Documentation of the Linux-Driver and Library for the PCI-VMEbus-Interface of W-IE-NE-R

Dirk Muehlenberg
H.-J. Mathes
Institut fuer Kernphysik
Forschungszentrum Karslruhe

July 13, 1999

1 Arwvme PCIBus to VMEbus Interface

1.1 Kernel Versions

There are many changes to the kernel since release 2.1, concerning macros like put_fs_..., get_fs_..., memcpy_fromfs, ... and the pci interface. I have adopted these macros, hoping it will work, I have not tested these changes on a kernel > 2.0.

1.2 Driver and Library Files

The library files are located in the directory foo, the driver files in foo/driver.

- Library Files

```
COPYING ........ GNU license
arwvme.c ....... the library
arwvme.h ....... the users header for the lib
test_vmdis.c ... test program for the CES VMDIS 8004
test_v462.c .... test program for the CAEN Mod V462
```

- Driver Files

```
HOWTO .......... contains information of the installing/
                 deinstalling procedure
README ......... describe features and usage like this file. The
                 usage is partly described on the basis of the
                 libary.
BUGS ........... actual bugs in the driver and library
Makefile ....... contains all the maintainance of the driver
arwvme.h ....... the header incuded from the library
arwvme_priv.h .. private header to the driver
arwvme.c ....... the driver source
list.* ......... list implementation
vic68.h ........ defines for the VIC68A from Cypress
plx9050.h ...... defines for the PLX9050 from PLXtechnology
arwpci.* ....... defines and function for readout of the PLX
                 chip via PCI bios
```

1.3 The interface

The hardware was designed by ARW-Elektronik, D-69168 Wiesloch, and is now sold by W-IE-NE-R, Plein Baus GmbH, Germany (www.wiener-d.com/vmepci.htm).

1.3.1 Hardware functionality

The hardware consists of two parts connected by a up to ten meters cable allowing 32 bit wide differential data transfer. A PCI card (PCIADA) containing the PC site local bus logic (i.e. PCIbus target chip PLX9050, www.plxtech.com) and interfacing to the VMEbus compatible card VMEMM (VME Master Module) which may act as a system controller. The VMEMM contains the VIC68A chip (from Cypress, www.cypress.com) as the VME site local bus driver. Due to the choice of the PCIbus as the PC site local bus, all hardware parameters (like memory addresses, timeouts, interrupt line ...) can be configured dynamically during the boot process or afterwards by simply writing to the PCI configuration space. There are two additional memory spaces available to the driver, the registers of the PLX chip and those of the VIC68A. The driver will setup the two chips to an operable state during initialization, the user of the driver has only very limited access to this initial configuration, thus resulting in a always well defined state of the hardware.

1.3.2 Features

The features of the driver (loadable module) are:

- Multi-Process access capability: an arbitrary number of processes (up to the maximum allowed by the kernel) is allowed to enter the driver with the constraint, that one process must not possess two file descriptors (fd) for the same crate.
- Multi-Hardware installation: the driver allows an arbitrary number of PCIADA/VMEMM combos in the configuration, you may have only up to 16 combos as the limit given by the VMEMM module number jumpers (or the numbers of your available PCI slots). If two VMEMM VMEbus system contollers have the same number, the second (in the sequence of PCIbus numbering) will be rejected.
- PnP: due to the PCIbus there aren't any parameter to set externally for the driver, all driver parameters are retrieved from the PCI bios.

- Shared interrupts with ROAK and RORA support: the driver supports shared interrupts (so the kernel version does), if there are many PCIADAs exhausting the available interrupts, the PCI bios will give some PCIADAs the same irq line, for which the driver is prepared.

- VFS: the driver supports the usual Virtual File System functions open, release, read, write, seek, ioctl and mmap. The following addressing modes are supported

  ```
  A16D32, A16D24 and A16D8
  A24D32, A24D16 and A24D8
  A32D32, A32D16 and A32D8
  ```

- MMapped access: the mmapped access is realized via page fault handling, which allows the user (process) to fetch a big window on the VMEbus (say 1M), the driver will map the actual user page on the hardware page every time the user accesses a new page. This process is transperant to the user and very fast.

