Technical Reference Manual
For
Linux® Enhanced VME
Device Drivers

Manual Order Code 555 0051   Rev 04   Nov 2010

Concurrent Technologies Inc
6 Tower Office Park
Woburn, MA 01801
USA
Tel: (781) 933 5900
Fax: (781) 933 5911

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 * after a name represents an active low signal - e.g. INIT# or INIT*
- h denotes a hexadecimal number - e.g. FF45h
- byte represents 8-bits
- word represents 16-bits
- dword represents 32-bits

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.

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

REVISION HISTORY

Revision	Summary of Changes	Date
01	Initial Release	May 2009
02	64 bit driver; additional Kernel API's added to Section 8 (8.41 and 8.42)	August 2009
03	Added Kernel APIs to Section 8 (8.43 and 8.44). Added multiple device access support.	November 2009
04	Added User Space API to Section 7 (7.42) Added Kernel API to Section 8 (8.45) Added Example Section 11 Added details to Section 12	November 2010

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 INTRODUCTION

1.1 Overview

The Concurrent Technologies Linux® Enhanced VME device driver allows the user to access the VME bus by providing an interface to a Tundra® Universe II™ or a Tundra® Tsi148™ device. These devices provide the PCI to VME bus bridge interface on Concurrent Technologies VME products.

This device driver allows a Linux user application to access many features of the devices 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 driver also provides a kernel space API which allows other drivers to access the VME bus from within the Linux kernel.

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 system is already installed on the target board(s).
- The Concurrent Technologies Linux Enhanced 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 Section 2 (Software Prerequisites).

This page is intentionally unused.

2 SOFTWARE PREREQUISITES

The software is distributed in Red Hat® Package format and requires the Red Hat Package Manager (RPM) to be installed.

The Linux Enhanced 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. Where the binary object module is not provided, the binary module can be built as the driver is provided in part source and part binary format. See the **`readme`** file supplied with the package for more details.

The Linux Enhanced 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 Enhanced 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 is intentionally unused.

3 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. Programming examples using User Space API are included in the Board Support Package – see Section 11 for more details.

The VME Device driver also provides kernel space API functions which can be called from other device drivers to access the VME bus. When the driver is loaded, these kernel space API functions are exported to become part of the kernel symbol table. Programming examples using Kernel Space API are included in the Board Support Package – see Section 12 for more details.

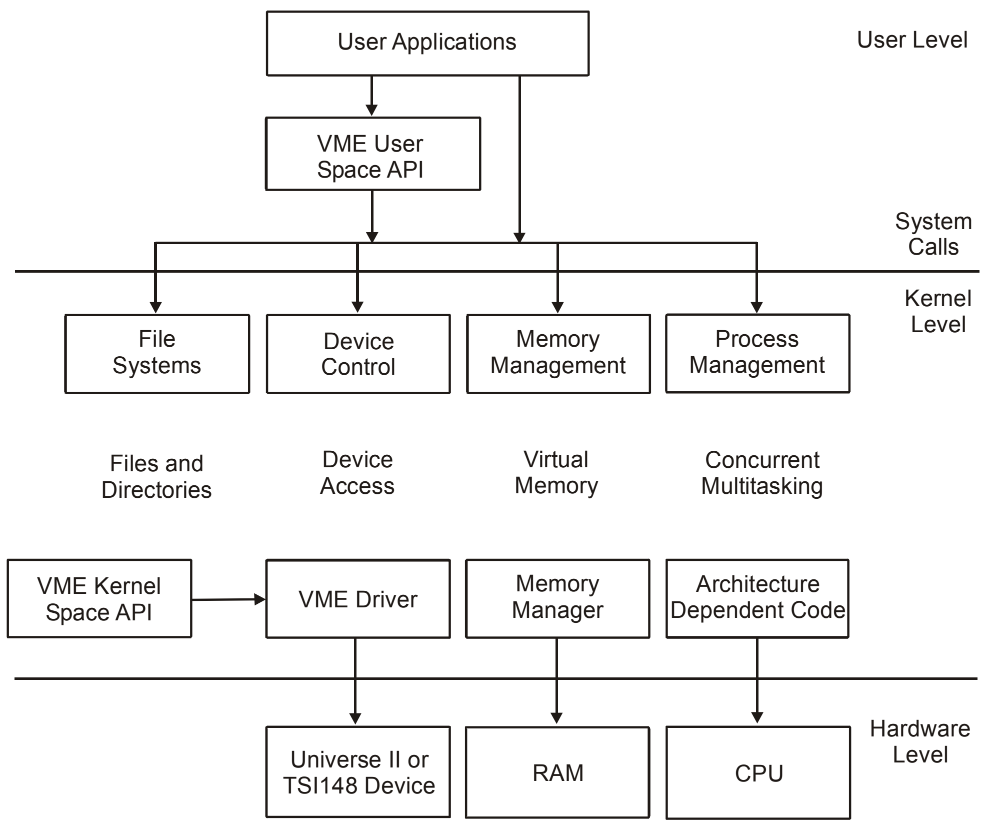

Figure 3-1 **System Overview**

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. The device file can be opened and accessed by multiple processes/ applications. It is necessary to synchronize between processes/ applications when devices are accessed simultaneously from different processes/ applications.

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 and Tsi148 devices. Operations using this device file can be summarized as follows:

- Read and write Universe II/Tsi148 registers.
- Set User Address Modifiers in the case of Universe II.
- Set hardware byte swapping, for boards that support it.
- Enable and disable Universe II/Tsi148 interrupts.
- Allows an application to wait for a specified Universe II/Tsi148 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.

NOTE: With the Tsi148 Enhanced API, there are two DMA device files, namely **`/dev/vme/dma0`** and **`/dev/vme/dma1`**, giving access to the two DMA controllers.

3.2.5 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.

3.3 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.

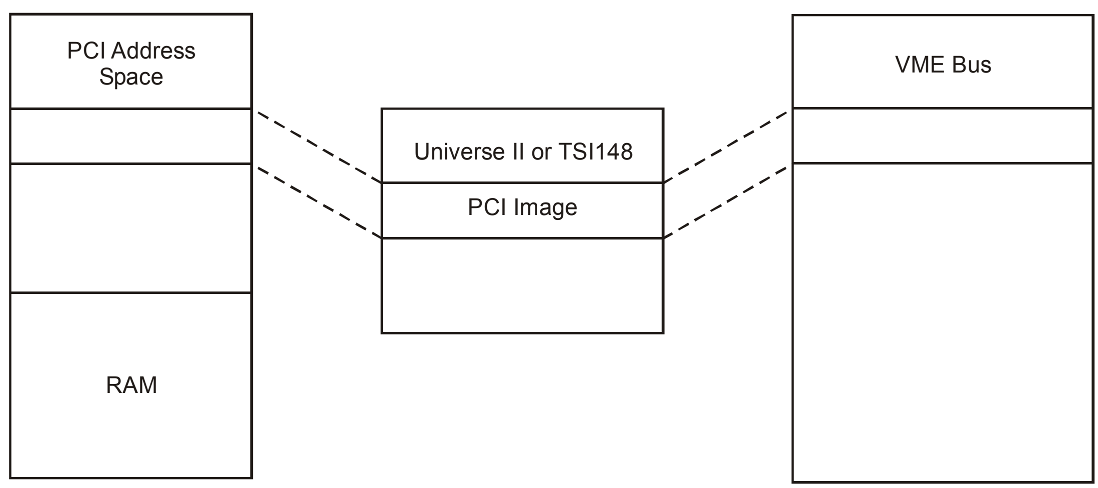

Figure 3-2 Image Window

A PCI image window can be setup and enabled by calling the VME device driver **`vme_enablePciImage`** and **`vmekrn_enablePciImage`** functions. Up to eight PCI images (numbered 0 to 7) can be used. In the case of Universe II, PCI images 0 and 4 have a 4 Kbyte resolution and PCI images 1, 2, 3, 5, 6, and 7 have a 64 Kbyte resolution. In the case of the Tsi148, all the PCI images have a resolution of 64Kbytes.

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`**, **`vmekrn_readImage`**, **`vme_write`** and **`vmekrn_write_Image`** 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. This is not applicable for kernel space APIs.

3.4 Universe II/Tsi148 Interrupt Handling

The VME device driver provides an interrupt routine to handle PCI interrupts generated by the Universe II/Tsi148 device. The Universe II and Tsi148 can generate 24 and 26 different PCI interrupts respectively, by default all these interrupts are disabled. However, the user can enable individual interrupts by calling the **`vme_enableInterrupt`** or **`vmekrn_enableInterrupt`** functions 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. In the case of kernel space APIs the **`vmekrn_waitInterrupt`** function can be used to wait for interrupts. Alternately a kernel user driver can register up to 32 callback functions to be invoked when an interrupt occurs. The API to register the kernel callback function is **`vmekrn_registerInterrupt`** and to remove is **`vmekrn_removeInterrupt`**.

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

VME bus interrupts can also be generated via software by calling the **`vme_generateInterrupt`** or **`vmekrn_generateInterrupt`** function. The contents of the Universe II Status ID register and the Tsi148 Status ID field of the VICR 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`** or **`vmekrn_setInterruptMode`** functions.

3.5 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 three ways of reserving memory, namely via BIOS configuration, during driver load time or by using the **`vme_reserveMemory`** API function. All of these methods are described in the following sections.

3.5.1 Reserving Memory at Load Time

In case of a 32 bit or a 64 bit system, first, an area of reserved 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. On a 32 bit system fitted with 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. Please note that the **`highmem=`** argument needs to be used only in case of a 32 bit system as in case of a 64 bit system the **`mem=`** argument needs to be used irrespective of the RAM size.

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** command line parameter in Section 3.12 for more details.

3.5.2 Reserving Memory at Run Time

Another way to reserve memory is to use the **vme_reserveMemory** API function. 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.

