VPCIVMED

Windows 95 driver for

PCI-VME

User’s Manual

*00437.A0

General Remarks

The only purpose of this manual is a description of the product. It must not be interpreted a declaration of conformity for this product including the product and software.

W-Ie-Ne-R revises this product and manual without notice. Differences of the description in manual and product are possible.

W-Ie-Ne-R excludes completely any liability for loss of profits, loss of business, loss of use or data, interrupt of business, or for indirect, special incidental, or consequential damages of any kind, even if **W-Ie-Ne-R** has been advises of the possibility of such damages arising from any defect or error in this manual or product.

Any use of the product which may influence health of human beings requires the express written permission of **W-Ie-Ne-R**.

Products mentioned in this manual are mentioned for identification purposes only. Product names appearing in this manual may or may not be registered trademarks or copyrights of their respective companies.

VPCIVMED is designed by ARW Elektronik, Germany

Table of contents:

List of tables:

1. VPCIVMED driver: General description

VPCIVMED provides an easy access to the VME bus for Windows95[1] and 98 users. It's major efforts are demonstrated by a small test program `pvmon.exe` which is supplied in the same package.

It is easy to use the driver for your own VME application. The driver is independent from the chosen programming language since Windos95 standard I/O functions are used for the communication.

VME access is performed via an interface window of an area of virtual memory which is defined by the driver. For user applications this window looks like normal memory. Read and write operations to the VME bus are converted into simple read and write operations into the (not real) memory.

The access to the driver is not limited to one process. Multiple processes can use the driver. Even one driver supports multiple VME interfaces.

Different levels of VME interrupts are handled by the interface. The driver provides several serviced to operate these interrupts.

[1] Windows95 and Windows98 are trademarks of the Microsoft Corporation.

2. Installation

Be sure that PCIADA card of the PCI-VME interface is installed in your PC. Please refer to the PCI-VME manual to insert the card.

After switching on you machine Windows recognizes the new hardware in you system and asks for a driver. Insert the supplied CD into your drive and enter the driver's path. If you CDROM is drive D type `D:\WIN95\DRIVER`.

In the next step the driver is copied to `WINDOWS\SYSTEM\VPCIVMED.VXD` and the interface is added to the WINDOWS registry. You will find the driver at `HKEY_LOCAL_MACHINE\ENUM\PCI\VEN10B5&DEV9050 ...`

You will find the interface at `start / settings / control panel / system / device manager` where Interrupt and I/O settings can be verified.

Note:	The driver only works for Windows 95 / 98 in 32 bit mode. Only real 32 bit applications can use the driver but it does not work for MS-DOS[2] or WINDOWS 3.11 programs.

So far WINDOWS NT[3] is not supported. A driver is under preparation.

[2] MS-DOS and Windows 3.11 are trademarks of the Microsoft Corporation.

[3] Windows NT is a trademark of the Microsoft Corporation.

3. Operating the driver

3.1. A simple test unit: `pvmon.exe`

The program is a useful tool to check the access to the VME bus and test VME modules. It expects the driver vpcivmed in `C:\WINDOWS\SYSTEM`. If it is not there the path has to be specified.

Open a DOS box and start pvmon by typing `pvmon -?`. A short help is displayed. Help can be obtained by typing `?` on the prompt, too.

Before accessing the VME bus pvmon has to be configured by typing `c`. Store the settings and restart the program. Now you can exchange data with the VME bus.

For more information please refer to the short form manual in APPENDIX B.

3.2. Using the driver for program code

Access to the driver is managed by Windows 95 Standard I/O functions which are independent from the programming language. Header files for c++ programs are supplied with the interface. They could easy adapted to other languages.

In your program include files `vpcivmed.h` and `windows.h`. Add `winerror.h` too if you want to use `GetLastError()` to decode error messages. Use window's function `CreateFile()` to open the interface, `DeviceIoControl()` to operate it and `CloseHandle ()` to close it.

At maximum `VPCIVMED_MAX_PCIADA` PCIADA cards (currently 4), `VPCIVMED_MAX_VMEMM` interfaces (16) and `VPCIVMED_MAX_WINDOWS` (8) different windows are supported by the driver. These parameters are defined in `vpcivmed.h`. Only the number of the VMEMM module is used to identify different modules and cards.

