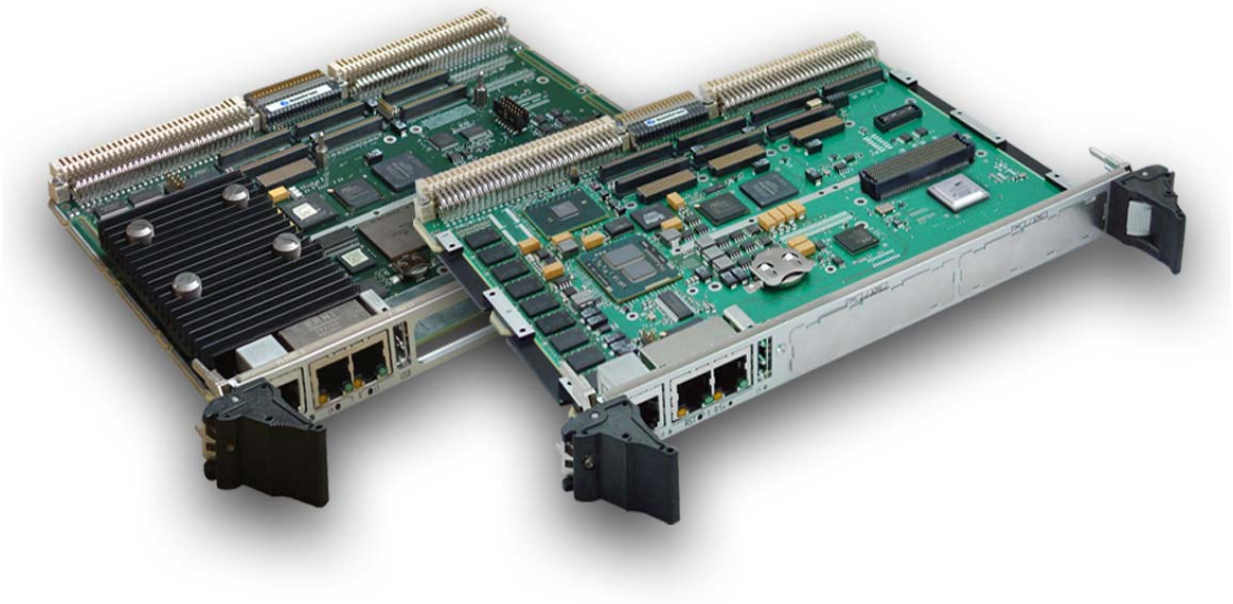
» VM6050 & VM6250 «



VxWorks 6.8
ALMA VME Bus Driver

SD.DT.G00-0e - February 2012

Revision History

Publication Title:	VxWorks 6.8 VME Driver on VM6050 & VM6250	
Doc. ID:	SD.DT.G00-0e	
Rev.	Brief Description of Changes	Date of Issue
0e	Initial Issue	02-2012

Proprietary Note

This document contains information proprietary to Kontron. It may not be copied or transmitted by any means, disclosed to others, or stored in any retrieval system or media without the prior written consent of Kontron or one of its authorized agents.

The information contained in this document is, to the best of our knowledge, entirely correct. However, Kontron cannot accept liability for any inaccuracies or the consequences thereof, or for any liability arising from the use or application of any circuit, product, or example shown in this document.

Kontron reserves the right to change, modify, or improve this document or the product described herein, as seen fit by Kontron without further notice.

Trademarks

This document may include names, company logos and trademarks, which are registered trademarks and, therefore, proprietary to their respective owners.

Environmental Protection Statement

This product has been manufactured to satisfy environmental protection requirements where possible. Many of the components used (structural parts, printed circuit boards, connectors, batteries, etc.) are capable of being recycled.

Final disposition of this product after its service life must be accomplished in accordance with applicable country, state, or local laws or regulations.

Environmental protection is a high priority with Kontron.

Kontron follows the DEEE/WEEE directive.

You are encouraged to return our products for proper disposal.

The Waste Electrical and Electronic Equipment (WEEE) Directive aims to:

- reduce waste arising from electrical and electronic equipment (EEE)
- make producers of EEE responsible for the environmental impact of their products, especially when they become waste
- encourage separate collection and subsequent treatment, reuse, recovery, recycling and sound environmental disposal of EEE
- improve the environmental performance of all those involved during the lifecycle of EEE

Conventions

This guide uses several types of notice: Note, Caution, ESD.

Note: this notice calls attention to important features or instructions.

Caution: this notice alert you to system damage, loss of data, or risk of personal injury.

ESD: This banner indicates an Electrostatic Sensitive Device.

All numbers are expressed in decimal, except addresses and memory or register data, which are expressed in hexadecimal. The prefix `0x' shows a hexadecimal number, following the `C' programming language convention.

The multipliers `k', `M' and `G' have their conventional scientific and engineering meanings of $*10^3$, $*10^6$ and $*10^9$ respectively. The only exception to this is in the description of the size of memory areas, when `K', `M' and `G' mean $*2^{10}$, $*2^{20}$ and $*2^{30}$ respectively.

When describing transfer rates, `k' `M' and `G' mean $*10^3$, $*10^6$ and $*10^9$ *not $*2^{10}$ $*2^{20}$ and $*2^{30}$.*

In PowerPC terminology, multiple bit fields are numbered from 0 to n, where 0 is the MSB and n is the LSB. PCI and CompactPCI terminology follows the more familiar convention that bit 0 is the LSB and n is the MSB.

Signal names ending with an asterisk (*) or a hash (#) denote active low signals; all other signals are active high.

Signal names follow the PICMG 2.0 R3.0 CompactPCI Specification and the PCI Local Bus 2.3 Specification.

For Your Safety

Your new Kontron product was developed and tested carefully to provide all features necessary to ensure its compliance with electrical safety requirements. It was also designed for a long fault-free life. However, the life expectancy of your product can be drastically reduced by improper treatment during unpacking and installation. Therefore, in the interest of your own safety and of the correct operation of your new Kontron product, you are requested to conform with the following guidelines.

High Voltage Safety Instructions

Warning!
All operations on this device must be carried out by sufficiently skilled personnel only.

Caution, Electric Shock!
Before installing a not hot-swappable Kontron product into a system always ensure that your mains power is switched off. This applies also to the installation of piggybacks. Serious electrical shock hazards can exist during all installation, repair and maintenance operations with this product. Therefore, always unplug the power cable and any other cables which provide external voltages before performing work.

Special Handling and Unpacking Instructions

ESD Sensitive Device!
Electronic boards and their components are sensitive to static electricity. Therefore, care must be taken during all handling operations and inspections of this product, in order to ensure product integrity at all times

Do not handle this product out of its protective enclosure while it is not used for operational purposes unless it is otherwise protected.

Whenever possible, unpack or pack this product only at EOS/ESD safe work stations. Where a safe work station is not guaranteed, it is important for the user to be electrically discharged before touching the product with his/her hands or tools. This is most easily done by touching a metal part of your system housing.

It is particularly important to observe standard anti-static precautions when changing piggybacks, ROM devices, jumper settings etc. If the product contains batteries for RTC or memory backup, ensure that the board is not placed on conductive surfaces, including anti-static plastics or sponges. They can cause short circuits and damage the batteries or conductive circuits on the board.

General Instructions on Usage

In order to maintain Kontron's product warranty, this product must not be altered or modified in any way. Changes or modifications to the device, which are not explicitly approved by Kontron and described in this manual or received from Kontron's Technical Support as a special handling instruction, will void your warranty.

This device should only be installed in or connected to systems that fulfill all necessary technical and specific environmental requirements. This applies also to the operational temperature range of the specific board version, which must not be exceeded. If batteries are present, their temperature restrictions must be taken into account.

In performing all necessary installation and application operations, please follow only the instructions supplied by the present manual.

Keep all the original packaging material for future storage or warranty shipments. If it is necessary to store or ship the board, please re-pack it as nearly as possible in the manner in which it was delivered.

Special care is necessary when handling or unpacking the product. Please consult the special handling and unpacking instruction on the previous page of this manual.

