MOTOROLA COMPUTER GROUP

VME DRIVER FOR LINUX

USER GUIDE

VME driver

INTRODUCTION

The VME driver provides a programmatic interface which application programs may use to configure and control board access to the VME bus. The VME driver provides the ability for an application to control the inbound and outbound VME address maps, generate DMA based transfers, cause and receive VME interrupts, and utilize any special purpose features of the VME controller (e.g. RMW transfers).

The VME driver is written for the main purpose of facilitating testing of the boards VME interface. Test applications may use the driver to configure boards within the chassis into the desired test configuration and generate whatever VME transactions that are needed for the test. However, the VME driver will also be useful in any situation where applications need to access VME bus resources.

The VME driver is written as a Linux module and is accessed via device nodes. It supports both the Tundra Universe and MCG Tempe VME bridges.

VME DRIVER FEATURES

The VME driver provides the following features to the application:

The ability to configure the outbound VME registers to create (or delete) address ranges in CPU space within which CPU accesses are translated into VME bus accesses. These are called ‘outbound windows’. The driver will support configuration of the outbound windows with any VME attributes (e.g. A16, A24, etc) which are supported by the board. The application may access the outbound windows through device nodes using the standard libc open(), close(), read (), & write() calls. (NOTE: one of the outbound decoders is required for internal driver use and will not be available to the application).

The ability to configure the inbound VME registers to create (or delete) address ranges in VME space within which VME accesses are translated into CPU bus accesses. These are called ‘inbound windows’. The driver will support configuration of the inbound windows with any configuration (e.g.. A16, A24, etc) which is supported by the board. (NOTE: the CR/CSR inbound decoder is required for internal driver use and will not be available to the application).

The ability to cause a transfer of data using the DMA feature of the VME controller. Any combination of source and destination addresses & attributes supported by the board are available to the application. For support of DMA performance testing, the VME driver will measure the time that elapses during the DMA transaction and make that measurement available to the application. The driver provides access to DMA via device nodes using the standard libc open(), close(), & ioctl() calls.

The ability to cause RMW (read modify write) transactions to occur on the VME bus. Any RMW attributes (e.g. address, data) supported by the boards VME bus controller are available to

the application. The driver provides access to RMW via a device node using the standard libc open(), close(), & ioctl() calls.

The ability to configure the VME bus location monitor to any attributes supported by the board. The driver will allow the application to determine if the location monitor has been triggered.

The ability to cause a VME bus interrupt at any VIRQ level and vector supported by the board.

The ability to determine whether or not a VME bus interrupt has occurred and if so, at what level and which vector was received during the VME IACK cycle at that level.

The ability to configure the VME bus system controller attributes (e.g. bus timeout value, arbitration level, etc) to any setting supported by the board. This feature is only effective if the board is the VME system controller.

The ability to directly access the VME bus controller registers. Normal user applications do not need this feature but it is provided for testing purposes.

The ability to determine information about each VME board present in the VME chassis. Each boards slot number, type of VME controller and whether it is system controller or not will be provided to the application by the driver.

The driver detects VME errors and provide a means for the user application to receive notification of VME errors.

The VME driver supports both the Tundra Universe II VME chip and the MCG Tempe VME chip. The driver will provide a consistent, device independent interface to these two VME controllers. If the application attempts to use a VME feature which is not present on the board (e.g. 2eSST on a Universe II board), the driver will return an error indication to the application.

VME DEVICE NODES

The VME driver services are provided to the user application via several device nodes. The major device number for all of the VME devices is 221. Major device 221 has been defined as the standard major device number for VME devices and so, should not conflict with any existing Linux device. Minor device numbers are used to distinguish the various VME device nodes. Device node usage is as follows:

Device Name	Major	Minor	Function
/dev/vme_m0	221	0	VME outbound window #0
/dev/vme_m1	221	1	VME outbound window #1
/dev/vme_m2	221	2	VME outbound window #2
/dev/vme_m3	221	3	VME outbound window #3
/dev/vme_m4	221	4	VME outbound window #4

/dev/vme_m5	221	5	VME outbound window #5
/dev/vme_m6	221	6	VME outbound window #6
/dev/vme_m7	221	7	VME outbound window #7 (note outbound window 7 is reserved for driver use).
/dev/vme_dma0	221	16	VME DMA #0
/dev/vme_dma1	221	17	VME DMA #1
/dev/vme_ctl	221	32	VME controller miscellaneous functions (e.g. interrupts).
/dev/vme_regs	221	33	Provides direct access to VME controller registers. (use with caution).
/dev/vme_rmw0	221	34	Provides access to the VME controller RMW function.
/dev/vme_lm0	221	35	Provides access to the VME controller location monitor function.

Details of the interface to each of the above nodes is provided in the application interface section of this document.