Intercommunication between driver and the users program is done via memory windows. The driver provides a window for each process who requested it returning a pointer into the window.

Size, Address Modifier and offset to access the VME bus is fixed for each window. Random access to different windows it possible. The driver itself takes care of Address Modifier and address offsets.

Any PCI-VME application using the driver contains three major parts:

1. **Startup**

   ```
   vxd_Handle = CreateFile(VxDpathName,0,0,NULL,0,
                           FILE_FLAG_DELETE_ON_CLOSE,NULL);
   ```

   During this procedure number and IDs of connected VMEMM modules is determined.

2. **Controlling the Interface** Each access to the interface is done by

   ```
   result = DeviceIoControl(vxd_Handle, .....);
   ```

   It is only necessary to pass the ID of the selected VMEMM module to the driver. The corresponding ID of the PCIADA card is calculated automatically.

3. **Shut down** On the command

   ```
   CloseHandle(vxd_Handle);
   ```

   the driver is closed for the application. It is removed of the memory after it's last process has finished.

3.3. Services

The driver provides different services which communicate via `DeviceIoControl()`with the application. Numbers and structures for this communication are defined in `vpcivmed.h` (see APPENDIX C). Define Pointers to in and out structures before calling the driver.

A call of the driver may look like:

```
VPCIVMED_STANDARD_COMMAND  sInterface;
VPCIVMED_VECTOR_LEVEL      sVectorLevel;
DWORD                      DIOC_count;
DWORD                      dwResult;
*
*
sInterface.dwInterface = 1;           // selection of 1st VMEMM

// poll if an interrupt is pending ------------------
dwResult = DeviceIoControl(vxd_Handle, VPCIVMED_READ_VECTOR,
             &sInterface, izeof(sInterface),&sVectorLevel,
             sizeof(sVectorLevel), &DIOC_count, NULL);

if (!dwResult)
   printf(„Error %d occured\n“, GetLastError());
else
   printf(„I have read a vector %d at level %d\n“,
          sVectorLevel.dwStatusID, wLevel);
*
*
```

Service `VPCIVMED_READ_VECTOR` is called. Pointers to in and out structure and it's sizes are necessary. An error code which is explained in `winerror.h` and the real size of the returned data is returned.

Description of the defined services:

`VPCIVMED_INIT_HARDWARE` initializes one VMEMM module. Standard initialization commands are summarized in APPENDIX D. Additional initialization commands can be passed to the interface. All `VPCIVMED_INIT_COMMAND`s have to be stored in a STOP terminated array. Example:

```
struct
{
   DWORD dwInterface;
   VPCIVMED_INIT_ELEMENT sVIC[3];
} sUserInitStruct = {0, {{VIC,  BYTE_ACCESS,     0x57, 0xAA},
                         {VIC,  BYTE_ACCESS,     0x53, 0x00},
                         {STOP, WORD_ACCESS, 0x00,  0x00}}};
```

Note:	If the array contains only the STOP element the standard initialization will be performed.

The interface will be initialized on the first call of the service. It has to be deinitialized before a new initialization is possible.

`VPCIVMED_DEINIT_HARDWARE` deinitializes the specified VMEMM board and it's PCIADA card. Additional commands are added as described above. APPENDIX E shows the standard commands.

`VPCIVMED_ATTACH_WINDOW` reserves a window for VME access. One process can open `VPCIVMED_MAX_WINDOWS` at maximum. Parameters which are required to open a window are passed in a `VPCIVMED_ADD_WINDOW` structure. The window size is limited to 256 Mbyte.

Address Modifier, a Base Address and size have to be specified for each window. Only values at the edge of a 4k page are possible for address and size. The driver's header file provides macro functions `PAGE_BASE()` and `PAGE_SIZE()` to calculate these numbers.

The driver maps the specified area of the VME bus into the (virtual) memory. A pointer to this memory region is returned. Each access to this region is mapped into the VME bus. Any access out of the window will be denied.

Errors during VME bus access are not reported as Windows errors.

`VPCIVMED_DETATCH_WINDOW` releases a previously reserved window. Use a `VPCIVMED_REMOVE_WINDOW` structure to define parameters.

`VPCIVMED_GET_STATIC_STATUS` returns status information of a VMEMM interface in a `VPCIVMED_STATIC_STATUS` structure.