- Mixing of MMap and Read/Write: there is no interaction between the mmapped and the read/write accesses.

- Notification of system failures (interrupts) via SIGBUS: the user will be notified in case of system errors (SYSFAIL, ACFAIL) or VMEBUS timeout and error via the SIGBUS signal. If the user is prepared (via signal()), the signal is catched, otherwise the process will terminate. This mechanism ensures, that the user won't miss an important error status of the system even in the case of mmapped access, otherwise resulting in difficult backtraces. The information available for the signal are the timelabel (jiffies), the reason (interrupt vector), the address of the last access (only correct in case of VMEBUS_TO or VMEBUS_ERR), and the module number.

1.4 The library

The library accompanying the driver is for the users convenience. All features of the driver are also accessible without this library. Where not stated otherwise, most functions are calls to ioctl() with appropriate parameters, and return 0, if successful, or -1 in the case of error (code in errno, defines in `<asm/errno.h>`). The system calls read, write and lseek will set errno to EIO, if the user has not called ArwvmeSetMode() with appropriate parameters.

1.4.1 Opening and closing the device

The library can be opened via

```
int fd = ArwvmeOpen(int module_nr);
```

Due to the possibility of using many PCIADAs, you have to support an index (the module number) of the VMEMM. If a module with this number is found and this process does not own this crate already, open() returns a valid fd, which is needed by all subsequent library calls, to identify your connection. In the case of error, open returns -1 and set errno to:

- ENODEV: the user has specified a wrong minor device number.
- EBUSY: the user tried to open the same device a second time.

The connection to the device can be closed via

```
int ArwvmeShutdown(int fd);
```

which calls close(fd) and exists only for symmetry.

1.4.2 Set the Address Modifier

The address modifier (AM code) can be set via

```
int ArwvmeSetMode(int fd, unsigned int mask);
```

mask contains the combination of the address, the access and the data bitmasks (ARWVME_A16, ARWVME_D16, ..., → driver/arwvme.h) necessary to setup the operation mode of the VMEbus for this process. A valid setup is:

```
ArwvmeSetMode(fd, ARWVME_A16|ARWVME_D16|ARWVME_DATA|ARWVME_USER);
```

1.4.3 read, write System Call

The read/write system calls for this device have a maximum number of bytes transferrable each call! Sorry, but I have chosen this way to prevent users from hogging the VMEbus for long times (say seconds). The maximum value is 16K, which is suitable for the hardware, because one read/write of 16K lasts around 20ms, which is near the heartbeat of the kernel (on Intel). This means, that a process will release the VMEbus in time. So you have to read/write in a loop to transfer all bytes. A normal user process will run for about 200ms, so if you have another process which must have minimal interrupt latency, you have to run it with SCHED_RR (see 1.6.4), in which case the normal user process will deschedule in time. I have included ArwvmeRead() and ArwvmeWrite() into the library, which do a sched_yield after each transfer of 16K. I recommend to use them for large amounts of data, to give other processes the chance of using the VMEbus. There is no easy way to do this cooperative work in the driver, because you can only call schedule() at the end of a system call (to release the CPU and the VMEbus to another process). By the way, you have no greater bandwidth without this limitation.

1.4.4 lseek System Call

The lseek call will set errno to EFAULT, if the given offset is unaligned. The only possible whence parameter is SEEK_SET, otherwise errno will be set to EINVAL.

1.4.5 Memory Mapping

```
The mmapped access can be prepared via
   void *ArwvmeMMap(int fd, int size);
```

which maps a buffer (size bytes) in user space on the 4-32K VMEbus window in kernel space. No mapping is really done, the mm layer handles the page faults via arwvme_vma_nopage(), which is registered for this requested memory area. The call returns the pointer to the user buffer or null, if the mapping was impossible. The start of the user buffer maps to VMEbus address 0x0. This mapping can be altered by using

```
   int ArwvmeSetVmeWindow(int fd, unsigned int addr);
```

which remaps the user buffer to the new VMEbus start address 'addr'. addr is handled as addr &~(win_size - 1), the user can only shift in entities of win_size.