Table Of Contents

Chapter 1 - Introduction 1
Chapter 2 - Including VME Bus Driver 2
Chapter 3 - Master and Slave VME Windows 3
3.1 Configuring Static VME Slave Windows 3
3.2 Configuring Dynamic VME Slave Windows 6
3.2.1 Example of VME slave Windows Creation mapping a buffer 6
3.2.2 Example to Create a slave windows on top of memory 8
3.3 VME Master Configuration 9
3.4 Configuring Static VME Master Windows 10
3.4.1 Example of define for a static VME Master Windows: 12
3.5 Configuring Dynamic VME Master Windows 12
3.5.1 Example code for dynamic VME Master window 12
Chapter 4 - Accessing a VME Address or Getting a Local Address 13
4.1 Description 13
4.2 Routines 13
Chapter 5 - Default VME Board Configuration 14
Chapter 6 - Simple Access on VME 15
6.1 VME Bus Take and Release services 15
Chapter 7 - VME DMA 16
7.1 Routines 16
Chapter 8 - VME 2eSST Support 18
8.1 VME ALMA Bridge Configuration for 2eSST Support 18
8.2 Slave VME 2ESST Windows 18
8.2.1 Routines for 2eSST Slave Windows 18
8.2.2 Example of Define for Static 2eSST VME Slave Windows 20
8.2.3 Example to Create 2eSST VME Slave Windows Dynamically 21
8.3 2eSST VME DMA 23
8.3.1 Description 23
8.3.2 Routines 23
Chapter 9 - VME Interrupt Line Routing 25
9.1 VME Interrupt Reception and Generation 25
9.1.1 Description 25

9.1.2 Routines 25
9.1.3 VME Interrupt Example 26

Chapter 10 - VME Mailbox 27
10.1 Description 27
10.2 Routines 27
10.3 Mailbox Examples 28

Chapter 11 - VME Shared Semaphore 29
11.1 VME Read Modify Write Cycles 29
11.2 Description 29
11.3 Semaphore Routines 30
11.4 VME Semaphore Example 31

Chapter 12 - VME ACFAIL Signal 33
12.1 Description 33
12.2 Routines 33

Chapter 13 - VME SYSFAIL Signal 34
13.1 Description 34
13.2 Routines 34

Chapter 14 - VME Bus ERROR and Probe 35
14.1 Description 35
14.2 Routines 35

Chapter 15 - Watchdog and Hardware Reset 36
15.1 Description 36
15.2 Routines 36

Chapter 1 - Introduction

This document describes the VxBus VME bus driver under VxWorks 6.8 developped for VM6050 Intel® Core™ I7 based board or VM6250 PowerPC board. The VME feature is accomplished trough the Kontron/IBM PCI to VME Bridge called ALMA bridge. This bridge provides full access to VME Bus so it gives access to :

- Dynamic master and slaves VME windows configuration,
- Simple VME data transfer,
- DMA data transfer services,
- 2eSST data transfer services,
- VME interrupt services,
- Routines to require VME Bus by Software,
- Shared semaphore routines,
- Bus error connect routines,
- ACFail and SysFail connect routines,
- Addressed mailbox interrupt routines,
- VME addresses probing (to check VME addresses validity) routines,
- Watchdog and reset routines.

Chapter 2 - Including VME Bus Driver

To include the ALMA VME bus the define `INCLUDE_VME_DRV` must be defined in `config.h`.

This define is set by default into the BSP delivery.

Look at the ALMA VME driver files located at `$WIND_BASE/target/3rdParty/Kontron/h/vxbAlma.h` for VM6050 or into the BSP for VM6250 to see all other available pre-defined values and to find all routines pre-definition.

Chapter 3 - Master and Slave VME Windows

The ALMA PCI-to-VME bridge provides a full VME64 interface. This chip is highly configurable for both VME master and slave addressing, and can appear a little overwhelming at first. ALMA uses "windows" for accessing and being accessed. A window is simply a mapping from one space to another. This can be from the CPU to the VME, VME to the PCI, VME to the local RAM.

In order to ease the initial learning curve for this chip, the mechanism described below provides a simple method for configuring slave and master addressing VME windows. It should be remembered however, that this chip can be configured to provide almost ANY slave and master mapping that might be conceived.

3.1 Configuring Static VME Slave Windows

The board can be configured to allow access to any part of the board from any address in any space on the VME bus.

The minimum granularity for VME slave spaces is 1 MB, i.e. a minimum of 1 MB of internal PCI memory space can be mapped onto a contiguous VME space. Thereafter, the size increases in powers of 2.

Applications requiring more flexibility with slave windows allocation can use the `sysVmeChannelAlloc()` and `sysVmeChannelFree()` routines bypassing those static mechanisms and configuring the chip directly.

Up to 8 distinct static mappings are allowed at once for mapping VME to board components, like RAM or peripherals. These static windows must be defined in **`hwconf.h`** file into the BSP directory. To active a slave static window that will be present at any boot then, uncomment one of the slave windows in **`hwconf.h`** file structure.

```
VME_STATIC_WIND vmeStaticWindow [] ;
```

Each entry corresponds to a VME_STATIC_WIND structure that must be filled with the following informations:

```
/* VME windows */
typedef struct vmeStaticWindow
   {
   int        type;                    /* window type master or slave */
   char       name[WIND_NAMEMAX];      /* window name */
   UINT32     cpuAddr;                 /* CPU address */
   UINT32     pciAddr;                 /* PCI address */
   UINT32     vmeAddr;                 /* VME address */
   UINT32     size;                    /* size of window (in bytes)*/
   UINT32     flag;                    /* any particular flags */
   } VME_STATIC_WIND;
```

For each window, there are 6 values that need to be defined. These values are :

1. `type`

   VME_SLAVE define must be set for slave windows

2. `name`

   The slave window name limited to 12 characters. It can be displayed with `sysVmeChannelShow()` under VxWorks shell.

3. `cpuAddr`

   Must be set to zero for slave windows. Not used.

4. `pciAddr`

   The PCI start address that is mapped to the slave address; for example: 0x00000000 is a mapping of VME to PCI address 0x00000000 for VM6050 (which happens to be the start of local memory on Pentium architecture) or 0x8000000 for PowerPC Board (start of local address for PowerPC Board). Since it is likely that most users will wish to map slave windows to the start of local RAM, a define named MAP_TO_RAM for parameter will just do that. If it is necessary to map the VME windows to the end of a huge RAM (1 or 2 GB) then this parameter must be set to a value that corresponds to the top of RAM minus the wished windows size and being 1M aligned. The top of RAM can be determined by routine sysMemTop();

5. `vmeAddr`

   The VME start address at which this slave window starts to respond.

   For example, a value of (0x10000000) requests the slave window 1 to start at 0x10000000 on the VME.

   If this window is for a A32 space, and this parameter is set to the value MAP_BY_VME_ID then the exact slave address configured on the board is dependent on the processor number, as defined in the VxWorks boot parameters. The calculation is the following:

   addressMapped = VME_DEFAULT_BASE + (sysVmeIdGet () * VME_DEFAULT_INCREMENT)
   where VME_DEFAULT_BASE and VME_DEFAULT_INCREMENT are defined in
   `3rdParty/Kontron/h/vxbAlma.h :`

```
#define VME_DEFAULT_BASE      0x08000000
#define VME_DEFAULT_INCREMENT 0x08000000
```

   This gives a table of default slave addresses as follows :

VME Board Id	VME Slave Address
0	0x08000000
1	0x10000000
2	0x18000000
3	0x20000000
4	0x28000000
5	0x30000000
6	0x38000000
7	0x40000000
8	0x48000000
9	0x50000000
a	0x58000000
b	0x60000000

6. `Size`

   The size (in bytes) of the slave window. Therefore, it must be a whole number of a power of 2 megabytes, since that is the granularity of a slave window. It is so possible to map 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 or 1024 MB in a slave window, since these are powers of 2.

7. `flags`

   Provides information about the mapping - the address space to use, the address modifiers to allow, whether posted writes are allowed etc.

This parameter is a logical ORing of the following:

VMEFG_AM_A32_ENABLE	enable A32 space accesses
VMEFG_AM_A24_ENABLE	enable A24 space accesses
VMEFG_AM_A16_ENABLE	enable A16 space accesses
VMEFG_AM_SUPERVISOR_ENABLE	enable supervisor accesses
VMEFG_AM_USR_ENABLE	enable user accesses
VMEFG_AM_64BIT_ENABLE	enable 64 bit accesses
VMEFG_AM_DATA_ENABLE	enable data accesses
VMEFG_AM_PGM_ENABLE	enable program accesses
VMEFG_AM_ASCENDING_ENABLE	enable ascending accesses
VMEFG_WRTPOST	allow writeposting
VMEFG_READAHEAD	allow readahead buffering

The following next 4 defines are mutually exclusive and define the mapping between big and little endian data, i.e. how data and address values get swapped around. If none is specified, the default is VMEFG_LEBE_ADDR.

