
    void *sMallocPtr;
    void *dMallocPtr;
    uint32_t sourceBuffer;
    uint32_t destinationBuffer;

    sMallocPtr = cacheDmaMalloc( bufferSize + width);
    if (sMallocPtr == NULL)
    {
        printf("cacheDmaMalloc FAILED
        return (ERROR);
    }
    dMallocPtr = cacheDmaMalloc(bufferSize + width);
    if (dMallocPtr == NULL)
    {
        printf("cacheDmaMalloc FAILED
        cacheDmaFree(sMallocPtr);
        return (ERROR);
    }

    sourceBuffer = (uint32_t)sMallocPtr +
        (width - ((uint32_t)sMallocPtr) % width);
    destinationBuffer = (uint32_t)dMallocPtr +
        (width - ((uint32_t)dMallocPtr) % width);

    status = doXferChained(width,
        bufferSize,
        sourceBuffer,
        destinationBuffer,
        descs);

    cacheDmaFree(sMallocPtr);
    cacheDmaFree(dMallocPtr);

    return (status);
}

```


11

MIDAS File System

11.1 The Midas File System (MFS)

Overview

The M5xxx contains a serial EEPROM that is accessible as a file system referred to as the Midas File System (MFS). The MFS can be accessed from the BSP or from the Midas Monitor. Please see the M5xxx BSP Monitor User's Guide[10] for information on how to access MFS files from the Midas Monitor. The functions used for interfacing with MFS from the BSP are documented in this section, along with the default files that are placed in MFS and their contents.

The MFS Functions

This section documents the MFS access functions available in the BSP. The include files `mfs.h` and `mfs_ini.h` should be included when using these functions.

mfs_open

Synopsis `int mfs_open(char *filename, int flags)`
 `filename` - the name of the file to open in MFS
 `flags` - 0 or `MFS_CREATE(1)`

Description This function opens a file in MFS, or creates it if it does not exist and the `MFS_CREATE` option is set.

Returns File number of opened file if OK, else `ERROR`

Example `int fd;`

 `if ((fd = mfs_open(filename, 0) == ERROR)`
 `return ERROR`

mfs_close

Synopsis STATUS mfs_close
 (
 int fd
)

fd - the file number that was previously returned from mfs_open

Description This function close a file in MFS that was previously opened with mfs_open.

Returns OK, or ERROR.

Example /* Close a previously opened file in MFS. */
 mfs_close(fd);

mfs_remove

Synopsis STATUS mfs_remove
 (
 char *filename
)

filename - the name of the MFS file to remove

Description This function removes a file in MFS.

Returns ERROR if file doesn't exist, else OK.

Example /* Remove the file in MFS named test.txt. */
 mfs_remove("test.txt");

mfs_dir

Synopsis STATUS mfs_dir
 (
)
)

Description This function lists the files in the MFS.

Returns Nothing

Example /* List the files in MFS */
 mfs_dir;

mfs_seek

Synopsis STATUS mfs_seek
 (
 int fd,
 int offset,
 int refpos
)
)

fd - the file number that was previously returned from mfs_open

offset - the position in the file to read/write next (in bytes)

refpos - The position from which to seek. The value can be either MFS_START (0), MFS_END (-1), or an offset in bytes.

Description This function seeks to a specified point in an MFS file so that future reads/writes to the file occur from the given point.

Returns If the seek cannot be performed, ERROR is returned. Otherwise OK is returned.

Example /* Seek to the end of a file in MFS. */
 mfs_seek(fd, 0, MFS_END);

mfs_stat

Synopsis STATUS mfs_stat
 (
 int fd,
 MFS_STAT *stat
)

fd - the file number that was previously returned from mfs_open
stat - pointer to an MFS_STAT structure which contains offset, and size of the file.

Description This function queries the status of a file in sprom.

Returns If the stat cannot be performed, ERROR is returned. Otherwise OK is returned.

Example /* Get the status of a previously opened file in MFS. */
 MFS_STAT mfs_stat;
 mfs_stat(fd, &mfs_stat);

mfs_tell

Synopsis int mfs_tell
 (
 int fd
)

fd - the file number that was previously returned from mfs_open

Description This function returns the current offset location for reading/writing from/to a file in SPROM.

Returns If the operation cannot be performed, ERROR is returned. Otherwise the offset is returned.

Example /* Get the current read/write location of a previously opened file in MFS. */
 int offset;
 offset = mfs_tell(fd);

mfs_eof

Synopsis

