Version

4

N4 COMMUNICATIONS CO

DriveVME Users Guide and Reference

Version 4.3, November 2006.

DriveVME

N4 COMMUNICATIONS CO

DriveVME User’s Guide and Reference

TABLE OF CONTENTS

DriveVME - Introduction

DriveVME is a collection of Windows NT/XP programs, libraries and device drivers that maximizes particular VME processor board features on both local and remotely networked chassis while offering standard interfaces to end users and programmers at all requirement and skill levels. DriveVME aims to be the definitive statement in Windows-to- VME interfacing.

DriveVME completely manages and shares all aspects of the processing environment's VME interface, making it possible for user programs and user created native NT device drivers that make use of VME resources to be developed and maintained independently -- yet run concurrently, and without unnecessary processing platform hardware dependencies.

NOTICE - NOTICE - NOTICE

Read the file DriveVMEReadME.txt to learn of last minute changes and important notices of hardware errata, operating limitations and capabilities specific to your hardware. It is displayed via the 'Getting Started' VME operations Menu Option.

At this writing DriveVME is supported on 20 processing platforms including designs incorporating the Tundra/Universe I, II and IIB and 148 PCI-VME interface chips, the BIT3 PCI – VME chassis links, Cypress chip as well as other Pentium/Athlon and compatible designs.

To use this manual or DriveVME successfully, you must have a basic knowledge what interaction the VME resource you wish to work with has with the VME bus. This includes knowledge about the VME device such as which VME addresses and interrupts it uses that are interesting to your requirements. It does not require understanding in particular how the processor board manipulates the VME bus.

This manual does not attempt to describe in detail what the VME bus is, or even most of its electrical and other specifications. That said, nearly everything relevant about the VME bus from the programmer's perspective and much about it's logical organization can be inferred from reading the chapter "AssistVME". For more information on the VME bus, see www.vita.com .

You do not need to be a Windows programmer in particular or even able to program at all to make full use of DriveVME's VME diagnostic and exploratory abilities. On the other end of the spectrum, experienced real-time C programmers will find a full suite of fully managed interface points, fully exposing every VME ability the hardware affords.

DriveVME is distributed in three generic versions:

1. "DVMEDDK". DriveVME Development Kit is a licensed fully featured VME development and diagnostic environment. Although it offers interface points to other device drivers, file systems, network users, DLL programmers and GUI application developers, its most visible end-user application is "AssistVME" – called by some as "Fully featured VME – but with removable training wheels". DVMEDDK is licensed like a compiler, one license per developer or end user.

2. "DVMERUN". Programs and device drivers developed using DVMEDDK can take advantage of DVMERUN. DVMERUN provides all the program access points of DVMEDDK and only error reporting end-user features, while consuming fewer memory and storage resources and at approximately one tenth

the cost. DVMERUN is licensed as though it were a part in a system, with license ownership transferring with the ownership of the system in which it is embedded.

3. "DriveVME Eval". For all the vendors whose processors are available to the public, N4 Communications offers a free, fully functional but time limited version of DVMEDDK on its website, www.n4comm.com for evaluation. DriveVME is a system's design level product, and as such users are assumed to be knowledgable about and assume responsibility for the technical complexities, risks and particular performance they require. The evaluation version of DriveVME exists to help make those determinations prior to purchase of a license.

DriveVME is not sold to end-users directly, but only through processor board manufacturers and embedded systems manufacturers. Contact your processor vendor about purchasing DriveVME.

Matching DriveVME's Abilities with Your Requirements

Use this section to determine which of DriveVME's features meet your needs, then go the relevant section of this manual for appropriate detail.

1. "I want an easy to use graphical program that fully exposes all local or network remote VME chassis resources for exploratory or diagnostic purposes, including a way to take snapshots of VME settings to email for support purposes."
   --If this describes your needs, read the chapter titled "ASSIST-VME".

2. "I want a nothing-fancy quick way to read a chunk of data from either a local or remote VME bus directly into my spreadsheet or database application".
   --If this describes your needs, read the chapter titled "ASSIST-VME" and the "file-oriented VME I/O" sub-section.