APPLICATION INTERFACE

The application access driver services by using open() to open one or more of the above device nodes and then using the standard close(), read(), write(), and ioctl() calls to perform various functions.. All VME device nodes support the open() & close() routines. Depending on the device node function, a VME device node may or may not support the other calls.

open(), close(), read(), write(), subroutine calls on VME nodes adhere to the normal semantics of such calls and are not specifically described further in this document unless there are exceptions to the normal semantics or restrictions.

Many of the VME driver features are available only via ioctl() calls. ioctl() calls are of the normal form ioctl(int fd, int cmd, void *arg); where cmd and arg are device node specific. The usage of cmd and arg in ioctl() calls is described for each of the VME device nodes.

Some VME devices cannot support open() by more than one application at a time. Once opened, these devices must be closed before they will be available to any other application. The following describes the usage of each of the VME device nodes in detail.

The data items and structures described in the following are defined in the include file ‘vmedrv.h’.

/dev/vme_m[0-7]

/dev/vme_m[0-7] nodes provide the application the ability to access addresses on the VME bus in a programmed io mode. An application opens a /dev/vme_m[0-7] node, configures it with ioctl, and then accesses the VME bus through it with read(), or write().

VME space must be accessed with loads and stores that are consistent with the VME space attributes (e.g. D16 space must be accessed as 16 bit wide loads & stores). The driver accesses VME space with the required data width during read() and write().

/dev/vme_m[0-7] nodes supports open(), close(),read(), write() and ioctl(). Each device node coorelates directly to a VME outbound window (e.g. /dev/vme_m3 correlates to VME outbound decoder 3). /dev/vme_m[0-7] nodes are exclusive use and may be opened by only one application at a time.

The highest numbered outbound decoder (#7 for Tempe), is reserved for internal use by the driver and should not be accessed by the application.

Both Universe II and Tempe VME controllers have 8 outbound decoders and so device nodes /dev/vme_m0 through /dev/vme_m6 are accessible to the application. However, the VME driver may be ported to VME controllers with more or less than 8 outbound decoders. To ease porting to those controllers, applications which use the VME driver should be written to be flexible in the number of outbound windows that it supports.

VME outbound windows are mapped through PCI memory space. PCI memory space, although large, is not unlimited. Attempts to simultaneously map multiple large VME outbound windows will fail if sufficient PCI memory space is not available to the driver..

Ioctl() cmd values and corresponding arg which are supported by this node are:

VME_IOCTL_SET_OUTBOUND which take an arg pointer to the following structure and initializes the VME outbound window to the attributes described by that structure.

See structure definition in appended file vmedrv.h

VME_IOCTL_GET_OUTBOUND which takes an arg pointer to the above structure and initializes it with the current attributes of the VME outbound window.

/dev/vme_dma*

/dev/vme_dma[0-1] nodes provide the application the ability to access the DMA feature of the VME controller. An application opens a /dev/vme_dma* node, configures it with ioctl, and then causes DMA transactions on the VME bus through it with ioctl().

/dev/vme_dma nodes supports open(), close(), and ioctl(). They do not support read(), write (), mmap() or munmap(). Each device node correlates directly to a VME DMA channel (e.g. / dev/vme_dma1 correlates to VME DMA channel #1).

/dev/vme_dma nodes are not exclusive use. They may be open simultaneously by several processes. However, a DMA channel may process only one request at a time. If multiple processes are requesting a DMA service at the same time, the requests are processed sequentially and one process may have to wait for another (or several) DMAs to complete before its request is processed.

The VME driver time stamps a DMA request with both the time the DMA operation was started by the DMA hardware and the time the DMA stopped (i.e. the DMA interrupt occurred). The start and stop time are in microseconds. Accuracy of the stop time may be affected by interrupt latency in a heavily loaded system. Applications which need to accurately measure DMA throughput should do so in a lightly loaded system environment.

All /dev/vme_dma ioctl() calls expect that arg points to the following structure.

See structure definition in appended file vmedrv.h

ioctl() cmd values which are supported by this node are:

VME_IOCTL_START_DMA which performs a VME DMA transaction with the attributes described by the structure.

/dev/vme_ctl

The /dev/vme_ctl node provides the application the ability to configure or query various attributes of the VME bus.

The /dev/vme_ctl node supports open(), close(), and ioctl(). It does not support read(), write (), mmap() or munmap().

The dev/vme_ctl node is not exclusive use. It may be open simultaneously by several processes.

Some of the functions available through this device node are available only to an application running on the board which is VME system controller. Other functions require specific poweron configuration of other boards in the chassis. Any such restrictions are documented in the following descriptions.

Ioctl() cmd values and corresponding arg which are supported by this node are:

VME_IOCTL_GET_SLOT_VME_INFO which takes an arg pointer to the following structure and fills it with information about the board (if any) that is present in the specified slot. The application may obtain information about the board it is currently running on by specifying a slot number of 0.

See structure definition in appended file vmedrv.h

VME_IOCTL_SET_REQUESTOR which takes an arg pointer to the following structure and configures the board the application is running on to arbitrate for the VME bus at the request level specified in the structure.