`VPCIVMED_GET_DYNAMIC_STATUS` informs about parameters of the interface which change during operation. Use a `VPCIVMED_DYNAMIC_STATUS` structure for communication.

`VPCIVMED_READ_VECTOR` returns interrupt information in a `VPCIVMED_VECTOR_LEVEL` structure.

`VPCIVMED_ACCESS_VIC68A` provides direct access to the VIC68A chip. Use a `VPCIVMED_VIC68A_ACTION` structure to program the chip and for the exchange of data.

The PCI-VME profits of the huge variety of features which are provided by the VIC68A chip, e. g. direct access to 68xxx processors and programmable delays by accessing the VIC68A directly. No limitations of this communication are installed.

Note:	Do not change any registers which may influence the Address Modifier Register. It will cause errors in the mechanism of interface windows.

`VPCIVMED_INSTALL_IRQ_HANDLER` installs the interrupt handler on the local interrupt priority level. The TCB (Thread Control Block) of the calling thread is stored when the service is accessed. If an interrupt is enabled and released and the thread is alertable it is possible to invoke the installed interrupt handler.

Either PCIADA or VMEMM interrupts cause the interrupt handler. Interrupt source is coded in a parameter which is described in Table 1.

Table 1: Coding of interrupt level and vector.

meaning	unused	interrupt level	unused	interrupt vector
bits	31 to 19	18 to 16	15 to 8	7 to 0

A BUS ERROR is handled as an VMEMM Interrupt. Since the driver is locked after each VMEMM interrupt it has to be released by the user's application. Interrupts caused by PCIADA are treated as virtual level 8.

`VPCIVMED_CONTROL_INTERRUPTS` controls the interrupt mechanism. It enables or disables specified interrupts of PCIADA or VMEMM.

`VPCIVMED_TAS` causes an uninterruptible cycle on the VME bus which is comparable to the TAS command of 68xxx processors.

`VPCIVMED_GET_PCIADA_STATUS` returns status of all connected PCIADA boards installed in the PC. It checks which VMEMM modules are connected and ready.

VPCIVMED_RESET controls different reset functions of the interface and the VME bus which are a local reset, a global reset and a VME bus reset.

Contents of all VIC68A registers are lost during a reset. Perform a deinitialization and a reinitialization after the reset to reload registers.

3.4. Interrupt vectors

Each interrupt caused by VMEMM has to be vectored. Normally vectors from 0x00 to 0x3F are used by the driver (internal use) and vectors from 0x40 to 0xFF are reserved for VME bus and it's peripherals. Refer to Table 2 for detailed information.

Note: The time out interrupt generated by PCIADA causes an interrupt vector number 1.

Table 2: Interrupt vectors for different sources.

Interrupt source	vector no.
Interrupt caused by PCIADA (time out)	1 (active)
Clock Tick Interrupt Generator	2
Reset push button on the front panel	6 (active)
VME bus Timeout (Bus-Error)	7 (active)
Interprocess communication global switch #0	8
Interprocess communication global switch #1	9
Interprocess communication global switch #2	10
Interprocess communication global switch #3	11
Interprocess communication module switch #0	12
Interprocess communication module switch #1	13
Interprocess communication module switch #2	14
Interprocess communication module switch #3	15
ACFAIL asserted	16
Write post Fail	17
Arbitration Timeout	18
SYSFAIL asserted	19
VME bus Interrupter acknowledge	20

Note: Pressing the reset button on the front panel causes an interrupt. Applications have to take care of any further action which should be performed.

Note: If more than one application use one window of the interface it is not possible to locate the cause of a VME BUS ERROR. In this case every only one action is performed.

If errors occur during interrupt operations check at `start / settings / control panel / system / device manager` if any interrupt reserved for the interface. The interface works without a reserved interrupt but interrupt functions are not available in this case.

APPENDIX A: Packing list:

The driver is delivered in one CD ROM which contains:

Directory `WIN95\DRIVER`:

`vpcivmed.vxd`	the driver
`pcivme.inf`	INF file for installation

Directory `WIN95\DRIVER\SOURCE`:

`vpcivmed.h`	header file to access the driver
`vic.h`	header file for the VIC68A chip
`vme.h`	header file to access the VME bus
	source files for the driver

Directory `WIN95\PVMON`:

`pvmon.exe`	a useful program

Directory `WIN95\PVMON\SOURCE`:

source files for pvmon.