VMEFG_LEBE_NO	do no swapping
VMEFG_LEBE_ADDR	maintain address coherency (default)
VMEFG_LEBE_DATA	maintain data coherency
VMEFG_LEBE_BTNS	byte translation (no swapping)

Example of define for static VME Slave Windows

```
{VME_SLAVE, "DramSlave0", 0, MAP_TO_RAM, MAP_BY_VME_ID, 0x04000000,
              ( VMEFG_AM_A32_ENABLE | VMEFG_AM_SUPERVISOR_ENABLE |
                VMEFG_AM_USR_ENABLE | VMEFG_AM_LONG_ENABLE |
                VMEFG_READAHEAD | VMEFG_WRTPOST),}
```

3.2 Configuring Dynamic VME Slave Windows

Applications requiring more flexibility in managing slave windows allocation can use the `sysVmeChannelAlloc()` routines. It is particularly interesting to open slave VME windows to map directly a previoulsy allocated buffer using `memalign()` routine for example. So the other VME board has only to access to the correct VME address to read/write directly the reserved buffer without knowing its real address into the RAM.

See the first example in section 3.2.1.

Routine Name	Description
`sysVmeChannelAlloc()`	Create a VME to PCI mapping in ALMA
`sysVmeChannelFree()`	Delete a VME to PCI mapping in ALMA

See section 3.1 page 3 on static windows to have a description of routine parameters.

3.2.1 Example of VME slave Windows Creation mapping a buffer

```
STATUS createSlaveVMEWindows() {
   void *localAddr ;
   VME_STATIC_WIND pWindSlave = {
   VME_SLAVE,
   "8MBuffMap",
   0,
   MAP_TO_RAM,
   MAP_BY_VME_ID,
   0x00800000, /*8M size*/
   ( VMEFG_AM_A32_ENABLE | VMEFG_AM_SUPERVISOR_ENABLE |
   VMEFG_AM_USR_ENABLE | VMEFG_AM_LONG_ENABLE |
             VMEFG_READAHEAD | VMEFG_WRTPOST)};

   /* Reserve a buffer aligned to 1M for VME slave windows mapping*/
   localAddr = memalign (0x100000, 0x800000);
   if (localAddr == (void *)NULL)
             {
                        printf ("Error mallocing 0x800000 aligned on 1M\n");
                        return ERROR;
             }
   printf("Local Malloc =  CPU address :0x%08x\n", localAddr);
   memset(localAddr, 0xFF, 0x800000);

   pWindSlave.vmeAddr = (UINT32)(VME_DEFAULT_BASE + (sysVmeIdGet() *
   VME_DEFAULT_INCREMENT));
   pWindSlave.pciAddr = localAddr; /*This will map the VME windows directly to the buffer*/
   if (sysVmeChannelAlloc(&pWindSlave) != OK)
             printf("Cannot Alloc Slave VME Channel");
   return OK;
}
```

When executing this gives on VMEID 0 board

```
-> createSlaveVMEWindows
Local Malloc =  CPU address :0x0cc00000
value = 0 = 0x0
-> sysVmeChannelShow

Master PCI-MEM to VME Bus
-------------------------------------------------------------------------------
|   Name     | CPU Addr | PCI Addr | VME Addr | Size |   AM   |Conv| WP | RH |
-------------------------------------------------------------------------------
|AlmaRemote  |0xed000000|0xed000000|0x00000000|   8Mb|  A16S  |ADDR| No | No |
-------------------------------------------------------------------------------

Slave VME to PCI Bus
-------------------------------------------------------------------------------
|Num|   Name     | VME Addr |   Addr   | Space | Size|   AM   |Conv| WP | RH |
-------------------------------------------------------------------------------
| 0 |8MBuffMap   |0x08000000|0x0cc00000| DRAM |   8Mb|D-03-ff |ADDR| Yes| Yes|
-------------------------------------------------------------------------------

2ESST Slave VME to PCI Bus
-------------------------------------------------------------------------------
|Num|  Name  |      VME Addr    |   Addr   | Space | Size |   AM   |Conv|WP|RH|
-------------------------------------------------------------------------------
|                   No 2eSST VME to PCI CHANNEL                               |
-------------------------------------------------------------------------------
value = 0 = 0x0
```

On the second VME board with a VME Master window where the VME address 0x08000000 is mapped, a simple display on the correct CPU address allows to dump the reserved buffer. Example

```
-> sysVmeChannelShow

Master PCI-MEM to VME Bus
-------------------------------------------------------------------------------
|   Name     | CPU Addr | PCI Addr | VME Addr | Size |   AM   |Conv| WP | RH |
-------------------------------------------------------------------------------
|DramMaster0 |0xe8000000|0xe8000000|0x08000000|  64Mb|A32SDATA|ADDR| No | No |

Etc

-> d 0xe8000000,40,1
NOTE: memory values are displayed in hexadecimal.
0xe8000000:  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  *................*
0xe8000010:  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  *................*
0xe8000020:  ff ff ff ff ff ff ff ff                         *................*
value = 0 = 0x0
->
```

The buffer content on VMEId 0 board is accessed

3.2.2 Example to Create a slave windows on top of memory

This example assumes that the USER_RESERVED_MEM define has been set to 0x08000000 for example to reserve a shared memory on the top of DD RAM.

```
VME_STATIC_WIND pWindSlave = {
        VME_SLAVE,
        "DramSlave0",
        0,
        MAP_TO_RAM,
        MAP_BY_VME_ID,
        0x04000000, /*size*/
        ( VMEFG_AM_A32_ENABLE | VMEFG_AM_SUPERVISOR_ENABLE |
        VMEFG_AM_USR_ENABLE | VMEFG_AM_LONG_ENABLE |
        VMEFG_READAHEAD | VMEFG_WRTPOST)};

pWindSlave.pciAddr = (UINT32)((UINT32)sysMemTop() & 0xFFF00000) + (UINT32) 0x100000;
/*1M aligned*/
if (sysVmeChannelAlloc(&pWindSlave) != OK)
            printf("Cannot Alloc Slave VME Channel");
```

Will create the following slave window

```
-> sysVmeChannelShow
...
Slave VME to PCI Bus
-------------------------------------------------------------------------------
|Num|   Name     | VME Addr |   Addr    | Space | Size|   AM    |Conv| WP | RH |
-------------------------------------------------------------------------------
| 0 |DramSlave0  |0x08000000|0xb8000000| DRAM |  64Mb|D-03-ff |ADDR| Yes| Yes|
-------------------------------------------------------------------------------
```

3.3 VME Master Configuration

The ALMA chip is responsible for generating VME accesses based on PCI. It is located on the 32-bit PCI bus.

There is a 1024 8 MB entry table for mapping PCI memory, PCI I/O, or CPU to VME bus.

The granularity of VME master spaces is 8 MB i.e. a minimum of 8 MB of VME can be mapped in a contiguous space. The VME address must be 8 MB aligned. The VME size that can be mapped is limited by the space in PCI memory space (128 MB mapped on Alma2f BAR 1).

3.4 Configuring Static VME Master Windows

The Static Master windows are configured in a way similar to the static slave windows. A master window may be created to map to any part of A16, A24 or A32 VME address space, the mapping being from either PCI Memory or PCI I/O space (flags `VMEFG_PCIMEM` or `VMEFG_PCIIO`).

Up to 4 separate static master windows may be configured through this mechanism.

Applications requiring more than 4 windows or requiring dynamic master windows allocation can use the `sysVmeChannelAlloc()` and `sysVmeChannelFree()` routines bypassing those static mechanisms and configuring the chip directly (see section 3.4.1 page 12).

For each static window, 6 values need to be defined in the `hwconf.h` file. The structure is identical to the VME slave windows:

```
VME_STATIC_WIND vmeStaticWindow [] ;
```

For each window, there are 6 values that need to be defined. These values are :

1. `type`

   VME_MASTER define must be set for master windows

2. `name`

   The master window name is limited to 12 characters. It can be displayed with sysVmeChannelShow() under vxWorks shell.

3. `cpuAddr`

   This corresponds to the CPU address that must be used to access the VME with this window. It is calculated by the driver itself so set it to 0.