3.5.3 Reserving Memory from BIOS

Memory can also be reserved using the setup option provided in the system BIOS. When using this method of reserving the memory, the size of the memory block to be reserved needs to be selected from the BIOS setup option called “**VSI Reserved Size**”. This setup option will normally be in the “**Universe**” or “**VMEBus**” setup menu of the BIOS depending upon the board being used. Once selected, from the next boot onwards the BIOS reserves the specified amount of physical memory which will then be used by the driver for the VME image and DMA buffer allocation. Please note that not all the boards support this option.

> **NOTE:** The reserve memory allocation during load time using the command line option and run time using the **vme_reserveMemory** API will take precedence over the BIOS memory reservation in cases where more than one memory reservation method are used.

3.5.4 Reserving Memory for Xen Virtualization Software Enabled Kernels

Some of the recent Linux distributions are delivered with support for Xen virtualization software. Please note that this section is applicable only if the virtualization support is specifically enabled.

The traditional way of reserving memory at the top of the physical memory using the **mem** or **highmem** command line parameters with the Linux kernel will not apply. In order to reserve memory for driver usage at the top of the physical memory, the **mem=** command line parameter needs to be passed to the Xen Hypervisor. For example, if you have a board with 1 Gbyte of physical memory and if you want to reserve 8 Mbytes at the top, then **mem=1016M** should be the command line argument for the Xen Hypervisor.

With **grub** this can be done in **grub.conf** as follows:

```
title red hat Enterprise Linux Server (2.6.18-3.e15xen)
root (hd0; 0)
kernel /xen.gz.-2.6.18-3.e15 mem=1016M
module /vmlinux-2.6.18-3.e15xen ro root=/dev/VolGroup00/LogVol00 rhgb quiet
module /initrd-2.6.18-3.e15xen.img
```

When the VME driver is loaded an additional command line parameter **`resMemStart`** needs to be mentioned along with **`resMemSize`**. The **`resMemStart`** specifies the physical start address of the driver's reserved memory. Hence, for the example stated above, the **`resMemStart=0x3F800000`** needs to be passed along with **`resMemSize=8`**.

3.5.5 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.

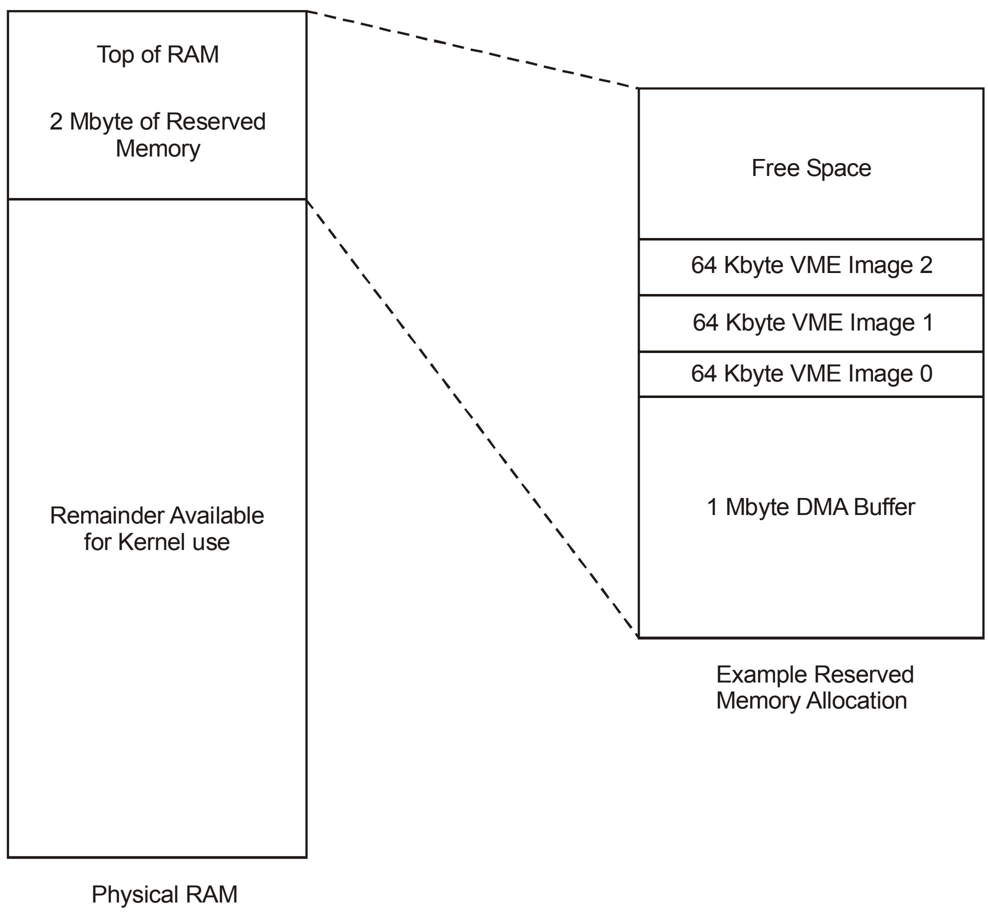

Figure 3-3 Reserved Memory Layout

3.6 DMA Transfer

The VME device driver allows the user to configure the single DMA controller with the Universe and two DMA controllers with the Tsi148 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 user's application program and kernel space drivers to perform the data transfers.

Before the DMA transfer functions can be used a DMA buffer must be allocated with the **`vme_allocDmaBuffer`** or **`vmekrn_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 devices file or by memory mapping the DMA buffer into user space with the **`mmap`** function. In the case of kernel space APIs the **`vmekrn_readDma`** and **`vmekrn_writeDma`** functions can be used to read from and write into the DMA buffer.

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. This is not applicable in the case of kernel space APIs.

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.

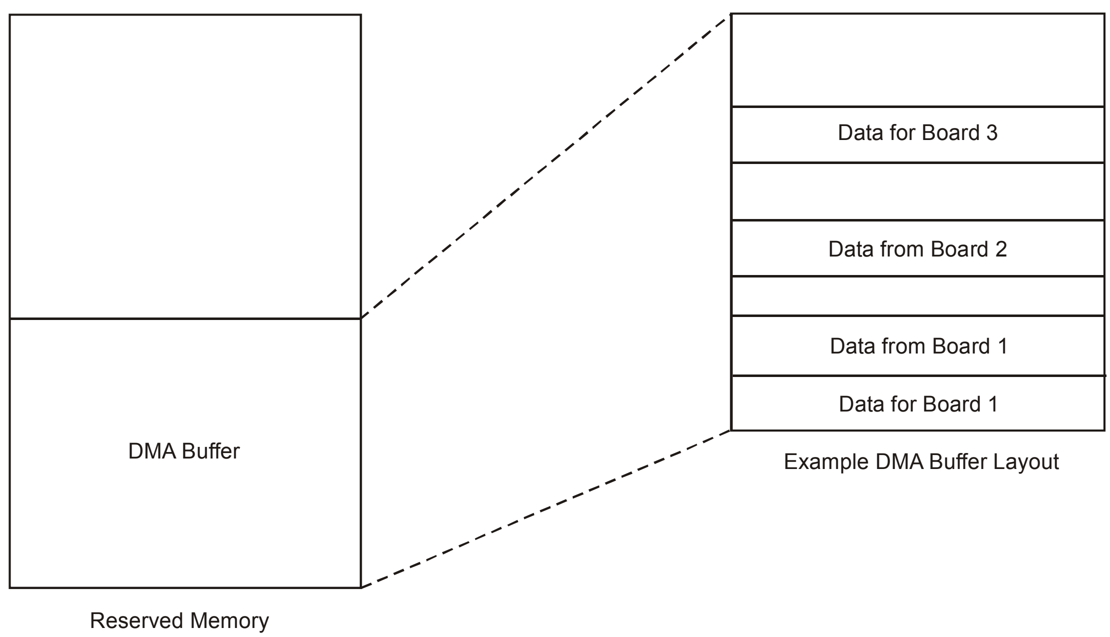

Figure 3-4 DMA Buffer Layout

In Figure 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.

3.6.1 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`** or **`vmekrn_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/Tsi148 DMA registers. The function will return on completion of the DMA, if an error is detected or if the specified timeout period expires.

3.6.2 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/Tsi148 DMA register layout. The structure of each command packet is the same, and contains all the necessary information to program the Universe II/Tsi148 DMA address and control registers. When a linked-list transfer is started, the Universe II/Tsi148 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`** or **`vmekrn_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, as the last command packet in the list is executed first.

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

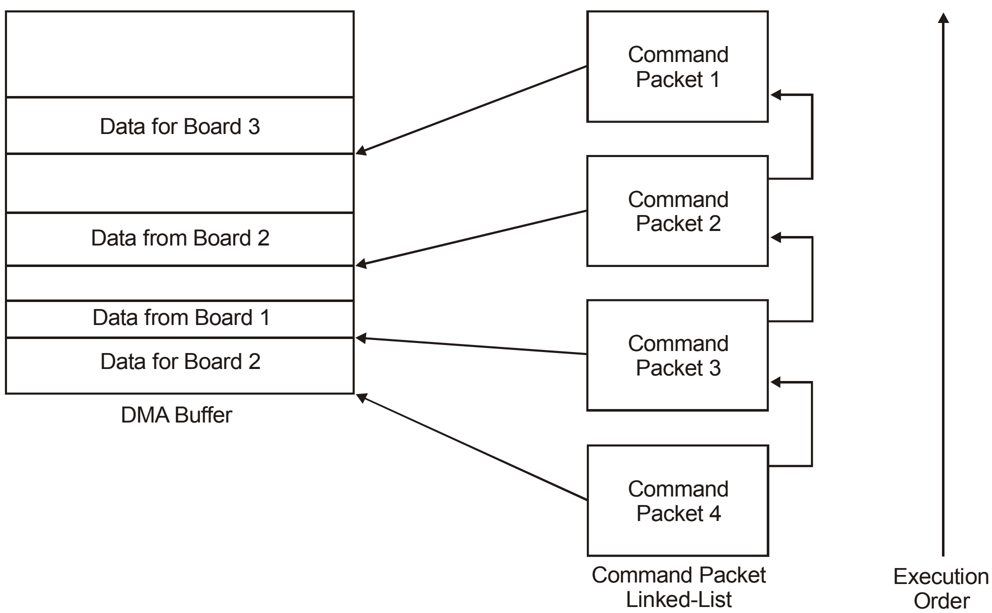

Figure 3-5 Command Packet Linked-List

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

3.7 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 Figure 3-6.