3. "I'm a Visual-Basic or FoxPro or Office comfortable programmer, and can open and close and read and write files, but I'm leery about DLL's. I want to do a few things with VME resources on a local or networked-remote chassis. Some of it may involve dealing with VME interrupts, but not more than a dozen or so per second and if there occasionally are short but variable delays between when the interrupt happens and my user program gets to it, that's ok."
   --If this describes your needs, read the chapter titled "file-oriented VME I/O" and "file-oriented interrupt handling" sub-sections.

4. "I'm able to write stand alone Windows user-level applications that have graphical interfaces and can use dynamic link libraries. I want to create a stand-alone program to have full access to a great variety of VME resources on a the local or a networked-remote chassis, and am willing to trade a little performance for rapid development and easier-to-understand code."
   --If this describes your needs, read the chapter titled "The DriveVME DLL".

5. "I'm a Windows 'C' or 'C++' programmer and wish to create a user application that uses only a few VME resources and/or requires the highest possible performance available to a user-level program on a local or remote chassis."
   --If this describes your needs, read the chapters including "file I/O initiated access to VME resources".

6. "I'm a programmer who knows the 'C' language and I may use Windows or may have programs based on other platforms. I want to create a high-performance VME program that is run and interpreted in the kernel of a local or remote Windows VME chassis, possibly to the exclusion of anything else on the system from time to time. I want this program to communicate its results and take direction from a user – level program on the same or possibly another chassis that may or may not be running Windows at all. And I don't want to have to deal with learning Windows device driver level or even Windows programming at all."
   --If this describes your needs, read the chapter "Interpreted Kernel 'C' VME via file-IO 'console'". (This feature allows a TCP/IP or network-sharable 'file' to accept a VME-capable K&R C program --which is then compiled to P-code and run in the kernel of the target processor. The C program's console I/O is accessed by further read and writes to the same file handle the program was sent in on).

7. "I have a VME device I want to integrate into the Windows/NT environment, just as though it were any other NT device like a serial port or network card. I am able to write NT/2000 device drivers and just need to know how to hook into DriveVME to gain access to the local VME bus."
   –If this describes your needs, read the chapter "Writing a VME – aware NT/2000/XP Device Driver"

8. "My application calls for the shortest possible delay between when a VME interrupt happens and when the processor responds to it, or must run critical operations without the possibility of interruptions. There may be thousands of interrupts per second to deal with in a versatile way. It's more than DriveVME's native kernel C interpreter can handle. I am willing to create a Windows device driver to respond to these interrupts at the highest possible performance, then have that accessed by my user-mode application for lower-priority processing. How do I hook into DriveVME at the device-driver kernel level?"
   –If this describes your needs, read the chapter "Writing a VME – aware NT/2000 Device Driver"

DRIVEVME INSTALLATION

DISTRIBUTION METHODS AND FILE NAMING

All versions of DriveVME are distributed complete in one executable file. The first two letters of the file name are the key to the general nature of the contents.

- **"DV….exe"** Distribution files that begin with "DV" are fully functional but time – limited, freely distributed evaluation versions of the DriveVME DDK.

- **"N4….exe"** Distribution files that begin with "N4" are fully functional, licensed versions of the DriveVME development kit.

- **"Nr…exe"** Distribution files that begin with "Nr" are fully functional, licensed DriveVME runtime environments.

The next set of up to 3 digits reflect the version number of the software, the first digit reflecting the major version number, and the rest the minor release number. For example, "DV312…exe" would be the evaluation DDK for version 3.12.

The remaining letters and digits are keys to the particular processor make and model that version of DriveVME is supports. Different processor manufacturers use varied approaches in their VME implementations, all of which have an effect on any programs that relate to the VME bus. There is variability in memory management, interrupt routing, data-lane (byte-swapping) hardware, lamps, switches, bus bridges, and even generic access. DriveVME connects the various features into standard interface points so that software written to DriveVME will run across VME processor platforms and models. For example, DV10XY4.exe specifies the DriveVME evaluation kit version 1.0 for a 486 class process made by Xycom Corp. in the mid 1990's – the first platform supported by DriveVME.

