Technical Reference Manual for Linux® VME Device Driver Software User Manual

Manual Order Code 555 0037 Rev 14   February 2007

Concurrent Technologies Inc
3840 Packard Road,
Suite 130
Ann Arbor, MI 48108
USA
Tel: (734) 971 6309
Fax: (734) 971 6350

Concurrent Technologies Plc
4 Gilberd Court
Newcomen Way
Colchester, Essex CO4 9WN
United Kingdom
Tel: (+44) 1206 752626
Fax: (+44) 1206 751116

E-mail: info@gocct.com   http://www.gocct.com

NOTES

CONVENTIONS

Throughout this manual the following conventions will apply:
** or $\overline{\quad}$ over a name represents an active low signal. e.g. INIT* or $\overline{INIT}$*
h denotes a hexadecimal number. e.g. FF45h
byte represents 8-bits
word represents 16-bits
dword represents 32-bits

GLOSSARY OF TERMS

API	*Application Program Interface*
DMA	*Direct Memory Access*
PCI	*Peripheral Component Interconnect*
ROAK	*Release on Acknowledge*
RORA	*Release on Register Access*
RPM	*Red Hat® Package Manager*
VME	*Versa Module Europe*

ASSOCIATED DOCUMENTS

1. Tundra® Universe II™ User Manual.

NOTATIONAL CONVENTIONS

NOTE	Notes provide general additional information.
WARNING	Warnings provide indication of board malfunction if they are not observed.
CAUTION	Cautions provide indications of board or system damage if they are not observed.

Revision	Revision History	Date
01	Initial Release	March 2001
02	Removed version number details from Chapter 2	April 2001
03	Added vme_reserveMemory function	July 2001
04	Added vme_readVerrInfo function	October 2001
05	Added RTLinux information	May 2002
06	Added vme_setInterruptMode function	January 2003
07	Added information for reserving memory using *grub*	June 2003
08	Added information for reserving high memory	March 2004
09	Added Command Line parameters to Chapter 3 and vme_readExtInterruptInfo function to Chapter 7	August 2004
10	Minor corrections to Programming Examples	March 2005
11	Notes for RTLinux added at 3.7.2, 7.30 and 7.31, and removed from 7.6 and 7.21, 7.38	May 2005
12	Added changes for PCI Space Allocation	August 2005
13	Revised description of reserving memory at load time and added note to PCI image data	January 2006
14	Added note to System Overview and revised the RTLinux notes	February 2007

Table of Contents

Table of Figures

Table of Tables

This page has been left intentionally blank

Introduction

1.1 Overview

The Concurrent Technologies Linux and RTLinux VME device drivers allow the user to access the VME bus by providing an interface to the Tundra Universe II device.

These device drivers allow a Linux user application or RTLinux task access to many features of this device including:

- Access to device registers.
- Off board (PCI) image windows.
- On board (VME) image windows.
- Direct and Linked List mode DMA transfers.
- Support for memory mapping of image windows.
- Handling of VME bus interrupts.
- Generation of VME bus interrupts.

The following sections provide the user with the information required to load and use this device driver on a Concurrent Technologies VME board. This document assumes:

- The user is familiar with the operation and programming of a Linux system.
- The user has some knowledge of VME bus operation.
- An operational Linux or RTLinux system is already installed on the target board(s).
- The Concurrent Technologies Linux or RTLinux VME device driver software has been installed in accordance with the instructions contained in the ***readme*** file supplied with the package.
- Software interdependencies are met. (See Software Prerequisites Chapter 2).

References to Linux in the remainder of this manual apply to both Linux and RTLinux unless indicated otherwise.

This page has been left intentionally blank

Software Prerequisites

The software is distributed in Red Hat® Package format and requires the Red Hat Package Manager (rpm) to be installed. Separate distribution packages are provided for Linux and RTLinux.

The Linux VME device driver is supplied as a binary object module. It requires specific Linux kernel versions, otherwise a version mismatch will occur when the module is loaded The distribution package may include drivers for several different kernel revisions. See the ***readme*** file supplied with the package for more details.

The Linux VME device driver was compiled with Red Hat Linux so this is the recommended system configuration. Please see the distribution media for supported Kernel versions.

The Linux VME utility program requires the ***ncurses*** library. The source code for the utility program is also provided and can be re-compiled if necessary. To do this the GNU C compiler and tools must be installed along with an appropriate version of the ***ncurses*** library.

This page has been left intentionally blank

System Overview

3.1 Overview

Figure 3-1 below shows how the VME device driver and API interact with the existing parts of the Linux kernel, user applications and the hardware.

The VME device driver is a kernel loadable module and as such operates in the Linux kernel space within the system.

The interface between a user application and the VME device driver is provided via an Application Programming Interface (API). This consists of a series of functions, which can be linked with a user application running in the user space. The API communicates with the VME device driver via standard low-level file access functions. Examples of how to use many of the key features of the API are shown in Chapter 10 Programming Examples.

The same API exists for both Linux and RTLinux tasks.

NOTE: The API will not allow access to the VME device driver from both Linux and RTLinux tasks at the same time. Exclusive access is given to whichever task opens the driver first and is maintained until all the VME device files are closed.

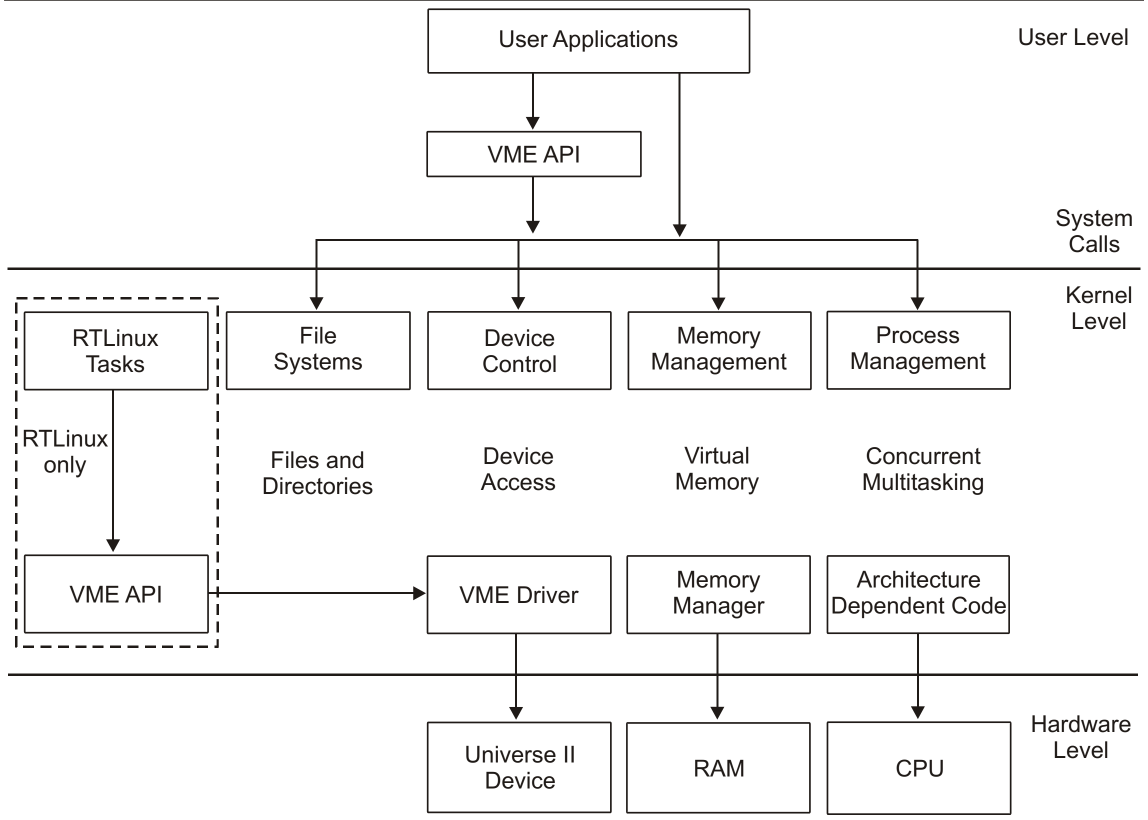

Figure 3-1 System Overview

3.2 Device Files

The VME device driver is implemented as a character device. Character devices are accessed through names (or “nodes”) in the Linux file system, usually located in the ***/dev*** directory.

When the VME device driver is loaded, a number of device file entries are created in the ***/dev/vme*** directory. These files are used to gain access to the various functions of the device driver. Typically a device file is opened, the required operation is carried out and then the device file is closed.

Each VME device file is described in the following sections.

3.2.1 Control file /dev/vme/ctl

This device file is used to control the behavior of the Universe II device. Operations using this device file can be summarized as follows:

- Read and write Universe II registers.
- Set User Address Modifiers.
- Set hardware byte swapping, for boards that support it.
- Enable and disable Universe II interrupts.
- Allows an application to wait for a specified Universe II interrupt to occur.
- Generate VME bus interrupts.
- Read VME interrupt information.
- Enable and disable Register Access and CR/CSR image windows.
- Enable and disable Location Monitors.

3.2.2 PCI Image files /dev/vme/lsi0 - 7

These device files are used to control and access the PCI image windows. Each of the eight available windows is accessed through a separate device file. Operations using these device files can be summarized as follows:

- Enable and disable a PCI image window.
- Read and write data to a PCI image window.
- Memory map a PCI image window.

3.2.3 VME Image files /dev/vme/vsi0 - 7

These device files are used to control and access the VME image windows. Each of the eight available windows is accessed through a separate device file. Operations using these device files can be summarized as follows:

- Enable and disable a VME image window.
- Read and write data to a VME image window.
- Memory map a VME image window.

3.2.4 DMA file /dev/vme/dma

This device file is used to gain access to the DMA facilities of the Universe II device. Operations using this device file can be summarized as follows:

- Allocation of a buffer for DMA transfers.
- Direct and Linked List DMA transfers.
- Creation of the command packet list for Linked List DMA transfers.
- Read and writing data to the DMA buffer.
- Memory mapping of the DMA buffer.

3.3 VME Device Driver Status Information

In Linux there is an additional mechanism for the kernel and kernel modules to send information to other processes: the ***/proc*** file system. The ***/proc*** file system is not associated with any device, the files living in ***/proc*** are generated by the kernel when they are read. These files are usually text files so they can be accessed easily with no special programming or tools required.