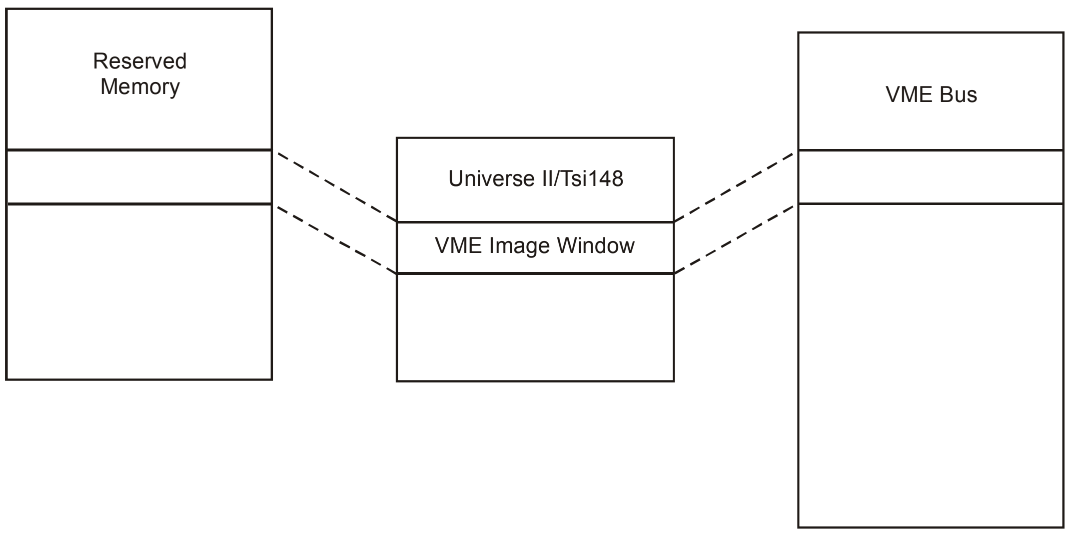

Figure 3-6 VME Image Window

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`** or **`vmekrn_enableVmeImage`** function. Up to eight VME images (numbered 0 to 7) can be used. For the Universe II, VME images 0 and 4 have a 4 Kbyte resolution and VME images 1, 2, 3, 5, 6 and 7 have a 64 Kbyte resolution. For the Tsi148, the granularity of the VME images depends upon the address modifiers being used. When using A16, A24, A32 and A64 address modifiers, the granularity is 16 byte, 4 Kbytes, 64 Kbytes and 64 Kbytes respectively.

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. In the case of kernel space APIs, the **`vme_readImage`** and **`vme_writeImage`** API functions can be used to access the VME image window.

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. This is not applicable to kernel space API.

3.8 Universe II/Tsi148 Register Access from the VME Bus

The Universe II Control and Status Registers (UCSR) and Tsi148 Combined Register Group (CRG) occupy 4 Kbytes of internal memory. There are two mechanisms to access the UCSR/CRG register space from the VME bus.

One method uses a VME bus Register Access Image that allows the user to put the UCSR/CRG in an A16, A24, A32 or A64 address space. A64 address space applies only to the Tsi148 bridge. This image can be setup and enabled by calling the VME device driver **`vme_enableRegAccessImage`** or **`vmekrn_enableRegAccessImage`** function.

The other way to access the UCSR/CRG 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`** or **`vmekrn_enableCsrImage`** function.

3.9 Mailbox Communications

The Universe II/Tsi148 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/Tsi148 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/Tsi148 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.

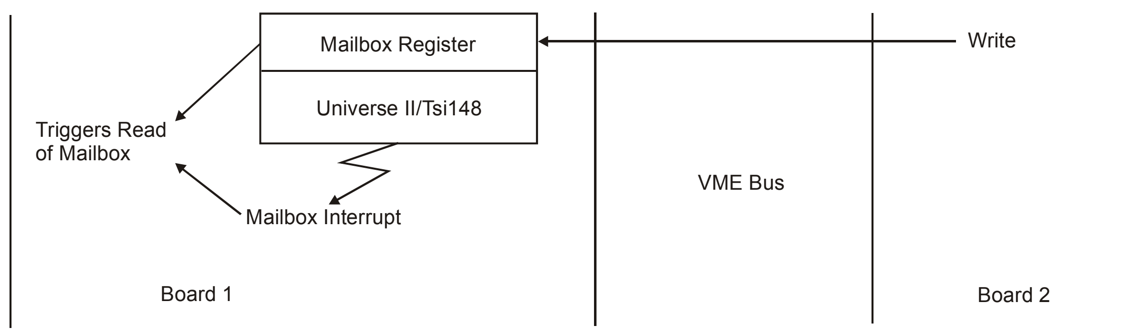

Figure 3-7 Mailbox Communications

The Mailboxes are accessible from the same address spaces and in the same manner as the other Universe II/Tsi148 registers, as described in Section 3.8.

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

The Tsi148 Mailbox registers are located at offsets:

0x610	Mailbox register 0
0x614	Mailbox register 1
0x618	Mailbox register 2
0x61C	Mailbox register 3

A user application can be informed when a Mailbox interrupt occurs by using the **`vme_waitInterrupt`** function. In the case of kernel space drivers, the **`vmekrn_waitInterrupt`** function can be used to wait for a mailbox interrupt.

3.10 Location Monitoring

The Universe II/Tsi148 has four Location Monitors to provide VME bus broadcast capability. In the case of Universe II, 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. In the case of Tsi148, four locations starting at the address configured in the LMBA register with each location being eight bytes long are monitored.

The Location Monitors can be setup and enabled by calling the VME device driver **`vme_enableLocationMon`** or **`vmekrn_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. The kernel device drivers can use the **`vmekrn_waitInterrupt`** function to wait for the location monitor interrupts.

3.11 User Address Modifiers

In addition to the standard address modifiers, A16, A24, and A32, it is also possible to program the Universe II device with two user defined address modifiers. This can be achieved by first calling the **`vme_setUserAmCodes`** or **`vmekrn_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. In the case of the Tsi148, **`vme_setUserAmCodes`** and **`vmekrn_setUserAmCodes`** functions are not supported as the first four bits of the User address modifiers are selected by choosing either User 1, 2, 3 or 4 address modifiers when selecting VME bus access and the last two bits are selected by the SUP and PGM bit.

3.12 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.

3.12.1 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

3.12.2 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"`**.

3.12.3 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.

3.12.4 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 PCI addresses obtained during initialization.

`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 as follows in case the values were not obtained from the BIOS.

`pciAddr`	0xB0000000
`pciSize`	0x10000000

4 SOFTWARE INSTALLATION

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

This page is intentionally unused.

5 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. 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 if the execute attribute is set on the script file.
- Check that the current directory is **/usr/local/linuxvmeen**

Then type **./ins** to run the installation script 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 as specified in Section 2 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 is intentionally unused.

6 LINUX VME UTILITY PROGRAM

The Linux utility program exercises many of the functions provided by the VME device driver. 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/linuxvmeen_util`**/ directory. To build the utility program for Linux make sure the current directory is **`/usr/local/linuxvmeen_util`** and then type:

```
make all
```

Once the build process has completed type **`./linuxvmeen`** 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 is intentionally unused.

7 USER SPACE API

7.1 Overview

The Concurrent Technologies Linux VME API allows the user to use the features provided by the Tsi148 and Universe II bridges. A detailed description of each API function is given in this section. The API is provided as a static library file i.e. **`libcctvmeen.a`**, which is found in the **`/usr/local/linuxvmeen_util`** directory. Along with the libraries, the header file **`vme_api_en.h`** is provided. This file contains the declarations for the API.

The 32-bit APIs are limited to 32-bit addressing capability. To address 64-bit memory space, there are 64-bit versions of the API. Where these are applicable, both 32-bit and 64-bit declarations are provided. The 64-bit APIs can only be used under the 64-bit Linux Operating System.

Some programming examples using the User Space API functions described in this section can be found in the Board Support Package – see Section 11 for more details.

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 II/Tsi148 device register at the given offset. Valid Universe II/Tsi148 register offsets are defined in the **`vme_api_en.h`** header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** In the case of the Tsi148, PCFS registers are little endian while all the other registers are big endian. To provide uniform access to all registers in both the Universe II and Tsi148, the API register access functions automatically perform byte-swapping where necessary, allowing the application to use little endian values throughout.

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 II/Tsi148 device register at the given offset. Valid Universe II/Tsi148 register offsets are defined in the **`vme_api_en.h`** header file.

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

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** In the case of the Tsi148, the PCFS registers are little endian while all the other registers are big endian. To provide uniform access to all registers in both the Universe II and Tsi148, the API register access functions automatically perform byte-swapping where necessary, allowing the application to use little endian values throughout.

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_ROAK`** or **`INT_MODE_RORA`**) as defined in **`vme_api_en.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 II/Tsi148 interrupt number.

Description:

Enables the given Universe II/Tsi148 device interrupt. Valid Universe II/Tsi148 interrupt numbers are enumerated in the **`vme_api_en.h`** header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

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 II/Tsi148 interrupt number.