4. `pciAddr`

   Must be set to 0 for Master windows. Not used

5. `vmeAddr`

   The VME start address. It must be a 8MB aligned address to map the start of this master window. For example: 0x08000000 is a request that master maps to VME address 0x08000000. Typically, for A16 or A24 space this would be 0, since it is usual to map these spaces entirely.

6. `size`

   The size (in bytes) of the master window. This must be a whole multiple of 8MB, since this is the minimum granularity of a master window.

7. `flags`

   Provides information about the mapping - the address space to map to, the address modifier to use, whether posted writes are allowed, etc.

   This parameter is a logical ORing of the following:

   One of:

`VMEFG_PCIIO`	map VME via PCI32 I/O space
`VMEFG_PCIMEM`	map VME via PCI32 MEM space

If neither is specified, the default is to use MEM space

VMEFG_WRTPOST	Allow writeposting
VMEFG_READAHEAD	Allow readahead buffering

Any ONE of the following provides the address modifier (and VME address space):

These map to VME A32 space

VMEFG_WRTPOST	Allow writeposting
VMEFG_AM_A32UMBLT	Use Address Modifier 0x08
VMEFG_AM_A32UDATA	Use Address Modifier 0x09
VMEFG_AM_A32UPROG	Use Address Modifier 0x0A
VMEFG_AM_A32UBLT	Use Address Modifier 0x0B
VMEFG_AM_A32SMBLT	Use Address Modifier 0x0C
VMEFG_AM_A32SDATA	Use Address Modifier 0x0D
VMEFG_AM_A32SPROG	Use Address Modifier 0x0E
VMEFG_AM_A32SBLT	Use Address Modifier 0x0F

These map to VME A24 space

VMEFG_AM_A24UMBLT	Use Address Modifier 0x38
VMEFG_AM_A24UDATA	Use Address Modifier 0x39
VMEFG_AM_A24UPROG	Use Address Modifier 0x3A
VMEFG_AM_A24UBLT	Use Address Modifier 0x3B
VMEFG_AM_A24SMBLT	Use Address Modifier 0x3C
VMEFG_AM_A24SDATA	Use Address Modifier 0x3D
VMEFG_AM_A24SPROG	Use Address Modifier 0x3E
VMEFG_AM_A24SBLT	Use Address Modifier 0x3F

These map to VME A16 space

VMEFG_AM_A16U	Use Address Modifier 0x29
VMEFG_AM_A16S	Use Address Modifier 0x2D
VMEFG_AM_A16LCK	Use Address Modifier 0x2C

The following next 4 defines are mutually exclusive and define the mapping between big and little endian data i.e. how data and address values get swapped around. If none is specified, the default is VMEFG_LEBE_ADDR.

VMEFG_LEBE_NO	Do no swapping
VMEFG_LEBE_ADDR	Maintain address coherency (default)
VMEFG_LEBE_DATA	Maintain data coherency
VMEFG_LEBE_BTNS	Byte translation (no swapping)

3.4.1 Example of define for a static VME Master Windows:

In `hwconf.h` add an entry in `VME_STATIC_WIND vmeStaticWindow [] =`

```
{VME_MASTER, "VmeDma", 0, 0, 0x01800000, 0x04000000,VMEFG_AM_A32SDATA | VMEFG_READAHEAD
| VMEFG_WRTPOST}
```

3.5 Configuring Dynamic VME Master Windows

Applications requiring more than four static windows or requiring more flexibility in managing master windows allocation can use the `sysVmeChannelAlloc()` and `sysVmeChannelFree()` routines.

Routine Name	Description
`sysVmeChannelAlloc()`	Create a PCI to VME mapping in ALMA
`sysVmeChannelFree()`	Delete a PCI to VME mapping in ALMA

3.5.1 Example code for dynamic VME Master window

The following code simply allocates a master VME window of 64M to map to VME Board with VME ID = 2

```
#define REMOTE_VMEID      2

VME_STATIC_WIND pWind = {
        VME_MASTER,
        "VmeDma",
        0,
        0,
        0x01800000,
        0x04000000,
        VMEFG_AM_A32SDATA | VMEFG_READAHEAD | VMEFG_WRTPOST
        };

pWind.vmeAddr = (UINT32)(VME_DEFAULT_BASE + (REMOTE_VMEID *
VME_DEFAULT_INCREMENT));

sysVmeChannelFree("DramMaster0");
sysVmeChannelFree("Master2");

if (sysVmeChannelAlloc(&pWind) != OK)
        printf("Cannot Alloc Master VME Channel");
```

Chapter 4 - Accessing a VME Address or Getting a Local Address

This chapter defines how to access a VME address or to get a local address mapped to VME.

4.1 Description

When configuring master spaces, no parameter specifies the CPU address to which the VME space is mapped - only the VME address is specified. The actual local address used to map the window depends on the set of master windows required, since the master spaces are allocated dynamically at run-time.

To find out what local address to use to access a given VME address, the standard function `sysBusToLocalAdrs()` must be used, this will provide the correct local address to use to access the required VME address.

The reverse function `sysLocalToBusAdrs()` convert a local CPU address to the VME address for a specified VME slave (Address Modifier).

The function `sysVmeChannelShow()` is useful for displaying the set of mappings currently in use.

4.2 Routines

Routine Name	Description
`sysBusToLocalAdrs()`	Convert a bus address to a local address
`sysLocalToBusAdrs()`	Convert a local address to a bus address
`sysVmeChannelShow()`	Display VME master and slave window configuration

Chapter 5 - Default VME Board Configuration

This is the output example from `sysVmeChannelShow()`, showing the default bootRom and VxWorks configuration as shipped. In BSP no slave windows are opened by default to avoid VME conflict that could lead to hardware damage (if a VME slave channel maps an area that is already mapped by another then hardware damage can occurs when this VME mapping is accessed) and only one Master windows is opened.

```
-> sysVmeChannelShow

Master PCI-MEM to VME Bus
------------------------------------------------------------------------------
|   Name     | CPU Addr | PCI Addr | VME Addr | Size |   AM   |Conv| WP | RH |
------------------------------------------------------------------------------
|AlmaRemote  |0xed000000|0xed000000|0x00000000|   8Mb|  A16S  |ADDR| No | No |
------------------------------------------------------------------------------

Slave VME to PCI Bus
------------------------------------------------------------------------------
|Num|   Name     | VME Addr |   Addr   | Space | Size|   AM   |Conv| WP | RH |
------------------------------------------------------------------------------
|                        No VME to PCI CHANNEL                                |
------------------------------------------------------------------------------

2ESST Slave VME to PCI Bus
------------------------------------------------------------------------------
|Num|  Name  |      VME Addr    |   Addr   | Space | Size |   AM   |Conv|WP|RH|
------------------------------------------------------------------------------
|                    No 2eSST VME to PCI CHANNEL                              |
------------------------------------------------------------------------------
value = 0 = 0x0
->
```

1. The board automatically detects whether or not it is system controller.
2. « `AlmaRemote` » named channel allow ALMA chip to detect the presence of any other ALMA device on the VME bus. It allows the access to peer ALMA service like mailbox and semaphore. ALMA always open a corresponding slave windows in A16S to allows this feature. This specific windows is not displayed in `sysVmeChannelShow()`.

Chapter 6 - Simple Access on VME

Simple VME accesses can be performed directly at any CPU address mapping a VME address.

However a well known deadlock condition can occurs on any bus system coupling PCI/VME bus.

Deadlocks can occurred when a VMEbus initiated request to the PCI host collided with a PCI host request to the VMEbus. This conditions comes from the fact that PCI bus supports RETRY mechanism while VMEbus doesn't support it.

ALMA2f implements a solution to prevent this deadlock situations. It is providing a mechanism allowing a PCI bus master device to obtain the VMEbus ownership before it starts its access to ALMA2f. By doing so, the device is guaranteed that ALMA2f will not retry its transaction. This feature is implemented by means of routine `sysVmeBusTake()` and `sysVmeBusGive()`. These routines must by called before any simple VME access and any ALMA services that implement a simple VME Access (like shared ALMA Semaphore, Addressed Mailbox, VME Interrupt acknowledgment). On the contrary it must not be used when invoking DMA and 2eSST data transfer services.

6.1 VME Bus Take and Release services

Routines `sysVmeBusTake()` and `sysVmeBusGive()` must be invoked before and after Master VME simple access if a local Slave VME windows exists and is also used for external VME request answer.