Status information from the VME device driver can be obtained by reading the files in the ***/proc/vme*** directory. For example typing ***more /proc/vme/ints*** at the shell prompt will display the Universe interrupt counter values maintained by the VME device driver.

3.4 Accessing Other Boards On The VME Bus

The simplest way to access other boards on the VME bus is via a PCI image window. A PCI image window allows part of the PCI address space to be mapped on to the VME bus. This is illustrated in Figure 3-2.

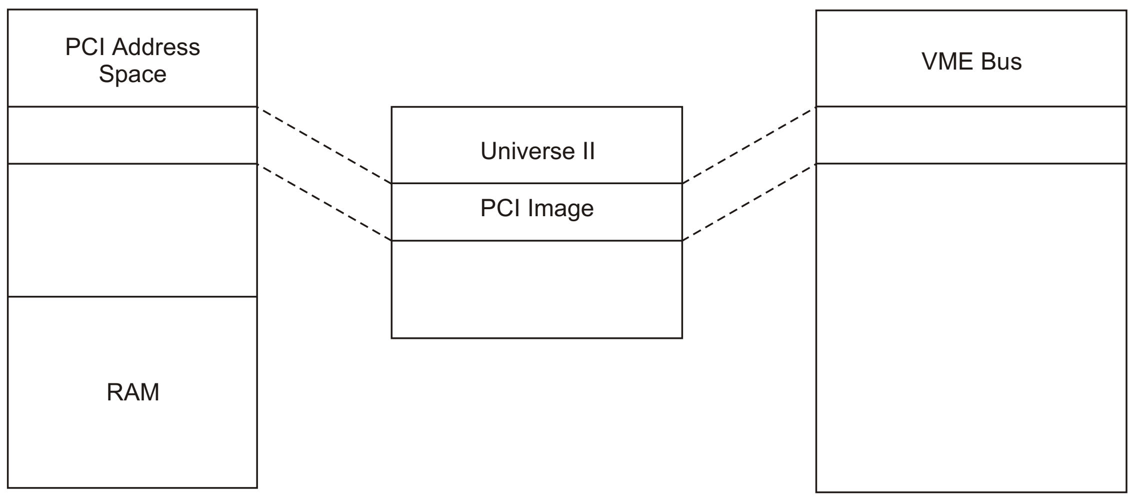

Figure 3-2 Image Window

A PCI image window can be setup and enabled by calling the VME device driver **`vme_enablePciImage`** function. Up to eight PCI images (numbered 0 to 7) can be used. PCI images 0 and 4 have a 4 Kbyte resolution. PCI images 1, 2, 3, 5, 6, and 7 have a 64 Kbyte resolution.

When a PCI image window is setup it can be mapped into Kernel memory space by setting the ioremap parameter. The PCI image window can then be accessed by using the **`vme_read`** and **`vme_write`** functions. When these functions are used, the VME device driver performs memory copying and VME bus error checking as the data is being transferred.

Alternatively, a PCI image window can be mapped into User memory space by using the **`vme_mmap`** function, allowing the image to be accessed with the standard **`memcpy`** function or by similar means. Using this approach provides an increase in performance, however it is then the responsibility of the user to manage VME bus error checking as the VME device driver has no visibility to the data being transferred. The user must also take care of unmapping the PCI image window, with the **`vme_mmap`** function, prior to closing the device file.

3.5 Universe II Interrupt Handling

The VME device driver provides an interrupt routine to handle PCI interrupts generated by the Universe II device. The Universe II can generate 24 different PCI interrupts, by default all these interrupts are disabled. However, the user can enable individual interrupts by calling the **`vme_enableInterrupt`** function and passing in the appropriate interrupt number.

A user application may wish to be informed when a certain interrupt occurs. This can be achieved by using the **`vme_waitInterrupt`** function. The **`vme_waitInterrupt`** function adds the calling process to a wait queue and then puts it to sleep. When the designated interrupt occurs or the timeout expires the process is woken up and the function returns.

When VME bus interrupts are enabled the VME device driver records vector information for each incoming VME bus interrupt. Calling the **`vme_readInterruptInfo`** function can retrieve this information from the driver.

VME bus interrupts can also be generated via software by calling the **`vme_generateInterrupt`** function. The contents of the Universe II Status ID register are used to supply an interrupt vector for generated VME bus interrupts.

ROAK (default) or RORA interrupts, for incoming VME bus interrupts are supported. The interrupt mode may be selected by calling the **`vme_setInterruptMode`** functions.

3.6 Reserving Memory

In order to use the VME image and DMA functions, a portion of physical RAM must be reserved for this purpose. There are two ways of reserving memory, either when the driver is loaded or via the **`vme_reserveMemory`** API function. Both methods are described in the following sections.

NOTE The **`VME_reserveMemory`** API function is not available to RTLinux tasks.

3.6.1 Reserving Memory At Load Time

First, an area of reseved memory must be allocated at the top of RAM. This is configured by passing the **`mem=`** argument to the Linux kernel when it is started. For example, if your board has 64 Mbytes of RAM, the argument **`mem=62M`** keeps the kernel from using the top 2 Mbytes.

If your board contains 1 Gbyte of RAM or more the **`mem=`** argument should be replaced with the **`highmem=`** argument. In this case, the amount of high memory will be reduced, to allow for the reserved memory region. The high memory area begins at 0x38000000 (896 Mbytes). For example, if your board has 1 Gbyte of RAM, the argument **`highmem=120M`** keeps the kernel from using the top 8 Mbytes of RAM for high memory.

NOTE: The **`highmem=`** argument only applies to x86 based boards, on Power-PC boards the **`mem=`** argument should be used for all memory sizes.

The most convenient place to do this is in the boot loader configuration file.

With ***lilo*** this is done in the ***lilo.conf*** file as shown below:

```
boot=/dev/sda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
linear
default=linux
image=/boot/vmlinuz-2.2.14-5.0
label=linux
initrd=/boot/initrd-2.2.14-5.0.img
read-only
root=/dev/sda2
append="mem=62M"
```

NOTE If the ***lilo.conf*** file is modified, ***lilo*** should be run from the command line to activate the changes.

With ***grub***, this is done in the ***grub.conf*** file as shown below:

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You do not have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /, eg.
#          root (hd0,0)
#          kernel /boot/vmlinuz-version ro root=/dev/hda1
#          initrd /boot/initrd-version.img
#boot=/dev/hda
default=0
timeout=10
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
title Red Hat Linux (2.4.18-14)
      root (hd0,0)
      kernel /boot/vmlinuz-2.4.18-14 ro root=LABEL=/ mem=62M
      initrd /boot/initrd-2.4.18-14.img
```

When the VME device driver is loaded, its initialization routine will determine the size of reserved memory either by probing or via the **resMemSize** command line parameter. Not all boards support the probing function as they have BIOS reserved memory areas at the top of RAM. For these boards probing is disabled and it is mandatory to specify the configured size of reserved memory with the **resMemSize** command line parameter. To check if probing is disabled, load the VME device driver and then type ***dmesg*** to display kernel log messages. If probing is disabled the following VME device driver messages will be seen in the kernel log:

"WARNING Reserved memory probe disabled"
"Please use the resMemSize command line parameter"

If you have configured the Linux kernel to leave the top 8 Mbytes of RAM free for example, you can then reload the VME device driver using the command line parameter **resMemSize=8**. See also the description of the **`resMemSize`** commamnd line parameter, in the Command Line Parameters section, for more details.

3.6.2 Reserving Memory At Run Time

As an alternative to allowing the driver to reserve memory at load time the **`vme_reserveMemory`** API function may be used. This function allows a user defined memory area to be reserved for DMA buffer and VME window use. The user memory area must be in RAM and be contiguous. If this function is used, it should be called once, as part of an applications initialization sequence, before any of the other DMA or VME window API functions are used.

NOTE Once this function is called, memory previously reserved will no longer be used by the driver, as only one reserved memory area is allowed.

NOTE This facility cannot be used by RTLinux tasks.

3.6.3 Reserved Memory Allocation

With reserved memory configured, the VME device driver can allocate space for the DMA buffer and each VME image that the user enables. Space is allocated from reserved memory with page size resolution. Figure 3-3 below shows an example of the reserved memory layout and how it might be allocated.

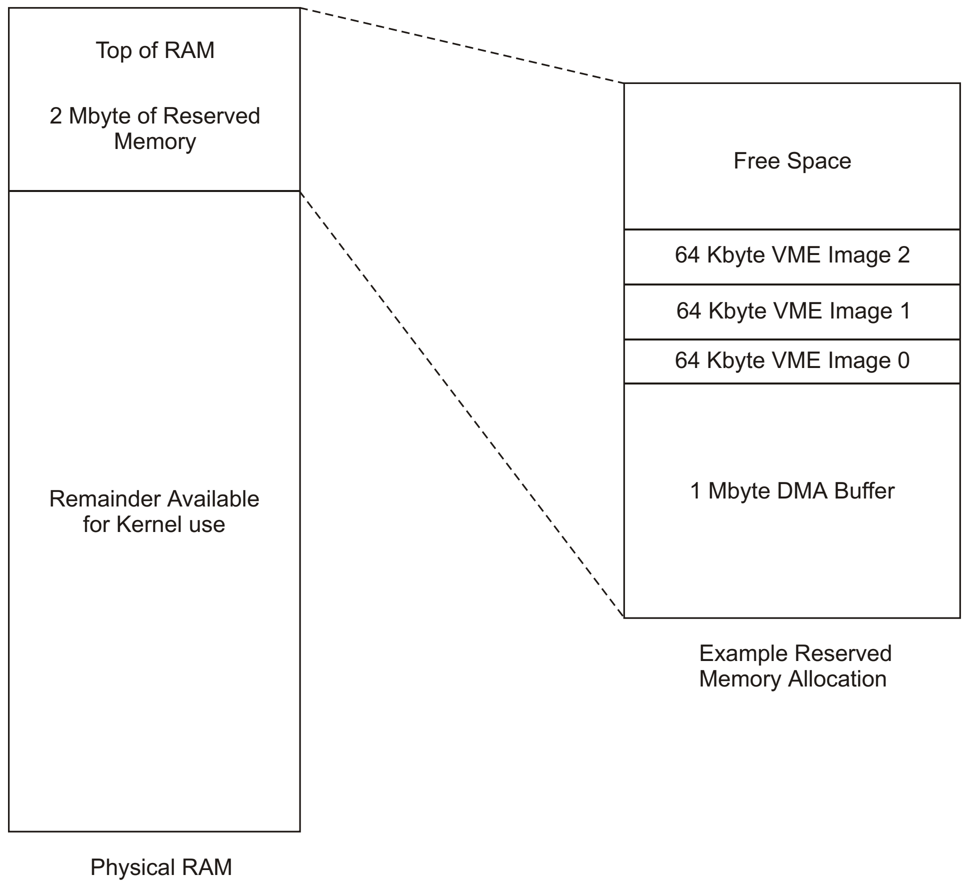

Figure 3-3 Reserved Memory Layout

3.7 DMA Transfer

The VME device driver allows the user to configure the Universe II DMA controller for high performance data transfer between the PCI and VME busses. The VME device driver allows a memory area to be reserved for DMA and provides the necessary functions for the users application program to perform the data transfers.

Before the DMA transfer functions can be used a DMA buffer must be allocated with the **`vme_allocDmaBuffer`** function. The DMA buffer resides in the reserved memory area shown in Figure 3-3 above. A user can access the DMA buffer by using the read and write functions on the DMA device file or by memory mapping the DMA buffer into user space with the mmap function.

Memory mapping the DMA buffer avoids the need for data to be copied and is thus faster. The user must however take care of unmapping the DMA buffer, with the **`vme_mmap`** function, prior to closing the device file.

It is up to the user to determine the layout of the DMA buffer, for example the DMA buffer could be divided up as shown in Figure 3-4.

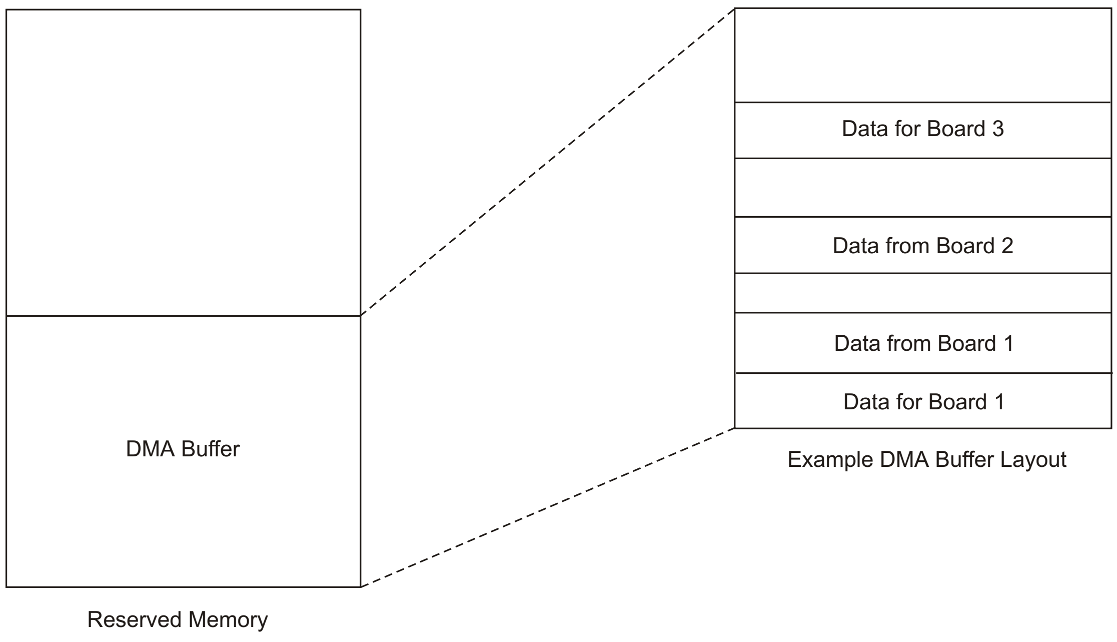

Figure 3-4 DMA Buffer Layout

In the Figuer 3-4 above, the DMA buffer has been divided into read and write data blocks. These can be transferred between the boards using either direct or linked list mode DMA.

3.7.1 Direct Mode Operation

In direct mode, a single block of data is transferred at a time. Each block of data can be transferred by calling the **`vme_dmaDirectTransfer`** function. The user passes information about the transfer via a data structure. When this function is called, the VME device driver will initiate the transfer by directly programming the Universe II DMA registers. The function will return on completion of the DMA, if an error is detected or if the specified timeout period expires.

3.7.2 Linked List Operation

Unlike direct mode, in which a single block of data is transferred at a time, linked-list mode allows a series of non-contiguous blocks of data to be transferred without software intervention. Each entry in the linked-list is described by a command packet, which resembles the Universe II DMA register layout. The structure of each command packet is the same, and contains all the necessary information to program the Universe II DMA address and control registers. When a linked-list transfer is started, the Universe II processes each command packet in turn, terminating the DMA when the last packet is processed or when an error occurs.

Before the VME device driver can initiate a linked-list DMA transfer, the command packet list must be created. The linked-list is maintained by the VME device driver, in Kernel memory not in the reserved space, and command packets are added by calling the **`vme_addDmaCmdPkt`** function. If the structure of the list does not change it only needs to be created once, that is DMA transfers can be repeated using the same command packet list.

NOTE Command packets should be added in reverse order, that is the last command packet in the list is executed first.

Figure 3-5 illustrates how a command packet linked-list might look.

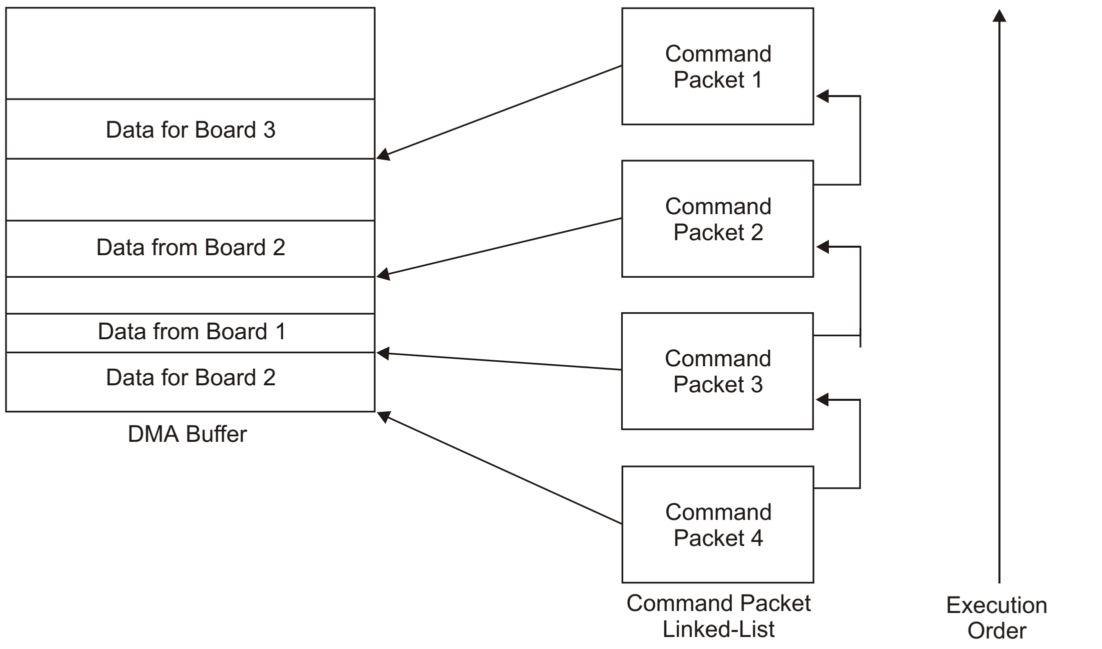

Figure 3-5 Command Packet Linked-list

Once a command packet list has been created, a linked-list DMA transfer can be initiated by calling the **`vme_dmaListTransfer`** function. The function will return on completion of the DMA, if an error is detected or if the specified timeout period expires.

NOTE With RTLinux tasks, the size of the DMA list is fixed when the VME driver is loaded. The maximum number of entries in the list is specified by the parameter "**`nDmaPkts`**". The default value for the maximum number of packets is 20. To change the value, load the driver with the following command:

```
    insmod vmedriver.o nDmaPkts=n
```

where n is the maximum number of packets, eg.