See structure definition in appended file vmedrv.h

VME_IOCTL_GET_REQUESTOR which takes an arg pointer to the following structure and initializes it with the current attributes of the VME arbiter.

See structure definition in appended file vmedrv.h

VME_IOCTL_SET_CONTROLLER which takes an arg pointer to the following structure and configures the system controller to the attributes specified by the structure. This cmd is only effective if the application is running on the board which is the VME system controller.

See structure definition in appended file vmedrv.h

VME_IOCTL_GET_CONTROLLER which takes an arg pointer to the following structure and initializes it with the current system controller attributes. This cmd is only effective if the application is running on the board which is the VME system controller.

See structure definition in appended file vmedrv.h

VME_IOCTL_GENERATE_IRQ which takes an arg pointer to the following structure and causes a VME interrupt request at the level and vector to be generated on the VME bus.

See structure definition in appended file vmedrv.h

VME_IOCTL_GET_IRQ_STATUS which takes an arg pointer to the following structure and determines if an interrupt at the specified level has been logged. The driver provides a handler for VME bus interrupts which logs the level and vector from each VME interrupt. Applications may query the interrupt log to determine if a particular VME bus interrupt has occurred.

See structure definition in appended file vmedrv.h

VME_IOCTL_CLR_IRQ_STATUS which takes an arg pointer to the following structure and removes the specified interrupt(s) from the interrupt log.

See structure definition in appended file vmedrv.h

VME_IOCTL_SET_INBOUND which takes an arg pointer to the following structure and initializes the VME inbound window to the attributes described by that structure.

See structure definition in appended file vmedrv.h

VME_IOCTL_GET_INBOUND which takes an arg pointer to the above structure and initializes it with the current attributes of the VME inbound window.

/dev/vme_regs

The /dev/vme_regs node provides the application the ability to directly access the VME controller registers.

The /dev/vme_regs node supports open(), close(), read(), and write().

The dev/vme_regs node is not exclusive use. It may be open simultaneously by several processes.

User applications normally do not (and should not) directly access the VME controller registers. This device node is intended for use by test applications which need to directly access the controller registers. Applications using this node must avoid modifying registers which are used by the driver or reading any register for which the read causes a side effect (e.g. a read that clears status bits).

The driver expects will 'byte swizzle' register accesses as needed so that the register data read or written by the application is in big endian format.

/dev/vme_rmw0

The /dev/vme_rmw0 node provide the application the ability to generate a RMW (read modify write) cycle on the VME bus.

The /dev/vme_rmw0 node supports open(), close(), and ioctl(). It does not support read(), write(), mmap() or munmap().

The /dev/vme_rmw0 node is exclusive use. It may be open by only one process at a time.

Ioctl() cmd values and corresponding arg which are supported by this node are:

VME_IOCTL_DO_RMW which takes an arg pointer to the following structure and causes a RMW transaction to occur on the VME bus with the attributes described by the structure:

See structure definition in appended file vmedrv.h

/dev/vme_lm0

The /dev/vme_lm0 node provide the application the ability to utilize the location monitor feature of the VME controller.

The /dev/vme_lm0 node supports open(), close(), and ioctl(). It does not support read(), write(), mmap() or munmap().

The /dev/vme_lm0 node is exclusive use. It may be open by only one process at a time.

Ioctl() cmd values and corresponding arg which are supported by this node are:

VME_IOCTL_SETUP_LM which takes an arg pointer to the following structure and initializes the location monitor circuitry of the VME controller to the attributes described by the structure:

See structure definition in appended file vmedrv.h

VME_IOCTL_WAIT_LM which takes an arg pointer to the following structure and waits (if necessary) for the location monitor to detect an access:

See structure definition in appended file vmedrv.h

DRIVER VME BUS USAGE

The driver uses some of the VME bus resources to facilitate its own operation. To allow for the driver to interoperate with other VME boards in the chassis, users must be able to configure the VME boards to avoid conflict with the VME driver. Additionally the VME driver provides some data structures that are specifically intended to facilitate testing of the VME bus interface. For these reasons, the VME resources and public data structures associated with the driver are documented here.

VME Initialization

The driver initializes the VME controller resources as follows.:

All VME outbound decoders are disabled with the exception of the last (normally #7) outbound decoder. The last decoder is reserved for driver use. Normally, decoder #7 will provide an outbound window into CS/CSR space which the driver uses for interboard communication. However, user applications should not depend on the outbound mapping that the driver sets up for its own use.

The driver allocates a kernel buffer each inbound decoder. The largest possible kernel buffer is allocated. This will be at least 128K bytes. The Linux kernel may need modification/recompilation to provide for inbound buffers larger than 128K bytes. Each of the inbound windows is mapped into A32/D32 space at an address determined by the boards slot number. The address is calculated as 0x80000000+(0x20000000*slotnum)+ (0x400000*inboundnum). Thus, in total, each board responds to a 32 Megabyte chunk of VME address space. The driver attempts to allocate as large a buffer as possible (up to 4Meg) for each inbound window. The actual size of the buffer allocated to each inbound window is provided to other boards via the interboard data structure which is available to other boards via CS/CSR space.