Here is an example of simple VME access with :

- `vmeAddr` variable (char*) that is the VME address to access
- `cpuVmeAddr` variable (char *) that is the conversion of VME addr to CPU address in A32 S DATA VME address space
- `VME_AM_EXT_SUP_DATA = 0xd` is the value for A32 Supervisor DATA Vme Address space defined in windriver file `target/h/vme.h`
- `sysBusToLocalAdrs()` is the routine to convert a VME address to a CPU address
- `vmeData` will take the data read on VME. This access must be protected by `sysVmeBusTake()`, `sysVmeBusGive()`

```
if (sysBusToLocalAdrs (VME_AM_EXT_SUP_DATA, (char *)vmeAddr, (char **)&cpuVmeAddr) ==
ERROR)
{
            printf("sysBusToLocalAdrs ERROR on VME Channel");
            return ERROR;
}

/* Loop on VME Bus taking, use timeout for security if whished*/
while (sysVmeBusTake() != OK);
vmeData = *(UINT32*)(cpuVmeAddr); /*Perform VME simple access */
sysVmeBusGive();
```

If `sysVmeBusTake()` returns OK then we are sure the VME Bus has been Granted and simple access can be performed. If VME bus cannot be granted then `sysVmeBusTake()` will returns ERROR after a 500 micros delay and after having removed its Bus Request. It is the application responsability to recall this routine to perform the expected VME access.

It is absolutly necessary to call `sysVmeBusGive()` to free the VME Bus after the VME transaction failing to block any other VME external access. `sysVmeBusTake()` and `sysVmeBusGive()` must be used in the same manner than previously for MailBox and Shared semaphore routines (see corresponding chapters).

Chapter 7 - VME DMA

This function performs high-speed direct memory access (DMA) to/from the VMEbus.

7.1 Routines

Routine Name	Description
sysVmeDmaRun()	Perform a DMA transfer form or two VME bus
sysDmaEnable()	Turn on the DMA facility
sysDmaDisable()	Turn off the DMA facility

The sysVmeDmaRun() function is the low level VME DMA routine. It takes a structure as parameters that should allow it to fulfill most user requirements.

The parameters for sysVmeDmaRun() is of type:

```
/* DMA */
typedef struct vmeDmaStruct
   {
   UINT32 localAddr;       /* Onboard address to transfer from/to as seen by CPU */
   UINT32 vmeAddr;         /* Address to transfer to/from as seen on VME bus */
   UINT32 len;             /* Transfer size in bytes */
   UINT32 flags;           /* Flags information about the transfer */
   SEM_ID *userSem;        /* Address of a WRS semaphore to wait on */
   } IOCTL_DMA;
```

The structure field are described below:

<localAddr> A pointer to either the start address of the data in local memory (if writing to VME), or the start address of the buffer to write into (if reading from the VME). Use CACHE_DMA_MALLOC() or memalign() method to assure the buffer alignement.

<vmeAddr> A pointer to either the start address of the VME buffer to write into (if writing to VME), or the VME start address of the data to be read (if reading from the VME). This must be the real address that will appear on the VME bus, not a local CPU mapping.

<len> The number of bytes to transfer. No limit on size, but it must be a whole number of transfers - i.e. for 32-bit DMA, exactly divisible by 4, for 64-bit transfers, exactly divisible by 8.

<flags> Options that modify the nature of the DMA. There are several values passed in the flags variable.

The format is (defined in 3rdParty/Kontron//h/vxbAlma.h):

One of:

```
VMEFG_PCIMEM            0x00000004 - buffer is in PCI memory space
VMEFG_PCIIO             0x00000008 - buffer is in PCI IO space
VMEFG_DRAM              0x0000000C - buffer is in DRAM
```

If nothing is specified, the default is to use: VMEFG_DRAM

One of:

```
VMEFG_DMA_TOVME          0x00000000 - write to VME
VMEFG_DMA_FROMVME        0x00000400 - read from VME
```

If nothing is specified, the default is to use: VMEFG_DMA_TOVME.

One of:

```
VMEFG_AM_A32UMBLT        0x00800000 - use User mode D64 transfers
VMEFG_AM_A32SMBLT        0x00C00000 - use Supervisor mode D64 transfers
VMEFG_AM_A32UBLT         0x00B00000 - use User mode D32 block transfers
VMEFG_AM_A32SBLT         0x00F00000 - use Supervisor mode D32 block transfers
VMEFG_AM_A32SDATA        0x00D00000 - use Supervisor Data D32 transfers
MEFG_AM_A32UDATA         0x00900000 - use User Data D32 transfers
```

i.e. Any address modifier left-shifted 20 places. If none is specified, the default is to use: VMEFG_AM_A32SDATA

One of:

```
VMEFG_LEBE_NO            0x00000020  - no conversion
VMEFG_LEBE_ADDR          0x00000040  - maintain address coherency
VMEFG_LEBE_DATA          0x00000060  - maintain data coherency
VMEFG_LEBE_BTNS          0x00000080  - bytes translation with no swapping
```

If none is specified, the default is to use: VMEFG_LEBE_ADDR

A DMA transfer size - a byte showing the number of back-to back transfers to perform before backing off the bus and rearbitrating. Bigger usually means faster, but increases latency for anything else trying to use the bus. This value must be in the range: ((0x00..0xFF)<<12) i.e. 0x00000000 to 0x000FF000 - value left shifted 12 places.

If none is specified, the default is to use: 0x40

Any combination of:

```
VMEFG_DMA_NOWAIT         0x20000000 - return if no DMA channel free
VMEFG_WRTPOST            0x00000001 - use write posting
VMEFG_READAHEAD          0x00000002 - use read ahead
```

Default is to use none of these options.

If VMEFG_DMA_NOWAIT is not used, a DMA call will block if no DMA channel is available to start the DMA immediately.

If VMEFG_DMA_NOWAIT is set, if no channel is free, the call will return immediately, with an error of EWOULDBLOCK.

<userSem> A handler on a WRS semaphore (type SEM_ID) that will be posted to on completion of the DMA. If this value is NULL, then the posting does not occur. This allows the calling task to monitor the status of the DMA. It can be waited on, i.e. the calling task will block until DMA completion, or it can be polled, or, if used for multiple DMAs, the number of completed DMAs can be found by checking this value. The semaphore needs to be created before using it with a semCCreate(SEM_Q_FIFO,0) call.

Chapter 8 - VME 2eSST Support

The ALMA2f PCI-VME bridge has VME 2eSST features.

8.1 VME ALMA Bridge Configuration for 2eSST Support

As described in VME 2eSST specification, each VME device on the VME bus should be configured and should provides information about its VME 2eSST capabilities.

Just to know, the VME ALMA bridge on Kontron computers boards is configured through the constant `ALMA_2ESST_CONFIG` defined in `vme/vxbAlma.c` using the appropriate values. Of course changing this configuration would necessitate to recompile the VME library.

8.2 Slave VME 2ESST Windows

In 2eSST bus mode, ALMA only uses "slave windows" for being accessed. Master Window are not used any more like in VME64 bus mode. So you don't need to open Master windows for 2ESST.

A slave window is simply a mapping from VME space to local DRAM space. There are 8 windows for mapping VME to local DRAM.

The minimum granularity of VME slave spaces is 1 MB, i.e. a minimum of 1 MB of local DRAM space can be mapped onto a contiguous VME space. Thereafter, the size increases in powers of 2.

8.2.1 Routines for 2eSST Slave Windows

VME driver. support provides flexibility in managing slave windows allocation with the following routines. This can be done in a static manner in `hwconf.h` or using dynamic windows allocation at run time with routines:

Routine Name	Description
`sysVmeChannelAlloc()`	Create a VME to PCI mapping in ALMA
`sysVmeChannelFree()`	Delete a VME to PCI mapping in ALMA

The VME slave window mappings require following information. This is pretty similar to standard slave VME windows described previously except for the flags parameter which is used to indicate that this window is 2eSST.

To active a Static 2eSSTslave Windows that will be present at any boot then uncomment one of the Slave Windows in `hwconf.h` file structure.

```
VME_STATIC_WIND vmeStaticWindow [] ;
```

Each entry correspond to a VME_STATIC_WIND structure that must be filled with the following information:

```
/* VME windows */
typedef struct vmeStaticWindow
   {
   int       type;                    /* window type master or slave */
   char      name[WIND_NAMEMAX];      /* window name */
   UINT32    cpuAddr;                 /* CPU address */
   UINT32    pciAddr;                 /* PCI address */
   UINT32    vmeAddr;                 /* VME address */
   UINT32    size;                    /* size of window (in bytes)*/
   UINT32    flag;                    /* any particular flags */
   } VME_STATIC_WIND;
```

For each window, there are 6 values that need to be defined. These values are :

1. `Type`

   `VME_2ESST` define must be set for slave 2eSST windows

2. `name`

   The slave window name is limited to 12 characters. It can be displayed with `sysVmeChannelShow()` under VxWorks shell.