APPENDIX B: Short form manual of pvmon

pvmon is a simple shell program to test the PCI-VME interface by ARW Elektronik. The code is OpenSource and is enclosed to the interface.

Overview of pvmon commands (type "?" to get this help):

a[h] [adrmode]	: Change address modifiers, h=help
c	: Configure interface
d[m] [start] [end]	: Dump memory area
e[m] <start> [value]	: Examine or change memory area
f<m> <start> <end> <x>	: Fill memory from <start> til <end> with <x>
g<m> <st> <en> [l] [x]	: Generate random memory test. (loop l, seed x)
h	: This help
i	: Interface init
l[m]	: Get VME interrupt status/ID
m<m> <src> <end> <dest>	: Move memory area
o	: Jump to OS
p[adrmode]	: Port search
q	: Quit program
r[x] <f> <start> [end]	: Read file <f> to VME, x= x or s (HEX)
s[m] <start> <end> <p>	: Search pattern <p>=different Items
t <start>	: TAS emulation, 'Test and Set' bit 7
v	: Generate VME SYSRESET
w[x] <f> <start> <end>	: Write VME into file <f>, h=Intel Hex
x <start> [val]	: Read/Write to interface register @ start
y[1/0]	: Read/set/clear SYSFAIL
z[0..3]	: Show interface internals

m = mode, e.g. b=byte, w=word, l=long (double) word; h = help, x= hex

start(address), end(address), src=source, dest=destination, []=option

pvmon is available for WIN 95/NT and Linux. The driver for the operating system has to be installed.

An error message is reported if no driver was found or the VME crate is not online.

The first time pvmon is started a configuration is mandatory. Simply type `c` on the command line.

Powerful commands are implemented in pvmon. Try `p` to look for ports or test the RAM on the VME bus with the command:

```
gw 0 10000 40.
```

In the address range from 0x00000 to 0x10000 RAM is tested for the predetermined address modifier in 0x40 runs using a random pattern.

Note:	Before using the command make sure that no important data is stored in the address range. All addresses will be overwritten.

To use pvmon interactively type e. g.

```
pvmon a39/p/a29/p
```

First address modifier is set to 0x39 and the address range is scanned readable addresses. The same is repeated for AM = 0x29.