```
int mfs_eof
(
    int fd
)
```

fd - the file number that was previously returned from mfs_open

Description This function tests the given stream for an end-of-file indicator. Once the indicator is set read operations on the file return the indicator until rewind is called or the file is closed. The end-of-file indicator is reset with each input operation.

Returns This function returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached.

Example

```
/* Determine if the given file is at its end-of-file marker. */
if (mfs_eof(fd)
    printf("This file is positioned at its end-of-file mark\n");
else
    printf("This file is not positioned at its end-of-file location\n");
```

mfs_ftrunc

Synopsis

```
int mfs_ftrunc
(
    int fd,
    int length
)
```

fd - the file number that was previously returned from mfs_open

length - the length to which the file will be truncated

Description This function truncates a file to the specified length. If the file is already shorter than this length, the length of the file is not changed.

Returns This function returns OK if the file was truncated or if the file is already shorter than the specified length. The function returns ERROR if an error condition is detected.

Example

```
/* Truncate the given file to 100 bytes. */
mfs_ftrunc(fd, 100);
```

mfs_gets

Synopsis `char *mfs_fgets`

```
(
    int *buf,
    int len,
    int fd
)
```

`buf` - buffer the put the read line into

`len` - the maximum length of the line to be read

`fd` - the file number that was previously returned from `mfs_open`

Description This function reads data from file `fd` until the next 0x0a character or until the next 0x0a character is found, whichever comes first. The data is stored into `buf`. This function is similar to the "C" library function `fgets`.

Returns This function returns `buf` if OK, or `NULL` if there is an error or EOF

Example `/* Read up to an 80 byte line from the given file in mfs */`

```
char buf[80];
if (mfs_gets(buf, 80, fd) != ERROR)
    printf("The line read is: %s\n", buf);
else
    printf("Error reading a line\n");
```

mfs_read

Synopsis

```
int mfs_read
(
    int fd,
    int *buf,
    int len
)
```

fd - the file number that was previously returned from mfs_open

buf - buffer to put the read data into

len - the maximum amount of data to be read

Description This function reads up to "len" bytes of data from file fd and stores the data into buf. This function is similar to the "read" C library function.

Returns This function returns the length read if OK, otherwise ERROR.

Example

```
/* Read up to 80 bytes from the given file in mfs */
char buf[80];
int val;
val = mfs_read(fd, buf, 80);
if (val != ERROR)
    printf("Read %d bytes: %s\n", val, buf);
else
    printf("Error reading data\n");
```

mfs_write

Synopsis STATUS mfs_write

```
(
    int fd,
    int *buf,
    int len
)
```

fd - the file number that was previously returned from mfs_open

buf - buffer to put the read data into

len - the amount of data to be written

Description This function writes up to "len" bytes of data from "buf" into the file fd. This function is similar to the "write" C library function.

Returns OK or ERROR.

Example /* Write some data to the given file in mfs */
char buf[80] = "ABCDEFGH";
int val;
mfs_write(fd, buf, 7);

mfs_pwd

Synopsis STATUS mfs_pwd

```
(
)
```

Description This function prints the current working directory in MFS.

Returns Nothing.

Example /* Print the current working directory in MFS */
mfs_pwd;

mfs_ini_gettext

Synopsis STATUS mfs_ini_gettext
 (
 char *filename,
 char *section,
 char *item,
 char *buf,
 int buflen
)

Description This function reads the value of the item "item" in section "section" from file "filename" and puts it into "buf", up to the length "buflen". Note the actual section name in the file has brackets around it, i.e. "[RACEdrv]", while the "section" parameter to this function should not have brackets, i.e. "RACEdrv".

Returns OK or ERROR

Example