3. `cpuAddr`

   Must be set to zero for slave windows. Not used.

4. `pciAddr`

   The PCI start address that is mapped to by the slave address; for example: 0x00000000 is a mapping of VME to PCI address 0x00000000 for VM6050 (which happens to be the start of local memory on Pentium architecture) or 0x8000000 for Power PC Board (start of local address for Power PC Board). Since it is likely that most users will wish to map slave windows to the start of local RAM, a define named MAP_TO_RAM for parameter will just do that. If it is necessary to map the VME windows to the end of a huge RAM (1 or 2 GB) then this parameter must be set to a value correspond to the top of RAM minus the wished windows size and being 1M aligned. The top of RAM can be determined by routine sysMemTop();

5. `vmeAddr`

   It is the base A64 2ESST VME decoding address or the VME start address at which this slave window starts to respond. MAP_BY_VME_ID can be used to get the exact slave A32 address according to the processor number.

   For example, a value of (0x10000000) requests that slave window 1 start at 0x10000000 on VME.

   If this window is for A32 space, and this parameter is set to the value MAP_BY_VME_ID then the exact slave address that the board is configured for is dependent on the processor number, as defined in the VxWorks boot parameters. The value is calculated from :

   ```
   addressMapped = VME_DEFAULT_BASE + (sysVmeIdGet () * VME_DEFAULT_INCREMENT)
   ```

   where `VME_DEFAULT_BASE` and `VME_DEFAULT_INCREMENT` are defined in `vxbAlma.h` :

   ```
   #define VME_DEFAULT_BASE          0x08000000
   #define VME_DEFAULT_INCREMENT     0x08000000
   ```

This gives a table of default slave addresses as follows :

VME Board ID	VME Slave Address
0	0x08000000
1	0x10000000
2	0x18000000
3	0x20000000
4	0x28000000
5	0x30000000
6	0x38000000
7	0x40000000
8	0x48000000
9	0x50000000
a	0x58000000
b	0x60000000
c	0x68000000
d	0x70000000
e	0x78000000
f	0x80000000

6. `size`

   The size (in bytes) of the slave window (1 Mb aligned, power of 2). Therefore, it must be a whole number of a power of 2 megabytes, since that is the granularity of a slave window.
   So it is possible to map 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 or 1024 MB in a slave window, since these are powers of 2.

7. `Flags`

   Provides information about the mapping, the address space to use, the address modifiers to allow, whether posted writes are allowed etc …

   - 2eSST XAM Mode : `SST_VME_XAM_MODE` (mandatory)
   - Extended Address modifier selection: `SST_VME_XAM_A32` or `SST_VME_XAM_A64` or `SST_VME_XAM_A32B` or `SST_VME_XAM_A64B`
   - Source address area type: `SST_VME_DRAM` or `SST_VME_PCIMEM` (not implemented yet).
   - Litlle Endian - Big Endian convertion: `VMEFG_LEBE_NO` or `VMEFG_LEBE_ADDR` or `VMEFG_LEBE_DATA` or `VMEFG_LEBE_BTNS`
   - Write_Posted enable: `SST_VME_WRTPOST` to enable it.
   - Read Ahead Enable: `SST_VME_READAHEAD` to enable it.

   To free a VME slave window, you need to run the `sysVmeChannelFree()` routine giving in argument the window name used previously to create the windows.

8.2.2 Example of Define for Static 2eSST VME Slave Windows

To be defined in `hwconf.h` structure vmeStaticWindow.

```
{VME_2ESST, "2eSstA32",     0, 0x03000000, MAP_BY_VME_ID, 0x04000000,
             (SST_VME_XAM_MODE | SST_VME_XAM_A32 | SST_VME_DRAM |
         VMEFG_LEBE_ADDR | SST_VME_WRTPOST | SST_VME_READAHEAD),},
```

8.2.3 Example to Create 2eSST VME Slave Windows Dynamically

```
void slave2esst ()
{
      void *localAddr;
      VME_STATIC_WIND pWind = {VME_2ESST, "2eSstA32", 0, 0, 0x01000000, 0x04000000,
      (SST_VME_XAM_MODE | SST_VME_XAM_A32 | SST_VME_DRAM | VMEFG_LEBE_ADDR |
      SST_VME_WRTPOST | SST_VME_READAHEAD)};


      /*Reserve an  8M size  memory area aligned on 1M */
      localAddr = memalign (0x100000, 0x800000);

      if (localAddr == (void *)NULL)
      {
            printf ("Error mallocing 0x800000 aligned on 1M\n");
            return;
      }
      printf("Local Malloc =  CPU address :0x%08x\n", localAddr);

      memset(localAddr, 0xFF, 0x800000);

      /* Calculate the VME address according to our VME ID, defined in proc num boot
      parameter */
      pWind.vmeAddr = (UINT32)(VME_DEFAULT_BASE + (sysVmeIdGet() *
      VME_DEFAULT_INCREMENT));

      /* The slave window maps VME to reserved localAddr in RAM */
      pWind.pciAddr = localAddr;

      if (sysVmeChannelAlloc(&pWind) != OK)
            printf("Cannot Alloc Master VME Channel");
}
```

After that `sysVmeChannelShow` indicates

```
-> slave2esst
Local Malloc =  CPU address :0x1bf00000
value = 0 = 0x0
-> sysVmeChannelShow

Master PCI-MEM to VME Bus
-------------------------------------------------------------------------------
|   Name     | CPU Addr | PCI Addr | VME Addr | Size |   AM   |Conv| WP | RH |
-------------------------------------------------------------------------------
|DramMaster0 |0xe8000000|0xe8000000|0x08000000|  64Mb|A32SDATA|ADDR| No | No |
|Master2     |0xec000000|0xec000000|0x18000000|  16Mb|A32SDATA|ADDR| No | No |
|AlmaRemote  |0xed000000|0xed000000|0x00000000|   8Mb|  A16S  |ADDR| No | No |
-------------------------------------------------------------------------------

Slave VME to PCI Bus
-------------------------------------------------------------------------------
|Num|   Name     | VME Addr |   Addr   | Space | Size|   AM   |Conv| WP | RH |
-------------------------------------------------------------------------------
|                        No VME to PCI CHANNEL                                |
-------------------------------------------------------------------------------

2ESST Slave VME to PCI Bus
-------------------------------------------------------------------------------
|Num|  Name  |      VME Addr    |   Addr   | Space | Size |   AM   |Conv|WP|RH|
-------------------------------------------------------------------------------
| 0 |2eSstA32|0x0000000008000000|0x1bf00000|  DRAM |  64Mb|SST_A32 |ADDR| Y| Y|
-------------------------------------------------------------------------------
value = 0 = 0x0
```

8.3 2eSST VME DMA

8.3.1 Description

This feature performs high-speed direct memory access (DMA) to/from the VME bus.

Master CPU can initiate directly a 2ESST DMA transfer using the sysVmeDmaRun() driver routine.

8.3.2 Routines

Routine Name	Description
`sysVmeDmaRun()`	Perform a 2eSST DMA transfer using the ALMA PC-VME bridge
`sysDmaEnable()`	Turn on the DMA facility
`sysDmaDisable()`	Turn off the DMA facility

`sysVmeDmaRun()` takes a structure as parameters that should allow it to fulfill most user requirements.

The parameters for `sysVmeDmaRun()` is of type:

```
/* DMA */
typedef struct vmeDmaStruct
   {
   UINT32 localAddr;       /* Onboard address to transfer from/to as seen by CPU */
   UINT32 vmeAddr;         /* Address to transfer to/from as seen on VME bus */
   UINT32 len;             /* Transfer size in bytes */
   UINT32 flags;           /* Flags information about the transfer */
   SEM_ID *userSem;        /* Address of a WRS semaphore to wait on */
   } IOCTL_DMA;
```

The structure fields are described below for 2eSST:

`<localAddr>` A pointer to either the start address of the data in local memory (if writing to VME), or the start address of the buffer to write into (if reading from the VME).