APPENDIX C: Header file vpcivmed.h

```
#ifndef __PCIVMEH_H__

//---------------------------------------------------------------------
---------------------
// PCIVMEH.H, shared between applications and VPCIVMED driver
//
// (c) 1999 ARW Elektronik
//
// this source code is published under GPL (Open Source). You can
use, redistribute and
// modify it unless this header   is not modified or deleted. No
warranty is given that
// this software will work like expected.
// This product is not authorized for use as critical component in
life support systems
// without the express written approval of ARW Elektronik Germany.
//
// Please announce changes and hints to ARW Elektronik
//
// What
Who  When
// first steps
AR   24.01.98
// added direct read write access to vic68a chip registers
AR   12.07.98
// rename PCR_* into LCR_*
AR   19.07.98
// TAS included
AR   17.02.99
// Corrections about interrupt handling
AR   20.02.99
// changes about PCIADA status
AR   25.02.99
// changes of IOCTL codes because of compatibility to WIN NT
AR   12.03.99
// PLX 9052 removed out of VPCIVMED_STATIC_STRUCT
AR   16.03.99
// VIC68A_WRITE_ONLY added
AR   17.03.99
// extension for VME reset
AR   18.04.99
// release of version 2.5 for driver
AR   18.04.99
//

//---------------------------------------------------------------------
---------------------
// constants to be used to access certain features of the PCIVME
interface
//
#define VPCIVMED_CTL_CODE(x)         (0x80002000 | (x << 2)) //
compatibility to WIN-NT

#define VPCIVMED_INIT_HARDWARE       (VPCIVMED_CTL_CODE( 0)) //
initializes the hardware with given parameters
#define VPCIVMED_DEINIT_HARDWARE     (VPCIVMED_CTL_CODE( 1)) //
uninitializes the hardware
#define VPCIVMED_ATTACH_WINDOW       (VPCIVMED_CTL_CODE( 2)) //
requests a base address to a vme window
```

```
#define VPCIVMED_DETACH_WINDOW       (VPCIVMED_CTL_CODE( 3)) //
frees a vme window
#define VPCIVMED_GET_STATIC_STATUS   (VPCIVMED_CTL_CODE( 4)) // asks
for INTERFACE structure
#define VPCIVMED_GET_DYNAMIC_STATUS  (VPCIVMED_CTL_CODE( 5)) // asks
for dynamic status
#define VPCIVMED_READ_VECTOR         (VPCIVMED_CTL_CODE( 6)) //
reads the level and vector of IRQ
#define VPCIVMED_ACCESS_VIC68A       (VPCIVMED_CTL_CODE( 7)) //
access vic68a register
#define VPCIVMED_INSTALL_IRQ_HANDLER (VPCIVMED_CTL_CODE( 8)) //
installs a handler function
#define VPCIVMED_CONTROL_INTERRUPTS  (VPCIVMED_CTL_CODE( 9)) //
enable, disable of interrupts
#define VPCIVMED_TAS                 (VPCIVMED_CTL_CODE(10)) // make
test and set
#define VPCIVMED_GET_PCIADA_STATUS   (VPCIVMED_CTL_CODE(11)) // get
the status of PCIADA(s) only
#define VPCIVMED_RESET               (VPCIVMED_CTL_CODE(12)) // make
a reset to VME or global

//--------------------------------------------------------------------
----------------------
// possible return codes
//
#define BOGUSADDRESS 0xffffffff      // Returned by MS routines

//--------------------------------------------------------------------
----------------------
// some built in limits
//
#define VPCIVMED_MAX_PCIADA     4 // maximum count of supported PCI
interfaces
#define VPCIVMED_MAX_VMEMM     16 // maximum number of supported
VMEMMs
#define VPCIVMED_MAX_WINDOWS    8 // maximum number of windows into
VME

//--------------------------------------------------------------------
----------------------
// switches and masks
//

// switches for VPCIVMED_INIT_COMMANDs --------------------------
#define LCR   (BYTE)0    // destination is LCR register
#define IFR   (BYTE)1    // destination is VME-Interface register
#define VIC   (BYTE)2    // destination is VIC68A register
#define STOP  (BYTE)255  // this command stops the init machine

#define BYTE_ACCESS (BYTE)1    // write byte wise
#define WORD_ACCESS (BYTE)2    //       word
#define LONG_ACCESS (BYTE)4    //       long

// switches for VPCIVMED_ACCESS_VIC68A --------------------------
#define VIC68A_READ       0    // read only access
#define VIC68A_WRITE      1    // write and read back access
#define VIC68A_OR         2    // read, bit wise 'or' content and
read back access