```
/* Find whether PXB initialization is enabled */
char init_flag[40];
if (mfs_ini_gettext("vxbsp.ini", "RACEdrv", "PxbInit",
    init_flag, sizeof(init_flag)) == OK)
    printf("Value of init_flag is: %s\n", init_flag);
else
    printf("Value of init_flag not found\n");
```

mfs_ini_settext

Synopsis STATUS mfs_ini_settext
 (
 char *filename,
 char *section,
 char *item,
 char *text
)

Description This function sets the value of the item "item" in section "section" of file "filename" to "text". Note the actual section name in the file has brackets around it, i.e. "[RACEdrv]", while the "section" parameter to this function should not have brackets, i.e. "RACEdrv". Setting the "text" parameter to a null string, i.e. "" will remove the item from the section.

Returns OK or ERROR

Example /* Set the PXB initialization flag to be enabled */
 mfs_ini_gettext("vxbsp.ini", "RACEdrv", "PxbInit", "TRUE");

mfs_ini_setlong

Synopsis STATUS mfs_ini_setlong
 (
 char *filename,
 char *section,
 char *item,
 long *value
)

Description This function sets the value of the item "item" in section "section" of file "filename" to the value "value". Note the actual section name in the file has brackets around it, i.e. "[RACEdrv]", while the "section" parameter to this function should not have brackets, i.e. "RACEdrv".

Returns OK or ERROR

Example /* Set the VmeA16SlaveBase flag to "16384" */
 int val = 16384;
 mfs_ini_setlong("vxbsp.ini", "VMEInterface", "Vme16SlaveBase", &val);

mfs_ini_setlong

Synopsis STATUS mfs_ini_setlong
 (
 char *filename,
 char *section,
 char *item,
 long *value
)

Description This function sets the value of the item "item" in section "section" of file "filename" to the value "value". The value is written as a hex value with a prefix of "0x". Note the actual section name in the file has brackets around it, i.e. "[RACEdrv]", while the "section" parameter to this function should not have brackets, i.e. "RACEdrv".

Returns OK or ERROR

Example /* Set the VmeA16SlaveBase flag to "0x10000" */
 int val = 0x10000;
 mfs_ini_setlong("vxbsp.ini", "VMEInterface", "Vme16SlaveBase", &val);

mfs_ini_getlong

Synopsis STATUS mfs_ini_getlong
 (
 char *filename,
 char *section,
 char *item,
 long *value
)

Description This function sets the value of the item "item" in section "section" of file "filename". The value is read as a 32-bit unsigned integer, and it can appear in the file in either decimal or hex format. The hex format has a prefix of "0x". Note the actual section name in the file has brackets around it, i.e. "[RACEdrv]", while the "section" parameter to this function should not have brackets, i.e. "RACEdrv".

Returns OK or ERROR

Example /* Get the value of the VmeA16SlaveBase flag */
 int val;
 mfs_ini_getlong("vxbsp.ini", "VMEInterface", "Vme16SlaveBase", &val);

mfs_usr_load_file

Synopsis STATUS mfs_usr_load_file
 (
 char *networkFilename,
 char *mfsFilename)

Description This function loads the specified file “networkFilename” from the network and saves it in MFS with the specified filename “mfsFilename”

Returns Returns OK or ERROR

Example status = mfs_usr_load_file("/home/WindRiver/c5000-bsp1.2-r1.1/pci.ini",
 "pci.ini");

11.2 The vxbsp.ini File

The RACEdrv Section

Flag name	Meaning	Values	Default
PxbInit	Initialize PXB	TRUE, FALSE	FALSE
PxbPrefMemSize	Size of prefetch memory pool for PXB	Power of 2, 1MiB - 512 MiB	0 (disabled)
PxbMemIoSize	Size of non-prefetch memory pool for PXB	Power of 2, 1 MiB - 64 MiB	0 (disabled)
PxbIoSize	Size of PCI I/O pool for PXB	Power of 2, 16 bytes -256 KiB	0 (disabled)

The VmeInterface Section

Flag name	Meaning	Values	Default
VmeA32Master2Base	VME A32 base address of secondary PCI slave base image (no such space if not present)	VME A32 base address	Not present
VmeA16SlaveBase	32-bit VME base address of A16 VME slave image used for booting other boards over VME	32-bit VME base address	Not present
VmeA32SlaveBase	VME A32 base address of secondary VME slave base image	VME A32 base address	Not present
VmeA32SlaveDisable	If present, secondary VME slave base image is disabled.	0, 1	Not present

The Universe DMA driver is part of the BSP and is documented in the Universe DMA Driver Software Reference Manual.

11.3 The mmon.ini File

Normally, the mmon.ini file does not need to be modified by the user.

The BoardInfo Section

Flag name	Meaning	Values	Default
Model	The model number of the board	(Set by factory)	Depends on board config.
SerialNo	The serial number of the board	(Set by factory)	Assigned by manufacturer
ECO level	The ECO level of the board	(Set by factory)	Assigned by manufacturer
TotalDramSize	The size of the SDRAM memory	(Set by factory)	Assigned by manufacturer

The AutoStart Section

Flag name	Meaning	Values	Default
StartAddr	The address from which to start booting an application (VxWorks).	0xf400 0000-0xf4ffffff	0xf400 0000

12 *Fibre Channel Support*

12.1 Fibre Channel Information

Overview