Description:

Disables the given Universe II/Tsi148 device interrupt. Valid Universe II/Tsi148 interrupt numbers are enumerated in the **`vme_api_en.h`** header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

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 Section 9 (Data Structures for Enhanced API).

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 STATUSID register for the Universe II or the Status ID field of the VICR register for the Tsi148 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 = Res/VOWN	bit 1 = VIRQ1	bit 2 = VIRQ2
bit 3 = VIRQ3	bit 4 = VIRQ4	bit 5 = VIRQ5
bit 6 = VIRQ6	bit 7 = VIRQ7	bit 8 = ACFAIL
bit 9 = SYSFAIL	bit 10 = IACK	bit 11 = VIE/SWINT
bit 12 = VERR	bit 13 = PERR	bit 14 = Reserved
bit 15 = Reserved	bit 16 = MBOX0	bit 17 = MBOX1
bit 18 = MBOX2	bit 19 = MBOX3	bit 20 = LM0
bit 21 = LM1	bit 22 = LM2	bit 23 = LM3
bit 24 = DMA0	bit 25 = DMA1	

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 II/Tsi148 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.

NOTE: Two interrupt bits have a different meaning depending on the bridge device used:

bit 0 = Res/VOWN is RESERVED on the Tsi148 and VOWN on the Universe II

bit 11 = VIE/SWINT is VIE on the Tsi148 and SWINT on the Universe II

NOTE: bit 25 = DMA1 is not supported by the Universe II.

7.13 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.

Description:

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

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.

> **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.

7.14 vme_enableRegAccessImage

Declaration:

```
int vme_enableRegAccessImage ( INT32 deviceHandle,
                               EN_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 Register Access image. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 Register Access with the given parameters. This image maps the Tsi148/Universe II registers onto the VME bus enabling other boards in the system to access them.

Returns:

Zero if successful or an error code upon failure.

7.15 vme_disableRegAccessImage

Declaration:

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

Parameters:

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

Description:

Disables the Universe II/Tsi148 Register Access image.

Returns:

Zero if successful or an error code upon failure.

7.16 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 Register (UCSR) and Tsi148 Combined Register Group 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.

7.17 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 the VME64 specification.

Description:

Disables the given CR/CSR image.

Returns:

Zero if successful or an error code upon failure.

7.18 vme_enableLocationMon

Declaration:

```
int vme_enableLocationMon ( INT32 deviceHandle,
                            EN_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 Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 Location monitors, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

7.19 vme_disableLocationMon

Declaration:

`int vme_disableLocationMon ( INT32 deviceHandle );`

Parameters:

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

Description:

Disables the Universe II/Tsi148 Location monitors.

Returns:

Zero if successful or an error code upon failure.

7.20 vme_getStats

Declaration:

`int vme_getStats( INT32 deviceHandle, UINT32 type, void *iPtr );`

Parameters:

`deviceHandle`	device handle obtained from a previous call to **`vme_openDevice`**
`type`	status information type as defined in **`vme_api_en.h`**.
`*iPtr`	pointer to the data structure that will receive the requested status information. The status information data structures are described in the Section 9 (Data Structures for Enhanced API).

Description:

Get the requested status information from VME device driver.

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,
                         EN_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 Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 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 Section 3.12.5 (Overriding PCI Space Usage) for the default values used for the PCI address pool.

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 II/Tsi148 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,
                         EN_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 Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 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 VME image window specified by the device handle.

Returns:

Zero if successful or an error code upon failure.

7.26 vme_read

Declaration:

32 Bit Version:

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

64 Bit Version:

```
int vme_read ( INT32 deviceHandle, ULONG offset, UINT8 *buffer,
               ULONG 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:

32 Bit Version:

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

64 Bit Version:

```
int vme_write ( INT32 deviceHandle, ULONG offset, UINT8 *buffer,
                ULONG 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 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:

32 Bit Version:

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

64 Bit Version:

```
int vme_mmap ( INT32 deviceHandle, ULONG offset, ULONG  length,
               ULONG *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 4 Kbytes.
***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:

32 Bit Version:

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

64 Bit Version:

```
int vme_unmap( INT32 deviceHandle, ULONG userAddress, ULONG 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 user's 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:

32 Bit Version:

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

64 Bit Version:

```
int vme_allocateDmaBuffer ( INT32 deviceHandle, ULONG *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 II/Tsi148 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 DMA0 or DMA1 device file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

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,
                            EN_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 Section 9 (Data Structures for Enhanced API).

Description:

Initiates a Universe II/Tsi148 direct mode DMA transfer, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

7.33 vme_addDmaCmdPkt

Declaration:

```
int vme_addDmaCmdPkt ( INT32 deviceHandle,
                       EN_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 Section 9 (Data Structures for Enhanced API).

Description:

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

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

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,
                           EN_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 Section 9 (Data Structures for Enhanced API).

Description:

Initiates a Universe II/Tsi148 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.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

7.36 vme_getApiVersion

Declaration:

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

Parameters:

`*buffer`	pointer to a buffer to hold the version string, maximum buffer length 1024.

Description:

Gets the API version information as a string.

Returns:

Size of version information string.

7.37 vme_reserveMemory

Declaration:

32 Bit Version:

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

64 Bit Version:

```
int vme_reserveMemory( INT32 deviceHandle, ULONG physicalAddress,
                       ULONG 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.

7.38 vme_readVerrInfo

Declaration:

32 Bit Version:

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

64 Bit Version:

```
int vme_readVerrInfo ( INT32 deviceHandle, ULONG *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 and 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 Tsi148 when a VERR interrupt occurs, therefore, the VERR interrupt must first be enabled by calling the **vme_enableInterrupt** function.

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_setUserAmCodes

Declaration:

`int vme_setUserAmCodes ( INT32 deviceHandle, EN_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 Section 9 (Data Structures for Enhanced API).

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.

> **NOTE:** **`vme_setUserAmCodes`** is not supported by the Tsi148 – invoking this API for Tsi148 will return an error code of VME_EPERM.

7.40 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 Section 9 (Data Structures for Enhanced API).

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.

7.41 vme_getBoardCap

Declaration:

```
int vme_getBoardCap( INT32 deviceHandle, UINT32 *boardFlags );
```

Parameters:

`deviceHandle`	device handle obtained from a previous call to `vme_openDevice`.
`*boardFlags`	Pointer to a 32 bit quantity specifying the board capabilities. Bit 3 – bit 11 specify the different capabilities supported by the board.

bits 0 – 2 - Reserved
bit 3 - Byte Swap support
bit 4 - Single Cycle Transfers support
bit 5 - Block Transfers support
bit 6 - Multi Block Transfers support
bit 7 - 2eVME Transfers support
bit 8 - 2eSST 160 Transfers support
bit 9 - 2eSST 267 Transfers support
bit 10 - 2eSST 320 Transfers support
bit 11 - 2eSST Broadcast support
bits 12 - 31 - Reserved

Description:

Get the information of the capabilities supported by the board.

Returns:

Zero if successful or an error code upon failure.

7.42 vme_getInstanceCount

Declaration

```
int vme_getInstanceCount( INT32 minorNum, UINT32 *pCount );
```

Parameters:

minorNum	number of the VME image as defined in **vme_api_en.h**.
***pCount**	pointer to obtain the count of the number of times the device is left opened.

Description:

Obtains the number of times the VME device is opened and left unclosed.

This API can only be used after opening a corresponding VME device and cannot be used after closing the VME device.

Returns:

Zero if successful or an error code upon failure.

8 KERNEL SPACE API

8.1 Overview

The kernel space application programming interface (API) consists of a series of functions which allow access to the Universe II and Tsi148 bridges from the kernel space. The kernel drivers can use these APIs to perform any desired VME operation.

A detailed description of each kernel space API is provided in this section. The kernel space API usage is similar to the user space API explained in the previous section with the same data structure definitions used between them. The user space and kernel space APIs can be used concurrently but at any given time only one set of API can have the ownership of a particular image. For example, if a kernel device driver has acquired the ownership of lsi0 image then any user space application will not be able to access the same unless relinquished by the kernel driver.

The kernel API declarations that need to be included whenever these APIs need to be used are provided in the file **`vme_api_en.h`**.

Some programming examples using the Kernel Space API functions can be found in the Board Support Package – see Section 12 for more details.

These APIs support 32-bit and 64-bit operating mode, based on the installed 32-bit or 64-bit drivers.

8.2 vmekrn_acquireDevice

Declaration:

`int vmekrn_acquireDevice ( UINT32 imageNumber );`

Parameters:

`imageNumber` number of the image as defined in `vme_api_en.h`.

Description:

Before a kernel device driver can perform any operation, the ownership of the required image must be acquired. Calling `vmekrn_acquireDevice` acquires the ownership of the specified VME device for the kernel API usage.

Returns:

Zero if successful or an error code upon failure.

8.3 vmekrn_releaseDevice

Declaration:

`void vmekrn_releaseDevice( UINT32 imageNumber );`

Parameters:

`imageNumber` number of the image as defined in **`vme_api_en.h`**.

Description:

Releases the specified image acquired by calling **`vmekrn_acquireDevice`**. Resources obtained by the VME device driver, while the image was acquired are freed and image windows are disabled where applicable.

Returns:

Zero if successful or an error code upon failure.

8.4 vmekrn_readRegister

Declaration:

```
int vmekrn_readRegister( UINT32 offset, UINT32 *reg );
```

Parameters:

`offset`	offset of register to read.
`*reg`	location to store register values.

Description:

Reads a Universe II/Tsi148 device register at the given offset. Valid Universe II/Tsi148 register offsets are defined in the **`vme_api_en.h`** header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** In the case of the Tsi148, PCFS registers are little endian while all the other registers are big endian. To provide uniform access to all registers in both the Universe II and Tsi148, the API register access functions automatically perform byte-swapping where necessary, allowing the application to use little endian values throughout.