The Tempe & Universe provide an inbound decoder specifically for CSR space. This decoder is initialized by hardware to map the board into CSR space at the address 0x80000*slotnum (each board gets a 512K window). The driver creates an interboard data structure and cause it to appear at the base address of its CSR window. The inter board data structure contains the following data:

Validity flags which indicate the driver is active on this board and the inter board data structure has been initialized and is ready for use.

The revision of the driver running on the board and the type and revision of VME controller present on the board.

Semaphore locations that remote boards can use to coordinate their access to this board during testing. Only one (or a few) boards at a time can be using this boards VME inbound windows for testing. The semaphore area is provided to allow applications a means to share access to this board. Additionally, the size of kernel memory that has been allocated to each inbound buffer is provided in the inter board data structure.

An area that has been initialized with pre-defined test patterns which remote boards may read and verify. This area is intended to be used as read only test data by applications. However the driver has no way of protecting this area from writes so applications are on their honor to comply with the read only restriction.

A small scratch area (1k) for each board which can be used by a remote board for any purpose it chooses. Typically, this area would be used by a remote for a quick and easy test of the connections to a remote board.

An area that is for private driver use. This area provides the fifos and other data structures needed by the driver to support inter-board communication.

All other inbound decoders are disabled by the driver

After Initialization

After initialization, the inbound and outbound decoders may be reconfigured as needed by test or other application software to the needs of that particular application. The driver provides maximum flexibility to the application. It does not (and in some cases cannot) detect that a requested configuration causes conflicts on the VME bus (e.g. two boards responding to the same VME address). It is up to the applications to avoid configurations that cause VME bus conflicts.

Other notes:

The driver detects the presence of a board in the chassis by probing CSR space at the address appropriate for the slot number.

To facilitate testing of VMEbus transactions to/from VMEChip2 based boards, the user may manually configure the VMEchip2 inbound windows to map 32Meg of RAM at the address (like above). Applications running on remote boards may then assume that a slot that contains a VMEchip2 based board provides a 32Meg window for testing at the expected address. Likewise, access to Tempe or Universe based boards can be verified without running a copy of

the driver if the user manually configures 32 Meg of the boards RAM to appear on the VME bus at the appropriate address as described above.

The initialization provided by the driver is intended to facilitate basic connectivity testing of boards within a chassis. Each board provides buffers that a remote board can access via the VME bus and known data patterns in expected locations that can be read and verified. Applications may reconfigure the inbound & outbound decoders as needed to do more extensive testing.

SAMPLE APPLICATION CODE

A test program which reads and verifies several VME chip registers:

```
/* VME Linux driver test/demo code */

/*

 * Include files

 */

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <sys/ioctl.h>

#include "../vmedrv.h"


vmeInfoCfg_t myVmeInfo;

int
```

```
getMyInfo()
{
	int	fd, status;
	fd = open("/dev/vme_ctl", 0);
	if (fd < 0) {
		return(1);
	}

	memset(&myVmeInfo, 0, sizeof(myVmeInfo));
	status = ioctl(fd, VME_IOCTL_GET_SLOT_VME_INFO, &myVmeInfo);
	if (status < 0) {
		return(1);
	}
	close(fd);
	return(0);
}


unsigned int	tempeExpected[] = {  /* registers to read with expected contents */
	/* address,  expected,	mask */
	0x00000000, 0x014810e3, 0xFFFFFFFF,
	0x00000004, 0x02000000, 0xC6800288,
	0x00000008, 0x06800000, 0xFFFFFF00,
	0xFFFFFFFF
};
```

```
unsigned int    uniExpected[] = {       /* registers to read with expected contents */
        /* address,  expected,     mask */
        0x00000000, 0x000010e3, 0xFFFFFFFF,
        0x00000004, 0x02000000, 0xC6800288,
        0x00000008, 0x06800000, 0xFFFFFF00,
        0xFFFFFFFF
};



int
main(int argc, char *argv[])
{
        int     fd;
        int     n;
        int     imatempe = 0;
        unsigned int    *regptr;
        unsigned int    expected, actual;

        if (getMyInfo()) {
                printf("%s: getMyInfo failed.  Errno = %d\n", argv[0], errno);
                _exit(1);
        }
```

```
fd = open("/dev/vme_regs", 0);
if (fd < 0) {
        printf("%s: Open failed.  Errno = %d\n", argv[0], errno);
        _exit(1);
}

regptr = uniExpected;
if (myVmeInfo.vmeControllerID == 0x014810e3) {
        imatempe = 1;
        regptr = tempeExpected;
}
while (*regptr < 0x1000) {
        lseek(fd, *regptr, SEEK_SET);
        n = read(fd, &actual, 4);
        if (n != 4) {
                printf("%s: Read at address %08x failed.  Errno = %d\n",
                   argv[0], *regptr, errno);
                _exit(1);
        }
        actual &= *(regptr + 2);

        expected = *(regptr + 1);
        expected &= *(regptr + 2);

        if (expected != actual) {
```

```
                printf("%s: Unexpected data at address %08x.\n",
                    argv[0], *regptr);
                printf("%s: Expected: %08x Actual: %08x.\n",
                    argv[0], expected, actual);
                _exit(1);
            }
            regptr += 3;

        }
        return(0);
}
```