Licensed versions of DriveVME are distributed on CD-ROM's, in most cases one per purchase order received. A single copy of each of the ordered software packages is recorded on the CD-ROM, along with a proof-of-license (which is in fact part of N4 Communication's invoice for the order.)

INSTALLING DRIVEVME

< FIRST: REMOVE ANY PREVIOUS VERSIONS THEN REBOOT >

Before installing DriveVME, it is essential that any previous version (no matter the type of the previous version) be uninstalled, and the system rebooted. No copy of DriveVME can be running in any fashion or the installation of the new version of DriveVME will certainly fail. This is because DriveVME manages the VME interface, a system level resource, and also incorporates itself into the file system (the "V" drive refers to DriveVME) – Windows does not support removing this type of software without rebooting. To uninstall DriveVME, choose the "Uninstall" option in the "VME Operations" section of the "Programs" part of the "Start" button menu. Or, run the program "remove.bat" in the directory "C:\DriveVME". Choose "OK" at the various prompts that appear, then do a "cold" reboot of the system.

Installing DriveVME is as simple as running the single program file appropriate to your processor. You must have administrative privileges to install DriveVME.

Use the naming system described above to select the appropriate file. If all goes well, the installation will "unpack" all the DriveVME components into a directory "C:\DriveVME", then go about placing files in the "system32\drivers" and "system32" directory of your Windows folder. It will update the registry then launch the device driver and supporting error alerting service, as well as VME 'viewable' ram disks. It will create a "VME Operations" program group, which provides access to "AssistVME" – the user-level VME exploratory and diagnostic and utility program.

If all goes well, you should see a message to the effect "DriveVME installed and running". Click "OK", at this point, installation is complete. DriveVME is fully available from that point forward, and will re-install itself every reboot until removed.

It is easy to check whether DriveVME is running properly on any system. DriveVME establishes what appears to be a "disk" drive, it uses the letter "V". Use windows to display the contents of the "V" drive. If it exists and DriveVME is running properly, you should see, among other things, a file titled "aboutn4.txt". Displaying this file should offer the version number of DriveVME running, and contact information.

System Administrators can control access to DriveVME, and thereby the VME bus by applying desired security measures to the "V" drive.

TROUBLESHOOTING INSTALLATION PROBLEMS

The most typical reason for installation failure is incomplete removal of evaluation DriveVME versions. See above for that procedure. Other reasons for installation failure are recorded in the Windows/NT "Event Viewer", viewable from the "Administrative Tools" launch menu. Be sure to have those messages available to you when you are seeking support help. The most common remaining reasons for launch failure are:

1. **Incomplete removal of other software which manages VME access hardware.** DriveVME manages all aspects of the relationship between Windows/NT/2K/XP and the VME bus. Any other software which attempts to alter the devices which control that relationship would render the system's operations unpredictable. You must completely remove from activity any other device drivers or programs which "claim" the resources related to the VME interface logic from the system then reboot. Some of the processor vendors supported by DriveVME offer their own basic VME management drivers, these must be uninstalled before DriveVME can function reliably.

   Note: N4 in the past has implemented "compatibility" packages that allow software written to vendor-specific VME driver specifications to operate with DriveVME with no change. If you have existing applications written to use vendor specific drivers, contact N4 relative to creating a such a compatibility package for you.

2. **Interrupt sharing conflicts with other devices block DriveVME from loading.** DriveVME requires that at least one interrupt attached to the VME interface logic be routed and available for use by Windows/NT. Usually the system "BIOS" setup screens offer users a way to enable or establish this interrupt access and routing. Contact your processor vendor for details about this setting.