```
    insmod vmedriver.o nDmaPkts=50
```

This will install the vmedriver with a maximum of 50 DMA command packets for RTLinux. See Section 3.13 for command line options.

3.8 Sharing Memory On The VME Bus

On-board memory can be shared on the VME bus by using a VME image window. This is illustrated in the Figure 3-6.

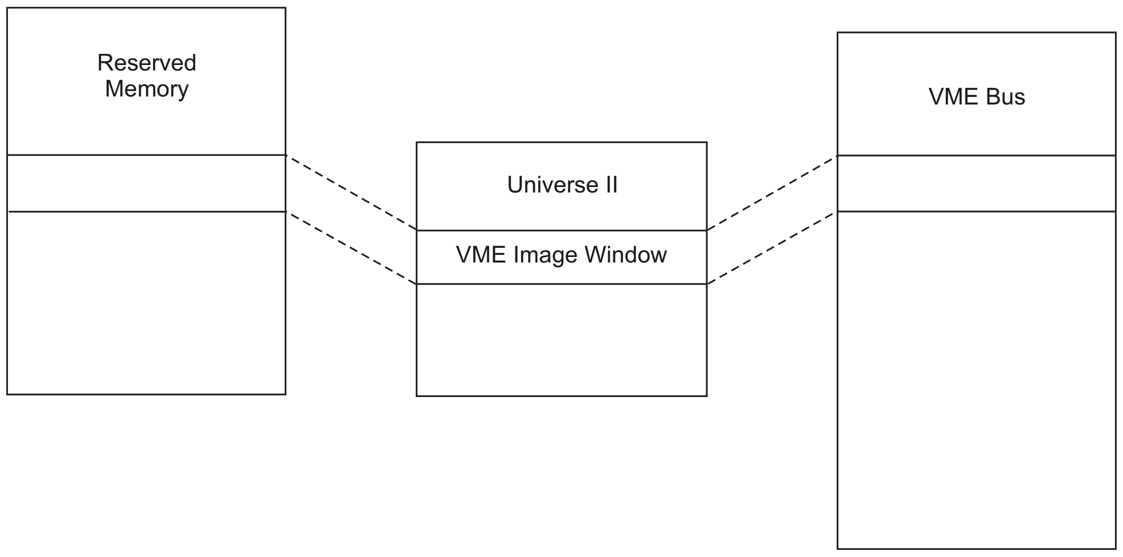

Figure 3-6 VME Image Window

Before a VME image window can be used, an area of reserved memory must be configured as described in the section above. A VME image window can be setup and enabled by calling the VME device driver **`vme_enableVmeImage`** function. Up to eight VME images (numbered 0 to 7) can be used. VME images 0 and 4 have a 4 Kbyte resolution. VME images 1, 2, 3, 5, 6, and 7 have a 64 Kbyte resolution.

When a VME image window is setup it can be mapped into Kernel memory space by setting the **`ioremap`** parameter. The VME image window can then be accessed by using the **`vme_read`** and **`vme_write`** functions. When these functions are used, the VME device driver performs memory copying of the data being transferred.

Alternatively, a VME image window can be mapped into User memory space by using the **`vme_mmap`** function, allowing the image to be accessed with the standard **`memcpy`** function or by similar means. The user must however take care of unmapping the VME image window, with the **`vme_mmap`** function, prior to closing the device file.

3.9 Universe II Register Access from the VME Bus

The Universe II Control and Status Registers (UCSR) occupy 4 Kbytes of internal memory. There are two mechanisms to access the UCSR register space from the VME bus.

One method uses a VME bus Register Access Image that allows the user to put the UCSR in an A16, A24 or A32 address space. This image can be setup and enabled by calling the VME device driver **`vme_enableRegAccessImage`** function.

The other way to access the UCSR is as CR/CSR space, as defined in the VME64 specification, where each slot in the VME bus system is assigned 512 Kbytes of CR/CSR space. This image can be setup and enabled by calling the VME device driver **`vme_enableCsrImage`** function.

3.10 Mailbox Communications

The Universe II has four 32-bit Mailbox registers, which provide an additional communication path between the VME bus and the PCI bus. Mailbox registers are useful for the communication of concise command, status, and parameter data. The Universe II can be programmed to generate an interrupt on the PCI bus when any one of its Mailbox registers is written to.

With Mailbox interrupts enabled, a user application on one board could write to a Mailbox of another board via the VME bus. The Universe II on the receiving board will interrupt via the local PCI bus and the interrupt service routine could then cause a read from this same Mailbox. This is illustrated in Figure 3-7.

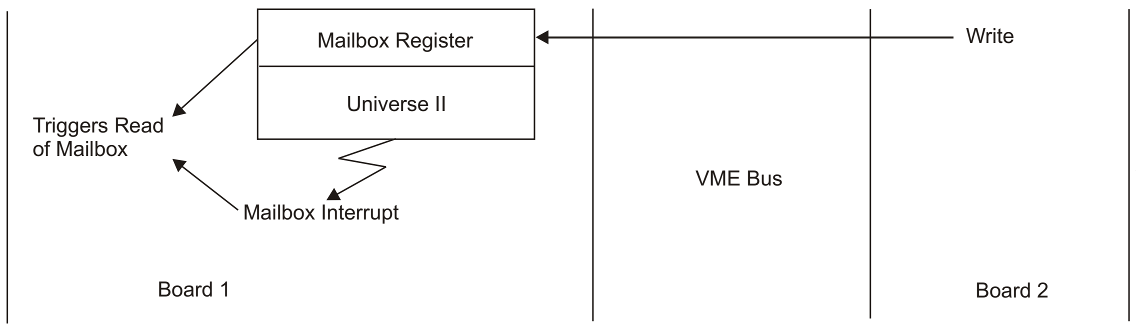

Figure 3-7 Mailbox Communications

The Mailboxes are accessible from the same address spaces and in the same manner as the other Universe II registers, as described in the Register Access Section 3-9. The Universe II Mailbox registers are located at offsets:

0x348	Mailbox register 0
0x34C	Mailbox register 1
0x350	Mailbox register 2
0x354	Mailbox register 3

A user application can be informed when a Mailbox interrupt occurs by using the **`vme_waitInterrupt`** function.

3.11 Location Monitoring

The Universe II has four Location Monitors to provide VME bus broadcast capability. The Location Monitor image can be programmed to monitor 4 Kbytes of the VME bus address space. When enabled, any accesses within this image window will cause the Universe II to generate Location Monitor interrupt(s). VME bus address bits 3 and 4 determine which Location Monitor will be used, and hence which of four Location Monitor interrupts to generate.

The Location Monitors can be setup and enabled by calling the VME device driver **`vme_enableLocationMon`** function. When called, this function sets up the Location Monitor image window with the given parameters and enables the Location Monitor interrupts. A user application can be informed when a Location Monitor interrupt occurs by using the **`vme_waitInterrupt`** function.

3.12 User Address Modifiers

In addition to the standard address modifiers, A16, A24, and A32, its also possible to program the Universe II device with two user defined address modifiers. This can be achieved by first calling the **`vme_setUserAmCodes`** function to set the Universe II **`USER_AM`** register with the desired address modifier values. Then either User1 or User2 address modifier can be selected for VME bus access, when for example a PCI image window is enabled.

3.13 Command Line Parameters

A number of command line parameters are available which can be used to change the default behavior of the VME device driver. The command line parameters are passed to the driver when it's loaded, for example: ***/sbin/insmod vmedriver.o resMemSize=8*** sets the reserved memory size used by the driver to 8 Mbytes. Each of the command line parameters are described in the following sections.

3.13.1 Specifying Reserved Memory Size

The **`resMemSize`** parameter can be used to manually set the amount of reserved memory used by the driver.

`resMemSize` values:	= 0 probe for user reserved memory (default) > 0 number of Mbytes of user reserved memory < 0 disables reserved memory detection

3.13.2 Overriding Board Type Auto Detection

The **`boardName`** parameter can be used to override board type auto detection. A valid board name string should be used, for example **`boardName="VP305/01x"`**.

3.13.3 Setting the Number of VME Vectors Captured

The **`vecBufSize`** parameter can be used to change the number of VME vectors captured in the buffer before it wraps around.

`vecBufSize` values:	range from 32 to 128 (default is 32) When the value is > 32 extended vector capture mode is used.

3.13.4 Overriding the DMA List Size (RTLinux operation only)

In list mode, th emaximum number of DMA command packets is 20. To override this specify the new value to **`nDmaPkts`**, for example, **`nDmaPkts = 50`**.

3.13.5 Overriding PCI Space Usage

The driver can assign PCI addresses for PCI images from a PCI address pool claimed during the driver installation. There are two parameters which override the default PCI address used:

`pciAddr`	Unused PCI address the driver will start allocating from
`pciSize`	The maximum amount of PCI space the driver will allocate

The default values for these are:

`pciAddr`	0xB0000000
`pciSize`	0x10000000

These values should not normally need changing.

Software Installation

Installation instructions for the Linux VME device driver and Utility program are provided in a ***readme*** file on the distribution media.

This page has been left intentionally blank

Loading the Device Driver Module

You must be the root user to load and unload kernel modules and the following assumes you installed the VME device driver in ***/usr/local***.

An install script file is provided to load the VME device driver. This is located in the same directory as the VME device driver object file (***linuxvme*** for Linux, ***rtlinux*** for RTLinux). When executed this script file will load the VME device driver module and create the required device file entries in ***/dev/vme***.

Before running the script file:

Check the execute attribute is set on the script file.

For Linux, check that the current directory is ***/usr/local/linuxvme***

For RTLinux, check that the current directory is ***/usr/local/rtlinuxvme***

Then type ***./ins*** to run the script and load the VME device driver.

Next, type ***dmesg*** to display kernel log messages. The VME device driver initialization messages should be seen towards the end of the log. If the module load occurred without error the VME device driver should report its initialization was successful and is now ready to be used.

The VME device driver also provides status information via the proc file system. This information can be obtained by viewing the files in the ***/proc/vme*** directory. For example type ***more /proc/vme/ctl*** to display general information about the VME device driver.

If you are trying to load the VME device driver using a kernel version other than that specified in the Software Prerequisites section above it is likely the ***insmod*** program will report a version mismatch. You may still be able to load the module by using the ***insmod -f*** option. An example of this is provided in the ***ins*** script file. Edit the file and run the script again.

If you need to unload the VME device driver, a script file is also provided for this purpose. Type ***./uns*** to unload the VME device driver.

NOTE The VME device driver can only be unloaded when it is not in use.

This page has been left intentionally blank

Linux VME Utility Program

The Linux utility program exercises many of the functions provided by the VME device driver. The utility program is supplied in source code form and as an executable program. The utility program uses ***ncurses*** library functions to display information on the screen and to facilitate user entry. The supplied executable program requires ***ncurses*** version 5.0-11. If this is not the version being used, the Linux utility program will need to be re-compiled before use.

A Makefile is provided along with all the other required files in the ***/usr/local/linuxvme_util*** directory for Linux or the ***/usr/local/rtlinuxvme_util*** for RTLinux.

To build the utility program for Linux make sure the current directory is ***/usr/local/linuxvme_util*** and then type:

make all

To build the utility program for RTLinux make sure the current directory is ***/usr/local/rtlinuxvme_util*** and then type

make all

Once the build process has completed type ***./linuxvme*** to run the utility program.

The Linux VME utility program was primarily designed for use on a system with a video display and requires a screen size of at least 80x24 characters. However, it may be possible to run it from a terminal or ***telnet*** session. The major problem in using the utility program in this manner is that screen updates will be rather slow.

Using the Linux VME utility program should be self-explanatory. Use the arrow keys to highlight an option and the return key to select. A sub-menu or option may be exited by pressing the ‘q’ key or by selecting the Quit option.

Most of the time you will need to open the appropriate device file before a device operation will be allowed. For example, the ***ctl*** (control) device must be opened before using the Read Register option. To open a device file, select the Open Device option and then select the file from the list.

Some device operations require parameters to be entered. The utility program sets default values which can of course be changed. To do this, highlight the parameter then press the return key. Enter the new value when prompted or use the arrow keys to cycle through the available values, then press return again. When you’re happy the parameters are correct press the key to execute the device operation.

A more detailed understanding of the utility program can always be gained from studying the supplied source code.

This page has been left intentionally blank

Application Programming Interface

7.1 Overview

The VME device driver Application Programming Interface (API) consists of a series of functions, which can be linked with a user application. It is provided in a static library file, ***libcctvme.a,*** which is located in the ***/usr/local/linuxvme_util*** directory for Linux (or ***/usr/local/rtlinuxvme_util*** for RTLinux) along with the ***vme_api.h*** header file.

A detailed description of each API function is given in this chapter.

NOTE RTLinux programs must not be linked with ***libcctvme.a***, they should only include the ***vme_api.h*** header file

7.2 vme_openDevice

Declaration:

`int vme_openDevice ( INT8 *deviceName );`

Parameters:

`*deviceName`	device file name string.

Description:

Before a device operation can be performed, the appropriate device file must be opened. Calling **`vme_openDevice`** opens the given VME device file and returns the device handle which can then be passed, as a parameter, to the other API routines.

Returns:

The device handle if successful or an error code upon failure.

7.3 vme_closeDevice

Declaration:

```
int vme_closeDevice ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.

