Documentation of a Linux driver and interface class for the Tundra Universe II PCI to VME bridge

Version 0.93, supporting Kernel 2.4.x and 2.6.x

Dr. Andreas Ehmanns

January 2006

http://universe2.sourceforge.net

Contents

1 Introduction

Writing software for VMEBus modules was very easy with 68K machines and operating systems like OS-9 or others. Accessing pointers with addresses outside the onboard memory (or reserved areas) were automatically transposed into cycles on the VMEBus. This transposition could be realized with a few standard IC's due to the similarity of the CPU-bus and VMEBus structure. Dumping the first 4096 bytes of an external memory at address 0xA2000000 was simply done by:

```
unsigned int *ptr;
ptr = (unsigned int *) 0xA2000000;

cout << hex;
for (i = 0; i < 1024; i++)
  cout << *ptr++ << " \n" ;
```

Since the production line of 68K CPUs reached its end with the 68060 chip more recent VMEBus CPUs (e.g. Intel or PowerPC based) are available. In most cases the implementation of the VMEBus on such boards is done with a PCI to VMEbus bridge from Tundra (www.tundra.com). The first version of this chip was called *Universe* the newer one *Universe II*. This chip is not only a bridge but provides many additional functions like interrupt handling, DMA transfers, FIFOs, mailboxes, semaphores and many more.
But there is no way to access the VMEBus directly like it is done in the previous example. A driver is needed which provides the user with access to the VMEBus and the other functions on the Universe chip.

For the Linux operating system there already exist two drivers.

- www.vmelinux.org (VMELinux project)
- http://lisa2.physik.uni-bonn.de/~hannappe/software/universe_doc/universe.html

Why writing a new driver?
First of all both drivers listed above are no longer maintained since quite a lot of time and additionally provide no support for 2.6.x kernel versions.
Secondly both drivers have the disadvantage that the user has to dive into the details of address mappings, calculation of different (virtual) addresses and all the related stuff.

To overcome this limitations this project provides a new driver together with an user friendly interface C++ class allowing access to the VMEBus with a set of VMEBus instructions but without the need of knowing details how to communicate with the driver. Especially all the dealing with virtual and mapped address is completely hidden from the user. Therefore the user only needs to think about VME addresses and what he wants to do with his VME module(s).

The actual driver of this project works with kernel versions 2.4.x and 2.6.x and provides:

- Access to VMEBus with A32/A24/A16, D32/D16/D8 and BLT/MBLT
- Access from VMEBus to PCI resources
- Easy-to-use read and write functions for VMEBus addresses
- A fast pointer-like access mode
- DMA transfers (including linked-list-operation)
- Interrupt handling for DMA, VMEBus irqs and mailboxes

2 Installation

Get a zipped archive of the complete software from *http://universe2.sourceforge.net*. Unzip and extract the archive with *tar xzvf universeII-0.93.tar.gz*. This will create a directory with name universeII and a small number of subdirectories which will be discussed in detail in the next sections.

2.1 Driver

Since the building of kernel modules significantly changed from kernel tree 2.4 to 2.6 there exist two subdirectories for the drivers. One for the 2.4.x kernel *universeII/driver-2.4* and one for the 2.6.x kernel tree *universeII/driver-2.6*. To compile the driver change to the corresponding subdirectory and type *make*. The object code (module) must then be inserted into the kernel with *insmod universeII.o* (2.4.x) *insmod universeII.ko* (2.6.x) or use the script *./ins*. Note that you must be root to do this! To remove the driver type *rmmod universeII* or *./uns*. Before you can use the driver you must create some devices under */dev*. This is simply be done by typing *make devices*. If you update from an older driver version to 0.93 or higher you have do delete the old devices with *rm /dev/vme** and create them again with *make devices*. The reason for this is the change of the vme driver device number from 70 to 221 to be compliant with the linux device number scheme.

A more elegant (and the prefered) way to load the driver is to use the module loader. The module is loaded automatically when one of the universeII devices is accessed. First add the following two lines to `/etc/modules.conf`:

```
# Universe II VMEBus driver
alias char-major-221 universeII
```

Then copy the driver (universeII.o or universeII.ko for 2.6.x) to
`/lib/modules/2.x.x/kernel/drivers/char` - where 2.x.x must be replaced by your actual kernel version - and last step type:

```
depmod -a
```

The driver has been compiled without any problems using gcc versions up to 3.3.5 2.95.2, and was tested on different Linux distributions up to Suse 10.0 and Debian 3.0 versions.
It was used with kernel versions up to 2.4.27/2.6.15 and worked properly. To check if the driver was successfully loaded have a look at */var/log/messages* and */proc/universeII*.

When the driver is loaded it sets the universeII to be VMEBus system controler. In cases where a VMEBus system controler is already installed in the crate, load the driver with option `sys_ctl=0`. This will prevent the driver to activate the onboard system controler. When using the module loader, add the following line to `/etc/modules.conf`:

```
options universeII sys_ctrl=0
```

There are some other VME parameters which can be set via options:

Option	Default	Range	Description
sys_ctrl	1	0..1	Sets VME board to be system controller
br_level	3	0..3	Sets the VMEBus request level
req_mode	0	0..1	VMEBus request mode, 0=demand, 1=fair
rel_mode	0	0..1	VMEBus release mode, 0=RWD, 1=ROR
vbto	3	0..7	VMEBus Time-out 0=disable 1=16μs 2=32μs 3=64μs 4=128μs 5=256μs 6=512μs 7=1024μs
varb	0	0..1	VMEBus Arbitration Mode, 0=Round Robin, 1=Priority
varbto	1	0..2	VMEBus Arbitration Time-out 0=disable Timer 1=16μs 2=256μs
vrai_bs			see comments below

In case of problems concerning the communication with a VMEBus module, first check if the module supports the VMEBus request level or the release mode which is used by the driver.

For external use of the four onboard mailboxes, the universeII registers must be accessible from VMEBus side. The base address can be set with:

```
options universeII vrai_bs=0xa0000000
```

In this case the registers can be access from VMEBus side with address 0xa0000000 + register offset. For more informations see chapter 4.7 mailbox access.

2.2 Driver interface

A user who wants to use the VMEbus is (in general) not interested in driver details. To hide the driver calls from the end-user there exists a driver interface with name *VMEBridge*. This C++ class provides the user with a set of functions which allow access to the VMEbus without the need of using driver calls explicitely.
To compile this class as a static and a shared library, change to the directory *universeII/vmelib* and type *make*. This creates the library *libvmelib.a* and *libvmelib.so.0.9.3*. To install the shared lib on your system type *make install*.

Due to changes in the kernel, it can not be avoided that the way of passing information from the interface class to the driver has slightly to be adapted from time to time. Therefore it is highly recommended to always use driver and interface class with same version numbers!

For simple VMEbus modules this class can be used directly. For more complex modules it is recommended to inherit a new class from the base class VMEBridge and to implement all module specific function in this new class.

2.3 VME debugger

An easy way to check and test VMEBus modules without writing test code are programs like the OS-9 debugger *debug*. A similar program that demonstrates the use of the class VME-Bridge and provides the VMEBus user with a simple but useful debugging tool can be found in *universeII/vmedebug*. With a small set of commands (start vmedebug and type help) it would become easy to test the VMEbus and most modules.

For A24 transfers replace the upper 8 bit (24..31) with ones, for A16 access replace the upper 16 bits with ones.
Example: To read from address 0xE00000 with A24, type `rl 0xFFE00000`.

2.4 Test and demonstration programs

A set of test and demonstration programs are delivered with this distribution to make you familiar with the use of driver and interface class. These files are the source code of the examples dicussed in chapter 4.You can find them in *universeII/test* and compile them with *make*.

3 General informations

This chapter describes in short words the principles of the universeII chip. For detailed informations take a look at the Tundra homepage (www.tundra.com) or the universe II manual.