   Also, in nearly every case, DriveVME is set by processor vendors not to "share" the interrupt associated with all VME interrupts with any other devices. That is because DriveVME seeks to offer known latencies between when a VME interrupt occurs and when it is processed by the user. If other devices can also use the interrupt line used by DriveVME, then the VME interrupt service latencies can vary with the interrupt traffic generated by non-VME devices. If DriveVME's reason it can't start is interrupt related, use the "Windows NT diagnostics" under "resources" / "IRQ" to check for the assignment of an additional device to the same line the VME interface is set to use. To resolve these issues, most vendors offer system BIOS settings to enable and disable devices and select the interrupt lines they use.

   Rest assured that if DriveVME is offered for a particular processor make and model, it has passed an evaluation and checkout suite of tests and is known to work with the version supplied to N4 by the processor vendor.

   Some very few vendors, with specific applications in mind for their hardware, intentionally share the VME interrupt with other resources, your vendor can inform you if this is the case for your processor.

3. **Memory mapping conflicts.** One of DriveVME's main functions is to offer access to VME addresses via the processor's address space. There are very many distinct and incompatible ways the VME bus can be addressed, so different applications will nearly certainly need to access the VME bus in a manner entirely incompatible with other applications. Most versions of DriveVME reserve a little over 3GB of the 4GB address space. If your application calls for more than 800 Meg or more of physical memory, contact N4 for a version of DriveVME that offers less VME options, but allows for more physical RAM in the system.

   Alternately, this conflict can arise when custom devices (not typically installed with the vendor's hardware) use PCI address space are located "smack in the middle" of the NT physical address space. Using the configuration programs associated with these devices to move them to the high or low end of the address space usually resolves these issues.

4. **Out of date BIOS versions.** Early in the development cycle, most processor vendor's BIOS versions fail to implement the VME interface setup fully. They either overlook to connect the interrupt system to the VME chip, or other issues that effectively "hide" the interface from Window/NT standard hardware "HAL" manipulation logic. As DriveVME is very careful to follow all the Windows conventions, it attempts to "reserve" or "claim" or "register" the use of the VME chip's resources. Incomplete BIOS implementations cause NT to report that no such resources are indeed available (even if in fact they are); or that even if available- they aren't actually usable. If none of the above items in the troubleshooting list appear to solve the problem, check with the processor vendor for the latest BIOS updates. Drivers other than DriveVME sometimes "work" in these degraded circumstances with this hardware because they don't follow Microsoft prescribed practices of notifying the O S about claimed resources so as to avoid conflicts.

   (NOTE: Be advised that in very rare circumstances, a very few processor vendors, mostly dedicated application vendors, made incompatible BIOS changes after their products were qualified with DriveVME – so that the earlier BIOS worked and the later ones would not. If your system had a recent BIOS change and DriveVME fails to load, and none of the issues 1-3 above seem to apply, check this option with your processor vendor.)

If none of the above solutions permits DriveVME to load properly, the first line of support is the group from whom you purchased the system. If your system uses VME processors as an embedded part of a larger application, then contact the embedded system company's support group first, then the processor vendor's support company.

As DriveVME is not sold directly to end users, N4 Communications will only participate in direct end-user support if the process is managed and includes the active involvement of support personnel from a company reselling DriveVME. That said, N4 Communications is dedicated to the success of any group using its products and will cooperate fully in the resolution of such problems as may arise.

The DriveVME Environment – Files, Connections & Resources

This section outlines the various major elements of DriveVME, where they reside and a brief description of their role in providing VME resources to you. While it doesn't discuss operational aspects of DriveVME, it provides a roadmap to DriveVME's impact on the system and a quick locator of which files support what features. It assumes that DriveVME is up and running in your VME / Windows environment.

One basic key fact about DriveVME to keep in mind is that DriveVME offers most of its abilities through using the regular Windows file system to access very special purpose "files" on a psuedo disk drive "V:\" created by DriveVME. Sharing, security, specifications and so forth are specified through carefully constructed names to particular "folders" on the "V" drive. Even reading the starting address of arrays in user programs that have relationships to VME operations is through these files.