Description:

Closes the given VME device file and releases the device handle obtained by calling **`vme_openDevice`**. Resources obtained by the VME device diver, while the device file was open, are freed and device services such as image windows are disabled where applicable.

NOTE If the user application aborts with a device file open, the device close function is called automatically by the Linux Kernel.

Returns:

Zero if successful or an error code upon failure.

7.4 vme_readRegister

Declaration:

```
int vme_readRegister( INT32 deviceHandle, UINT16 offset, UINT32 *reg );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`offset`	offset of register to read.
`*reg`	location to store register value.

Description:

Reads a Universe device register at the given offset. Valid Universe register offsets are defined in the ***vme_api.h*** header file.

Returns:

Zero if successful or an error code upon failure.

7.5 vme_writeRegister

Declaration:

```
int vme_writeRegister( INT32 deviceHandle, UINT16 offset, UINT32 reg );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`offset`	offset of register to write.
`reg`	register value.

Description:

Writes to a Universe device register at the given offset. Valid Universe register offsets are defined in the ***vme_api.h*** header file.

CAUTION Care should be taken when using **`vme_writeRegister`** as it may effect the operation of other functions.

Returns:

Zero if successful or an error code upon failure.

7.6 vme_setInterruptMode

Declaration:

```
int vme_setInterruptMode( INT32 deviceHandle, UINT8 mode);
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`mode`	selected interrupt mode (**`INT_MODE_ROA`**K or **`INT_MODE_RORA)`** defined in ***vme_api.h***.

Description:

Sets the interrupt mode to ROAK (default) or RORA for incoming VIRQ's.

Returns:

Zero if successful or an error code upon failure.

7.7 vme_enableInterrupt

Declaration:

```
int vme_enableInterrupt ( INT32 deviceHandle, UINT8 intNumber );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`intNumber`	Universe interrupt number.

Description:

Enables the given Universe device interrupt. Valid Universe interrupt numbers are enumerated in the ***vme_api.h*** header file.

NOTE The effects of enabling the VME bus error interrupt (VERR), on boards without the proper hardware support, is undefined.

Returns:

Zero if successful or an error code upon failure.

7.8 vme_disableInterrupt

Declaration:

```
int vme_disableInterrupt ( INT32 deviceHandle, UINT8 intNumber );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`intNumber`	Universe interrupt number.

Description:

Disables the given Universe device interrupt. Valid Universe interrupt numbers are enumerated in the ***vme_api.h*** header file.

Returns:

Zero if successful or an error code upon failure.

7.9 vme_generateInterrupt

Declaration:

```
int vme_generateInterrupt ( INT32 deviceHandle, UINT8 intNumber );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`intNumber`	VME bus interrupt number, VIRQ 1 - 7.

Description:

Generates the given VME bus interrupt.

Returns:

Zero if successful or an error code upon failure.

7.10 vme_readInterruptInfo

Declaration:

```
int vme_readInterruptInfo ( INT32 deviceHandle, VME_INT_INFO *iPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*iPtr`	pointer to a data structure where interrupt information will be stored. The parameters are described in the Data Structure section below.

Description:

Reads VME interrupt information from the driver. Up to 32 vectors, for the specified interrupt, are returned along with the number of interrupts since the last read.

Returns:

Zero if successful or an error code upon failure.

NOTE: The function will return an error if the size of the VME vector buffer has been increased above 32 using the **`vecBufSize`** command line parameter. To retrieve the extended range of vectors the **`vme_readExtInterruptInfo`** function should be used instead.

7.11 vme_setStatusId

Declaration:

```
int vme_setStatusId ( INT32 deviceHandle, UINT8 statusId );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`statusId`	status ID value.

Description:

Sets the Universe STATUSID register with the given value. The contents of this register will be used to supply an interrupt vector for subsequent VME bus interrupts generated by this board, for example when the **`vme_generateInterrupt`** call is used.

Returns:

Zero if successful or an error code upon failure.

7.12 vme_waitInterrupt

Declaration:

```
int vme_waitInterrupt ( INT32 deviceHandle, UINT32 selectedInts,
                        UINT32 timeout, UINT32 *intNum );
```

Parameters:

`deviceHandle` device handle obtained from a previous call to **`vme_openDevice`**.

`selectedInts` select interrupts to wait for, by setting the appropriate bit, where:

bit 0 = VOWN	bit 1 = VIRQ1	bit 2 = VIRQ2
bit 3 = VIRQ3	bit 4 = VIRQ4	bit 5 = VIRQ5
bit 6 = VIRQ6	bit 7 = VIRQ7	bit 8 = DMA
bit 9 = LERR	bit 10 = VERR	bit 11 = reserved
bit 12 = SW_IACK	bit 13 = SW_INT	bit 14 = SYSFAIL
bit 15 = AC_FAIL	bit 16 = MBOX0	bit 17 = MBOX1
bit 18 = MBOX2	bit 19 = MBOX3	bit 20 = LM0
bit 21 = LM1	bit 22 = LM2	bit 23 = LM3

`timeout` timeout value in system ticks (jiffies) or zero to wait forever.

`*intNum` returns the interrupt received or the conflicting interrupt in the same format as **`selectedInts`** parameter above. In addition, bit 31 is used to indicate whether the returned value is valid or not, where:

valid: bit 31 = 1
invalid: bit 31 = 0

Description:

Waits for one of the specified Universe interrupts to occur.

The interrupt received is returned in the **`intNum`** parameter. If a wait call is already pending, on any one of the selected interrupts, the function will return an error code and the conflicting interrupt bit will be set in the **`intNum`** parameter.

NOTE **`vme_waitInterrupt`** makes sure the selected interrupts are enabled so there is no need to call **`vme_enableInterrupt`** first.

NOTE The effects of enabling the VME bus error interrupt (VERR), on boards without the proper hardware support, is undefined.

Returns:

Zero if successful or an error code upon failure.

7.13 vme_setUserAmCodes

Declaration:

```
int vme_setUserAmCodes ( INT32 deviceHandle, VME_USER_AM *amPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*amPtr`	pointer to a data structure containing the parameters necessary to set the user address modifiers. The parameters are described in the Data Structure section below.

Description:

Sets the Universe user address modifier register with the given value. The contents of this register will be used when a User address modifier is selected for VME bus access.

Returns:

Zero if successful or an error code upon failure.

7.14 vme_setByteSwap

Declaration:

```
int vme_setByteSwap ( INT32 deviceHandle, UINT8 enable );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`Enable`	set or clear bits in this parameter to enable/disable byte swapping. Setting bit 3 enables master VME byte swap, clear to disable. Setting bit 4 enables slave VME byte swap, clear to disable. Setting bit 5 enables fast write, clear to disable.

Description:

Enables or disables hardware byte swapping on the VME bus, for those boards that support it.

NOTE The operation of this function on boards that do not support hardware byte swapping is undefined.

Returns:

Zero if successful or an error code upon failure.

7.15 vme_enableRegAccessImage

Declaration:

```
int vme_enableRegAccessImage ( INT32 deviceHandle,
                               VME_IMAGE_ACCESS *iPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*iPtr`	pointer to a data structure containing the parameters necessary to enable the Register Access image. The parameters are described in the Data Structure section below.

Description:

Enables the Universe Register Access, with the given parameters. This image maps the Universe registers onto the VME bus enabling other boards in the system to access them.

Returns:

Zero if successful or an error code upon failure.

7.16 vme_disableRegAccessImage

Declaration:

```
int vme_disableRegAccessImage ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.

Description:

Disables the Universe Register Access image.

Returns:

Zero if successful or an error code upon failure.

7.17 vme_enableCsrImage

Declaration:

```
int vme_enableCsrImage ( INT32 deviceHandle, UINT8 imageNum );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`imageNum`	image number specifies one of thirty-one available CR/CSR image windows as defined in theVME64 specification.

Description:

Enables the given CR/CSR image, mapping the Universe II Control and Status Registers (UCSR) onto the VME bus enabling other boards access. The VME64 specification assigns a total of 16 Mbytes of CR/CSR space for the entire VMEbus system. This 16 Mbytes is broken up into 512 Kbytes per slot for a total of 32 slots. The first 512 Kbyte block is reserved for use by the Auto-ID mechanism. The CR/CSR space occupies the upper 4 Kbytes of the 512 Kbytes available for its slot position.

Returns:

Zero if successful or an error code upon failure.

7.18 vme_disableCsrImage

Declaration:

`int vme_disableCsrImage ( INT32 deviceHandle, UINT8 imageNum );`

Parameters:

`DeviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`imageNum`	image number specifies one of thirty-one available CR/CSR image windows as defined in theVME64 specification.

Description:

Disables the given CR/CSR image.

Returns:

Zero if successful or an error code upon failure.

7.19 vme_enableLocationMon

Declaration:

```
int vme_enableLocationMon ( INT32 deviceHandle,
                            VME_IMAGE_ACCESS *iPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`*iPtr`	pointer to a data structure containing the parameters necessary to enable the Location monitors. The parameters are described in the Data Structure section below.