8.5 vmekrn_writeRegister

Declaration:

```
int vmekrn_writeRegister( UINT32 offset, UINT32 reg );
```

Parameters:

`offset`	offset of register to write.
`reg`	register value.

Description:

Writes to a Universe II/Tsi148 device register at the given offset. Valid Universe II/Tsi148 register offsets are defined in the **`vme_api_en.h`** header file.

> **CAUTION:** Care should be taken when using **`vmekrn_writeRegister`** as it may affect the operation of other functions.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** In the case of the Tsi148, the PCFS registers are little endian while all the other registers are big endian. To provide uniform access to all registers in both the Universe II and Tsi148, the API register access functions automatically perform byte-swapping where necessary, allowing the application to use little endian values throughout.

8.6 vmekrn_setInterruptMode

Declaration:

```
int vmekrn_setInterruptMode( UINT8 mode );
```

Parameters:

mode	selected interrupt mode (**INT_MODE_ROAK** or **INT_MODE_RORA**) as defined in **vme_api_en.h**.

Description:

Sets the interrupt mode to ROAK (default) or RORA for incoming VIRQs.

Returns:

Zero if successful or an error code upon failure.

8.7 vmekrn_enableInterrupt

Declaration:

```
int vmekrn_enableInterrupt( UINT8 intNumber );
```

Parameters:

`intNumber` Universe II/Tsi148 interrupt number.

Description:

Enables the given Universe II/Tsi148 device interrupt. Valid Universe II/Tsi148 interrupt numbers are enumerated in the `vme_api_en.h` header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.8 vmekrn_disableInterrupt

Declaration:

```
int vmekrn_disableInterrupt ( UINT8 intNumber );
```

Parameters:

`intNumber` Universe II/Tsi148 interrupt number.

Description:

Disables the given Universe II/Tsi148 device interrupt. Valid Universe II/Tsi148 interrupt numbers are enumerated in the `vme_api_en.h` header file.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.9 vmekrn_generateInterrupt

Declaration:

```
int vmekrn_generateInterrupt ( UINT8 intNumber );
```

Parameters:

`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.

8.10 vmekrn_readInterruptInfo

Declaration:

```
int vmekrn_readInterruptInfo ( EN_VME_INT_INFO *iPtr );
```

Parameters:

`*iPtr`	pointer to a data structure where interrupt information will be stored. The parameters are described in Section 9 (Data Structures for Enhanced API).

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 **`vmekrn_readExtInterruptInfo`** function should be used instead.

8.11 vmekrn_setStatusId

Declaration:

```
int vmekrn_setStatusId(  UINT8 statusId );
```

Parameters:

statusId	status ID value.

Description:

Sets the STATUSID register for the Universe II or the Status ID field of the VICR register for the Tsi148 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 **vmekrn_generateInterrupt** call is used.

Returns:

Zero if successful or an error code upon failure.

8.12 vmekrn_waitInterrupt

Declaration:

```
int vmekrn_waitInterrupt( UINT32 selectedInts, UINT32 timeout, UINT32
                          *intNum );
```

Parameters:

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

bit 0 = Res/VOWN	bit 1 = VIRQ1	bit 2 = VIRQ2
bit 3 = VIRQ3	bit 4 = VIRQ4	bit 5 = VIRQ5
bit 6 = VIRQ6	bit 7 = VIRQ7	bit 8 = ACFAIL
bit 9 = SYSFAIL	bit 10 = IACK	bit 11 = VIE/SWINT
bit 12 = VERR	bit 13 = PERR	bit 14 = Reserved
bit 15 = Reserved	bit 16 = MBOX0	bit 17 = MBOX1
bit 18 = MBOX2	bit 19 = MBOX3	bit 20 = LM0
bit 21 = LM1	bit 22 = LM2	bit 23 = LM3
bit 24 = DMA0	bit 25 = DMA1	

`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 II/Tsi148 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:** **`vmekrn_waitInterrupt`** makes sure the selected interrupts are enabled so there is no need to call **`vmekrn_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.

> **NOTE:** Some interrupt bits have a different meaning in case of Tsi148 or Universe II:
> bit 0 = Res/VOWN is Reserved on Tsi148 and VOWN on Universe II
> bit 11 = VIE/SWINT is VIE on Tsi148 and SWINT on Universe II

> **NOTE:** bit 25 = DMA1 is not supported on Universe II

8.13 vmekrn_setByteSwap

Declaration:

```
int vmekrn_setByteSwap ( UINT8 enable );
```

Parameters:

`enable` set or clear bits in this parameter to enable/disable byte swapping.

Description:

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

- 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.

> **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.

8.14 vmekrn_enableRegAccessImage

Declaration:

```
int vmekrn_enableRegAccessImage ( EN_VME_IMAGE_ACCESS *iPtr );
```

Parameters:

`*iPtr`	pointer to a data structure containing the parameters necessary to enable Register Access image. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 Register Access with the given parameters. This image maps the Tsi148/Universe II registers onto the VME bus enabling other boards in the system to access them.

Returns:

Zero if successful or an error code upon failure.

8.15 vmekrn_disableRegAccessImage

Declaration:

```
int vmekrn_disableRegAccessImage ( void );
```

Parameters:

None

Description:

Disables the Universe II/Tsi148 Register Access image.

Returns:

Zero if successful or an error code upon failure.

8.16 vmekrn_enableCsrImage

Declaration:

```
int vmekrn_enableCsrImage ( UINT8 imageNumber );
```

Parameters:

`imageNumber`	image number specifies one of thirty-one available CR/CSR image as defined in theVME64 specification.

Description:

Enables the given CR/CSR image, mapping the Universe II Control and Status Register (UCSR) and Tsi148 Combined Register Group onto the VME bus enabling other boards' access. The VME64 specification assigns a total of 16 Mbytes of CR/CSR space for the entire VME bus 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.

8.17 vmekrn_disableCsrImage

Declaration:

```
int vmekrn_disableCsrImage ( UINT8 imageNumber );
```

Parameters:

`imageNumber`	image number specifies one of thirty-one available CR/CSR image windows as defined in the VME64 specification.

Description:

Disables the given CR/CSR image.

Returns:

Zero if successful or an error code upon failure.

8.18 vmekrn_enableLocationMon

Declaration:

```
int vmekrn_enableLocationMon ( EN_VME_IMAGE_ACCESS *iPtr );
```

Parameters:

`*iPtr`	pointer to a data structure containing the parameters necessary to enable the Location monitors. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 Location monitors, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

8.19 vmekrn_disableLocationMon

Declaration:

```
int vmekrn_disableLocationMon ( void );
```

Parameters:

`None`

Description:

Disables the Universe II/Tsi148 Location monitors.

Returns:

Zero if successful or an error code upon failure.

8.20 vmekrn_getStats

Declaration:

```
int vmekrn_getStats( UINT32 type, void *iPtr );
```

Parameters:

`type`	status information type as defined in **`vme_api_en.h`**.
`*iPtr`	pointer to the data structure that will receive the requested status information. The status information data structures are described in the Section 9 (Data Structures for Enhanced API).

Description:

Get the requested status information from VME device driver.

Returns:

Zero if successful or an error code upon failure.

8.21 vmekrn_clearStats

Declaration:

```
int vmekrn_clearStats ( void );
```

Parameters:

None

Description:

Resets the statistical information maintained by the VME device driver.

Returns:

Zero if successful or an error code upon failure.

8.22 vmekrn_enablePciImage

Declaration:

```
int vmekrn_enablePciImage( UINT32 imageNumber,
                           EN_PCI_IMAGE_DATA *iPtr );
```

Parameters:

`imageNumber`	number of the PCI image as defined in **`vme_api_en.h`**.
`*iPtr`	pointer to a data structure containing the parameters necessary to open the image window. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 PCI image window specified by the image number, 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 Section 3.12.5 (Overriding PCI Space Usage) for the default values used for the PCI address pool.

8.23 vmekrn_disablePciImage

Declaration:

```
int vmekrn_disablePciImage( UINT32 imageNumber );
```

Parameters:

`imageNumber` number of the PCI image as defined in **`vme_api_en.h`**

Description:

Disables the Universe II/Tsi148 PCI image window specified by the image number.

Returns:

Zero if successful or an error code upon failure.

8.24 vmekrn_enableVmeImage

Declaration:

```
int vmekrn_enableVmeImage( UINT32 imageNumber,
                           EN_VME_IMAGE_DATA *iPtr );
```

Parameters:

`imageNumber` number of the VME image as defined in **`vme_api_en.h`**.

`*iPtr` pointer to a data structure containing the parameters necessary to open the image window. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Enables the Universe II/Tsi148 VME image window specified by the image number, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

8.25 vmekrn_disableVmeImage

Declaration:

`int vmekrn_disableVmeImage( UINT32 imageNumber );`

Parameters:

`imageNumber` number of the VME image as defined in `vme_api_en.h`.

Description:

Disables the VME image window specified by the image number.

Returns:

Zero if successful or an error code upon failure.

8.26 vmekrn_readImage

Declaration:

```
int vmekrn_readImage( UINT32 imageNumber, ULONG offset, UINT8 *buf,
                      ULONG count );
```

Parameters:

imageNumber	number of the PCI or VME image as defined in **vme_api_en.h**.
offset	relative offset from device start address.
***buf**	buffer to hold data.
count	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.

Returns:

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

8.27 vmekrn_writeImage

Declaration:

```
int vmekrn_writeImage( UINT32 imageNumber, ULONG offset,
                       const UINT8 *buf, ULONG count );
```

Parameters:

imageNumber number of the PCI or VME image as defined in **vme_api_en.h**.

offset relative offset from device start address.

***buf** buffer to hold data.

count 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.