3.1 Master and slave images

Communication between PCI and VMEBus is implemented by the concept of images. One can define an area of PCI addresses (image) which will be mapped to the VMEBus, or an area of VMEBus addresses which are mapped to the PCI bus. The first one is called master image, the second one slave image. The maximum number of images is eight for the master and eight for the slave images.
To access a VMEBus module one first sets up an image which includes the addresses of the module and then uses read and write commands to transfer data to/from the module.

3.2 DMA

For fast data transfers and block transfers between PCI and VMEBus the universeII chip includes a high performance onboard DMA. Operations between these two busses are decoupled by a bidirectional FIFO. The DMA provides normal data transfers cycles but also blocktransfers (BLT) or multiplexed blocktransfers (MBLT).
An additional feature is the linked-list operation mode. In this mode the DMA transfers a series of non-contiguous blocks of data without software intervention.

3.3 Interrupt handling

The universeII provides a flexible scheme to map interrupts from one bus to the other one. The interrupt output pins on PCI INT#[7:0] and on VMEBus VIRQ#[7:1] are software programmable. In addition interrupts can be generated from different hardware and software sources, e.g. DMA event, mailbox access, location monitor or error states like SYSFAIL* or ACFAIL*.

4 How to use this driver

To explain the usage of the driver and the VMEBridge class, let's start with a simple program which reads the first 4096 bytes of an external memory at VMEBus address 0xA2000000.
Note: In the next subsections this example will be expanded to introduce additional function like memory mapped transfers or DMA. All these test programs can be found in directory *universeII/test*.

4.1 Simple read and write functions

```
//test_simple.cpp

  #include <iostream.h>
  #include "vmelib.h"

  int main()
  {
    int image,i;
    unsigned int dummy32;
    VMEBridge vme;

    image = vme.getImage(0xA2000000, 0x10000, A32, D32, MASTER);
    if (image < 0)
    {
      cerr << "Can't allocate master image !\n";
      return 0;
    }

    cout << hex;

    for (i = 0; i < 1024; i++)
    {
      vme.rl(image, 0xA2000000 + i*4, &dummy32);
      cout << i << ": " << dummy32 << "!\n";
    }

}
```

The first step to do is the declaration of a class VMEBridge.

```
  VMEBridge vme;
```

This calls the constructor of this class which initializes some variables and checks that the driver, the universe control device and the DMA device are accessible.

Next step is to define a master image to access the VMEBus memory.

```
int getImage(unsigned int base, unsigned int size, int vas, int vdw, int ms);
```

The function getImage() allocates an image which allows access to VMEBus resources from address `base` to `base+size`. The parameters `vas` and `vdw` define the addessing type (A32/A24/A16/...) and the data with (D64/D32/D16/...)[1] used for VMEBus cycles. These types are defined in `vmelib.h` and should always be used. The last parameter `ms` indicates if this images should be a master or slave image. Predefined values are `MASTER` and `SLAVE` (How to use slave images is explained at the end of this chapter). Since the number of images that can be allocated by one application is limited only by the number of free images, the number of the allocated image is returned from `getImage(...)`. This number is needed as parameter for most functions of the VMEBridge class. If an error occured (e.g. no free image available) the returned image number is negative.

The (theoretical) maximum size of an image is the range of a 32-bit value but the main limitation is the availability of I/O memory. Have a look at *cat /proc/iomem* to see what is allocated by different resources (You will also find entries from the universeII driver here). When requesting a new image, the driver searches for free I/O address space to cover a mapping to the VMEBus. If no free address range can be found to cover the requested image size an error will be returned.
You see that it is a good idea to save resources and request an image with only that size which is really needed to handle the module.
The minimum size of an image (64K) is limited by the universeII chip and will automatically expanded to 64K if a smaller image size is requested.

Just to remind you: All the address parameters for the functions in the VMEBride class are VME addresses. Don't care about address mappings or re-calculation of addresses like you possibly know from other software. All this stuff is handled inside the VMEBridge class.

Note: Each available image can be programmed with any combination of `vas` and `vdw`. There is no predefined relation between image number and data/address width like it is done in other drivers.

When leaving the scope where `vme` is defined (in this case leaving the program) the destructor is called and all allocated images are released. The released image is free again and can be reallocated by the program or another application. Sometimes it can be useful to release an image manually, e.g. when it is needed no longer but the scope is not left. This can be done

[1] This is the maximum (!) date size on VMEbus. Example: Setting D32 allows also D16 and D8 cycles.

by calling

```
void releaseImage(int image);
```

where `image` is the image number returned by getImage(...).

For read or write transfers to/from a VMEBus module there are a set of six functions. The first letter indicates if the function performs a read or a write cycle, the second letter defines the data width. long for 32-bit transfers, word for 16-bit cycles and byte for 8-bit data.

```
int rl(int image, unsigned int addr, unsigned int *data);
int wl(int image, unsigned int addr, unsigned int *data);

int rw(int image, unsigned int addr, unsigned short *data);
int ww(int image, unsigned int addr, unsigned short *data);

int rb(int image, unsigned int addr, unsigned char *data);
int wb(int image, unsigned int addr, unsigned char *data);
```

Parameter `image` is the number returned by `getImage`, `addr` is the VMEBus address and `*data` contains the value to read/write[2].
Note: If you use an Intel based system, the data on the VMEBus data lines is swapped! (See chapter 6.1 for more information.)

In addition these simple functions can be used to transfer blocks of data. In this case there is one more parameter:

```
int rl(int image, unsigned int addr, unsigned int *data, int size);
int wl(int image, unsigned int addr, unsigned int *data, int size);

int rw(int image, unsigned int addr, unsigned short *data, int size);
int ww(int image, unsigned int addr, unsigned short *data, int size);

int rb(int image, unsigned int addr, unsigned char *data, int size);
int wb(int image, unsigned int addr, unsigned char *data, int size);
```

The size of the data block in bytes (!) has to be passed to this functions and `data` is always a pointer to the data buffer. This solution is much faster than calling rl/wl/... for each data word since the loop over the data block is done in the driver and only one driver call is needed (see also chapter 5 Performance tests).

[2]In Addition the functions wl, ww and wb can be used with passing parameter `data` as call-by-value instead of call-by-reference.

This is the easiest way to perform read and write cycles on the VMEBus but there is one disadvantage of this method. Depending on your CPU power the cycle time is in the order of a few microseconds (1...5 μs). For some applications this is no problem, but for system which are optimized for high speed data transfers (e.g. data acquisition systems) faster cycles are desirable. For transfers of complete blocks of data this can be done using the onboard DMA device. For all other cases there exists one additional method which will be discussed now.

4.2 Memory mapped transfers

In this mode accessing VMEBus modules is nearly the same like on 68K machines. The allocation of an image is still needed but the for-loop in the example above will be replaced:

```
//test_mmap.cpp

  #include <iostream.h>
  #include "vmelib.h"

  int main()
  {
    int image,i;
    unsigned int *ptr;

    VMEBridge vme;

    image = vme.getImage(0xA2000000, 0x10000, A32, D32, MASTER);
    if (image < 0)
    {
      cerr << "Can't allocate master image !\n";
      return 0;
    }

    ptr = (unsigned int *) vme.getPciBaseAddr(image);

    cout << hex;

    for (i = 0; i < 1024; i++)
      cout << "i: " << *ptr++ << "!\n";

  }
```

The first thing is to determine the PCI base address of the image specified by `image`. When setting up an image, the driver reserves an area of free PCI addresses which are mapped to the range of VMEBus addresses specified by `getImage`. These addresses can only be accessed by the kernel. When requesting a new image the driver maps these PCI addresses into user

space. The first address in user space corresponds to the first VMEBus address allocated by `getImage`. [3]
Accessing *ptr in the upper example means reading address 0xA2000000. Since ptr is declared as `unsigned int` the VMEBus cycle is D32. Using a pointer of `unsigned short` will lead to a D16 cycle (if the image was allocted with D16!). Write cycles are straight forward.

For more complex VMEBus modules just define a struct with the same address layout as implemented in the VMEBus module, define a pointer of the struct type, assign the base address and access the different register of the VMEBus module in this direct way.