Description:

Enables the Universe Location monitors, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

7.20 vme_disableLocationMon

Declaration:

```
int vme_disableLocationMon ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.

Description:

Disables the Universe Location monitors.

Returns:

Zero if successful or an error code upon failure.

7.21 vme_clearStats

Declaration:

```
int vme_clearStats ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.

Description:

Resets the statistical information maintained by the VME device driver.

Returns:

Zero if successful or an error code upon failure.

7.22 vme_enablePciImage

Declaration:

`int vme_enablePciImage ( INT32 deviceHandle, PCI_IMAGE_DATA *iPtr );`

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*iPtr`	pointer to a data structure containing the parameters necessary to open the image window. The parameters are described in the Data Structure section below.

Description:

Enables the Universe PCI image window specified by the device handle, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

NOTE The **`pciAddress`** element of the **`iPtr`** can be set to either a valid free PCI address or 0. If 0 is used, the physical PCI address for the image will be allocated by the driver.

When assigning a PCI address manually, care should be taken not to overlap one PCI address range with any other or with the PCI address pool used by the driver. Refer to the section titled Overriding PCI Space Usage for the default values used for the PCI address pool.

When calling the **`vme_enablePciImage`** function from RTLinux, the **`pciAddress`** element must be set to 0 or the call will return an error.

7.23 vme_disablePciImage

Declaration:

```
int vme_disablePciImage ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.

Description:

Disables the Universe PCI image window specified by the device handle.

Returns:

Zero if successful or an error code upon failure.

7.24 vme_enableVmeImage

Declaration:

```
int vme_enableVmeImage ( INT32 deviceHandle, VME_IMAGE_DATA *iPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`*iPtr`	pointer to a data structure containing the parameters necessary to open the image window. The parameters are described in the Data Structure section below.

Description:

Enables the Universe VME image window specified by the device handle, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

7.25 vme_disableVmeImage

Declaration:

```
int vme_disableVmeImage ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.

Description:

Disables the Universe VME image window specified by the device handle.

Returns:

Zero if successful or an error code upon failure.

7.26 vme_read

Declaration:

```
int vme_read ( INT32 deviceHandle, UINT32 offset, UINT8 *buffer,
               UINT32 length );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`offset`	relative offset from device start address.
`*buffer`	buffer to hold data.
`length`	amount of data to read in bytes.

Description:

Reads data from the device, typically a PCI image window that has been ***ioremapped*** into the kernel memory space. As data is read from the VME bus, a check is made for bus errors and if detected an error is returned.

When used on the DMA device, data is read from the DMA buffer using a memory copy to user space. The same thing occurs with VME image windows, so data is read from the memory allocated for the window using a memory copy to user space.

Returns:

Number of bytes read if successful or an error code upon failure.

7.27 vme_write

Declaration:

```
int vme_write ( INT32 deviceHandle, UINT32 offset, UINT8 *buffer,
                UINT32 length );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`offset`	relative offset from device start address.
`*buffer`	buffer to containing data.
`length`	amount of data to write in bytes.

Description:

Writes data to the device, typically a PCI image window that has been ***ioremapped*** into the kernel memory space. As data is written to the VME bus, a check is made for bus errors and if detected an error is returned.

When used on the DMA device, data is written to the DMA buffer using a memory copy. The same thing occurs with VME image windows, so data is written to the memory allocated for the window using a memory copy.

Returns:

Number of bytes written if successful or an error code upon failure.

7.28 vme_mmap

Declaration:

```
int vme_mmap ( INT32 deviceHandle, UINT32 offset, UINT32 length,
               UINT32 *userAddress );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`offset`	relative offset from device start address, must be PAGE aligned.
`length`	size of area to map in bytes, must be a multiple of PAGE_SIZE which is typically 4KB.
`*userAddress`	returned address of memory mapped area.

Description:

Memory maps the device into user space.

Returns:

Zero if successful or an error code upon failure.

7.29 vme_unmap

Declaration:

`int vme_unmap( INT32 deviceHandle, UINT32 userAddress, UINT32 length );`

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`userAddress`	address of previously mapped area.
`length`	size of mapped area in bytes.

Description:

Removes the memory mapping for the device at the given address. The length parameter must match the length given in the corresponding `vme_mmap` call.

NOTE It is the users responsibility to remove the memory mapping before closing a device file.

Returns:

Zero if successful or an error code upon failure.

7.30 vme_allocateDmaBuffer

Declaration:

```
int vme_allocateDmaBuffer ( INT32 deviceHandle, UINT32 *size );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*size`	pointer to size of buffer in bytes. The actual buffer size returned, is always a multiple of PAGE_SIZE, regardless of the size requested.

Description:

Allocates a buffer for use with the Universe DMA functions. The DMA buffer is mapped in to Kernel memory space but may be re-mapped to User space using the **`vme_mmap`** function on the DMA device file.

Returns:

Zero if successful or an error code upon failure.

7.31 vme_freeDmaBuffer

Declaration:

```
int vme_freeDmaBuffer ( INT32 deviceHandle );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.

Description:

Frees a previously allocated DMA buffer.

Returns:

Zero if successful or an error code upon failure.

7.32 vme_dmaDirectTransfer

Declaration:

```
int vme_dmaDirectTransfer ( INT32 deviceHandle,
                            VME_DIRECT_TXFER *dPtr );
```

Parameters:

deviceHandle	device handle obtained from a previous call to **vme_openDevice**.
***dPtr**	pointer to a data structure containing the parameters necessary to do a direct DMA transfer. The parameters are described in the Data Structure section below.

Description:

Initiates a Universe direct mode DMA transfer, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

7.33 vme_addDmaCmdPkt

Declaration:

```
int vme_addDmaCmdPkt ( INT32 deviceHandle, VME_CMD_DATA *cmdPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*cmdPtr`	pointer to a data structure containing the parameters necessary to add a command packet to the linked list. The parameters are described in the Data Structure section below.

Description:

Adds a command packet to the DMA linked list, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

7.34 vme_clearDmaCmdPkts

Declaration:

`int vme_clearDmaCmdPkts ( INT32 deviceHandle );`

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.

Description:

Clears the DMA command packet linked list.

Returns:

Zero if successful or an error code upon failure.

7.35 vme_dmaListTransfer

Declaration:

```
int vme_dmaDirectTransfer( INT32 deviceHandle,
                           VME_TXFER_PARAMS *tPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`*tPtr`	pointer to a data structure containing the parameters necessary to do a linked list DMA transfer. The parameters are described in the Data Structure section below.

Description:

Initiates a Universe linked list mode DMA transfer, with the given parameters. The linked list of command packets must already have been created with calls to the `vme_addDmaCmdPkt` function.

Returns:

Zero if successful or an error code upon failure.

7.36 vme_getApiVersion

Declaration:

```
int vme_getApiVersion ( char *buffer );
```

Parameters:

`*buffer`	pointer to a buffer to hold the version string.

Description:

Gets the API version information as a string.

Returns:

Size of version information string.

7.37 vme_reserveMemory

Declaration:

```
int vme_reserveMemory( INT32 deviceHandle, UINT32 physicalAddress,
                       UINT32 size );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`physicalAddress`	physical address of memory area, should be page aligned.
`size`	size of memory area, should be a multiple of page size. If **`size=0`** the driver will just release reserved memory resources.

Description:

Allows a user defined memory area to be reserved for DMA buffer and VME window use. The user memory area must be in RAM and be contiguous. It can be used to select an alternative reserved memory area to that configured by the driver when it is loaded. If this function is used, it should be called once, as part of an applications initialization sequence, before any of the other DMA or VME window API functions are used.

NOTE Once this function is called, memory previously reserved will no longer be used by the driver, as only one reserved memory area is allowed.

WARNING Great care must be exercised when using this function, as passing incorrect parameters could result in corruption of memory and possible system failure.

Returns:

Zero if successful or an error code upon failure.

NOTE This function is not supported in the RTLinux distribution.

7.38 vme_readVerrInfo

Declaration:

```
int vme_readVerrInfo ( INT32 deviceHandle, UINT32 *Address,
                       UINT8 *Direction, UINT8 *AmCode )
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*Address`	pointer to location to store the address that caused the VME bus error.
`*Direction`	pointer to location to store the transfer direction, where 0=write, 1=read.
`*AmCode`	pointer to location to store the address modifier code. See below for typical values.

Description:

This function returns the VME bus error information collected by the Linux VME driver when a VME bus error occurs. The driver reads VME bus error information from the hardware when a VERR interrupt occurs, therefore, the VERR interrupt must first be enabled by calling the **`vme_enableInterrupt`** function.

VME bus error information is captured and latched by the hardware. When the **`vme_readVerrInfo`** function is called, it retrieves the captured information from the Linux VME driver and then unlatches the hardware ready to capture the next VME bus error.

NOTE If posted writes are enabled, VME bus error detecton may be delayed.

NOTE This function is only supported on boards which have the necessary hardware to capture this information.

Returns:

Zero if successful or an error code upon failure.

Typical address modifier codes:

0x2D	A16 supervisory access
0x29	A16 non-privileged access
0x3F	A24 supervisory block transfer (BLT)
0x3E	A24 supervisory program access
0x3D	A24 supervisory data access
0x3C	A24 supervisory 64 bit block transfer (MBLT)
0x3B	A24 non-privileged block transfer (BLT)
0x3A	A24 non-privileged program access
0x39	A24 non-privileged data access
0x38	A24 non-privileged 64 bit block transfer (MBLT)
0x0F	A32 supervisory block transfer (BLT)
0x0E	A32 supervisory program access
0x0D	A32 supervisory data access
0x0C	A32 supervisory 64 bit block transfer (MBLT)
0x0B	A32 non-privileged block transfer (BLT)
0x0A	A32 non-privileged program access
0x09	A32 non-privileged data access
0x08	A32 non-privileged 64 bit block transfer (MBLT)

Details of other address modifier codes can be found in the ANSI VME64 specifications.