Returns:

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

8.28 vmekrn_allocDmaBuffer

Declaration:

```
int vmekrn_allocDmaBuffer( UINT32 imageNumber, ULONG *size );
```

Parameters:

imageNumber	DMA device number as specified in **vme_api_en.h**.
***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 II/Tsi148 DMA functions.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.29 vmekrn_freeDmaBuffer

Declaration:

`int vmekrn_freeDmaBuffer( UINT32 imageNumber );`

Parameters:

`imageNumber` DMA device number as specified in **`vme_api_en.h`**.

Description:

Frees a previously allocated DMA buffer.

Returns:

Zero if successful or an error code upon failure.

8.30 vmekrn_dmaDirectTransfer

Declaration:

```
int vmekrn_dmaDirectTransfer( UINT32 imageNumber,
                              EN_VME_DIRECT_TXFER *dPtr );
```

Parameters:

imageNumber DMA device number as specified in **vme_api_en.h**.

***dPtr** pointer to a data structure containing the parameters necessary to do a direct DMA transfer. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Initiates a Universe II/Tsi148 direct mode DMA transfer, with the given parameters.

Returns:

Zero if successful or an error code upon failure.

NOTE: The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.31 vmekrn_addDmaCmdPkt

Declaration:

```
int vmekrn_addDmaCmdPkt( UINT32 imageNumber, EN_VME_CMD_DATA *cmdPtr );
```

Parameters:

imageNumber	DMA device number as specified in **vme_api_en.h**.
***cmdPtr**	pointer to a data structure containing the parameters necessary to add a command packet to the linked list. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

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

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.32 vmekrn_clearDmaCmdPkts

Declaration:

```
int vmekrn_clearDmaCmdPkts( UINT32 imageNumber );
```

Parameters:

`imageNumber` DMA device number as specified in **`vme_api_en.h`**.

Description:

Clears the DMA command packet linked list.

Returns:

Zero if successful or an error code upon failure.

8.33 vmekrn_dmaListTransfer

Declaration:

```
int vmekrn_dmaListTransfer( UINT32 imageNumber,
                            EN_VME_TXFER_PARAMS *tPtr );
```

Parameters:

imageNumber DMA device number as specified in **vme_api_en.h**.

***tPtr** pointer to a data structure containing the parameters necessary to do a linked list

DMA transfer. The parameters are described in Section 9 (Data Structures for Enhanced API).

Description:

Initiates a Universe II/Tsi148 linked list mode DMA transfer, with the given parameters. The linked list of command packets must already have been created with calls to the **vmekrn_addDmaCmdPkt** function.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The Universe II device does not support DMA1 – invoking this API with DMA1 for Universe II will return an error code of VME_EPERM.

8.34 vmekrn_readDma

Declaration:

```
int vmekrn_readDma( UINT32 imageNumber, ULONG offset, UINT8 *buf,
                    ULONG count );
```

Parameters:

imageNumber	DMA device number as specified in **vme_api_en.h**.
offset	relative offset from buffer start address.
***buf**	buffer to hold data.
count	amount of data to read (in bytes).

Description:

Reads data from the DMA device buffer that has been **ioremapped** into the kernel memory space.

Returns:

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

8.35 vmekrn_writeDma

Declaration:

```
int vmekrn_writeDma( UINT32 imageNumber, ULONG offset,
                     const UINT8 *buf, ULONG count );
```

Parameters:

imageNumber	DMA device number as specified in **vme_api_en.h**.
offset	relative offset from buffer start address.
***buf**	buffer to hold data.
count	amount of data to write (in bytes).

Description:

Writes data to the DMA device buffer that has been **ioremapped** into the kernel memory space.

Returns:

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

8.36 vmekrn_getDmaBufferAddr

Declaration:

`int vmekrn_getDmaBufferAddr( UINT32 imageNumber, ULONG *bufferAddress );`

Parameter:

`imageNumber`	DMA device number as specified in `vme_api_en.h`.
`*bufferAddress`	Physical address of the DMA buffer allocated.

Description:

Gets the physical address of the DMA buffer allocated by the `vmekrn_allocDmaBuffer` call.

Returns:

Zero if successful or an error code upon failure.

8.37 vmekrn_readVerrInfo

Declaration:

```
int vmekrn_readVerrInfo( ULONG *Address,UINT8 *Direction,
                         UINT8 *AmCode);
```

Parameters:

`*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 and 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 Tsi148 when a VERR interrupt occurs, therefore, the VERR interrupt must first be enabled by calling the **`vmekrn_enableInterrupt`** function.

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.

8.38 vmekrn_setUserAmCodes

Declaration:

`int vmekrn_setUserAmCodes( EN_VME_USER_AM *amPtr );`

Parameters:

`*amPtr`	pointer to a data structure containing the parameters necessary to set the user address modifiers. The parameters are described in Section 9 (Data Structures for Enhanced API).

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.

> **NOTE:** **`vmekrn_setUserAmCodes`** is not supported by the Tsi148 – invoking this API for Tsi148 will return an error code of VME_EPERM.

8.39 vmekrn_readExtInterruptInfo

Declaration:

```
int vmekrn_readExtInterruptInfo( EN_VME_EXTINT_INFO *iPtr );
```

Parameters:

`*iPtr`	pointer to a data structure where interrupt information will be stored. The parameters are described in Section 9 (Data Structures for Enhanced API).

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.

8.40 vmekrn_getBoardCap

Declaration:

```
int vmekrn_getBoardCap( UINT32 *boardFlags );
```

Parameters:

`*boardFlags`	Pointer to a 32 bit quantity specifying the board capabilities. Bit 3 – bit 11 specify the different capabilities supported by the board. bits 0 – 2 – Reserved bit 3 - Byte Swap support bit 4 - Single Cycle Transfers support bit 5 - Block Transfers support bit 6 - Multi Block Transfers support bit 7 - 2eVME Transfers support bit 8 - 2eSST 160 Transfers support bit 9 - 2eSST 267 Transfers support bit 10 - 2eSST 320 Transfers support bit 11 - 2eSST Broadcast support bits 12 – 31 - Reserved

Description:

Get the information of the capabilities supported by the board.

Returns:

Zero if successful or an error code upon failure.

8.41 vmekrn_PciImageAddr

Declaration:

```
int vmekrn_PciImageAddr( UINT32 imageNumber, ULONG *iPtr );
```

Parameters:

`imageNumber`	number of the PCI image as defined in **`vme_api_en.h`**.
`*iptr`	pointer of the PCI Image Address.

Description:

Obtains the I/O remapped PCI Image address of the corresponding image number.

Returns:

Zero if successful or an error code upon failure.

8.42 vmekrn_VmeImageAddr

Declaration:

`int vmekrn_VmeImageAddr( UINT32 imageNumber, ULONG *iPtr );`

Parameters:

`imageNumber`	number of the VME image as defined in `vme_api_en.h`.
`*iptr`	pointer of the VME Image Address.

Description:

Obtains the I/O remapped VME Image address of the corresponding image number.

Returns:

Zero if successful or an error code upon failure.

8.43 vmekrn_registerInterrupt

Declaration:

`int vmekrn_registerInterrupt( EN_VME_INT_DATA *iPtr );`

Parameters:

`*iPtr`	pointer to a data structure containing the parameters necessary to register a callback. The parameters are described in Section 9 (Data Structures for Enhanced API)

Description:

Registers kernel driver's callback function to be invoked upon the registered interrupt occurrence. Up to 32 callback functions can be registered for any one interrupt. It is necessary to remove the registered callback functions at the exit of user's kernel driver.

The registered functions for any one interrupt will be called in the order in which they were registered. The same vector number will be supplied to each callback function via the **`EN_VME_INT_USR_DATA`** structure parameter.

Returns:

Zero if successful or an error code upon failure.

> **NOTE:** The registered callback function has to be interrupt safe.
> The APIs **`vmekrn_readInterruptInfo`** and **`vmekrn_readExtInterruptInfo`** cannot be used to obtain vector numbers inside the callback function if multiple callbacks are registered for a single interrupt, instead use the vector number obtained from **`EN_VME_INT_USR_DATA`**.

8.44 vmekrn_removeInterrupt

Declaration:

```
int vmekrn_removeInterrupt( EN_VME_INT_DATA *iPtr );
```

Parameters:

`*iPtr`	pointer to a data structure containing the parameters necessary to register a callback. The parameters are described in Section 9 (Data Structures for Enhanced API)

Description:

Removes the registered kernel driver's callback function of an interrupt. It is necessary to remove all the registered callback functions when the user's kernel driver exits. The callback functions may be removed in any order.

Returns:

Zero if successful or an error code upon failure.

8.45 vmekrn_getInstanceCount

Declaration:

```
int vmekrn_getInstanceCount( INT32 minorNum, UINT32 *pCount );
```

Parameters:

minorNum	number of the VME image as defined in **vme_api_en.h**
***pCount**	pointer to obtain the count of the number of times the device is left opened.

Description:

Obtains the number of times the VME device is opened and left unclosed.

This API can only be used after opening a corresponding VME device and cannot be used after closing the VME device.

Returns:

Zero if successful or an error code upon failure.

This page is intentionally unused.

9 DATA STRUCTURES FOR USER/KERNEL SPACE API

The following sections describe the data structures used by the VME device driver user/kernel space API functions.

Please note that the structure elements not applicable or supported by either Universe II or Tsi148 must be set to zero.