But be warned! In this mode a direct communication with the universeII chip is used and therefore no support from the driver is available. This is the fastest way to perform VME transactions but no checking for bus errors and other *unexpected* cases is performed. Accessing a wrong address leads directly to a segmentation fault. For testing of a new module it's a safer way to start with simple read and write functions and after debugging change to faster cycles.

4.3 DMA transfers

The fastest data transfers to/from VMEBus can be achived using the onboard DMA. Due to the direct copy between VME-device and main memory there is no CPU power needed for the transfer. This is very important for data acquisition systems where multiple processes or threads are running. Two modes of DMA operation are available: *Direct Mode* and *Linked List Mode* which will be discussed seperately.
Common to both modes is the use of interrupts for indicating that DMA operation is finished. There is no polling on registers, so no CPU time is wasted by waiting for the DMA to complete. In addition there is a timer which aborts the DMA after one second if it is still in running or in hangup state. All implemented DMA calling function (DMAread(), DMAwrite() or execCmdPktList()) are blocking which means that one can be sure that the DMA job is done (or timed out) when returning from the call.

4.3.1 Direct mode

In direct mode the DMA registers are programmed directly. This is the standard mode for DMA transfers. Returning to our small example code, readout of the external memory can be done by:

```
//test_dma.cpp

  #include <iostream.h>
  #include "vmelib.h"

  int main()
```

[3] This is the only use-case where a non-VME address has to be handled by the application.

```
  {
    int i, offset;
    unsigned int base, *ptr;
    VMEBridge vme;

    base = vme.requestDMA();
    if (!base)
    {
      cerr << "Can't allocate DMA !\n";
      return 0;
    }

    vme.setOption(DMA, BLT_ON);

//----------------------------

    offset = vme.DMAread(0xA2000000, 4096, A32, D32);
    if (offset < 0)
      return;

    ptr = (unsigned int *) (base + offset);
    cout << hex;

    for (i = 0; i < 1024; i++)
      cout << "i: " << *ptr++ << "!\n";

//----------------------------

    vme.releaseDMA();

  }
```

Since the DMA can be used by several applications ownership of the DMA must be allocated and released with two functions:

```
int requestDMA(void)
void releaseDMA(void)
```

The architecture of Linux doesn't allow the universe DMA to transfer data directly to/from a buffer in user space. The solution to this restriction is the same used for slave images namely using a memory area of 128 kB which is allocated by the driver. This area is mapped to the user space of the application when calling `requestDMA()`. The return value is the address to the first byte of this memory area. All DMA transfers are executed to/from this buffer. After a

read call, the data is stored in this buffer, before calling the write function all data to transfer must be written to this buffer.
After a `releaseDMA` this buffer is no longer available and accessing it leads to a segmentation fault!
It is obvious that requesting and releasing the DMA device needs some time. So it should be calculated very well how often and at which places in your code the DMA will be allocated. If your application is the only one using the DMA device, requestDMA and releaseDMA should be called only once at the beginning and the end of your program.
The DMA read and write functions are:

```
int DMAread(unsigned int source, unsigned int count, int vas, int vdw);
int DMAwrite(unsigned int dest, unsigned int count, int vas, int vdw);
```

The parameters `source` and `dest` are the VMEBus addresses to read/write from/to, count is the number of bytes (!) to transfer, `vas` and `vdw` are the VMEBus addressing and data widths (see also function `getImage()`).

Return values are negativ in case of an error or greater/equal zero representing an byte-offset which in some cases is needed due to a limitation of the universeII chip when dealing with DMA cycles. Although the PCI and VMEBus addresses may be programmed to any byte aligned address, they must be 8-byte aligned to each other (e.g. the lower three bits of each must be identical)!!!
When starting DMA cycles (DMAwrite or DMAread) the driver checks for this limitation and in case of difference changes (increases) the PCI address to fulfill this condition. In this cases the data is not written/read to/from the base address returned by requestDMA() but `offset` bytes (!) higher (see example above).

In the above exmaple the 32 bit VME block transfer mode is enabled with the setOption-command (see chapter `Image options`). Make sure that your VME module support the BLT mode, there is no automatical fallback to non-BLT mode if the slave module does not support the BLT mode. If you want to use the multiplexed BLT mode (MBLT), call DMAread with D64 instead of D32/D16/D8.

Note: There is no need to allocate an image for using the DMA. The DMA has direct access to all possible VME addresses.

4.3.2 Linked List Mode

Unlike direct mode, in which DMA performs a transfer of a single block of data at a time, linked-list mode allows the DMA to transfer a series of non-contiguous blocks of data without software intervention. The DMA transfer parameters are stored in a linked list and each entry of this list is called `command packet`. The data structure for the command packet stores all necessary informations for the DMA register which are directly loaded from this structure.

Returning to our basic example with an external memory at 0xA2000000 a linked list can be created with the following code.

```
//test_llist.cpp

#include <iostream.h>
#include "vmelib.h"

int main()
{
  int i, list;
  unsigned int base, *ptr, packets[5];
  VMEBridge vme;

  list = vme.newCmdPktList();
  cout << "List number is " << list << "!\n";

  packets[0] = vme.addCmdPkt(list,0,0xa2000000,16,A32,D32);
  packets[1] = vme.addCmdPkt(list,0,0xa2000100,32,A32,D32);
  packets[2] = vme.addCmdPkt(list,0,0xa2000300,4,A32,D32);
  packets[3] = vme.addCmdPkt(list,0,0xa2000404,16,A32,D32);
  packets[4] = vme.addCmdPkt(list,0,0xa2000508,16,A32,D32);

  base = vme.requestDMA();

  cout << "Executing list ..." << flush;
  if (!vme.execCmdPktList(list))
    cout << "done !\n";

  ptr = (unsigned int *) (base + packets[0]);
  for (i = 0; i < 4; i++)
    cout << i << " = " << hex << *ptr++ << dec << "!\n";

  vme.releaseDMA();
  vme.delCmdPktList(list);

}
```

First step is creating a new list with

```
newCmdPktList();
```

The driver can manage up to 256 list and if no new list can be created, the returned list number is negativ. Elements (called command packets) can be added to this list with

```
addCmdPkt(int list, int rw, unsigned int vmeAddr, int size, int vas, int vdw);
```

and are limited in number only by the amount of available kernel memory. Since it is possible to deal with more than one list it must be specified to which list the command packets should be appended. The second parameter indicates if the DMA transfer is a read (rw = 0) or write cycle (rw = 1). Other values are not allowed. Next two parameters are the VMEBus source address and the size of the DMA transfer in bytes (!). The VMEBus address width and data width are set with `vas` and `vdw` for which predefined values exist (e.g. A32, D32, ...). The return value is the start address where the data are stored after a DMA read, or where the data are taken from for a DMA write cycle. Due to some restrictions of the universeII chip the data areas returned by multiple `addCmdPkt(...)` commands are NOT always contiguous (8-byte alignment). For each command packet there is an own memory area for the data. In addition these memory areas are shared between all lists and to avoid lost data, these areas are only accessible after requesting the DMA with `requestDMA()`! In addition make sure that you don't access these areas after releasing the DMA (see example above)!

This method seems to be a little bit complicated but the important advantage of it is the fact that a second copy process from kernel- to user space is avoided.

An existing list can be executed by calling

```
int execCmdPktList(int list);
```

And removing this list is done by

```
int delCmdPktList(int list);
```

For both functions the return parameter is zero for success and negative if an error occured.

4.3.3 DMA multi-buffer handling

When dealing with DMA and slave images it can be very useful to divide the 128kB wide buffer (or slave image) into several smaller buffers. This decouples transfers and speeds up data troughput when e.g. one task reads data from one buffer while another task already writes to another one.
Dividing the DMA memory space into several buffers is done by requesting DMA ownership with the number of buffers as parameter.

```
unsigned int requestDMA(int nrOfBuffers);
```

Where nrOfBuffers must be in [1..128] and of 2^n (1, 2, 4, 8, ...).