1.4.6 Miscalleanous

The process can test the connection between PCIADA and VMEMM at any time via

```
int ArwvmeCheckConnection(int fd);
```

which is already done during the init_module() routine. It returns TRUE, if the connection was ok, FALSE otherwise.

```
int ArwvmeResetVme(int fd);
```

initiates a global reset in the VMEMM module and is accessible only to superuser and group members of this device.

```
int ArwvmeInitialize(int fd);
```

inits the hardware of the VMEMM module and is accessible only superuser and group members of this device.

```
void ArwvmeTieFpos(int fd);
```

ties the f_pos (position in the device space i.e. file position) to the read or write action.

```
void ArwvmeUntieFpos(int fd);
```

unties the f_pos, sucessive reads or writes won't alter the file position (useful for fifos).

```
int ArwvmeCtrlArbiter(int fd, int arb_mode, int br_lev,
                      int fairness);
```

controls the arbiter paramters of the VMEMM. There are two possible arb_modes - ArwvmeArbPri (Priority -) and ArwvmeArbRR (Round-Robin-arbitration). The bus request level is br_lev (0..3) and the fairness duration counts from 0 (disabled), 1 (2mus), 2 (4mus), .., 15 (none).

```
int ArwvmeSetVMEbusTO(int to);
```

controls the timeout value of the VMEbus timeout, the values are 4us .. 512us .. infinity, whereas user values > 512 are mapped to infinity.

1.4.7 System error detection (i.e. VMEbus error)

The irq notification is enabled by default, so error detection is always possible.

```
ArwvmeSysErrInfo *ArwvmeGetSysErrInfo(int fd);
```

returns a structure containing the information about the last (!) VMEbus error generated by this pid. The structure contains the module_number, the vme_addr, the time (jiffies) and the reason, at which the handler was called. The returned pointer is private and must not be freed! In the cases of PCIADA_TO, ACFAIL and SYSFAIL the handler will aggressively kick the users (via SIGBUS). If a process catches the signal, it will inspect the reason and decides what to do next.

```
int ArwvmeCheckSysError(int fd);
```

returns TRUE, if a bus error occured since the last request. Both functions reset the sys err info, so a ArwvmeCheckSysError(fd); ArwvmeGetSysErrInfo(fd); will reveal no error for the second call.

1.5 I/O and memory base addresses

The driver configures itself via the PCI-Bios.

1.6 Interrupts

1.6.1 Configuring the adaptor

The adaptor gets its irq via PCI setup (Bios or kernel). There are two interrupt sources:

1. PCIADA(ptor):
   - access to VMEMM timeout (24us),
   - data cable not connected,
   - vme-crate not powered up,
   - PCIADA not activated,
2. VMEMM: (VIC68a)
   - all 7 VMEbus interrupt level,

- mailbox interrupt,
- interrupt due to successful interrupt generation,
- SYSFAIL, ACFAIL,
- bus timeout
- arbiter timeout
- timer interrupts
- reset button

The vectors 0x0..0x3f are reserved for the driver and will be catched by the irq_handler!

1.6.2 Interrupt types

Vme knows two different types of interrupters, ROAK and RORA.

ROAK (Release On Acknowledge) interrupters release their interrupt request signal during the IACK (interrupt acknowledge) cycle on the Vme bus. This type of interrupters is supported.

RORA (Release On Register Access) interrupters in addition want a module specific register access to remove the interrupt request signal from the bus. This type of interrupter is now also supported, but still in the testing phase.

The driver handles the IACK cycles automatically (in case of ROAK or RORA-ArwvmeRoraByHandler) and appends information about occurred interrupts in a list with a length of maximum ARWVME_MAX_INTS_QUEUED entries (defined in driver/arwvme.h)

1.6.3 How to program and use interrupts

Interrupts are forwarded to the user program by means of the select() system call or by delivering the SIGUSR1 signal to the user program. The select() call returns if an interrupt occurred or if a predefined time interval is over. Afterwards the users program can query the interrupts with the function ArwvmeGetNextPendingInt().

```
int ArwvmeSelect(int fd, struct timeval *);
```