> *New in Version 4, under Windows XP:* By default the security is set so that only the system administrators, kernel drivers and system processes can access the V drive. As access to the V drive is in effect a window into the kernel operations, this level of protection is mandatory. The file DriveVMEDD.h defines:
>
> GUID_DRIVEVME_DEVICE_INTERFACE_CLASS
>
> Under which the V:\ is opened. Therefore, users who wish to modify this device setup class can override the V:\ drive's type, exclusivity settings (shared) and security parameters (restrictive by default). See "Setting Device Object Registry Properties After Installation" for details in Microsoft's Windows DDK Kernel Mode Architecture document for detail.

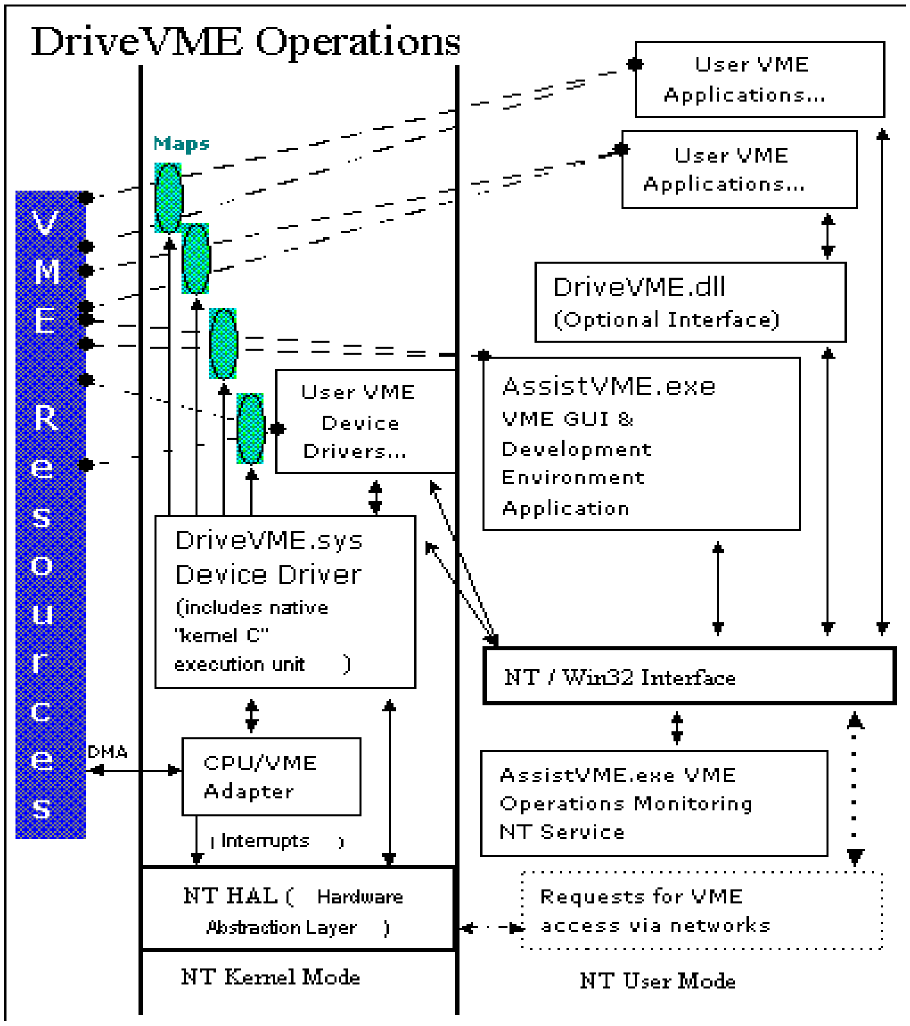
The helpful graphical user program "AssistVME", key to doing development with DriveVME, has as it's principal function translating human-understandable desires relative to VME activity into the appropriate “VME” file name and transactions with the specially created VME “file”.

DriveVME offers specific services requested during the time the “file” is held open, and ceases to undertake the specified activity when the “file” is closed. Various features of DriveVME’s DLL or user library all boil down to calls and files routed through this venue.