The known DMA write and read functions can be used as usual only the buffer number has to be passed in addition.

```
int DMAread(unsigned int source, unsigned int count, int vas, int vdw,
            unsigned int bufNr);
int DMAwrite(unsigned int dest, unsigned int count, int vas, int vdw,
             unsigned int bufNr);
```

There is a negativ return value if the buffer number is invalid or the transfer size exceeds the buffer size. But there is no implementation of a full buffer handler which distributes the buffers and checks for collisions. If this is needed, it has to be added by the user.

4.4 Slave images

Slave images are needed when another VMEBus master wants to access PCI resources. Such a master can be another VMEBus CPU, or a VMEBus module with an onboard microcontroller or DMA device. Allowing access to an area of PCI addresses can be done, by allocating a slave image. Example:

```
int imageSlave;
unsigned int slaveBase;

imageSlave = vme.getImage(0xA0000000, 0x10000, A32, D32, SLAVE);
if (imageSlave < 0)
{
  cerr << "Can't allocate slave image!!!\n";
  return;
}

slaveBase = vme.getPciBaseAddr(imageSlave);
```

Most work is be done by calling
`vme.getImage(0xA0000000, 0x10000, A32, D32, SLAVE)`
Now the universeII chip responds to VMEBus cycles with A32 and D32 and addresses in the range from 0xA0000000 to 0xA0010000. These addresses are translated into a memory area inside kernel space which is mapped to user space. Due to some kernel restrictions the size of a slave image is limited to 128 kB! The base address to this memory area is returned by `vme.getPciBaseAddr(imageSlave)` and corresponds in our example to VMEBus address 0xA0000000. The advantage of this method is that the data which was written/read by the external VMEBus master is directly accessible without additional copy activity between user- and kernel-space.

The universeII VME slave interface also allows block transfers to slave images. If the slave image was allocated as D32, the normal BLT mode is automatically supported, only for MBLT transfers the image needs to be allocated as D64.

4.5 VMEBus Interrupt handling

There are many applications where it is neccessary to wait for an external module to finish some work. This can simply be done by polling on a register on this module, but this wastes CPU time and can only be taken into account in applications where only one thread is running. A better way is to deal with VMEBus interrupt which are supported by nearly all more comlex VMEBus modules. The idea is not to poll on a register but to go to sleep until the VMEBus module creates an interrupt. This interrupt is serviced by the driver which then awakens the sleeping application. During the sleep process no CPU time is wasted and can therefore be used by another process.
The handling is a little bit complicated but if you need interrupts it is worth the work trying to understand it.

Before VMEBus interrupts can be used, the driver must be prepared in a way that he knows how to handle incoming interrupts. The needed function is declared as:

```
int setupIrq(int image, unsigned int irqLevel, unsigned int StatusID,
             unsigned int addrSt, unsigned int valSt,
             unsigned int addrCl, unsigned int valCl);
```

The VMEBus specification defines seven different interrupt levels VIRQ1 .. VIRQ7 and a 8 bit status/ID word which is read from the module during an interrupt acknowledge cycle. This gives the module the possibility to distinguish different interrupts with the same level. For using a VMEBus irq the corresponding irq level and the StatusID must be defined with `irqLevel` and `StatusID` in setupIrq. Any combination of VmeIrq and StatusID can be used but only once. In the case when a combination is already in use, setupIrq returns with an error.

Most VMEBus modules do not clear the VMEBus interrupt line when an interrupt acknowledge cycle is finished. Thus the interrupt handler is called again directly after servicing the previous interrupt. This locks the system and the only solution to break the lock is a reboot. To avoid this hangup, the driver must clear the VMEBus interrupt line before it finishes and the interrupt capability is enabled again. Most VMEBus modules clear the interrupt line when a specific register is addressed, or a specific value is written to this register. The parameters `addrCl` and `valCl` of setupIrq are used by the driver to clear this interrupt by writing the value of `valCl` to VMEBus address `addrCl`. The VMEBus address is an absolut address and not relativ to the base address of the image. Additionally it must lie in the address range which is covered by the image with number `image`.

The parameters `addrSt` and `valSt` are used by the driver to start the action on a VMEBus module which will produce the interrupt when it is finished.
Often VMEBus interrupt are used to indicate that the module has finished some work. This action is started by writing a specific value to a specific register in this module. When doing this in an user application and then calling the driver to sleep for the interrupt, it can happen

that the interrupt occures before the process went to sleep. This will result in an infinite sleep. This can be avoided by using `addrSt` and `valSt` from inside the driver. When calling waitIrq (see below) the driver first writes the value of `valSt` to VMEBus address `addrSt` and then goes to sleep. The difference is, that these two instructions are atomic and can not be interrupted by the scheduler.
If the VMEBus address of `addrSt` (or `addrCl`) is zero there will be no write action by the driver.

Now all preparation is done and waiting (and sleeping) for a VMEBus interrupt with VIRQ level `irqLevel` and status/ID word `StatusId` can be done by calling:

```
waitIrq(unsigned int irqlevel, unsigned int StatusID);
```

4.6 VMEBus Interrupt generation

In the previous chapter we discussed the handling of incoming VMEBus interrupts. Now we will have a look at the other situation where we generate VMEBus interrupt.
Calling the function

```
int generateVmeIrq(unsigned int irqLevel, unsigned int statusID);
```

generates a VMEBus interrupt with the given interrupt level. If there is an interrupt handler in the VMEBus crate the handler performs an interupt acknowledge cycle (IACK) and reads the status/ID. As soon as the IACK cycle is finished the functions returns. In other words: generateVmeIrq() is blocking (sleeping) as long as the generated VMEBus interrupt is not handled.

The statusID word can be every number between 1 .. 256 but with the limitation that only even numbers are allowed. This is a restriction from the universeII chip which uses the lowest bit to indicate if VMEBus interrupt are generated by software or are forwarded from PCI side. This bit is NOT writeable and in our situation always zero. Therefore the function generateVmeIrq() checks the statusID parameter and returns with an error if an odd number is passed.

The use of VMEBus interrupts is primarily dedicated for inter-module communication but can also be used for two processes running on the same system. Example:
You can start one application setting up and waiting for a given combination of interupt level and statusID using setupIrq() and waitIrq() and then run a second application calling generateVmeIrq() with the same combination. If this makes sense for inter-process communication should be decided by yourself but it's a good tool to test your application if you are using VMEBus interrupts.

4.7 Mailbox access

One on the new features of the universeII chip was the implementation of four 32-bit mailbox registers. These registers support access from either bus and can be enabled to generate interrupts on either bus when they are written to.

In this driver they can be used for communication with other VMEBus masters without polling on e.g. external memory locations.
One simple example: We want to wait (sleep) for an message from another VMEBus master which indicates that data is available or valid. First the access to the universeII registers from VMEBus side must be enabled as described in chapter Installation. This opens a 4K wide image allowing access to the registers in A32, A24 or A16 address space depending on the base address. If one of the upper eight bits of the base address (parameter vrai_bs) is set, the driver enables A32 cycles. If the upper eight bits are zero, A24 will be chosen and A16 if the upper 16 bits are zero. The offset of all registers relative to this base address can be found in the Tundra universeII manual. The four mailbox offsets are:

mailbox	offset
0	0x348
1	0x34C
2	0x350
3	0x354

Secondly a mailbox must be enabled to generate interrupts which can be done by calling

```
setupMBX(int mailbox);
```

with mailbox number (0..3) as parameter. Last step, waiting for an interrupt (which means write access to this mailbox) is included with

```
unsigned int waitMBX(int mailbox);
unsigned int waitMBX(int mailbox, unsigned int timeout);
```

This functions returns the (32-bit) value which was written to the mailbox and triggered the interrupt. In case of error 0xFFFFFFFF is returned, so make sure not to use this value for communication.

Using no timeout value, this functions returns after one second if no access to the mailbox has been done. If another timeout value (in seconds) is needed it can be passed as second parameter to `waitMBX`. Valid values are in [1..10000].