#define VIC68A_AND        3    // read, bit wise 'and' content and
read back access
#define VIC68A_WRITE_ONLY 4    // do not read back after write
```

```
// switches for VPCIVMED_VECTOR_CMD ------------------------------
#define READ_CURRENT_LEVEL 0   // try to get the current IRQ level
#define READ_VECTOR        1   // (if level == 0) read vector @
current LEVEL else @ level

// switches for the VPCIVMED_RESET ------------------------------
#define VME_RESET_CMD      0   // raise a VME reset only
#define LOCAL_RESET_CMD    1   // raise a local reset only
#define GLOBAL_RESET_CMD   2   // raise a global reset
#define POLL_RESET_CMD     3   // ask if reset is finished

// address masks for the pager - to use for offset and size @ window
alignment -----------
#define HI_ADDRESS_MASK   (DWORD)0xFFFFF000    // masks the high
part of a vme address
#define LO_ADDRESS_MASK   (~HI_ADDRESS_MASK)   // masks the low
part of a vme address
#define ONE_PAGE_SIZE     (LO_ADDRESS_MASK + 1)  // size of 1 page
(hardware related)

// macros to calculate the real base and the real size of demand
pages -------------------
#define PAGE_BASE(base)   (base & HI_ADDRESS_MASK)  // makes an
aligned base for a page
#define PAGE_SIZE(base, size) (((base + size + LO_ADDRESS_MASK) /
ONE_PAGE_SIZE) * ONE_PAGE_SIZE)

//--------------------------------------------------------------------
---------------------
// ERROR RETURNS in dIfcStatus
//
#define E_NO_ERROR         0   // all OK
#define E_INCOMPATIBLE     1   // incompatible hardware
#define E_NO_ADDRESS       2   // cant get lcr or ifr addresses
#define E_NOT_CONNECTED    3   // no VMEMM hardware connected
#define E_CON_ERROR        4   // data transfer failure
#define E_EMPTY           -1   // no PCI interface associated

//--------------------------------------------------------------------
---------------------
// shared structures between PCIVME-IF and Application - COMMANDS
//
typedef struct
{
  DWORD dwInterface;         // some command only need this input
into requests
} VPCIVMED_STANDARD_COMMAND;

typedef struct               // one command element to initialize
interface or deinitialize
{
  BYTE     range;            // 0 = lcr, 1 = vme-interface, -1 =
stop, default = vme-if
  BYTE     type;             // 1 = byte access, 2 = word access, 4
= dword access, default byte
  WORD     offset;           // offset into interface address range
for initialisation
  DWORD    value;            // value to initialize
} VPCIVMED_INIT_ELEMENT;
```

```
typedef struct
{
  DWORD    dwInterface;      // targets to interface number
  VPCIVMED_INIT_ELEMENT sVie[1];  // at least one zero element must
be the last
} VPCIVMED_INIT_COMMAND;

typedef struct
{
  DWORD    dwInterface;       // targets to interface number ...
  DWORD    base;              // offset into VME address range.
(base + size) must be less than
  DWORD    size;              // 128 Mbytes for ext, 16 Mbytes for
std, 64k for short
  WORD     modifier;          // VME address modifier for this
window
} VPCIVMED_ADD_WINDOW;

typedef struct
{
  DWORD dwInterface;          // targets to interface number ...
  DWORD *pdwLinAdr;           // linear address of window to remove
} VPCIVMED_REMOVE_WINDOW;

typedef struct
{
  DWORD    dwInterface;       // targets to interface number ...
  DWORD    dwAddress;         // tas to address
  WORD     wModifier;         // VME address modifier for this
window
  BYTE     bContent;          // byte content to store and get back
} VPCIVMED_TAS_STRUCT;

typedef struct
{
  DWORD dwInterface;          // targets to interface number ...
  WORD  wRegisterAddress;     // address offset of vic68a register
  WORD  wAccessMode;          // read, write, or, and
  BYTE  bContent;             // content to write, and, or
} VPCIVMED_VIC68A_ACTION;

typedef struct
{
  DWORD dwInterface;           // targets to the interface number
...
  DWORD dwIrqHandler;          // void (*IrqHandler)(DWORD) = User
Handler
} VPCIVMED_IRQ_HANDLER;        // BOGUSADDRESS deinstalled

typedef struct
{
  DWORD dwInterface;           // targets to the interface number
...