`<vmeAddr>` A pointer to either the start address of the VME buffer to write into (if writing to VME), or the VME start address of the data to be read (if reading from the VME). This must be the real address that will appear on the VME bus, not a local CPU mapping.

`<len>` The number of bytes to transfer. No limit on size, but it must be a whole number of transfers - i.e. for 32-bit DMA, exactly divisible by 4, for 64-bit transfers, exactly divisible by 8.

`<flags>` Options that modify the nature of the DMA. There are several values passed in the flags variable. The format is (defined in `vxbAlma.h`):

One of:

```
VMEFG_PCIMEM            0x00000004 - buffer is in PCI memory space
VMEFG_PCIIO             0x00000008 - buffer is in PCI IO space
VMEFG_DRAM              0x0000000C - buffer is in DRAM
```

If nothing is specified, the default is to use: `VMEFG_DRAM`

One of:

```
VMEFG_DMA_TOVME         0x00000000 - write to VME
VMEFG_DMA_FROMVME       0x00000400 - read from VME
```

If nothing is specified, the default is to use: `VMEFG_DMA_TOVME`.

One of:

```
SST_VME_XAM_A32          0x00001100 - use 2ESST A32/D64 mode
SST_VME_XAM_A64          0x00001200 - use 2ESST A64/D64 mode
SST_VME_XAM_A32B         0x00002100 - use 2ESST A32/D64
```

Broadcast mode (not implemented yet)

```
SST_VME_XAM_A32B 0x00002200 - use 2ESST A64/D64 Broadcast mode
```

(not implemented yet)

One of:

```
VMEFG_LEBE_NO            0x00000020   - no conversion
VMEFG_LEBE_ADDR          0x00000040   - maintain address coherency
VMEFG_LEBE_DATA          0x00000060   - maintain data coherency
VMEFG_LEBE_BTNS          0x00000080   - bytes translation with no swapping
```

If none is specified, the default is to use: `VMEFG_LEBE_ADDR`

Any combination of:

```
VMEFG_DMA_NOWAIT         0x20000000 - return if no DMA channel free
VMEFG_DMA_TURBO          0x10000000 - use Alma TURBO mode
                                      (not implemented yet)
```

Default is to use none of these options.

If `VMEFG_DMA_NOWAIT` is not used, a DMA call will block if no DMA channel is available to start the DMA immediately.

If `VMEFG_DMA_NOWAIT` is set, if no channel is free, the call will return immediately, with an error of EWOULDBLOCK.

One of:

```
SST_VME_RATE_160         0x00000000 - use 160Mb/s VME 2esst rate compliance
SST_VME_RATE_267         0x00004000 - use 267Mb/s VME 2esst rate compliance
SST_VME_RATE_320         0x00008000 - use 320Mb/s VME 2esst rate compliance
```

`<userSem>` The address of a semaphore (type SEM_ID) that will be posted to on completion of the DMA. If this value is NULL, then the posting does not occur. This allows the calling task to monitor the status of the DMA. It can be waited on, i.e. the calling task will block until DMA completion, or it can be polled, or, if used for multiple DMAs, the number of completed DMAs can be found by checking this value. The semaphore needs to be created before using it with a `semCCreate(SEM_Q_FIFO,0)` call.

Chapter 9 - VME Interrupt Line Routing

9.1 VME Interrupt Reception and Generation

9.1.1 Description

The Kontron ALMA PCI to VME bridge provides the functionality of sending and receiving the seven standard VME bus interrupts.

The routine sysBusIntGen() is the sending routine that has two parameters, the first is the VME interrupt level with a range from one to seven. The second parameter is the VME vector. Because the sending ALMA register is an 8 bits register that contains both the VME interrupt level and the VME vector, the value of the 3 last bits of the VME vector must be the same as the level.

VECTOR							
					LEVEL		
7	6	5	4	3	2	1	0
V	V	V	V	V	V/L	V/L	V/L

V: Vector bits

L: Level bits

9.1.2 Routines

Routine Name	Description
`sysIntDisable()`	Turn on a VME bus interrupt level
`sysIntEnable()`	Turn off a VME bus interrupt. Level level can be 1 to 7 in range
`sysBusIntGen()`	Generate a VME bus interrupt
`sysBusIntCon-nect()`	Connect/Disconnect a routine to one of the seven VME interrupts

9.1.3 VME Interrupt Example

Interrupt routines can be used directly from VxWorks shell, example:

```
Board 1 -> sysIntEnable(1)
value = 0 = 0x0
Board 1 -> sysBusIntConnect(1,0x41,kkprintf,"\nIT VME 1 RECEIVED\n")
value = 0 = 0x0

Board 2-> sysBusIntGen(1,0x41)
value = 0 = 0x0
Board 2-> sysBusIntGen(1,0x41)
value = 0 = 0x0
Board2 -> sysBusIntGen(1,0x41)
value = 0 = 0x0
->


Board 1 ->
IT VME 1 RECEIVED

IT VME 1 RECEIVED

IT VME 1 RECEIVED
```

Chapter 10 - VME Mailbox

10.1 Description

The Kontron ALMA PCI to VME bridge provides the functionality of sending and receiving eight specific addressed interrupts, called addressed interrupt mailboxes.

When one of the mailboxes [0-7] register is addressed by VME (or locally) an interrupt is generated to the CPU. It is of course possible to connect a User routine to every mailbox addressed interrupt.

`sysVmeBusTake()` and `sysVmeBusGive()` service routines must be used before posting a mailbox (call to `sysVmeMailboxPost()`) as it is done in the case of a Simple VME access to prevent any deadlock condition if needed (See Chapter 6 page 15 - Simple Access VME).

10.2 Routines

Routine Name	Description
`sysVmeMailboxEnable()`	Enable a VME Addressed interrupt [0-7]
`sysVmeMailboxDisable()`	Disable a VME Addressed interrupt [0-7]
`sysVmeMailboxConnect()`	Connect/Disconnect a routine to the addressable interrupt
`sysVmeMailboxPost()`	Signal to a specified Mailbox on a specified VME target using VME ID and mailbox number [0-7]
`sysVmeMailboxAddrGet()`	Obtain the VME address of the inter board mailbox according the VME ID and mailbox number [0-7]
`sysVmeMailboxValGet()`	Obtain the value to write at the VME address returned by
`sysVmeMailAddrGet()`	Get Addr to signal a mailbox

10.3 Mailbox Examples

Mailbox can be used directly from VxWorks shell, for example:

Board 1 is VME ID = 0 (proc num = 0) and Board 2 is VME ID = 2 (proc num = 2)

```
Board 1->  sysVmeMailboxEnable(1)
value = 0 = 0x0

Board 1->   sysVmeMailboxConnect(1,kkprintf,"\nReceive mailbox 1\n")
value = 0 = 0x0

->


Board 2 ->  vmeid = 0
New symbol "vmeid" added to kernel symbol table.
vmeid = 0xbe8df50: value = 0 = 0x0

Board 2-> mailboxnum=1
New symbol "mailboxnum" added to kernel symbol table.
mailboxnum = 0xbed3fe0: value = 1 = 0x1

Board 2-> sysVmeMailboxPost(vmeid,mailboxnum)
value = 0 = 0x0
->

Board 1->
Receive mailbox 1
```

Mailbox interrupt 1 has been received on Board 1 VME ID 0

`sysVmeMailPost()` routine when run in full VME system with DMA must be protected using the `sysVmeBusTake()` and `sysVmeBusGive()` routines. See `mailboxVme.c` example file for a coding and test demonstration (or ask Kontron support).

Chapter 11 - VME Shared Semaphore

11.1 VME Read Modify Write Cycles

The ALMA PCI-to-VME bridge does not support VME RMW cycles. It cannot generate them as a master, and when accessed as a slave, the cycle is broken into individual read and write cycles. However, a mechanism exists for synchronizing VME boards using hardware semaphores. This mechanism uses standard read and write cycles and can greatly simplify interprocess and inter boards synchronization routines, enhancing system performance.

11.2 Description

The Kontron ALMA PCI to VME bridge has four eight bits semaphores that are very useful for synchronizing accesses and tasks between processors on the same board and between processors on separated boards on the VME backplane.