Supports the kernel with the timeout for this process and schedules this calling process for the selection, if there is no pending interrupt in the queue, or an interrupt waiting on the bus.

```
int ArwvmeEnableInterrupt(int fd); (deprecated)
```

Globally enables VMEbus interrupt notification (i.e. installs the irq handler). This function is a dummy and only available for compatibilty to the ATVME library. The irq is always enabled, if a process is registered in the driver.

```
ArwvmeDisableInterrupt(int fd); (deprecated)
```

Removes the irq handler from the kernel and disables interrupts. This function is also a dummy, the irq is disabled while no process is registered in the driver.

```
int ArwvmePostInterrupt(int fd, ArwvmeAckType ack_type,
         unsigned int rora_addr, int int_vector,
         unsigned int acc_mask);
```

allows to specify the interrupt acknowledge type. In the case of RORA there are the alternatives ArwvmeRoraByHandler and ArwvmeRoraByUser. ArwvmeRoraByHandler means the user supports the handler with a register address linked to an interrupt vector. On RORA interrupt the handler will access this register which ends the acknowledge cycle. On ArwvmeRoraByUser it's up to the user to access the register (which is also necessary in cases of more complicated RORA protocols), well knowing, that interrupt generation is blocked 'til user acknowledges.

```
int ArwvmeRemoveInterrupt(int fd, int int_vector);
```

removes the notification for interrupt vector int_vector.

```
ArwvmeIntAck *ArwvmeGetNextPendingInt(int fd);
```

gets the next interrupt from the list of pending interrupts. The returned IntAckStruct contains the interrupt vector, timelabel, the acknowledge type and the RORA register (if supplied). The user must free this space.

```
int ArwvmeSignal(int fd, void (*sig_handler)(int));
```

attaches the signal SIGUSR1 to the caller. If an interrupt occures, the user will be informed via SIGUSR1.

1.6.4 Interrupts and Linux (Interrupt latency)

Linux is no RTOS (even with its POSIX extensions like sched_setscheduler()), so it gives no guarantees for response on interrupts. On a complete idle system, the device drivers interrupt handler will be called in 2 us, but on a well loaded system you have up to 600us. You may say, that even this worst case is enough for you, but the real story is another one. Your user process will be informed of the pending interrupt by signal (SIGUSR1), and will process it (via its signal handler) at the next scheduling. So on a well loaded system (hogging CPU, much I/O) your process will catch the pending interrupt about a time of hundreds of ms later! This can be too much, above all in the case of RORA/ArwvmeRoraByUser (which means, that during this time you may miss some interrupts). There is a solution called RTLinux (its a RTOS kernel running Linux as its idle task), which makes guarantees of schedule times with an accuracy of 35us. But its API is not compatible with standard Linux device driver writing. Any data exchange of such a driver (its really a module) is done via fifos, which are devices to the user process. So you had to rewrite the driver completely, and it would never be a normal Linux driver. Such a solution is excellent for device drivers with specific and limited features (like a frame grabber). So your only way out (in the case of ArwvmeRoraByUser) is to use sched_setscheduler with SCHED_RR (be gentle) for your user process, and hope it will be enough for your task. I have tested this solution with heavy load and measured worst cases of 10ms (which is exactly the interrupt frequency of the clock) until the user process get informed. You can not get better with this extension! To really get better, you have to use another clock with increased interrupt frequency or use a variable programmed clock. For the sake of fairness, without sched_setscheduler() you may get worst cases of 1s, so it is a tremendous improvement given by the POSIX RT extensions, but it will be smoked by RTLinux. Another way out is to extend the device driver (by using rt_request_irq) and do whatever has to be done now at this place, exchanging the data via rt_fifos.

1.7 Future plans

For the future there will be a speed up of local bus transfers through bursts and use of fifos (PLX9050).

1.8 Acknowledgements

I would like to thank

- Albert Rausch (rausch@physi.uni-heidelberg.de)

for loaning me the interface and for the support in hardware debugging during the first phase of the driver development.

- Lueder v. Horn (vhorn@wiener-d.com),
- Klaus Hitschler (Klaus.Hitschler@gmx.de)
- Juergen Hannappel (hannappe@lisa2.physik.uni-bonn.de)

for the testing of the driver and library.