A test program for generating VME bus interrupts:

```
/* VME Linux driver test/demo code */
/*
 * Include files
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
```

```
#include <errno.h>
#include <sys/ioctl.h>
#include "../vmedrv.h"

vmeInfoCfg_t myVmeInfo;
int
getMyInfo()
{
	int	fd, status;
	fd = open("/dev/vme_ctl", 0);
	if (fd < 0) {
		return(1);
	}

	memset(&myVmeInfo, 0, sizeof(myVmeInfo));
	status = ioctl(fd, VME_IOCTL_GET_SLOT_VME_INFO, &myVmeInfo);
	if (status < 0) {
		return(1);
	}
	close(fd);
	return(0);
}

/*
```

```
 * Sequentially generate all even vectors on all levels
 */
int
main(int argc, char *argv[])
{
	int	fd;
	int	status;
	int	level, vec;

	virqInfo_t vmeVirq;

	if (getMyInfo()) {
		printf("%s: getMyInfo failed.  Errno = %d\n", argv[0], errno);
		_exit(1);
	}

	fd = open("/dev/vme_ctl", 0);
	if (fd < 0) {
		printf("%s: Open failed.  Errno = %d\n", argv[0], errno);
		_exit(1);
	}

	for (level = 1; level < 8; level++) {
		for ( vec = 2; vec < 256; vec += 2) {
			memset(&vmeVirq, 0, sizeof(vmeVirq));
```

```
                vmeVirq.level = level;
                vmeVirq.vector = vec;
                vmeVirq.waitTime = 10;
                status = ioctl(fd, VME_IOCTL_GENERATE_IRQ, &vmeVirq);
                if (status < 0) {
                    printf("%s: Ioctl failed.  Errno = %d\n", argv[0], errno);
                    _exit(1);
                }
                if (vmeVirq.timeOutFlag) {
                    printf("%s: VIRQ level %d, VEC %02x not acknowledged\n",
                       argv[0], level, vec);
                    _exit(1);
                }
            }
        }

        return(0);
}
```