One short example code (test_mbx.cpp) to show the use of these functions.

```
#include <iostream.h>
#include "vmelib.h"

int main()
{
  VMEBridge vme;

  vme.setupMBX(2);
  cout << "Waiting for mailbox 2 ..." << flush;
  cout << "\nBack with message " << hex << vme.waitMBX(2,10) << dec << endl;

  vme.releaseMBX(2);
}
```

When this program is started, you will see the message
`Waiting for mailbox 2 ...`
and it sleeps without any CPU usage.

Now start vmedebug on the same or a second machine in your VME crate and type
`debug: wl 0xa0000350 12345678`

The test program returns with output: `Back with message 12345678`

After 10 seconds the function returns with 0xFFFFFFFF if no access to mailbox 2 has been done.
Releasing an allocated mailbox is not done automatically. So make sure to release the used mailbox when it's no longer needed with:

```
int releaseMBX(int mailbox);
```

4.8 Image options

When allocating an image with `getImage()` some options like address and data width can be set via parameters but the remaining available options are set to the most common values which are the right choice in most cases. Nevertheless in some cases it can be necessary to change some options. For this purpose the function `setOption()` is provided.

```
void setOption(int image, unsigned int options);
```

Following options are available (The columns labeled master, slave and DMA indicate for which image type this option is allowed):

Option	Master	Slave	DMA	Description
DATA_AM	x	x	x	Sets the AM code to data access
PROG_AM	x	x	x	Sets the AM code to program access
NON_PRIV_AM	x	x	x	Sets the AM code to non-privileged access
SUPER_AM	x	x	x	Sets the AM code to supervisory access
BLT_ON, BLT_OFF	x	-	x	Enables/Disables block transfers
POST_WRITE_EN	x	x	-	Enables posted write cycles
POST_WRITE_DIS	x	x	-	Disables posted write cycles
PREF_READ_EN	-	x	-	Enables prefetched read cycles
PREF_READ_DIS	-	x	-	Disables prefetched read cycles
PCI_LCK_RMW_EN	-	x	-	Enables PCI bus lock for VME RMW cycles
PCI_LCK_RMW_DIS	-	x	-	Disables PCI bus lock for VME RMW cycles

The default settings for a new created master image are DATA_AM, NON_PRIV_AM, BLT_OFF, and POST_WRITE_EN and for a slave image in addition PREF_READ_DIS and PCI_LCK_RMW_DIS. Setting more than one option with one function call can be done by OR-ing the options. Example:

```
setOption(image, SUPER_AM | BLT_ON);
```

Note that not all combination make sence: Using e.g. BLT_ON and BLT_OFF in one function call will result in disabling block transfers. In General: using ambiguous options will lead to the disabling of this option.

4.9 VMEBus error handling and test functions

VMEbus error handling is very poorly supported by the Tundra universeII chip itself and therefore NOT a lack of this driver. The functionality provided by the universeII chip is used to provide as much VMEBus error handling as possible. Depending on the used read/write functions, VMEBus errors are handled in different ways.

- For simple read/write functions there is a checking for bus errors implemented in the driver and the interface class. If a VMEBus access causes a bus error, a message will be printed to screen or to the redirected ostream (see chapter message logging).

If `posted write` is enabled for this image (option POST_WRITE_EN, which is default !!!) a bus error for write access will NOT be detected because the transfers between PCI and VME are decoupled. If this is not desired, disable posted write with `setOption(int image, POST_WRITE_DIS);` But note that this could decrease the transfer speed slightly.

- Memory mapped transfers: On some systems a bus error directly leads to a segmentation fault, on other systems this will be undetected. So it will be wise to use this mode after debugging the system.
- For DMA-transfers there is a checking for different errors (including bus errors) after completion. Error are printed to screen or are redirected.

There are two additional functions that deal with this topic. Checking if a module is accessible before reading or writing to it can be done with the following function:

```
int there(unsigned int addr);
```

First there will be a scan over all existing images to find out if this address is covered by any image. Then a read cycle with the attributes of the found image (address with, data width, ...) will be executed and finally a bus error check will be done.
Since this function always uses the maximum data width of the image supporting the requested address, additional functions have been introduced to allow tests with defined data width:

```
int there8(unsigned int addr);
int there16(unsigned int addr);
int there32(unsigned int addr);
```

These functions perform a read test with the requested data width (D8, D16 or D32) if the allocated image allows the desired data with.

The return value will be 0 if the check failed, any other value for success.

In cases where memory mapped transfers are used a check of the bus error status can be carried out using the function:

```
int testBerr();
```

A value of 1 is returned if an uncleared bus error was found, 0 in the other case. In addition a possible existing bus error is cleared by this function.
Caveat: By using this function one has to keep in mind that there is no information which cycle causes the bus error. Memory mapped transfers don't check (and don't clear!) this status because the driver is not involved in this type of transfers. When the driver is used by multiple

applications there is furthermore no information which application caused the bus error!

Note: For simple read and write functions as well as for DMA transfers this function makes no sense because in this cases a possible bus error is automatically reported and cleared and a call of this function will always return with 0.

4.10 Using the /proc filesystem

Status information about the driver and the used images can simply be achieved by using the /proc filesystem. Type

```
cat /proc/universeII
```

and see what happens.

Note: This interface is dedicated to status and debugging informations. Polling on this interface reduces the system and driver performance.

5 Performance tests

The creation of a stable driver was one important point in the developement. But in most cases there is a second question to deal with: How fast is the driver or what is the time of one complete VME cycle. To answer this question several screenshots of a VME tracer have been made to show the performance of the different types of read/write functions.

All the following tests are carried out using an intel PIII CPU running at 866 MHz and an 16MB external VME memory (MMI 6390). An older VMEBus tracer (VMETRO) with a sampling rate of 16 MHZ was used to provide the following timing lists. Tests with a P4 CPU at 1.4 GHz showed the expected result of faster VME cycles but for a better presentation of the differences between the available VME transfer types the results from the slower CPU have been choosen for this chapter.
Note: The specification of the VME external memory states a cycle time of ≥ 120 ns for read cycles.

5.1 Simple read/write functions

Using the functions `rl`, wl, rw, ... is the easiest way to generate VMEBus cycles but it's the slowest type of cycle since there is a driver call needed for every function call. In our test system this leads to nearly 1.3 μs for one VMEBus cycle.

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC
     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0        0   A2000000  12345678  W  LONG   OK   1111111  1  1  09
0001 !  1.73 us     0   A2000004  12345678  W  LONG   OK   1111111  1  1  09
0002 !  1.24 us     0   A2000008  12345678  W  LONG   OK   1111111  1  1  09
0003 !  1.30 us     0   A200000C  12345678  W  LONG   OK   1111111  1  1  09
0004 !  1.30 us     0   A2000010  12345678  W  LONG   OK   1111111  1  1  09
0005 !  1.30 us     0   A2000014  12345678  W  LONG   OK   1111111  1  1  09
0006 !  1.24 us     0   A2000018  12345678  W  LONG   OK   1111111  1  1  09
0007 !  1.30 us     0   A200001C  12345678  W  LONG   OK   1111111  1  1  09
0008 !  1.30 us     0   A2000020  12345678  W  LONG   OK   1111111  1  1  09
0009 !  1.30 us     0   A2000024  12345678  W  LONG   OK   1111111  1  1  09
0010 !  1.24 us     0   A2000028  12345678  W  LONG   OK   1111111  1  1  09
0011 !  1.30 us     0   A200002C  12345678  W  LONG   OK   1111111  1  1  09
0012 !  1.30 us     0   A2000030  12345678  W  LONG   OK   1111111  1  1  09
0013 !  1.30 us     0   A2000034  12345678  W  LONG   OK   1111111  1  1  09
0014 !  1.24 us     0   A2000038  12345678  W  LONG   OK   1111111  1  1  09
0015 !  1.24 us     0   A200003C  12345678  W  LONG   OK   1111111  1  1  09

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

5.2 Simple block read/write functions

For several cycles with ascending VME addresses (a complete block of data) the same functions as above can be used, but additionally the number of bytes (!) to transfer has to be passed to the functions. In this case there is only one driver call since the loop over the data block in done by the driver. In our example this increases the data rate by a factor of two.