7.39 vme_readExtInterruptInfo

Declaration:

```
int vme_readExtInterruptInfo ( INT32 deviceHandle,
                               VME_EXTINT_INFO *iPtr );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**.
`*iPtr`	pointer to a data structure where interrupt information will be stored. The parameters are described in the Data Structure section below.

Description:

Reads VME interrupt information from the driver. Up to 128 vectors, for the specified interrupt, are returned along with the number of interrupts since the last read.

Returns:

Zero if successful or an error code upon failure.

NOTE: The function will return an error if the size of the VME vector buffer has not been increased above the default 32. To increase the VME vector buffer size, use the **`vecBufSize`** command line parameter.

This page has been left intentionally blank

Data Structures

The following sections describe the data structures used by the VME device driver API functions.

8.1 PCI Image Data

Before calling the **`vme_enablePciImage`** function, a PCI image data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	**Values**	**Description**
pciAddress	e.g. 0xC0000000	Base PCI address. The address must be in the valid PCI address range and aligned in accordance with image resolution. If the pciAddress is set to 0, the driver will allocate the physical PCI address used for the image.
vmeAddress	e.g. 0x10000	Base VME address. The address must be aligned in accordance with image resolution.
size	e.g. 4096 or 65536	Image size. The size should be set in accordance with image resolution.
postedWrites	0 = disabled 1 = enable	Whether to enable posted writes.
dataWidth	0 = 8 bit, 1 = 16 bit, 2 = 32 bit, 3 = 64 bit	Maximum data width.
addrSpace	0 = A16, 1 = A24, 2 = A32 5 = CR/CSR 6 = User1, 7 = User2	Address space modifier.
type	0 = data, 1 = program	Program/Data AM code.
mode	0 = non-privileged 1 = supervisor	Supervisor/User AM code.
vmeCycle	0 = no BLT's, 1=single BLT's	VME bus cycle type.
pciBusSpace	0 = PCI memory space 1 = PCI I/O space	PCI bus space memory space.
ioremap	0 = no 1 = yes	Whether to ioremap image. Note: PCI images can ioremapped or memory mapped to user space with **`vme_mmap`**.

Table 8-1 PCI Image Data

NOTE: If posted writes are enabled, VME bus error detecton may be delayed.

8.1.1 VME Image Data

Before calling the **`vme_enableVmeImage`** function, a VME image data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
vmeAddress	e.g. 0x10000	Base VME address. The address must be aligned in accordance with image resolution.
size	e.g. 4096 or 65536	Image size. The size should be set in accordance with image resolution.
postedWrites	0 = disabled 1 = enable	Whether to enable posted writes.
prefetchRead	0 = disabled 1 = enable	Whether to enable prefetch read.
type	1 = data, 2 = program, 3 = both	Program/Data AM code.
mode	1 = non-privileged 2 = supervisor 3 = both	Supervisor/User AM code.
addrSpace	0 = A16, 1 = A24, 2 = A32 6 = User1, 7 = User2	Address space modifier.
pciBusSpace	0 = PCI memory space 1 = PCI I/O space	PCI bus space memory space.
pciBusLock	0 = disabled 1 = enable	PCI bus lock of VME read modify write.
ioremap	0 = no 1 = yes	Whether to ioremap image. NOTE VME images can be ioremapped or memory mapped to user space with **`vme_mmap`**.

Table 8-2 VME Image Data

8.1.2 VME Interrupt Information

Before calling the **`vme_readInterruptInfo`** function, an interrupt information data structure must be created. The user need only assign a value to the intNum parameter.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
intNum	1 - 7	VME interrupt number to read information.
numOfInts		Number of VME interrupts since last call.
vecCount	0 - 32	Number of vectors stored in vectors array.
vectors[]		Array to contain the STATUSID vectors.

Table 8-3 VME Interrupt Data

Before calling the **`vme_readExtInterruptInfo`** function, an extended interrupt information data structure must be created. The user need only assign a value to the **`intNum`** parameter.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
intNum	1 - 7	VME interrupt number to read information.
numOfInts		Number of VME interrupts since last call.
vecCount	0 - 128	Number of vectors stored in vectors array.
vectors[]		Array to contain the STATUSID vectors.

Table 8-4 Extended VME Interrupt Data

8.1.3 Wait Interrupt Data

This data structure is used internally within the API and is described here for information purposes only. The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
intNum	bit 0 = VOWN bit 1 = VIRQ1 bit 2 = VIRQ2 etc.	VME interrupt number selection. Bit 31 is used for validation, see the **`vme_waitInterrupt`** function
timeout	e.g. 200 jiffies 2 seconds 0 = wait forever.	Time to wait for interrupt. Parameter given in system ticks (jiffies).

Table 8-5 Wait Interrupt Data

8.1.4 Image Access Data

Before calling the **`vme_enableRegAccessImage`** or **`vme_enableLocationMon`** functions, an image access data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
vmeAddress	e.g. 0x10000	Base VME address of the image. Note: the Universe device fixes size of these images.
type	1 = data, 2 = program, 3 = both	Program/Data AM code.
mode	1 = non-privileged 2 = supervisor 3 = both	Supervisor/User AM code.
addrSpace	0 = A16, 1 = A24, 2 = A32	Address space modifier.

Table 8-6 Image Access Data

8.1.5 Direct DMA Transfer Data

Before calling the **`vme_dmaDirectTransfer`** function, a direct DMA transfer data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
direction	0 = VME to PCI bus (read) 1 = PCI to VME bus (write)	Direction of transfer.
vmeAddress	e.g. 0x10000	VME address. The address must be 8 byte aligned with the offset.
offset	e.g. 0x1000	Offset from start of DMA buffer. See vmeAddress for alignment.
size	e.g. 1024	Size to transfer
txfer	See above.	VME Transfer Parameter data structure
access	See above.	VME Access Parameter data structure.

Table 8-7 Direct DMA Transfer Data

8.1.6 Command Packet Data

Before calling the **`vme_addCmdPkt`** function, a command packet data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
direction	0 = VME to PCI bus (read) 1 = PCI to VME bus (write)	Direction of transfer.
vmeAddress	e.g. 0x10000	VME address. The address must be 8 byte aligned with the offset.
offset	e.g. 0x1000	Offset from start of DMA buffer. See vmeAddress for alignment.
size	e.g. 1024	Size to transfer
access	See above.	VME Access Parameter data structure.

Table 8-8 Command Packet Data

8.1.7 VME Transfer Parameters

Before calling the **`vme_dmaListTransfer`** function, a transfer parameter data structure must be created and its parameters assigned. The user must assign a value to all parameters. This data structure also forms part of the direct DMA transfer data structure.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
timeout	e.g. 200 jiffies 2 seconds	Maximum time allowed for DMA transfer to complete. Parameter given in system ticks (jiffies)
ownership	0 = don't release bus until done. Bits 0 -3 used for VOFF Bits 4 - 6 used for VON	VME bus On/Off counters (VON/VOFF) For a description of values other than zero please refer to the Universe II User Manual.

Table 8-9 VME Transfer Parameters

8.1.8 VME Access Parameters

This data structure allows the user to assign VME bus access parameters and forms part of the direct DMA transfer and Command packet data structures. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
dataWidth	0 = 8 bit, 1 = 16 bit, 2 = 32 bit, 3 = 64 bit	Maximum data width.
addrSpace	0 = A16, 1 = A24, 2 = A32 6 = User1, 7 = User2	Address space modifier.
type	0 = data, 1 = program	Program/Data AM code.
mode	0 = non-privileged 1 = supervisor	Supervisor/User AM code.
vmeCycle	0 = no BLT's, 1=single BLT's	VME bus cycle type.

Table 8-10 VME Access Parameters

8.1.9 User Address Modifier Data

Before calling the **`vme_setUserAmCodes`** function, a user address modifier data structure must be created and its parameters assigned. The user must assign a value to all parameters.

The data structure is defined in ***vme_api.h*** and its parameters are described in the table below:

Parameter	Values	Description
user1	16 - 31	User 1 address modifier code.
user2	16 - 31	User 2 address modifier code.

Table 8-11 User Address Modifier Data

Error Codes

The VME device driver API function may fail for a number of reasons. When they do, they return an error code indicating failure. It may also be useful to inspect the global **`errno`** variable, immediately after the problem, to gain an indication of the failure. The **`strerror`** function can be used to map the error code into a string describing the type of error that has occurred.

All error codes are less than zero. The values and meanings of the errors are listed in the standard Linux header file ***error.h***. Some typical errors include:

`EPERM`	operation not permitted.
`ENODEV`	no such device.
`ENXIO`	no such device or address.
`EINVAL`	invalid argument.
`EFAULT`	bad address.
`EBUSY`	device or resource busy.
`ENOMEM`	out of memory.

This page has been left intentionally blank

Programming Examples

This section contains a number of example programs illustrating the use of the Linux VME device driver API functions.

10.1 Example 1 - Generating A VME Bus Interrupt From Software

The following routine shows how to set the Status ID, then generate a VIRQ1 interrupt.

```
void generate_vme_interrupt( void )
{
    int devHandle;
    int result;
    devHandle = vme_openDevice( "ctl" );
    if ( devHandle >= 0 )
    {
        result = vme_setStatusId( devHandle, 0x2 );
        if ( result < 0 )
        {
            printf("Error - failed to set status ID (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("status ID set, generating VIRQ1\n");

            result = vme_generateInterrupt( devHandle, 1 );
            if ( result < 0 )
            {
                printf("Error - failed to generate interrupt (%s)\n",
                        strerror(errno));
            }
            else
            {
                printf("VIRQ1 generated\n");
            }
        }
        vme_closeDevice( devHandle );
    }
}
```

10.2 Example 2 - Reading VME Bus Interrupt Information

The following routine shows how to read VME bus interrupt information for VIRQ1.

```
void read_vme_interrupt( void )
{
    int devHandle;
    int result;
    int i;
    VME_INT_INFO info;

    devHandle = vme_openDevice( "ctl" );
    if ( devHandle >= 0 )
    {
        info.intNum = 1; /* get info for VIRQ1 */

        result = vme_readInterruptInfo( devHandle, &info );
        if ( result < 0 )
        {
            printf("Error - failed to read interrupt info (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("Number of interrupts: %u\n", info.numOfInts);

            for ( i = 0; i < info.vecCount; i++ )
            {
                printf("%02d     0x%02X\n", i, info.vectors[i]);
            }
        }
        vme_closeDevice( devHandle );
    }
```

10.3 Example 3 - Wait For An Interrupt

The following routine shows how a Mailbox could be used.