STRUCTURE DEFINITIONS - VMEDRV.H

```
/*
 * Module name: vmedrv.h
```

```
 *      Application interface to VME device nodes.
 */

/*
 * The reserved elements in the following structures are
 * intended to allow future versions of the driver to
 * maintain backwards compatibility.  Applications should
 * initialize reserved locations to zero.
 */

#ifndef VMEDRV_H
#define VMEDRV_H

#define VMEDRV_REV      0x0301

#define VME_MINOR_TYPE_MASK     0xF0
#define VME_MINOR_OUT   0x00
#define VME_MINOR_DMA   0x10
#define VME_MINOR_MISC  0x20
#define VME_MINOR_SLOTS1        0x30
#define VME_MINOR_SLOTS2        0x40

#define VME_MINOR_CTL   0x20
#define VME_MINOR_REGS  0x21
#define VME_MINOR_RMW   0x22
```

```
#define VME_MINOR_LM    0x23
#ifndef PAGESIZE
#define PAGESIZE 4096
#endif
#ifndef LINESIZE
#define LINESIZE 0x20
#endif
#define VME_MAX_WINDOWS 8
#ifndef PCI_DEVICE_ID_TUNDRA_TEMPE
#define PCI_DEVICE_ID_TUNDRA_TEMPE 0x148
#endif

/*****************************************************************************
 * COMMON definitions
 */

/*
 *  VME access type definitions
 */
#define VME_DATA             1
#define VME_PROG             2
#define VME_USER             4
#define VME_SUPER            8

/*
```

```
 *  VME address type definitions
 */
typedef enum {
        VME_A16,
        VME_A24,
        VME_A32,
        VME_A64,
        VME_CRCSR,
        VME_USER1,
        VME_USER2,
        VME_USER3,
        VME_USER4
} addressMode_t;

/*
 *  VME Transfer Protocol Definitions
 */
#define VME_SCT             0x1
#define VME_BLT             0x2
#define VME_MBLT            0x4
#define VME_2eVME           0x8
#define VME_2eSST           0x10
#define VME_2eSSTB          0x20

/*
```

```
 *  Data Widths
 */
typedef enum {
        VME_D8 = 8,
        VME_D16 = 16,
        VME_D32 = 32,
        VME_D64 = 64
} dataWidth_t;

/*
 *  2eSST Data Transfer Rate Codes
 */
typedef enum {
        VME_SSTNONE = 0,
        VME_SST160 = 160,
        VME_SST267 = 267,
        VME_SST320 = 320
} vme2esstRate_t;


/*
 *  Arbitration Scheduling Modes
 */
typedef enum {
        VME_R_ROBIN_MODE,
```

```
        VME_PRIORITY_MODE
} vme2ArbMode_t;



/*****************************************************************************
 *  /dev/vme_m* Outbound Window Ioctl Commands
 */
#define VME_IOCTL_SET_OUTBOUND          0x10
#define VME_IOCTL_GET_OUTBOUND          0x11
/*
 *  VMEbus OutBound Window Arg Structure
 *  NOTE:
 *      If pciBusAddr[U,L] are 0, then kernel will dynamically assign
 *      pci start address on PCI bus.
 */
struct  vmeOutWindowCfg
{
        int             windowNbr;      /*  Window Number */
        char            windowEnable;   /*  State of Window */
        unsigned int    pciBusAddrU;    /*  Start Address on the PCI Bus */
        unsigned int    pciBusAddrL;    /*  Start Address on the PCI Bus */
        unsigned int    windowSizeU;    /*  Window Size */
        unsigned int    windowSizeL;    /*  Window Size */
        unsigned int    xlatedAddrU;    /*  Starting Address on the VMEbus */
        unsigned int    xlatedAddrL;    /*  Starting Address on the VMEbus */
```

```
        int             bcastSelect2esst;       /*  2eSST Broadcast Select */
        char            wrPostEnable;           /*  Write Post State */
        char            prefetchEnable; /*  Prefetch Read Enable State */
        int             prefetchSize;   /*  Prefetch Read Size (in Cache Lines) */
        vme2esstRate_t  xferRate2esst;  /*  2eSST Transfer Rate */
        addressMode_t   addrSpace;      /*  Address Space */
        dataWidth_t     maxDataWidth;   /*  Maximum Data Width */
        int             xferProtocol;   /*  Transfer Protocol */
        int             userAccessType; /*  User/Supervisor Access Type */
        int             dataAccessType; /*  Data/Program Access Type */
        int             reserved;       /* For future use */

};
typedef struct  vmeOutWindowCfg vmeOutWindowCfg_t;



/*****************************************************************************
 *  /dev/vme_dma* DMA commands
 */
#define VME_IOCTL_START_DMA             0x30
#define VME_IOCTL_PAUSE_DMA             0x31
#define VME_IOCTL_CONTINUE_DMA          0x32
#define VME_IOCTL_ABORT_DMA             0x33
#define VME_IOCTL_WAIT_DMA              0x34
```

```
typedef enum {
	/* NOTE: PATTERN entries only valid as source of data */
	VME_DMA_PATTERN_BYTE,
	VME_DMA_PATTERN_BYTE_INCREMENT,
	VME_DMA_PATTERN_WORD,
	VME_DMA_PATTERN_WORD_INCREMENT,
	VME_DMA_USER,
	VME_DMA_KERNEL,
	VME_DMA_PCI,
	VME_DMA_VME
} dmaData_t;


/*
 *  VMEbus Transfer Attributes
 */
struct  vmeAttr
{
	dataWidth_t     maxDataWidth;   /*  Maximum Data Width */
	vme2esstRate_t  xferRate2esst;  /*  2eSST Transfer Rate */
	int             bcastSelect2esst; /*  2eSST Broadcast Select */
	addressMode_t   addrSpace;      /*  Address Space */
	int             userAccessType; /*  User/Supervisor Access Type */
	int             dataAccessType; /*  Data/Program Access Type */
	int             xferProtocol;   /*  Transfer Protocol */
```

```
        int             reserved;       /* For future use */
};
typedef struct  vmeAttr vmeAttr_t;

/*
 *  DMA arg info
 *  NOTE:
 *      structure contents relating to VME are don't care for
 *       PCI transactions
 *      structure contents relating to PCI are don't care for