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC

     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0        0   A2000000  00000000  W  LONG   OK   1111111  1  1  09
0001 !  0.74 us     0   A2000004  00000001  W  LONG   OK   1111111  1  1  09
0002 !  0.68 us     0   A2000008  00000002  W  LONG   OK   1111111  1  1  09
0003 !  0.68 us     0   A200000C  00000003  W  LONG   OK   1111111  1  1  09
0004 !  0.68 us     0   A2000010  00000004  W  LONG   OK   1111111  1  1  09
0005 !  0.68 us     0   A2000014  00000005  W  LONG   OK   1111111  1  1  09
0006 !  0.68 us     0   A2000018  00000006  W  LONG   OK   1111111  1  1  09
0007 !  0.68 us     0   A200001C  00000007  W  LONG   OK   1111111  1  1  09
0008 !  0.68 us     0   A2000020  00000008  W  LONG   OK   1111111  1  1  09
0009 !  0.68 us     0   A2000024  00000009  W  LONG   OK   1111111  1  1  09
0010 !  0.68 us     0   A2000028  0000000A  W  LONG   OK   1111111  1  1  09
0011 !  0.68 us     0   A200002C  0000000B  W  LONG   OK   1111111  1  1  09
0012 !  0.68 us     0   A2000030  0000000C  W  LONG   OK   1111111  1  1  09
0013 !  0.68 us     0   A2000034  0000000D  W  LONG   OK   1111111  1  1  09
0014 !  0.68 us     0   A2000038  0000000E  W  LONG   OK   1111111  1  1  09
0015 !  0.68 us     0   A200003C  0000000F  W  LONG   OK   1111111  1  1  09

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

5.3 Memory mapped transfers

The fastest transfers can be achieved using memory mapped access to the VMEBus. But one has to be aware of the fact, that there is no checking for bus error since the driver is not involved into this cycles (and this is the reason why it's faster)! But take into account, that these times are extremly depending on the CPU load. In our example there was no other user application running.

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC

     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0        0   A2000000  00000000  W  LONG   OK   1111111  1  1  09
0001 !  0.37 us     0   A2000004  00000001  W  LONG   OK   1111111  1  1  09
0002 !  0.24 us     0   A2000008  00000002  W  LONG   OK   1111111  1  1  09
0003 !  0.18 us     0   A200000C  00000003  W  LONG   OK   1111111  1  1  09
0004 !  0.24 us     0   A2000010  00000004  W  LONG   OK   1111111  1  1  09
0005 !  0.24 us     0   A2000014  00000005  W  LONG   OK   1111111  1  1  09
0006 !  0.18 us     0   A2000018  00000006  W  LONG   OK   1111111  1  1  09
0007 !  0.24 us     0   A200001C  00000007  W  LONG   OK   1111111  1  1  09
0008 !  0.24 us     0   A2000020  00000008  W  LONG   OK   1111111  1  1  09
0009 !  0.37 us     0   A2000024  00000009  W  LONG   OK   1111111  1  1  09
0010 !  0.24 us     0   A2000028  0000000A  W  LONG   OK   1111111  1  1  09
0011 !  0.18 us     0   A200002C  0000000B  W  LONG   OK   1111111  1  1  09
0012 !  0.24 us     0   A2000030  0000000C  W  LONG   OK   1111111  1  1  09
0013 !  0.24 us     0   A2000034  0000000D  W  LONG   OK   1111111  1  1  09
0014 !  0.18 us     0   A2000038  0000000E  W  LONG   OK   1111111  1  1  09
0015 !  0.37 us     0   A200003C  0000000F  W  LONG   OK   1111111  1  1  09

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

5.4 DMA

For transfers of complete data blocks the universeII onboard DMA has the advantage of being fast, having the ability to use VME block transfer modes and there is no intervention needed by the CPU. Comparing these values with the example above, there is (at the first look) no gain in transfer speed, but in the DMA case the cycle times are independent of the CPU load!

But one should remember, that this transfer is only preferable when larger data block are involved. Settingx up the DMA needs time and one has to balance this against normal transfers.

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC

     !   TIME      BUS   ADDRESS    DATA    R/W SIZE  STAT   IRQ*    IACK*  AM
     !   rel.     LEVEL                                      7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0         0   A2000000  00000000  W  LONG   OK   1111111  1  1  09
0001 !  0.18 us      0   A2000004  00000001  W  LONG   OK   1111111  1  1  09
0002 !  0.24 us      0   A2000008  00000002  W  LONG   OK   1111111  1  1  09
0003 !  0.24 us      0   A200000C  00000003  W  LONG   OK   1111111  1  1  09
0004 !  0.24 us      0   A2000010  00000004  W  LONG   OK   1111111  1  1  09
0005 !  0.24 us      0   A2000014  00000005  W  LONG   OK   1111111  1  1  09
0006 !  0.24 us      0   A2000018  00000006  W  LONG   OK   1111111  1  1  09
0007 !  0.24 us      0   A200001C  00000007  W  LONG   OK   1111111  1  1  09
0008 !  0.24 us      0   A2000020  00000008  W  LONG   OK   1111111  1  1  09
0009 !  0.24 us      0   A2000024  00000009  W  LONG   OK   1111111  1  1  09
0010 !  0.24 us      0   A2000028  0000000A  W  LONG   OK   1111111  1  1  09
0011 !  0.24 us      0   A200002C  0000000B  W  LONG   OK   1111111  1  1  09
0012 !  0.18 us      0   A2000030  0000000C  W  LONG   OK   1111111  1  1  09
0013 !  0.24 us      0   A2000034  0000000D  W  LONG   OK   1111111  1  1  09
0014 !  0.24 us      0   A2000038  0000000E  W  LONG   OK   1111111  1  1  09
0015 !  0.18 us      0   A200003C  0000000F  W  LONG   OK   1111111  1  1  09

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

5.5 Blocktransfer, BLT and MBLT

Using VME block transfers increases the data rate typically by a factor of two or three since only one address cycle is needed at the begin of the transfer.

Note that the used VME tracer runs with a sampling rate of 16 MHz (62.5ns) and therefore reaches it's limitations of measuring cycle times in this example.

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC

     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0        0   A2000000  00000000  W  LONG   OK   1111111  1  1  0B
0001 !  0.18 us     0   A2000004  00000001  W  LONG   OK   1111111  1  1  0B
0002 !  0.12 us     0   A2000008  00000002  W  LONG   OK   1111111  1  1  0B
0003 !  0.12 us     0   A200000C  00000003  W  LONG   OK   1111111  1  1  0B
0004 !  0.12 us     0   A2000010  00000004  W  LONG   OK   1111111  1  1  0B
0005 !  0.12 us     0   A2000014  00000005  W  LONG   OK   1111111  1  1  0B
0006 !  0.12 us     0   A2000018  00000006  W  LONG   OK   1111111  1  1  0B
0007 !  0.12 us     0   A200001C  00000007  W  LONG   OK   1111111  1  1  0B
0008 !  0.12 us     0   A2000020  00000008  W  LONG   OK   1111111  1  1  0B
0009 !  0.12 us     0   A2000024  00000009  W  LONG   OK   1111111  1  1  0B
0010 !  0.12 us     0   A2000028  0000000A  W  LONG   OK   1111111  1  1  0B
0011 !  0.12 us     0   A200002C  0000000B  W  LONG   OK   1111111  1  1  0B
0012 !  0.12 us     0   A2000030  0000000C  W  LONG   OK   1111111  1  1  0B
0013 !  0.12 us     0   A2000034  0000000D  W  LONG   OK   1111111  1  1  0B
0014 !  0.12 us     0   A2000038  0000000E  W  LONG   OK   1111111  1  1  0B
0015 !  0.12 us     0   A200003C  0000000F  W  LONG   OK   1111111  1  1  0B

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

These cycle times (120 ns) are exactly what the specification of the external VME memory states as the minimum cycle time.