  WORD  wEnable;               // a 1 enables, a 0 disables
} VPCIVMED_IRQ_CONTROL;

typedef struct
{
  DWORD dwInterface;           // targets to interface number ...
  WORD  wAction;               // read current irq level, read
vector @ level
```

```
  WORD  wType;                 // must be set to 1
} VPCIVMED_VECTOR_COMMAND;

typedef struct
{
  DWORD dwInterface;           // targets to interface number ...
  WORD  wCommand;
} VPCIVMED_RESET_COMMAND;

//--------------------------------------------------------------------
---------------------
// shared structures between PCIVME-IF and Application - RESPONSE
//

// includes static information about driver parameters ------
typedef struct                // caution: very sensitive on
alignment
{
  DWORD dwInterface;          // comes from the interface No.
  DWORD dIfcStatus;           // usable ? fits to driver? OK?
  DWORD dwLinkCount;          // how often this interface is
requested

  WORD  wNumMemWindows;       // from actual configuration
  WORD  wNumIOPorts;
  WORD  wNumIRQs;
  WORD  wNumDMAs;

  DWORD dLCR_MemBase;         // from actual configuration
  DWORD dLCR_MemLength;

  WORD  wLCR_IOBase;
  WORD  wLCR_IOLength;
  WORD  wLCR_IRQ;
  WORD  wReserve1;

  DWORD dUSR_MemBase;
  DWORD dUSR_MemLength;

  WORD  wModuleType;          // read from connected hardware
  WORD  wFPGAVersion;
  WORD  wModuleNumber;
  WORD  wWordMode;

  WORD  wSysControl;
  WORD  wConnected;

  PVOID pvLcr;                // virtual address of LCR
  PVOID pvIfr;                // virtual address of IFR

                              // some addresses to tune performance
  WORD  *pwCSR;               // pointer to csr register
  WORD  *pwIRQStat;           // pointer to irq status
  BYTE  *pbVector;            // pointer to vector read register
  DWORD *pdwVMEAdr;           // pointer to VME address register
  BYTE  *pbModifier;          // pointer to address modifier
register
  void  *pvVME;               // pointer into VME window
  DWORD dwPagePhysVME;        // physical page number of the VME
window
```

```
  void  *psIrqDescriptor;      // pointer to associated irq
descriptor
  DWORD dwActivePage;          // the current active page of this
interface

  WORD  wReserve2;

  char  cszHWRevision[10];
} VPCIVMED_STATIC_STATUS;

typedef struct
{
  DWORD dwInterface;           // comes from the interface No.

  WORD  wVMEMM_connected;      // status: VMEMM is connected and
powered
  WORD  wVMEMM_enable;         // status: VMEMM access is enabled
  WORD  wPCIADAIrq;            // status: PCIADA timeout IRQ pending
  WORD  wVMEMMIrq;             // status: VMEMM IRQ pending
} VPCIVMED_DYNAMIC_STATUS;

typedef struct
{
  DWORD dwInterface;           // comes from the interface No.

  DWORD dwStatusID;            // interrupt-vector (byte, word, long)
  WORD  wLevel;                // interrupt-level
  WORD  wPCIIrq;               // pending PCIADA Irq detected and
cleared
} VPCIVMED_VECTOR_LEVEL;

typedef struct
{
  DWORD dwDummy;               // nothing useful in here
  WORD  wVersion;              // Version of driver
  WORD  wNumberOfInterfaces;   // number of detected PCIADA
  struct
  {
    DWORD dIfcStatus;          // connection status of PCIADA-VMEMM
    DWORD dwLinkCount;         // how often this interface is
requested

    WORD  wModuleType;         // if connected: type of connected
module
    WORD  wFPGAVersion;        // if connected: Version of VMEMM FPGA
    WORD  wModuleNumber;       // if connected: Number of Connected
VMEMM
    WORD  wWordMode;           // if connected: Mode of operation

    WORD  wSysControl;         // if connected: VMEMM sysctl status
    WORD  wConnected;          // connected or not

    WORD  wDummy;
    char  cszHWRevision[10];   // revision of PCI interface
  } sPCIAda[VPCIVMED_MAX_PCIADA]; // status of each one
} VPCIVMED_PCIADA_STATUS;

typedef struct
{
  DWORD dwInterface;            // targets to interface number ...
  WORD  wResult;
```

```
} VPCIVMED_RESET_RESULT;       // polling result: in progress if
(wResult != 0)

#define __PCIVMEH_H__
#endif
```