 *       VME transactions
 *      If source or destination is user memory and transaction
 *      will cross page boundary, the DMA request will be split
 *      into multiple DMA transactions.
 */
typedef struct  vmeDmaPacket
{
        int             vmeDmaToken;    /*  Token for driver use */
        int             vmeDmaWait;     /*  Time to wait for completion */
        unsigned int    vmeDmaStartTick; /*  Time DMA started */
        unsigned int    vmeDmaStopTick; /*  Time DMA stopped */
        unsigned int    vmeDmaElapsedTime; /*  Elapsed time */
        int             vmeDmaStatus;   /*  DMA completion status */

        int     byteCount;      /*  Byte Count */
```

```
int     bcastSelect2esst;       /*  2eSST Broadcast Select */

/*
 *  DMA Source Data
 */
dmaData_t       srcBus;
unsigned int            srcAddrU;
unsigned int            srcAddr;
int             pciReadCmd;
struct          vmeAttr srcVmeAttr;
char            srcfifoEnable;

/*
 *  DMA Destination Data
 */
dmaData_t       dstBus;
unsigned int            dstAddrU;
unsigned int            dstAddr;
int             pciWriteCmd;
struct          vmeAttr dstVmeAttr;
char            dstfifoEnable;


/*
 *  BUS usage control
```

```
        */
        int             maxCpuBusBlkSz; /*  CPU Bus Maximum Block Size */
        int             maxPciBlockSize;        /* PCI Bus Maximum Block Size */
        int             pciBackOffTimer;        /* PCI Bus Back-Off Timer */
        int             maxVmeBlockSize;        /* VMEbus Maximum Block Size */
        int             vmeBackOffTimer;        /* VMEbus Back-Off Timer */

        int             channel_number; /* Channel number */
        int             reserved;       /* For future use */
        /*
         *      Ptr to next Packet
         *      (NULL == No more Packets)
         */
        struct          vmeDmaPacket *  pNextPacket;

} vmeDmaPacket_t;



/*****************************************************************************
 *  /dev/vme_ctl ioctl Commands
 */
#define VME_IOCTL_GET_SLOT_VME_INFO     0x41
/*
 *  VMEbus GET INFO Arg Structure
 */
```

```
struct  vmeInfoCfg
{
        int             vmeSlotNum;     /*  VME slot number of interest */
        int             boardResponded; /* Board responded */
        char             sysConFlag;     /*  System controller flag */
        int             vmeControllerID; /*  Vendor/device ID of VME bridge */
        int             vmeControllerRev; /*  Revision of VME bridge */
        char             osName[8];      /*  Name of OS e.g. "Linux" */
        int             vmeSharedDataValid;     /*  Validity of data struct */
        int             vmeDriverRev;   /*  Revision of VME driver */
        unsigned        int vmeAddrHi[8];  /* Address on VME bus */
        unsigned        int vmeAddrLo[8];  /* Address on VME bus */
        unsigned        int vmeSize[8];  /* Size on VME bus */
        unsigned        int vmeAm[8];  /* Address modifier on VME bus */
        int             reserved;       /* For future use */
};
typedef struct  vmeInfoCfg      vmeInfoCfg_t;

#define VME_IOCTL_SET_REQUESTOR         0x42
#define VME_IOCTL_GET_REQUESTOR         0x43
/*
 *  VMEbus Requester Arg Structure
 */
struct  vmeRequesterCfg
{
```

```
        int             requestLevel;   /*  Requester Bus Request Level */
        char            fairMode;       /*  Requester Fairness Mode Indicator */
        int             releaseMode;    /*  Requester Bus Release Mode */
        int             timeonTimeoutTimer;     /*  Master Time-on Time-out Timer */
        int             timeoffTimeoutTimer;    /*  Master Time-off Time-out Timer */
        int             reserved;       /* For future use */
};
typedef struct  vmeRequesterCfg vmeRequesterCfg_t;

#define VME_IOCTL_SET_CONTROLLER        0x44
#define VME_IOCTL_GET_CONTROLLER        0x45
/*
 *  VMEbus Arbiter Arg Structure
 */
struct  vmeArbiterCfg
{
        vme2ArbMode_t   arbiterMode;    /*  Arbitration Scheduling Algorithm */
        char            arbiterTimeoutFlag;     /*  Arbiter Time-out Timer Indicator */
        int             globalTimeoutTimer;     /*  VMEbus Global Time-out Timer */
        char            noEarlyReleaseFlag;     /*  No Early Release on BBUSY */
        int             reserved;       /* For future use */
};
typedef struct  vmeArbiterCfg   vmeArbiterCfg_t;

#define VME_IOCTL_GENERATE_IRQ          0x46
```

```
#define VME_IOCTL_GET_IRQ_STATUS        0x47
#define VME_IOCTL_CLR_IRQ_STATUS        0x48
/*
 *  VMEbus IRQ Info
 */
typedef struct  virqInfo
{
        /*
         *  Time to wait for Event to occur (in clock ticks)
         */
        short               waitTime;
        short               timeOutFlag;

        /*
         *  VMEbus Interrupt Level and Vector Data
         */
        int             level;
        int             vector;
        int             reserved;       /* For future use */

} virqInfo_t;

#define VME_IOCTL_SET_INBOUND           0x49
#define VME_IOCTL_GET_INBOUND           0x50
/*
```

```
*  VMEbus InBound Window Arg Structure
*  NOTE:
*      If pciBusAddr[U,L] and windowSize[U,L] are 0, then kernel
*      will dynamically assign inbound window to map to a kernel
*      supplied buffer.
*/
struct  vmeInWindowCfg
{
        int             windowNbr;      /*  Window Number */