The locking mechanism is the following:

There are 4, 8-bit semaphore registers on an ALMA chip, accessible from A16 space. Each processor that wishes to "take" a semaphore writes a unique (to that processor) 7-bit value (with the top (control bit) set) to the semaphore and then reads it back. If the value read back is the same than the value written, this processor was successful in obtaining the semaphore. The value of the semaphore cannot be altered until cleared by writing a 0 in it.

At board power on the VME driver initializes a special slave VME window with address modifier A16 that provides access from the VME bus to the ALMA registers in particular the ALMA semaphore register. This has a fixed size of 0x100 and the VME address localization depends on the VmeID so on the proc num set into the boot parameter field.The address rule is :

ALMA Base Address Register (VME A16) = 0x100 * VmeId

VmeId	ALMA BAR (VME AM_A16)	VME slave window Range
0	0x0000	0x0000 to 0x00FF
1	0x0100	0x0100 to 0x01FF
2	0x0200	0x0200 to 0x02FF
3	0x0300	0x0300 to 0x03FF
4	0x0400	0x0400 to 0x04FF
5	0x0500	0x0500 to 0x05FF
x	0x0100 * X	(0x0100*X) to (0x0100 * (X+1)-1)

During VxWorks kernel init the driver scans the VME bus within a VME master window with AM A16 to find ALMA chip id for all VmeId values, and stores the remote ALMA semaphore VME address converted to local CPU address. `sysVmeSemShow()` displays the semaphores address and status (lock or unlock) of the local and remote semaphore, for example:

```
-> sysVmeSemShow
-----------------------------------
|   Semaphore status on VME bus   |
-----------------------------------
| VmeId | Sem # | Status |        |
-----------------------------------
|   00  |   00  | Unlock | Remote |
|   00  |   01  | Unlock | Remote |
|   00  |   02  | Unlock | Remote |
|   00  |   03  | Unlock | Remote |
|   02  |   00  | Unlock | Local  |
|   02  |   01  | Unlock | Local  |
|   02  |   02  | Unlock | Local  |
|   02  |   03  | Unlock | Local  |
-----------------------------------
value = 0 = 0x0
```

All the Kontron boards equipped with ALMA chip use the same scanning semaphore algorithm, so all the Kontron boards present on the VME bus have the same semaphore list in the same order.

11.3 Semaphore Routines

These routines provide an easy way to synchronize multiple tasks on CPUs on the same board or between board on VME backplane. The number of semaphore available on a system is 4 times the number of ALMA chips found during the VME bus scan (4 semaphores per ALMA). sysVmeBusTake() and sysVmeBusGive() service routine must be used like done for Simple VME access to prevent any deadlock condition (See Simple Access VME Chapter) if necessary.

Routine Name	Description
`sysVmeSemShow()`	Display ALMA semaphore status
`sysVmeSemTake()`	Attempt to lock a specific ALMA semaphore (use VME ID + Sema num)
`sysVmeSemGive()`	Clear a specific ALMA semaphore (use VME ID + Sema num)

11.4 VME Semaphore Example

Here is a very simple example with 2 VM6050 (one VmeID 0 and one VmeId 2).

First the semaphores list and their status is displayed with `sysVmeSemShow()`. Then semaphore number 3 on board VmeId 2 is taken from board VmeId 0 (with a timeout of 10 ms). The status is displayed again to check semaphore is taken and semaphore is given back.

All commands are entered from board VMEId = 0

```
-> sysVmeSemShow
-----------------------------------
|   Semaphore status on VME bus   |
-----------------------------------
| VmeId | Sem # | Status |        |
-----------------------------------
|   00  |   00  | Unlock | Local  |
|   00  |   01  | Unlock | Local  |
|   00  |   02  | Unlock | Local  |
|   00  |   03  | Unlock | Local  |
|   02  |   00  | Unlock | Remote |
|   02  |   01  | Unlock | Remote |
|   02  |   02  | Unlock | Remote |
|   02  |   03  | Unlock | Remote |
-----------------------------------
value = 0 = 0x0
-> sysVmeSemTake(2,3,10)
value = 0 = 0x0
-> sysVmeSemShow
-----------------------------------
|   Semaphore status on VME bus   |
-----------------------------------
| VmeId | Sem # | Status |        |
-----------------------------------
|   00  |   00  | Unlock | Local  |
|   00  |   01  | Unlock | Local  |
|   00  |   02  | Unlock | Local  |
|   00  |   03  | Unlock | Local  |
|   02  |   00  | Unlock | Remote |
|   02  |   01  | Unlock | Remote |
|   02  |   02  | Unlock | Remote |
|   02  |   03  |  Lock  | Remote |
-----------------------------------
value = 0 = 0x0
```

```
-> sysVmeSemGive(2,3)
value = 0 = 0x0
-> sysVmeSemShow
----------------------------------
|   Semaphore status on VME bus   |
----------------------------------
| VmeId | Sem # | Status |        |
----------------------------------
|   00  |   00  | Unlock | Local  |
|   00  |   01  | Unlock | Local  |
|   00  |   02  | Unlock | Local  |
|   00  |   03  | Unlock | Local  |
|   02  |   00  | Unlock | Remote |
|   02  |   01  | Unlock | Remote |
|   02  |   02  | Unlock | Remote |
|   02  |   03  | Unlock | Remote |
----------------------------------
value = 0 = 0x0
```

Chapter 12 - VME ACFAIL Signal

12.1 Description

The ALMA VME driver provides the following functions to control VME signals ACFAIL.

The user can define his own routine to be called on each ACFAIL VME signal occurrence and enable or disable ACFAIL VME signal handling.

12.2 Routines

Routine Name	Description
`sysAcFailEnable()`	Enable AC Fail interrupt line
`sysAcFailDisable()`	Disable AC Fail interrupt line
`sysAcFailConnect()`	Install an ISR to be called if AC Fail is detected

Chapter 13 - VME SYSFAIL Signal

13.1 Description

The ALMA VME Driver provides the following functions to control VME signals SYSFAIL. The user can define his own routine to be called on each SYSFAIL VME signal occurrence and enable or disable SYSFAIL VME signal handling.

13.2 Routines

Routine Name	Description
`sysSysFailEnable()`	Enable SYS Fail interrupt line
`sysSysFailDisable()`	Disable SYS Fail interrupt line
`sysSysFailConnect()`	Install an ISR to be called if SYSFAIL is detected

Chapter 14 - VME Bus ERROR and Probe

14.1 Description

The ALMA VME Driver provides the following routines to install a user-supplied ISR to be called in the event that the VME bus error interrupt is received. (i.e. VME BUS ERROR asserted) so the user may attempt to bring the system down gracefully, or do anything else required.

This routine is not called when using `vxMemProbe` on an invalid VME bus address.

14.2 Routines

```
* sysBusErrorConnect() - Install an ISR to be called if bus error is detected
  sysBusErrorConnect (
  FUNCPTR routine            /* routine called at each bus error event */
  UINT32 arg                 /* argument with which to call routine */
  )
```

Chapter 15 - Watchdog and Hardware Reset

15.1 Description

All Kontron boards equipped with ALMA chip have an hardware watchdog that is included in the ALMA chip. This watchdog is based on a 16-bit counter that is incremented every 4 milliseconds.

The routine `sysVmeWdStart()` can be used to set, reset and disable this watchdog. When the timer count is elapsed the ALMA chip invokes a general RESET of the board. To re-arm the watchdog timer before the specified timer count is reached call `sysVmeWdStart()` again with the same timer count. To cancel a watchdog timer before the specified timer count is reached call `sysVmeWdStart()` with timer count null.

The parameter value passed to `sysVmeWdStart()` is delay in milliseconds.

To include ALMA Watchdog functionality `INCLUDE_VME_DRV` must be defined in configuration file `config.h`. This is enabled by default.

`sysVmeReset()` routine can be used to hardware reset the board. If the used parameter is zero then the reset is local to the board. If the parameter is 1, `sysVmeReset(1)` then the reset is propagated to the Bus using `VME SYS_RESET` signal.

15.2 Routines

Routine Name	Description
`sysVmeWdStart()`	Start/Stop or Disable the ALMA watchdog timer
`sysBoardReset()`	Reset board (and possibly system)

MAILING ADDRESS
Kontron Modular Computers S.A.S.
150 rue Marcelin Berthelot - BP 244
ZI TOULON EST
83078 TOULON CEDEX - France

TELEPHONE AND E-MAIL
+33 (0) 4 98 16 34 00
sales@kontron.com
support-kom-sa@kontron.com

For further information about other Kontron products, please visit our Internet web site: www.kontron.com.