9.1 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_en.h`** and its parameters are described in Table 9-1 below:

Parameter	Values	Description
`pciAddress`	e.g. 0xC0000000	Lower 32 bits of 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.
`pciAddressUpper`	0	Upper 32 bits of Base PCI address **NOTE:** Only 32 bit addressing is supported.
`vmeAddress`	e.g. 0x10000	Lower 32 bits of Base VME address. The address must be aligned in accordance with image resolution.
`vmeAddressUpper`	e.g. 0x00000001	Upper 32 bits of base VME address.
`size`	e.g. 4096 or 65536	Lower 32 bits of Image size. The size should be set in accordance with image resolution.
`sizeUpper`		Upper 32 bits of image size. **NOTE:** Only 32 bit addressing is supported, hence this field has not effect
`readPrefetch`	0 = disable 1 = enable	Memory read prefetch control. **NOTE:** This field has no effect if using the Universe II.
`prefetchSize`	0 = 2CL, 1 = 4CL 2 = 8CL, 3 = 16CL	Memory prefetch size in cache lines. **NOTE:** This field has no effect if using the Universe II.
`postedWrites`	0 = disable 1 = enable	Enable or disable posted writes on Universe II. **NOTE:** This field has no effect if using the Tsi148.

Table 9-1 PCI Image Data

(continued on next page)

Parameter	Values	Description
`dataWidth`	0 = 8 bit 1 = 16 bit 2 = 32 bit 3 = 64 bit	Maximum data width. **NOTE:** The Tsi148 does not support 8 and 64 bit **`dataWidth`** for SCT and BLT. **NOTE:** The Tsi148 uses a **`dataWidth`** of 64 bit when configured for MBLT, 2eVME and 2eSST
`addrSpace`	0 = A16, 1 = A24 2 = A32, 4 = A64 5 = CR/CSR 6 = Universe User 1 7 = Universe User 2 8 = Tsi148 User1 9 = Tsi148 User2 10 = Tsi148 User 3 11 = Tsi148 User 4	Address space modifier. **NOTE:** Values 6 and 7 can be selected only if using the Universe II. **NOTE:** Values 4, 8, 9, 10 and 11 can be selected only if using the Tsi148.
`sstMode`	0 = SST160 1 = SST267 2 = SST320	2eSST transfer rate. **NOTE:** This field has no effect if using the Universe II.
`type`	0 = data 1 = program	Program/Data AM code.
`mode`	0 = non-privileged 1 = supervisor	Supervisor/User AM code.
`vmeCycle`	0 = SCT 1 = BLT 2 =MBLT 3 = 2eVME 4 = 2eSST 5 = 2eSSTB	VME bus cycle type. **NOTE:** SCT and BLT can be configured with the Universe II.
`vton`	0	Not used.
`vtoff`	0	Not used.
`sstbSel`	e.g. 0x000000ff	2eSSTB broadcast select. **NOTE:** This field has no effect if using the Universe II.
`pciBusSpace`	0 = PCI memory space 1 = PCI I/O space	PCI bus space memory space. **NOTE:** This field has no effect if using the Tsi148.
`ioremap`	0 = no 1 = yes	Whether to **`ioremap`** image. **NOTE:** PCI images can be memory mapped to user space with **`vme_mmap`**. **NOTE:** If used with kernel space API, this field must be set to 1

Table 9-1 PCI Image Data (continued)

9.2 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_en.h`** and its parameters are described in Table 9-2 below:

Parameter	Values	Description
`vmeAddress`	e.g. 0x10000	Lower 32 bits of Base VME address. The address must be aligned in accordance with image resolution.
`vmeAddressUpper`	e.g. 0x00000001	Upper 32 bits of base VME address. **NOTE:** This field has no effect if using the Universe II.
`size`	e.g. 4096 or 65536	Image size. The size should be set in accordance with image resolution.
`sizeUpper`		Upper 32 bits of image size. **NOTE:** This field has no effect if using the Universe II.
`postedWrites`	0 = disable 1 = enable	Enable or disable posted writes on Universe II. **NOTE:** This field has no effect if using the Tsi148.
`prefetchRead`	0 = disable 1 = enable	Enable or disable prefetch read on Universe II. **NOTE:** This field has no effect if using the Tsi148.
`threshold`	0 = prefetch on FIFO full empty 1 = prefetch on FIFO half empty	Threshold for prefetch. **NOTE:** This field has no effect if using the Universe II.
`virtualFifoSize`	0 = 64 1 = 128 2 = 256 3 = 512	FIFO Size. **NOTE:** This field has no effect if using the Universe II.
`vmeCycle`	0 = SCT 1 = BLT 2 = MBLT 3 = 2eVME 4 = 2eSST 5 = 2eSSTB	VME bus cycle type. **NOTE:** This field has no effect if using the Universe II.
`sstMode`	0 = SST160 1 = SST267 2 = SST320	2eSST transfer rate. **NOTE:** This field has no effect if using the Universe II.
`type`	1 = data 2 = program 3 = both	Program/Data AM code.
`mode`	1 = non-privileged 2 = supervisor 3 = both	Supervisor/User AM code.

Table 9-2 VME Image Data

(continued on next page)

`addrSpace`	0 = A16 1 = A24 2 = A32 4 = A64 6 = User 1 7 = User 2	Address space modifier. **NOTE:** Values 6 and 7 can be selected only if using the Universe II. **NOTE:** Value 4 can only be selected if using the Tsi148.
`pciBusSpace`	0 = PCI memory space 1 = PCI I/O space	PCI bus space memory space. **NOTE:** This field has no effect if using the Tsi148.
`pciBusLock`	0 = disable 1 = enable	Enable or disable PCI bus lock (for read modify write) on Universe II. **NOTE:** This field has no effect if using the Tsi148.
`ioremap`	0 = no 1 = yes	Whether to **`ioremap`** image. **NOTE:** PCI images can be memory mapped to user space with **`vme_mmap`**. **NOTE:** If used with kernel space API, this field must be set to 1

Table 9-2 VME Image Data (Continued)

9.3 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_en.h`** and its parameters are described in Table 9-3 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 9-3 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_en.h`** and its parameters are described in Table 9-4 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 9-4 Extended VME Interrupt Data

9.4 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_en.h`** and its parameters are described in Table 9-5 below:

Parameter	Values	Description
`intNum`	bit 0 = Reserved 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 9-5 Wait Interrupt Data

9.5 User Interrupt Data

Before calling the **vmekrn_registerInterrupt** function, a user interrupt data structure must be created and filled out.

The data structure **EN_VME_INT_DATA** is defined in **vme_api_en.h** and its parameters are described in Table 9-6 below:

Parameter	Values	Description
intNum	0 to 25 for Tsi148 0 to 23 for Universe	VME interrupt number to register user handler to. To be filled by user.
userInt	Valid function pointer	User handler to be invoked at the occurrence of interrupt. To be filled by user.
usrPtr in EN_VME_INT_USR_DATA	User passed pointer	User defined pointer to be passed to user's handler. To be filled by user.

Table 9-6 User Interrupt Register Data

The user handler will be invoked from Linux VME Enhanced driver with a **void*** pointer. Typecast it to a pointer to a data structure of type **EN_VME_INT_USR_DATA** to access the data described in Table 9-7:

Parameter	Values	Description
intNum	0 to 25 for Tsi148 0 to 23 for Universe	VME interrupt number for user reference.
intVec	Vector data	Status ID for user reference.
usrPtr	User passed pointer	User defined pointer passed to user.

Table 9-7 User Interrupt Received Data

When calling the **vmekrn_removeInterrupt** function, the user must fill out the **intNum** and **userInt** fields of the **EN_VME_INT_DATA** structure and pass this structure to the function. These fields are described in Table 9-8:

Parameter	Values	Description
intNum	0 to 25 for Tsi148 0 to 23 for universe	VME interrupt number to remove user handler from. To be filled by user.
userInt	Valid function pointer	User handler to be removed from interrupt. To be filled by user.

Table 9-8 User Interrupt Remove Data

9.6 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_en.h`** and its parameters are described in Table 9-9:

Parameter	Values	Description
`vmeAddress`	e.g. 0x10000	Lower 32 bits of Base VME address of the image. **NOTE:** The Universe II/Tsi148 devices fix the size of these images.
`vmeAddressUpper`	e.g. 0	Upper 32 bits of base VME address.
`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, 4 = A64	Address space modifier. **NOTE:** A64 address modifier is not supported by the Universe II.

Table 9-9 Image Access Data

9.7 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_en.h`** and its parameters are described in Table 9-10:

Parameter	Values	Description
`direction`	0 = VME to PCI bus (read) 1 = PCI to VME bus (write)	Direction of transfer.
`vmeAddress`	e.g. 0x10000	Lower 32 bits of VME address. The address must be 8 byte aligned with the offset.
`VmeAddressUpper`	e.g. 0	Upper 32 bits of VME address. **NOTE:** This field has no effect if using the Universe II.
`offset`	e.g. 0x1000	Offset from start of DMA buffer. See **`vmeAddress`** for alignment.
`size`	e.g. 1024	Size to transfer.
`txfer`	See Section 9.8 below.	VME Transfer Parameter data structure.
`access`	See Section 9.9 below.	VME Access Parameter data structure.

Table 9-10 Direct DMA Transfer Data

9.8 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_en.h`** and its parameters are described in Table 9-11:

Parameter	Values	Description
`direction`	0 = VME to PCI bus (read) 1 = PCI to VME bus (write)	Direction of transfer.
`vmeAddress`	e.g. 0x10000	Lower 32 bits of VME address. The address must be 8 byte aligned with the offset.
`vmeAddressUpper`	e.g. 0	Upper 32 bits of VME address. **NOTE:** This field has no effect if using the Universe II.
`offset`	e.g. 0x1000	Offset from start of DMA buffer. See **`vmeAddress`** for alignment.
`size`	e.g. 1024	Size to transfer.
`access`	See Section 9.9 below.	VME Access Parameter data structure.