For MBLT (D64) VME cycles the timing is the same as above but the number of transferred bytes is 8 insted of 4 increasing the data rate by a factor of two.

5.6 Linked list operation

In the linked list operating mode the single DMA cycles are as fast as in normal mode but there is time needed when a new packet has to be loaded. This time is in the order of 3-4μs where the time of one cycles has to be subtracted.

Note that in this example the time for creating a linked list is not taken into account!

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC

     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-------------------------------------------------------------------------
TRIG !  00.0        0   A2000000  00000000  W  LONG   OK   1111111  1  1  09
0001 !  0.24 us     0   A2000004  00000001  W  LONG   OK   1111111  1  1  09
0002 !  0.24 us     0   A2000008  00000002  W  LONG   OK   1111111  1  1  09
0003 !  0.18 us     0   A200000C  00000003  W  LONG   OK   1111111  1  1  09
0004 !  3.90 us     0   A2000100  00000000  W  LONG   OK   1111111  1  1  09
0005 !  0.24 us     0   A2000104  00000001  W  LONG   OK   1111111  1  1  09
0006 !  0.24 us     0   A2000108  00000002  W  LONG   OK   1111111  1  1  09
0007 !  0.24 us     0   A200010C  00000003  W  LONG   OK   1111111  1  1  09
0008 !  0.24 us     0   A2000110  00000004  W  LONG   OK   1111111  1  1  09
0009 !  0.18 us     0   A2000114  00000005  W  LONG   OK   1111111  1  1  09
0010 !  0.24 us     0   A2000118  00000006  W  LONG   OK   1111111  1  1  09
0011 !  0.24 us     0   A200011C  00000007  W  LONG   OK   1111111  1  1  09
0012 !  3.28 us     0   A2000300  00000000  W  LONG   OK   1111111  1  1  09
0013 !  4.15 us     0   A2000404  00000000  W  LONG   OK   1111111  1  1  09
0014 !  0.31 us     0   A2000408  00000001  W  LONG   OK   1111111  1  1  09
0015 !  0.55 us     0   A200040C  00000002  W  LONG   OK   1111111  1  1  09

Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

5.7 Interrupt service time

Since the used linux distribution was not a real time version the interrupt service time question came up. In the following example an additional VME master write data into a slave image. After the last cycle (nr. 948) an interrupt is generated. The VME interrupt is serviced by the universeII chip which means reading the status/ID word (cycle nr. 949) and generating the corresponding PCI interrupt. Afterwards the interrupt servies routine of the universeII driver is called which first checks that the interrupt came from the universeII chip. This is necessary since we allow the use of shared interrupts. Then this routines cleares the still active VME interrupt by writing to the a specific register of the second VME master (cycle 950). If there is a sleeping process waiting for this interrupt and this combination of VME interrupt level and status/id word, this process will be waken up and the service routine is left. Cycle nr. 951 is the first instruction in the user application, so that the complete time from creating a VME interrupt until waking up can be calculated. For our example it's in the order of twelve microseconds.
It is obvious that these timings are extremly hardware depended and this example should only be used to get a feeling for the order of the interrupt service time!

```
V M E b u s   T r a c e ,   group 1 of 3      Sampling mode : SYNC
     !   TIME     BUS   ADDRESS    DATA   R/W SIZE  STAT   IRQ*   IACK*  AM
     !   rel.    LEVEL                                    7654321 OC I/O
-----!-----------------------------------------------------------------------
0948 !  0.18 us    3   B05E00EA  B05E0119  W  LONG   OK   1111111  1  1  08
0949 !  2.10 us    0   0000000F  xxxxxx02  R  LBYTE  OK   0111111  0  0  29
0950 !  5.89 us    0   00E02238  00000000  W  LONG   OK   1111111  1  1  39
0951 !  4.21 us    0   00E0220C  FFFFA033  R  LONG   OK   1111111  1  1  39
Time abs/rel:T Scroll:<,> Other group:X Jump:J Search:S Print:P Quit: Q
```

6 Miscellaneous

6.1 Byte swapping

The universeII always performs address invariant translation between the PCI and the VME-Bus. Depending on the used platform, the byte ordering in the application can be different from the byte ordering on the VMEBus. This is the case e.g. for intel based platforms where the byte ordering is `little endian` whereas the VMEbus uses `big endian`.

Since the driver is written to run under linux and under different platforms (Intel, PowerPC, Alpha, ...) it does not do any byte swapping by itself. In addition the VMEBridge class leaves the byte unchanged. The swapping (if wished) must be done by the user application. Before writing to or after reading data from VMEBus this data must be swapped, which can be done by the following two functions (note that the byte swapping for 16 bit and 32 bit words is different!):

```
unsigned int swap32(unsigned int val)
{
  return ((val & 0xFF000000) >> 24) | ((val & 0x00FF0000) >> 8) |
         ((val & 0x0000FF00) << 8) | ((val & 0x000000FF) << 24);
}

unsigned short swap16(unsigned short val)
{
  return ((val & 0xFF00) >> 8) | ((val & 0x00FF) << 8);
}
```

There are some manufactures who add a hardware swapping device to their boards, so there is no need to care about it. See also chapter 6.5.

6.2 Compiler Optimization

Be careful when using compiler optimization! Make sure that your program runs without any problems before switching on optimization.
Many VMEBus boards use a technique which is called "'key-address"' technique. This means that reading a specific address on this module causes it to do some action. The value of the read cycle is without interest. One simple example:

```
void doSomeAction(void)
{
  unsigned int dummy32, *ptr;

  *ptr = (unsigned int *) (MODULE_BASE);
  dummy32 = *ptr;
}
```

`MODULE_BASE` is the mapped VMEBus address of the module (see chapter memory mapped access). This small function reads a 32-bit word but leaves the function without using the read value. When compiler optimization is switched off (-O0) this code works without problems. Using optimization (-O2) this functions does not perform any cycle on the VMEBus!
The compiler knows nothing about key-address technique. It seems that a memory location is read but nothing will be done with the read value. So the instruction `dummy32 = *ptr;` is removed from the code!
This can be avoided by using the `volatile` attribute.

```
volatile unsigned int dummy32;
```

The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. So it prevents the compiler doing any optimization with instructions where dummy32 is involved.

In addtion, using loops together with optimization can lead to unexpected results. Let's assume that reading an specific address triggers a VME module to do some action and we want to do this 100 times.

```
void doSomeAction(unsigned int)
{
  int i;
  unsigned int dummy32, *ptr = (unsigned int *) (MODULE_BASE);

  for (i = 0; i < 100; i++)
    dummy32 = *ptr;

  return dummy32
}
```

Since we return dummy32 the read intruction will not be removed by the compiler but the loop. For the compiler it makes no sence to read a memory location several times without modifying it. Thus the read instruction will be executed only one time.

6.3 memcpy and others

Be careful using builtin functions like memcpy. Passing a memory mapped VME address to this function leads to a byte-wise (!) copy process. Most parameters of memory handling functions (copy memory block, clear memory, ...) are declared as char * and only a small number of VME modules support D8 cycles. Normally it will be faster to readout data with D16 or D32.

6.4 Message logging

In some cases it can be useful to redirect message output of the VMEBridge interface class from screen to files, a logging system or something else. For this purpose there are two functions implemented which can be used to redirect output from `cout` and/or `cerr` (which is the default) to anything of type `ostream *`.

```
void setStdlog(ostream *log)
void setErrorlog(ostream *log)
```

6.5 VMIC define

There are some manufacters which solve the problem with byte-swapping by adding an additional chip between VME- and PCI-bus. This chip can be programmed to do byte swapping or not. For VMEBus CPUs boards from VMIC byte swapping can be switched on with a `#define VMIC` in the driver. This solution is very fast, since the byte-swapping is done by hardware and not by software. In addition it makes code easier, more elegant and avoids programming errors.