        char            windowEnable;   /*  State of Window */
        unsigned int            vmeAddrU;       /*  Start Address responded to on the VMEbus */
        unsigned int            vmeAddrL;       /*  Start Address responded to on the VMEbus */
        unsigned int            windowSizeU;    /*  Window Size */
        unsigned int            windowSizeL;    /*  Window Size */
        unsigned int            pciAddrU;       /*  Start Address appearing on the PCI Bus */
        unsigned int            pciAddrL;       /*  Start Address appearing on the PCI Bus */
        char            wrPostEnable;   /*  Write Post State */
        char            prefetchEnable; /*  Prefetch Read State */
        char            prefetchThreshold;      /*  Prefetch Read Threshold State */
        int             prefetchSize;   /*  Prefetch Read Size */
        char            rmwLock;        /*  Lock PCI during RMW Cycles */
        char            data64BitCapable;       /*  non-VMEbus capable of 64-bit Data */
        addressMode_t   addrSpace;      /*  Address Space */
        int             userAccessType; /*  User/Supervisor Access Type */
        int             dataAccessType; /*  Data/Program Access Type */
```

```
        int             xferProtocol;   /*  Transfer Protocol */
        vme2esstRate_t  xferRate2esst;  /*  2eSST Transfer Rate */
        char            bcastRespond2esst;      /*  Respond to 2eSST Broadcast */
        int             reserved;       /* For future use */

};
typedef struct  vmeInWindowCfg  vmeInWindowCfg_t;



/*******************************************************************************
 *  /dev/vme_rmw RMW Ioctl Commands
 */
#define VME_IOCTL_DO_RMW                0x60
/*
 *  VMEbus RMW Configuration Data
 */
struct  vmeRmwCfg
{
        unsigned int    targetAddrU;    /*  VME Address (Upper) to trigger RMW cycle */
        unsigned int    targetAddr;     /*  VME Address (Lower) to trigger RMW cycle */
        addressMode_t   addrSpace;      /*  VME Address Space */
        int     enableMask;     /*  Bit mask defining the bits of interest */
        int     compareData;    /*  Data to be compared with the data read */
        int     swapData;       /*  Data written to the VMEbus on success */
        int     maxAttempts;    /*  Maximum times to try */
```

```
        int     numAttempts;    /*  Number of attempts before success */
        int     reserved;       /* For future use */

} ;
typedef struct vmeRmwCfg        vmeRmwCfg_t;


/*****************************************************************************
 *  /dev/vme_lm location Monitor Ioctl Commands
 */
#define VME_IOCTL_SETUP_LM              0x70
#define VME_IOCTL_WAIT_LM               0x71
/*
 *  VMEbus Location Monitor Arg Structure
 */
struct  vmeLmCfg
{
        unsigned int            addrU;  /*  Location Monitor Address upper */
        unsigned int            addr;   /*  Location Monitor Address lower */
        addressMode_t   addrSpace;      /*  Address Space */
        int             userAccessType; /*  User/Supervisor Access Type */
        int             dataAccessType; /*  Data/Program Access Type */
        int             lmWait;         /* Time to wait for access */
        int             lmEvents;       /* Lm event mask */
        int             reserved;       /* For future use */
```

```
};
typedef struct  vmeLmCfg        vmeLmCfg_t;

/*
 *  Data structure created for each board in CS/CSR space.
 */
struct  vmeSharedData {
        /*
         * Public elements
         */
        char    validity1[4];   /* "VME" when contents are valid */
        char    validity2[4];   /* "RDY" when contents are valid */
        int     structureRev;   /* Revision of this structure */
        int     reserved1;

        char    osname[8];      /* OS name string */
        int     driverRev;      /* Revision of VME driver */
        int     reserved2;

        char    boardString[16];        /* type of board */

        int     vmeControllerType;
        int     vmeControllerRev;
        int     boardSemaphore[8];      /* for use by remote */
        unsigned int    inBoundVmeAddrHi[8];    /* This boards VME windows */
```

```
        unsigned int    inBoundVmeAddrLo[8];    /* This boards VME windows */
        addressMode_t   inBoundVmeAM[8];        /* Address modifier */
        int     inBoundVmeSize[8];      /* size available to remotes */
        char    reserved3[0x1000-248];  /* Pad to 4k boundary */

        int     readTestPatterns[1024];
        int     remoteScratchArea[24][256];     /* 1k scratch for each remote*/
        /*
         * Private areas for use by driver only.
         */
        char    driverScratch[4096];
        struct  {
                char    Eye[4];
                int     Offset;
                int     Head;
                int     Tail;
                int     Size;
                int     Reserved1;
                int     Reserved2;
                int     Reserved3;
                char    Data[4096-32];
        } boardFifo[23];
};
```

```
/*
 *  Driver errors reported back to the Application (other than the
 *  stanard Linux errnos...).
 */
#define VME_ERR_VERR            1       /* VME bus error detected */
#define VME_ERR_PERR            2       /* PCI bus error detected */


#endif  /* VMEDRV_H */
```