Table 9-11 Command Packet Data

9.9 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_en.h`** and its parameters are described in Table 9-12:

Parameter	Values	Description
`timeout`	e.g. 200 jiffies » 2 seconds	Maximum time allowed for DMA transfer to complete. Parameter given in system ticks (jiffies)
`vmeBlkSize`	0 = 32, 1 = 64 2 = 128, 3 = 256 4 = 512, 5 = 1024 6 = 2048, 7 = 4096	VME Bus block size in bytes. **NOTE:** This field has no effect if using the Universe II.
`vmeBackOffTimer`	0 = 0, 1 = 1 2 = 2, 3 = 4 4 = 8, 5 = 16 6 = 32, 7 = 64	VME Bus Back off timer in microseconds. **NOTE:** This field has no effect if using the Universe II.
`pciBlkSize`	0 = 32, 1 = 64 2 = 128, 3 = 256 4 = 512, 5 = 1024 6 = 2048, 7 = 4096	PCI Bus block size in bytes. **NOTE:** This field has no effect if using the Universe II.
`pciBackOffTimer`	0 = 0, 1 = 1 2 = 2, 3 = 4 4 = 8, 5 = 16 6 = 32, 7 = 64	PCI Bus Back off timer in microseconds. **NOTE:** This field has no effect if using the Universe II.
`vton`	0	Not used.
`vtoff`	0	Not used.
`ownership`	Bits 0 – 3 used for VOFF: 0 = 0, 1 = 16 2 = 32, 3 = 64 4 = 128, 5 = 256 6 = 512, 7 = 1024. Bits 4 – 7 used for VON: 0 = Until done 1 = 256, 2 = 512 3 = 1024, 4 = 2048 5 = 4096, 6 = 8192 7 = 16348	VME bus On/Off counters (VON/VOFF). **NOTE:** This field has no effect if using the Tsi148.

Table 9-12 VME Transfer Parameters

9.10 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_en.h`** and its parameters are described in Table 9-13:

Parameter	Values	Description
`dataWidth`	0 = 8 bit 1 = 16 bit 2 = 32 bit 3 = 64 bit	Maximum data width. **NOTE:** The Tsi148 does not support 8 bit or 64 bit **`dataWidth`** for SCT or BLT. All the protocols above BLT use 64 bit by default.
`addrSpace`	0 = A16, 1 = A24 2 = A32, 4 = A64 5 = CR/CSR 6 = Universe User 1 7 = Universe User 2 8 = Tsi148 User1 9 = Tsi148 User2 10 = Tsi148 User 3 11 = Tsi148 User 4	Address space modifier. **NOTE:** Values 6 and 7 can be selected only if using the Universe II. **NOTE:** Values 4, 5, 8, 9, 10 and 11 can be selected only if using the Tsi148.
`sstbSel`	e.g. 0x00003	2eSST Broadcast select. **NOTE:** This field has no effect if using the Universe II.
`type`	0 = data 1 = program	Program/Data AM code.
`mode`	0 = non-privileged 1 = supervisor	Supervisor/User AM code.
`vmeCycle`	0 = SCT, 1= BLT 2 =MBLT, 3= 2eVME 4=2eSST, 5 = 2eSSTB	VME bus cycle type. **NOTE:** If using the Universe II it is only possible to select SCT or BLT values for this field.
`sstMode`	0=SST160 1=SST267 2=SST320	2eSST Transfer rate. **NOTE:** This field has no effect if using the Universe II.

Table 9-13 VME Access Parameters

9.11 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_en.h`** and its parameters are described in Table 9-14:

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

Table 9-14 User Address Modifier Data

NOTE: **`vme_setUserAmCodes`** is not supported by the Tsi148.

9.12 Control Status Data

To request control status information with **`vme_getStats`** function, set the type parameter to **`VME_STATUS_CTRL`** and pass a pointer to the union of the type **`EN_VME_DRIVER_STAT`** defined in the **`vme_api_en.h`** file. The driver will assign values to the members of structure **`EN_CTL_STATUS_DATA`** that is a member of the **`EN_VME_DRIVER_STAT`** union. The parameters of the structure **`EN_CTL_STATUS_DATA`** are described in Table 9-12:

Parameter	Values	Description
`version`		Driver version information
`brdName`		Board name
`devId`		VME Bridge device ID, Universe II or Tsi148
`regBase`		Memory base address of VME bridge registers

Table 9-15 Control Status Data

9.13 PCI Image Status Data

To request PCI image status information with **`vme_getStats`** function, set the type parameter to **`VME_STATUS_LSIx`** and pass a pointer to the union of the type **`EN_VME_DRIVER_STAT`** defined in the **`vme_api_en.h`** file. The driver will assign values to the members of structure **`EN_PCI_STATUS_DATA`** that is a member of the **`EN_VME_DRIVER_STAT`** union. The parameters of the structure **`EN_PCI_STATUS_DATA`** are described in Table 9-16:

Parameter	Values	Description	
devId		VME Bridge device ID, Universe II or Tsi148	
readCount		vme_read count	
writeCount		vme_write count	
errorCount		vme_read/write error count	
		Device register values:	
		Universe II	Tsi148
devReg1		LSIx_CTL	OTSAU
devReg2		LSIx_BS	OTSAL
devReg3		Not used	OTEAU
devReg4		LSIx_BD	OTEAL
devReg5		Not used	OTOFU
devReg6		LSIx_TO	OTOFL
devReg7		Not used	OTBS
devReg8		Not used	OTAT

Table 9-16 PCI Image Status Data

9.14 VME Image Status Data

To request VME image status information with **`vme_getStats`** function, set the type parameter to **`VME_STATUS_VSIx`** and pass a pointer to the union of the type **`EN_VME_DRIVER_STAT`** defined in the **`vme_api_en.h`** file. The driver will assign values to the members of structure **`EN_VME_STATUS_DATA`** that is a member of the **`EN_VME_DRIVER_STAT`** union. The parameters of the structure **`EN_VME_STATUS_DATA`** are described in Table 9-17:

Parameter	Values	Description	
`devId`		VME Bridge device ID, Universe II or Tsi148	
`readCount`		vme_read count	
`writeCount`		vme_write count	
`errorCount`		vme_read/write error count	
		Device register values:	
		Universe II	Tsi148
`devReg1`		VSIx_CTL	ITSAU
`devReg2`		VSIx_BS	ITSAL
`devReg3`		Not used	ITEAU
`devReg4`		VSIx_BD	ITEAL
`devReg5`		Not used	ITOFU
`devReg6`		VSIx_TO	ITOFL
`devReg7`		Not used	ITAT

Table 9-17 VME Image Status Data

9.15 DMA Status Data

To request DMA status information with **vme_getStats** function, set the type parameter to **VME_STATUS_DMAx** and pass a pointer to the union of the type **EN_VME_DRIVER_STAT** defined in the **vme_api_en.h** file. The driver will assign values to the members of structure **EN_DMA_STATUS_DATA** that is a member of the **EN_VME_DRIVER_STAT** union. The parameters of the structure **EN_DMA_STATUS_DATA** are described in Table 9-18:

Parameter	Values	Description	
devId		VME Bridge device ID,Universe II or Tsi148	
readCount		vme_read count	
writeCount		vme_write count	
errorCount		vme_read/write error count	
txferCount		DMA transfers count	
txferErrors		Number of DMA transfer errors	
timeoutCount		Number of DMA timeouts	
cmdPktCount		Command packet count	
cmdPktBytes		Number of bytes to transfer in linked list	
		Device register values:	
		Universe II	Tsi148
devReg1		DCTL	DCTL
devReg2		DTBC	DSAU
devReg3		Not used	DSAL
devReg4		DLA	DDAU
devReg5		DVA	DDAL
devReg6		DGCS	DSAT
devReg7		Not used	DDAT
devReg8		DCPP	DNLAU
devReg9		Not used	DNLAL
devReg10		Not used	DCNT
devReg11		Not used	DDBS

Table 9-18 DMA Status Data

9.16 Interrupt Status Data

To request interrupt image status information with **vme_getStats** function, set the type parameter to **VME_STATUS_INT** and pass a pointer to the union of the type **EN_VME_DRIVER_STAT** defined in the **vme_api_en.h** file. The driver will assign values to the members of structure **EN_INT_STATUS_DATA** that is a member of the **EN_VME_DRIVER_STAT** union. The parameters of the structure **EN_INT_STATUS_DATA** are described in Table 9-19:

Parameter	Values	Description
devId		VME Bridge device ID, Universe II or Tsi148
intCounter[26]		Interrupt counters
totalIntCount		VME bridge interrupt count
otherIntCount		Other shared interrupt count
mode		Interrupt mode

Table 9-19 Interrupt Status Data

10 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`** in the **`/usr/include`** directory. 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 is intentionally unused.

11 PROGRAMMING EXAMPLES USING USER SPACE API

A number of example programs, illustrating the use of the user space API functions are included as part of the VME Board Support Package. These include the following:

- Example 1 - Generating A VME Bus Interrupt From Software
- Example 2 - Reading VME Bus Interrupt Information
- Example 3 - Wait For An Interrupt
- Example 4 - Using a PCI Image Window
- Example 5 - Using a VME Image Window
- Example 6 - Using Direct Mode DMA
- Example 7 - Using Linked List Mode DMA
- Example 8 - Getting device statistics
- Example 9 - Obtain the Instance count of opened VME devices

The source code of the example programs are included as part of a tar ball which is installed at the location **`/usr/local/linuxvmeen_examples`** on the Board Support Package CD.

This page is intentionally unused.

12 PROGRAMMING EXAMPLES USING KERNEL SPACE API

An example driver illustrating the use of the kernel space API functions is included as part of the VME Board Support Package. The example driver includes the functions demonstrating the following operations:

- Using PCI Image Window
- Using VME Image Window
- Using direct mode DMA transfer
- Using link-list mode DMA transfer
- Wait for an Interrupt
- Register kernel driver interrupt handler, obtain interrupt count and remove interrupt handler for VME interrupts
- Obtain the Instance count of opened VME devices

The source code for the example driver is included as part of a tar ball which is installed at the location **`/usr/local/linuxvmeen_kapi_example`** on the Board Support Package CD. Please refer to driver source and the associated **`readme`** file for instructions on how to use the example driver.

This page is intentionally unused.