The “V” drive can be shared over a network just as any other drive can be, and through that mechanism VME resources can be offered to off-card and even off-chassis programs.

Even DriveVME’s special purpose kernel mode ‘C’ compiler and P-Code interpreter is accessed through reading and writing to a special “file” on the “V” drive.

General Relationship of DriveVME to Windows NT


DriveVME Operations
VME Resources
Maps
DMA
User VME Applications...
User VME Applications...
DriveVME.dll (Optional Interface)
AssistVME.exe VME GUI & Development Environment Application
User VME Device Drivers...
DriveVME.sys Device Driver (includes native "kernel C" execution unit)
NT / Win32 Interface
CPU/VME Adapter
(Interrupts)
AssistVME.exe VME Operations Monitoring NT Service
NT HAL (Hardware Abstraction Layer)
Requests for VME access via networks
NT Kernel Mode
NT User Mode

DriveVME and The Registry

DriveVME adds entries to the Windows/NT registry. In particular, the “current control set” “services” key includes configuration information for AssistVME (the graphical VME exploratory program’s support service); DriveVME (the main VME interface device driver); and VMEDisk (the facility that allows local or VME memory to act as a “RAM” disk on the system). There shouldn’t be any typical occasion for users to understand or directly manipulate these registry items directly. Windows itself adds items to the “Resourcemap” keys when DriveVME specifies VME interface chip IO, interrupt and other requirements.

C:\DRIVEVME – DriveVME's file repository

The single DriveVME distribution file "unpacks" by default into the directory at c:\DriveVME. While all files needed for active operations are copied elsewhere, many development and informational files exist only there.

After any installation, take a moment to check the contents of "DriveVMEReadMe.Txt" in this directory. The latest features and version change reports are loaded there. Here is a catalog:

- DriveVME.SYS
  This file is a Windows kernel mode device driver which actually provides the great majority of all the VME related services. It is the same for development and runtime versions. The active copy of this file resides in the active windows\system32\drivers directory. It is loaded very early on in the boot process. Its use is essential to DriveVME operations.

  - AssistVME.EXE
    This user mode file has many different roles. It is run as a system service every system boot, where without user intervention, it watches for unusual events (such as bus errors) from DriveVME. When they occur, it launches another copy of itself in the typical interactive 'user help' mode, explaining the event and offering assistance. If a copy is already running in user mode, then it pops to the foreground with an updated screen discussing the VME event. The active copy of this file is also in the drivers directory.

    The development version of AssistVME is very large, offering extensive diagnostic and development services. The runtime version is very streamlined, offering error logging services only.

    It is not necessary to run AssistVME to use DriveVME. To disable AssistVME, go to the "services" section of the "control panel", look for "VME operations assistance service", then select "stop". This will disable the service until the next reboot. If you want to stop the service from being launched automatically until further action, choose the startup button, then select "manual" startup options.

    Note that even when not launched automatically, running "AssistVME -Helpuser" is available at any time to get the latest VME operations information.

    AssistVME is not essential to DriveVME operations, but provides tremendous added value.

    ASSISTVME.HLP -Optional AssistVME text help file

- VMEDisk.sys

  This device driver provides VME RamDrive Support. This is an add on product. It allows VME memory or PC memory exposed to VME either on a local or remote chassis to be accessed like a local hard drive with a FAT file format. Choose the disk icon in AssistVME to review current selections, Choose either the slave, master or memory mapped "Software" options to establish the disk accessing VME with the selected resources.

  NOTE: If you wish to establish a local ram disk which refers to VME memory on a chassis over a network, then AFTER NORMAL INSTALLATION you must: go to the services option of the control panel- look for VME Operations Assistance Service. Choose Startup. Then change "Log On As" to refer to a specific account and password that has access to the desired remote network resource. Save the changes then reboot. From then on, the system will make the remote VME resource mapped appear to be a local "disk" drive.