The M5xxx models with 'F' in the model name contain an onboard Qlogic ISP2312 dual Fibre Channel controller. Models with a single 'F' have front panel access to one Fibre Channel port, and models with two 'F's in the model name have front panel access to both Fibre Channel ports. The Fibre Channel driver is sold as part of a separate product called "VMFC driver for VxWorks" (VMFC-DRV-VXWORKS) and is documented in the VMFC Driver Software Reference Manual[12].

APPENDIXES

A

Troubleshooting

This section covers common problems encountered with the M5000.

Gigabit Ethernet network communication does not work.

For Gigabit Ethernet Switches

The default configuration of the M5000 BSP requires that all Gigabit Ethernet switches in the data path (between the M5000 and other Gigabit Ethernet peers) support Ethernet packets larger than the MTU (Maximum Transmission Unit) being used, and that this feature (jumbo packets) is enabled for all switches in the network.

Note – Many Gigabit Ethernet switches that support jumbo packets are shipped with this feature disabled by default.

If you are not using a Gigabit Ethernet switch that supports jumbo packets, a work around for this problem is to configure `EMAC2_MTU` and `EMAC3_MTU` to 1500 in `config.h`. This turns off jumbo packet usage in software. New “`bootrom/vxWorks`” or “`vxWorks.st_rom`” images must be compiled and used with these new configurations.

For Fast Ethernet Switches (100 Mbps)

Fast Ethernet switches will work with the M5000 regardless of whether jumbo packets are configured or not.

Command line compilation of the BSP fails.

In order to compile the BSP from a shell, there are two steps that must be completed:

1. Installation of Tornado 2.2.1 with patch 90451.
2. Set up the compilation environment using the `torVars` script.

Wind River has installation files for installing Tornado 2.2.1 for PowerPC directly. It is also acceptable to install Tornado 2.2 for PowerPC, followed by installing the "Tornado 2.2 Cumulative Patch 1. Patch 90451 must be installed on top of Tornado 2.2.1 (or Tornado 2.2 with Tornado 2.2 Cumulative Patch 1 applied).

Before compiling the BSP, it is also necessary to run the `torVars` script. Run `torVars.bat/ torVars.sh` or `torVars.csh` script depending on which platform and shell is used.

In Solaris, the `torVars` script is run using `'source torVars.sh'` or `'source torVars.csh'`. Running `torVars.csh` or `torVars.sh` without using the `'source'` command won't work.

Having installed Tornado 2.2.1 with patch 90451 and run the `torVars` script in the current shell, the BSP should be able to compile using commands such as `'make'`, `'make vxWorks'`, `'make vxWorks.st_rom.bin'` or `'make bootrom.bin ADDED_CFLAGS=-DBOOT_ROM'` (from the BSP directory).

The `bootrom.bin` should be compiled with the `'ADDED_CFLAGS=-DBOOT_ROM'` option in order to not include Flash and Built In Self Test functionality in the bootrom image.

If you encounter problems compiling the BSP after having installed Tornado 2.2.1 with Patch 90451 and run the `torVars` script, check that the `WIND_BASE` environment variable correctly points to the Tornado 2.2.1-90451 installation.

-

B

Deprecated Functions

pciToLocalAdrs (replaced with sysBusToLocalAdrs)

Synopsis. int pciToLocalAdrs (

 int adrsSpace,

 char *busAdrs,

 char **localAdrs,

 int busId)

adrsSpace - specifies which PCI address space. Must be one of the following defined constants: PCI_MEMORY_SPACE (1) or PCI_IO_SPACE (2).

busAdrs - PCI bus address to convert.

localAdrs - where the converted address is returned.

busId - This parameter is not used.

Description. The PPC440GX accesses the PCI buses via address translation mappings set up by the BSP. Say you have a PCI device that maps its registers to certain PCI addresses. In order for the PPC440GX to read/write those registers, the PCI address must be converted to a local address. This function converts a PCI bus address to its equivalent local address.

Returns. OK, or ERROR if the translation is not valid, in which case localAdrs will be NULL.

Example. struct my_device_regs *devRegs;

 int value;

 /* The device registers start at memory address 0xC0000000 on the Primary bus */

 if (pciToLocalAdrs (PCI_MEMORY_SPACE, (char *)0xC0000000,

 (char **)&devRegs, 0) == ERROR)

 return (ERROR);

 /* Read a 32-bit value from the device's status register */

 value = devRegs->status;

pciLocalToPciAdrs (replaced with sysLocalToBusAdrs)

Synopsis. int pciLocalToPciAdrs (

 int adrsSpace,

 char *localAdrs,

 char **busAdrs,

 int busId)

adrsSpace - specifies which PCI address space. Must be one of the following defined constants:
PCI_MEMORY_SPACE (1) or PCI_IO_SPACE (2).
localAdrs - local address to convert.
busAdrs - where the converted address is returned.
busId - This parameter is not used.

Description. Say the application code allocates a buffer in the PPC440 DRAM. In order for other PCI masters to access this buffer, the buffer's PCI address must be obtained from its corresponding local address.

Returns. OK, or ERROR.

Example. char *localAdrs, *busAdrs;