6.6 readUniReg, writeUniReg

These two functions allow read and write access to registers of the universeII chip which are not supported by the driver. But be careful and make sure that the register is not used by the driver or unexpected things may happen.

```
unsigned int readUniReg(int reg)
void writeUniReg(int reg, unsigned int data)
```

The register to read/write from/to must be passed to `reg` and is the offset address of the register (see Tundra User Manual for details of register offsets). Example: Read the Location Monitor Base Address Register (LM_BS) with

```
cout << "LM_BS = " << readUniReg(0xF68) << "!\n";
```

Take a look at the Tundra manual for more details.

6.7 VMEBus SYSRESET*

The VMEBus SYSRESET* signal is implemented on most VMEBus modules and can be used for reset functions or whatever the designer of the module assigned to it. This is a most common way to bring modules back in a defined state after crashes or hangups. For CPUs this SYSRESET* signal has the same effect as pressing the reset button at the front panel. In some cases it makes no sence that all modules react to this signal. For this reason the most VMEBus modules have jumpers onboard which define if the module should react to a SYSRESET* or

not. For every module one should have a look in the manual to check how to enable this function and what this signal triggers.
The corresponding function is:

```
void vmeSysReset(void);
```

Note: When your CPU is jumpered to allow incoming SYSRESET* it will reboot when this signal occurs! Even if the CPU generated this signal itself by calling the upper function.
On some boards there is an additional jumper to allow the generation of an outgoing SYSRESET* signal. Make sure that all these jumpers are in the right postion to avoid undesired reset signals.

6.8 Driver reset

During test and debugging of applications it can happen that something hangs or crashes. In this case the allocated images or the universeII DMA remain allocated and can't be used anymore. Removing the driver with `rmmod` is one possibility to reset the driver but one has to be root to do this.
A better solution has been implemented in the interface class with the function:

```
int resetDriver();
```

This resets the driver by doing the following actions:

- Release all allocated images.
- Release DMA.
- Remove all DMA command packet lists.
- Clear all previous error bits.
- Release allocated mailboxes.
- Free all irq/StatusID combinations and finally
- Reset all statistic counters.

But be sure to exit or kill all existing programs which use the universeII driver before calling this function!

7 Quick reference

```
VMEBridge();
~VMEBridge();
int bridge_error;
```

The constructor initalizes some internal variables and checks that the control and the DMA device of the universe driver are accessible. Error codes (< 0) are stored in the variable `bridge_error`.

The destructor removes all existing lists (see DMA linked list operation) and closes all open images or other devices. Releasing an allocated image (see function releaseImage) which is not needed anymore can be done with the release function or will be at least done by the destructor.

```
int getImage(unsigned int base, unsigned int size, int vas, int vdw, int ms);
```

Allocates an image from address `base` to `base`+size for VME cycles with address and data size defined by `vas` and `vdw`. Setting the allocated image as master or slave image can be done with parameter `ms` (use the predefined values MASTER or SLAVE). Return value greater or equal zero is the number of the allocated image. Values smaller zero are error codes.

```
void releaseImage(int image);
```

An allocated image is automatically released by the destructor. In some cases it is necessary to release an image by hand. This can be done with this function where `image` is the number of the image to release.

```
unsigned int getPciBaseAddr(int image);
```

This function returns an address in user space which corresponds to the lowest VMEBus address of image `image` and is needed for memory mapped transfers.

```
void setOption(int image, unsigned int opt);
```

The option for an image are set to the mostly needed values. If other settings are required this function provides the necessary interface. Modifies options like block transfers on/off of image `image`. Use the predefined values/options in vmelib.h.

```
// Read/Write access to VME resources, data size 1,2,4 byte(s)

    int rl(int image, unsigned int addr, unsigned int *data);
    int wl(int image, unsigned int addr, unsigned int data);
```

```
    int rw(int image, unsigned int addr, unsigned short *data);
    int ww(int image, unsigned int addr, unsigned short data);

    int rb(int image, unsigned int addr, unsigned char *data);
    int wb(int image, unsigned int addr, unsigned char data);

// Block read/write functions

    int rl(int image, unsigned int addr, unsigned int *data, int size);
    int wl(int image, unsigned int addr, unsigned int *data, int size);

    int rw(int image, unsigned int addr, unsigned short *data, int size);
    int ww(int image, unsigned int addr, unsigned short *data, int size);

    int rb(int image, unsigned int addr, unsigned char *data, int size);
    int wb(int image, unsigned int addr, unsigned char *data, int size);
```

For read or write transfers to/from a VMEBus module there are a set of functions. The first letter indicates if the function performs a read or a write cycle, the second letter defines the data width. long for 32-bit transfers, word for 16-bit cycles and byte for 8-bit data. If parameters size is passed to the function, a transfer of a complete data block will be performed.

```
// Test and VMEBus error related functions

    int there(unsigned int addr);
    int there8(unsigned int addr);
    int there16(unsigned int addr);
    int there32(unsigned int addr);

    int testBerr();
```

Function there() tests if the given address is readable using the data width of an allocated image convering `address`. The other three function use the given data width (D8, D16 or D32) for testing `address`.

To check if an uncleared VMEBus error is still available, the function testBerr() can be used.

```
  int setupIrq(int image, int irqLevel, int statusID, unsigned int addrSt,
               unsigned int valSt, unsigned int addrCl, unsigned int valCl);
```

Setting up irq handling is a little bit complicated and is explained in chapter 4.5

```
int waitIrq(int irqLevel, int statusID);
```

Waits for a VMEBus intterupt with VME level irqLevel and status/ID word statusID.

```
int generateVmeIrq(unsigned int irqLevel, unsigned int statusID);
```

Generates a VMEBus interrupt with VME irq level irqLevel and provides status/ID during IACK cycle.

```
int setupMBX(int mailbox);
unsigned int waitMBX(int mailbox);
unsigned int waitMBX(int mailbox, unsigned int timeout);
int releaseMBX(int mailbox);
```

Allocating, waiting for and releasing a mailbox. Valid mailbox numbers are in [0..3]. setupMBX(mailbox) and releaseMBX(mailbox) return 0 for success, negative numbers in case of an error. waitMBX(mailbox) returns the value which was written to the mailbox and triggered the interupt. Error code is 0xFFFFFFFF, so take care to not write this value into a mailbox.

```
unsigned int requestDMA(void);
unsigned int requestDMA(nrOfBuffers);
```

Requests ownership of the universeII onboard DMA. If DMA is not available zero will be returned in case of success the address of the DMA buffer in user space. When using multi buffer handling, nrOfBuffers must be in [1..128] and of 2^n.

```
void releaseDMA(void);
```

Release ownership of universeII onboard DMA.

```
int DMAread(unsigned int source, unsigned int count, int vas, int vdw);
int DMAread(unsigned int source, unsigned int count, int vas, int vdw,
            unsigned int bufNr);
int DMAwrite(unsigned int dest, unsigned int count, int vas, int vdw);
int DMAwrite(unsigned int dest, unsigned int count, int vas, int vdw,
             unsigned int bufNr);
```

Writes or reads count bytes (!) from/to source/dest using VMEBus address and data width vas/vdw. For multi buffer handling the buffer number must be passed in addition.

```
int newCmdPktList(void);
```

Creates a new list for DMA linked list operation mode. The number of the new list is returned. Negative values in case of error.

```
delCmdPktList(int list);
```

Removes an existing list and all command packets of it.

```
unsigned int addCmdPkt(int list, int write, unsigned int vmeAddr,
                       int size, int vas, int vdw);
```

Adds a command packet to an existing list. In case of error 0xFFFFFFF (-1) is returned, else the address to the buffer space of this command packet.

```
int execCmdPktList(int list);
```

Executes list `list` and returns after completion of all command packets. Negative return values in case of error.

```
unsigned int readUniReg(int reg);
void writeUniReg(int reg, unsigned int data);
```

Read and write access to universe II register (for use of unsupported features) is carried out with these two functions.

```
int resetDriver();
```

Completely resets the universeII driver. Make sure to exit or kill all programs using the driver before calling this function.

```
void vmeSysReset();
```

Generates a VMEBus SYSRESET* signal.