```
void wait_vme_interrupt( void )
{
    int devHandle;
    int result;
    UNIT32 intNum;
    UNIT32 selectedInts;
    VME_IMAGE_ACCESS idata;

    devHandle = vme_openDevice( "ctl" );
    if ( devHandle >= 0 )
    {
        idata.vmeAddress = 0x2000000;
        idata.addrSpace = VME_A32;
        idata.type = VSI_BOTH;  /* both Program and Data */
        idata.mode = VSI_BOTH;  /* both Supervisor and non-privileged */

        result = vme_enableRegAccessImage( devHandle, &idata );
        if ( result < 0 )
        {
            printf("Error - failed to enable Register Access (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("Register Access enabled\n");
            printf("Waiting for someone to write to Mailbox...\n");

            /* wait for mailbox interrupt to occur
               note: vme_waitInterrupt makes sure the interrupt is enabled
               so there no need to call vme_enableInterrupt first
           */
            selectedInts = 1L <<LINT_MBOXO;
            intNum = 0
            result = vme_waitInterrupt( devHandle, selectedInts, &intNum );
            if ( result < 0 )
            {
                printf("Error - failed to receive interrupt (%s)\n",
                        strerror(errno));
            }
            else
            {

                printf (”LINT_MBOX0 RECEIVED (0x%08X)\N”, intNum);
                printf("Interrupt %u received\n", LINT_MBOX0);
                result = vme_readRegister( devHandle, MBOX0, &reg );
                if ( result < 0 )
                {
                    printf("Error - failed to read register (%s)\n",
                                strerror(errno));
                }
                else
                {
                    printf("MBOX0: 0x%08X\n", reg);
                }
            }

            result = vme_disableRegAccessImage( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed to disable Register Access (%s)\n",
                        strerror(errno));
            }

        }
        vme_closeDevice( devHandle );
    }
}
```

10.4 Example 4 - Using a PCI Image Window

The following routine shows how to use an ioremapped PCI image window.

```
void enable_pci_image( void )
{
    int devHandle;
    int result;
    int i;
    UINT8 buffer[8];
    PCI_IMAGE_DATA idata;

    devHandle = vme_openDevice( "lsi0" );
    if ( devHandle < 0 )
    {
        printf("Error - failed to open lsi0\n");    }
    else
    {
        idata.pciAddress = 0xd0000000;
        idata.vmeAddress = 0x1000000;
        idata.size = 0x10000;
        idata.dataWidth = VME_D32;  /* 32 bit data */
        idata.addrSpace = VME_A32;  /* 32 bit address */
        idata.postedWrites = 1;
        idata.type = LSI_DATA;      /* data AM code */
        idata.mode = LSI_USER;      /* non-privileged */
        idata.vmeCycle = 0; /* no BLT's on VME bus */
        idata.pciBusSpace = 0;  /* PCI bus memory space */
        idata.ioremap = 1;  /* use ioremap */

        result = vme_enablePciImage( devHandle, &idata );
        if ( result < 0 )
        {
            printf("Error - failed to enable PCI image 0 (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("PCI image 0 enabled, reading data...\n");

            result = vme_read( devHandle, 0, buffer, 8 );
            if ( result < 0 )
            {
                printf("Error - failed to read data (%s)\n",
                        strerror(errno));
            }
            else
            {
                for ( i = 0; i < 8; i++ )
                {
                    printf("0x%02X\n", buffer[i] );
                }
                }
            result = vme_disablePciImage( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed to disable PCI image 0 (%s)\n",
                        strerror(errno));
            }
        }
        vme_closeDevice( devHandle );
    }
}
```

10.5 Example 5 - Using a VME Image Window

The following routine shows how to use a memory mapped VME image window.

```
void enable_vme_image( void )
{
    int devHandle;
    int result;
    int i;
    UINT8 buffer[8];
    UINT32 userMemAddrs;
    VME_IMAGE_DATA idata;


    devHandle = vme_openDevice( "vsi0" );
    if ( devHandle < 0 )
    {
        printf("Error - failed to open lsi0\n");
    }
    else
    {
        idata.vmeAddress = 0x2000000;
        idata.size = 0x10000;
        idata.addrSpace = VME_A32;  /* 32 bit address */
        idata.postedWrites = 1;
        idata.prefetchRead = 1;
        idata.type = VSI_BOTH;  /* both program and data */
        idata.mode = VSI_BOTH;  /* both supervisor and non-privileged */
        idata.pciBusLock = 0;   /* PCI bus lock disable */
        idata.pciBusSpace = 0;  /* PCI bus memory space */
        idata.ioremap = 0;      /* don,t ioremap */


        result = vme_enableVmeImage( devHandle, &idata );
        if ( result < 0 )
        {
            printf("Error - failed to enable VME image 0 (%s)\n",
                        strerror(errno));
        }
        else
        {
            result = vme_mmap( devHandle, 0, PAGE_SIZE, &userMemAddrs );

            if ( result < 0 )
            {
                 printf("Error - failed to memory map VME image 0 (%s)\n",
                         strerror(errno));

            }
            else
            {
                 printf("VME image 0 enabled and memory mapped, reading
                         data...\n");

                memcpy( buffer, (UINT8 *) userMemAddrs, 8 );

                for ( i = 0; i < 8; i++ )
                {
                    printf("0x%02X\n", buffer[i] );
                }

                result = vme_unmap( devHandle, userMemAddrs, PAGE_SIZE );

                if ( result < 0 )
                {
                    printf("Error - failed to  unmap VME image 0 (%s)\n",
                            strerror(errno));

                }
            }

            result = vme_disableVmeImage( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed to disable VME image 0 (%s)\n",
                        strerror(errno));
            }

        }

        vme_closeDevice( devHandle );
    }
}
```

10.6 Example 6 - Using Direct Mode DMA

The following routine shows how to read some data using direct mode DMA.

```
void do_direct_dma( void )
{
int devHandle;
    UINT8 buffer[8];
    VME_DIRECT_TXFER tdata;
    UINT32 size;
    int result;

    devHandle = vme_openDevice( "dma" );
    if ( devHandle >= 0 )
    {
        size = 4096;
        result = vme_allocDmaBuffer( devHandle, &size );
        if ( result < 0 )
        {
            printf("Error - failed to allocate DMA buffer (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("DMA buffer allocted, %u bytes available\n",
                    size);

            tdata.direction = UNI_DMA_READ;
            tdata.offset = 0;       /* start of DMA buffer */
            tdata.size = 4096;  /* read 4KB */
            tdata.vmeAddress = 0x1000000;

            tdata.txfer.timeout = 200;  /* 2 second timeout */
            tdata.txfer.ownership = 0;  /* hold VME bus until done */

            tdata.access.dataWidth = VME_D64;
            tdata.access.addrSpace = VME_A32;
            tdata.access.type = LSI_DATA;   /* data AM code */
            tdata.access.mode = LSI_USER;   /* non-privileged */
            tdata.access.vmeCycle = 1;  /* single BLT's */


            result = vme_dmaDirectTransfer( devHandle, &tdata );
            if ( result < 0 )
            {
                printf("Error - DMA transfer failed (%s)\n",
                        strerror(errno));
            }
            else
            {
                printf("DMA transfer successful\n");

                /* read and display the first 8 bytes of the buffer */
                result = vme_read( devHandle, 0, buffer, 8 );

                if ( result > 0 )
                {
                    for ( i = 0; i < 8; i++ )
                    {
                        printf("0x%02X\n", buffer[i] );
                    }
                }
            }

            result = vme_freeDmaBuffer( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed to free DMA buffer (%s)\n",
                        strerror(errno));
            }

        }

        vme_closeDevice( devHandle );
    }
}
```

10.7 Example 7 - Using Linked-list Mode DMA

The following routine shows how to read some data using linked list mode DMA.

```
void do_list_dma( void )
{
    int devHandle;
    UINT8 buffer[8];
    VME_CMD_DATA cmddata;
    VME_TXFER_PARAMS tdata;
    UINT32 size;
    int result;

    devHandle = vme_openDevice( "dma" );
    if ( devHandle >= 0 )
    {
        size = 4096;
        result = vme_allocDmaBuffer( devHandle, &size );
        if ( result < 0 )
        {
            printf("Error - failed to allocate DMA buffer (%s)\n",
                        strerror(errno));
        }
        else
        {
            printf("DMA buffer allocted, %u bytes available\n",
                    size);

            /* clear any existing command packets */
            result = vme_clearDmaCmdPkts( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed clear list (%s)\n",
                        strerror(errno));
            }
            else
            {
                cmddata.direction = UNI_DMA_READ;
                cmddata.offset = 0; /* start of DMA buffer */
                cmddata.size = 4096;    /* read 4KB */
                cmddata.vmeAddress = 0x1000000;

                cmddata.access.dataWidth = VME_D64;
                cmddata.access.addrSpace = VME_A32;
                cmddata.access.type = LSI_DATA;  /* data AM code */
                cmddata.access.mode = LSI_USER;  /* non-privileged */
                cmddata.access.vmeCycle = 1;    /* single BLT's */

                result = vme_addDmaCmdPkt( devHandle, &cmddata );
                if ( result < 0 )
                {
                     printf("Error - failed to add command packet (%s)\n",
                             strerror(errno));

                }
                else
                {
                    /* 2 second timeout, hold VME bus until done */
                       tdata.txfer.timeout = 200;
                    tdata.txfer.ownership = 0;
                    result = vme_dmaListTransfer( devHandle, &tdata );
                    if ( result < 0 )
                    {
                        printf("Error - DMA transfer failed (%s)\n",
                                strerror(errno));
                    }
                    else
                    {
                        printf("DMA transfer successful\n");

                        /* read and display the first 8 bytes */
                        result = vme_read( devHandle, 0, buffer, 8 );

                        if ( result > 0 )
                        {
                            for ( i = 0; i < 8; i++ )
                            {
                                printf("0x%02X\n", buffer[i] );
                            }
                        }
                    }
                }
            }

            vme_clearDmaCmdPkts( devHandle );

            result = vme_freeDmaBuffer( devHandle );
            if ( result < 0 )
            {
                printf("Error - failed to free DMA buffer (%s)\n",
                        strerror(errno));
            }

        }

        vme_closeDevice( devHandle );
    }
}
```

This page has been left intentionally blank