```

        /* Allocate a 1KB buffer in MPC8240 DRAM */

localAdrs = malloc (1024);

if (localAdrs == NULL)

    return (ERROR);

        /* Find PCI address of buffer */

if (pciLocalToPciAdrs (PCI_MEMORY_SPACE,

localAdrs, &busAdrs, 0) == ERROR)

    return (ERROR);

printf ("buffer local addr=0x%x, buffer PCI

        bus address=0x%x\n", localAdrs, busAdrs);

```


C

Built In Self Test

(BIST) API.

C-1 Built In Self Test API Contents

The Built In Self Test API (BIST API) is a set of functions that tests the M5000 hardware and not any external IO. The tests are grouped as follows:

- **Processor device tests.** Tests SDRAM, Flash, SPROM, PLD and temperature sensors. (`bist_proc_*`).
- **P2P tests.** Tests the three P2P bridges on the M5000 board. (`bist_p2p_*`).
- **Fibre Channel tests.** Tests the Fibre Channel interface on the M5000 board. (`bist_p2fc_*`).
- **VME tests.** Tests the VME interface on the M5000 board. (`bist_p2vme_*`).
- **Raceway tests.** Tests the RaceWay interface on the M5000 board. (`bist_p2race_*`).
- **I/O Spacer tests.** Tests Gigabit Ethernet extension modules mounted on some M5000 boards. (`bist_iospacer_*`).
- **Mezzanine tests.** Tests the Mezzanine connection that is present on some M5000 boards, the M55xx boards. (`bist_mezz_*`).
- **Base board tests.** Tests other items on the board, primarily the Ethernet interrupt. (`bist_baseboard_*`).

Each group has an "all" function (I.E. `bist_proc_all`) that runs all the tests for that groups. There are also two functions "bist_all" and "bist_all_show" that run all the tests appropriate for the board on which they are being run.

A full list and explanation of the BIST API functions can be found in the BIST API Reference Guide. A link directly to the API Reference Guide can be found on the main documentation webpage located in the documentation directory.

Technical Support

In order for us to provide fast technical support, please provide the following information:

- Any modifications made to the default BSP.
- Any changes to the default versions of the FLASH files, such as mmon.ini and pci.ini.
- Detailed description of all symptoms observed, including serial port output and PCI or VME analyzer trace files if applicable

Online Support

<http://www.vmetro.com/support>

North and South America

Telephone Support (281) 584-0728
Fax (281) 584-9034

United Kingdom

Telephone Support +44 (0) 1494 476000
Fax +44 (0) 1494 464472

Singapore

Telephone Support +65 6238 6010
Fax +65 6238 6020

Europe and the rest of the world

Telephone Support +47 23 17 28 00
Fax +47 23 17 28 01

References

The Fibre Channel Industry Association (FCIA)

<http://www.fibrechannel.org>

American National Standards Institute

<http://www.ansi.org>