APPENDIX D: Standard initialization procedure

The standard initialization procedure is summarized in the following array:

```
{LCR,  WORD_ACCESS, 0x4c, 0x0009}               // disable
interrupts
{LCR,  WORD_ACCESS, 0x50, 0x4180}              // enable
interface

{VIC,  BYTE_ACCESS, (WORD)0x03, 0xf8+1}    // VIICR

{VIC,  BYTE_ACCESS, (WORD)0x07, 0x78+1}    // VICR1
{VIC,  BYTE_ACCESS, (WORD)0x0b, 0x78+2}
{VIC,  BYTE_ACCESS, (WORD)0x0f, 0x78+3}
{VIC,  BYTE_ACCESS, (WORD)0x13, 0x78+4}
{VIC,  BYTE_ACCESS, (WORD)0x17, 0x78+5}
{VIC,  BYTE_ACCESS, (WORD)0x1b, 0x78+6}
{VIC,  BYTE_ACCESS, (WORD)0x1f, 0x78+7}     // VICR7

{VIC,  BYTE_ACCESS, (WORD)0x23, 0xf8+0}     // DSICR

{VIC,  BYTE_ACCESS, (WORD)0x27, 0xf8+1}     // LICR1
{VIC,  BYTE_ACCESS, (WORD)0x2b, 0xf8+2}
{VIC,  BYTE_ACCESS, (WORD)0x2f, 0xf8+3}
{VIC,  BYTE_ACCESS, (WORD)0x33, 0xf8+4}
{VIC,  BYTE_ACCESS, (WORD)0x37, 0xf8+5}
{VIC,  BYTE_ACCESS, (WORD)0x3b, 0x38+6}
{VIC,  BYTE_ACCESS, (WORD)0x3f, 0x38+7}     // LICR7

{VIC,  BYTE_ACCESS, (WORD)0x43, 0xf8+2}     // ICGS
{VIC,  BYTE_ACCESS, (WORD)0x47, 0xf8+3}     // ICMS

{VIC,  BYTE_ACCESS, (WORD)0x4b, 0xe8+6}     // EGICR

{VIC,  BYTE_ACCESS, (WORD)0x4f, 0x08}         // ICGS-IVBR (!)
{VIC,  BYTE_ACCESS, (WORD)0x53, 0x0c}         // ICMS-IVBR (!)

{VIC,  BYTE_ACCESS, (WORD)0x57, 0x00}        // LIVBR (!)

{VIC,  BYTE_ACCESS, (WORD)0x5b, 0x10}        // EGIVBR (!)

{VIC,  BYTE_ACCESS, (WORD)0x5f, 0x00}        // ICSR

{VIC,  BYTE_ACCESS, (WORD)0x63, 0x00}       // ICR0
{VIC,  BYTE_ACCESS, (WORD)0x67, 0x00}
{VIC,  BYTE_ACCESS, (WORD)0x6b, 0x00}
{VIC,  BYTE_ACCESS, (WORD)0x6f, 0x00}
{VIC,  BYTE_ACCESS, (WORD)0x73, 0x00}       // ICR4

{VIC,  BYTE_ACCESS, (WORD)0x83, 0xfe}       // VIRSR

{VIC,  BYTE_ACCESS, (WORD)0x87, 0x0f}       // VIVR1
{VIC,  BYTE_ACCESS, (WORD)0x8b, 0x0f}
{VIC,  BYTE_ACCESS, (WORD)0x8f, 0x0f}
{VIC,  BYTE_ACCESS, (WORD)0x93, 0x0f}
{VIC,  BYTE_ACCESS, (WORD)0x97, 0x0f}
{VIC,  BYTE_ACCESS, (WORD)0x9b, 0x0f}
{VIC,  BYTE_ACCESS, (WORD)0x9f, 0x0f}       // VIVR7

{VIC,  BYTE_ACCESS, (WORD)0xa3, 0x3c}       // TTR  - 16 usec
```

```
{VIC,  BYTE_ACCESS, (WORD)0xb3, 0x40}       // ARCR
{VIC,  BYTE_ACCESS, (WORD)0xb7, 0x29}       // AMSR
{VIC,  BYTE_ACCESS, (WORD)0xd3, 0x00}       // RCR

{IFR,  LONG_ACCESS, (WORD)ADRHL, 0xF0F0F0F0}   // ADR-H, ADR-L
{IFR,  WORD_ACCESS, (WORD)CSR  , 0x0000}  // Contr-Reg

{VIC,  BYTE_ACCESS, (WORD)0x7f, 0x80}         // ICR7

{LCR,  WORD_ACCESS, 0x4c, 0x0009}                // disable
interrupts

{STOP, WORD_ACCESS, 0,     0}
```

APPENDIX E: Standard deinitialization procedure

Deinitialization is divided into two part. Part one is run before the user deinitialization:

```
{VIC,  BYTE_ACCESS, (WORD)0x7f, 0x00},      // ICR7 - set
SYSFAIL
{LCR,  WORD_ACCESS, 0x4c, 0x0009},             // disable
interrupts
{STOP, WORD_ACCESS, 0,    0}};
```

Part two starts after the user commands:

```
{LCR,  WORD_ACCESS, 0x50, 0x4080},             // disable
interface
{STOP, WORD_ACCESS, 0,    0}};
```