- Files associated with easy access to DriveVME though the user mode dynamic link library "DLL":

1. DriveVME.dll -Handy user routines which run a kernel "C" Java program delivering common DriveVME features to user mode programs.
2. DVMEDLL.h -Include file for DriveVME.DLL users (contains many examples and documentation).
3. DriveVME.exp -File some developers need to use ""
4. DriveVME.lib -File some developers need to link for ""

- System "Include" files for DriveVME's kernel "C" Interpreter (copied to the system32\drivers directory) NOTE – THESE ARE NOT ASSOCIATED WITH THE DLL MENTIONED ABOVE:

1. CTYPE.H -Kernel "C" include file for strings.
2. DIR.H -Kernel "C" file search includes
3. Fcntl.h -Kernel "C" Java file handler header.
4. MATH.H -Kernel "C" Java math function header.
5. Ports.h -Kernel "C" Java I/O port access header.
6. Realtime.h -Kernel "C" Java realtime services header
7. SETJMP.H -Kernel "C" Java setjmp/longjmp declarators
8. STDARG.H -Kernel "C" Java varags declarations.
9. Stdio.h -Kernel "C" Java fileIO declrations.
10. Stdlib.h -Kernel "C" Java library functions

11. String.h -Kernel "C" Java string functions.
12. Universe.h -Tundra/Newbridge Universe Chip declarations.
13. Tsi148.h -Tundra 148 PCI-x / VME Chip declarations.
14. Vmeints.h -Kernel "C" Java VME interrupt declarations.

- Kernel "C" Interpreter Sample Code files (Open with AssistVME) NOTE: THESE PROGRAMS RUN IN THE KERNEL UNDER DRIVEVME'S C RUNTIME ENVIRONMENT. THEY ARE NOT ASSOCIATED WITH THE DLL.

1. ConioDemo.c -Kernel "C" function showing I/O functions
2. demo.c -Kernel "C" VME interrupt demonstration.
3. FcntlCheck.c -Check file for routines in Fcntl.h
4. PortsCheck.c -Checkout program for routines in Ports.h
5. RealtimeCheck.c -Kernel "C" Java realtime services demo
6. StdIOCheck.c -Checkout program for routines in stdio.h
7. StdlibCheck.c -Checkout program for stdlib.h
8. StringCheck.c -Checkout program for routines in string.h
9. UniverseCheck.c -Kernel "C" Java program to check out accessing the Universe chip.
10. Tsi148Check.c -Kernel "C" Java program to check out accessing the Tsi148 chip.

- Source examples and details relative to writing Windows NT/2000 device drivers which call DriveVME:

1. DDKDemo.zip -Sample NT kernel driver using Drive/VME
2. dvmeddk.c -sample kernel mode DriveVME device driver
3. DVMEDDK.h -Include file for device driver authors

- Miscellaneous Example and Info Files:

1. DriveVMEReadME.txt – Recent updates and latest news.
2. DvmeMFCDemo.exe -Demo Windows app to catch bus error messages, sample program
3. DvmeMFCDemo.zip - Sample Source and development studio for ""
4. REMOVE.BAT -Uninstall Drive/VME

5. TestDriveVME.exe -Console application exercising DriveVME.DLL
6. TstDLSrc.zip -Source for TestDriveVME.exe
7. vmeassist.bat -AssistVME launcher (obsolete)
8. NOTE: Early Xycom emulation libraries are restricted for use on Xycom hardware only --not currently provided for non Intel platforms.

xvme.dll	-N4's Xycom library DriveVME emulator
xvme.exp	-N4's export library for ""
xvme.lib	-N4's link library for ""
xvmeisr.dll	-N4's Xycom library DriveVME emulator ""
xvmeisr.exp	-N4's export library for ""
xvmeisr.lib	-N4's link library for ""

AssistVME – Your Window on VME Operations

Once DriveVME is installed and running, most users will explore its abilities using first by running AssistVME. AssistVME can be launched from the "start" menu under the "VME Operations" section under "VME Operations Assistance". In the sections following this one, it is presumed that the reader is running a copy of AssistVME (available for free in evaluation mode on the internet at http://www.n4comm.com).

AssistVME is a graphic user interface Windows program that has many abilities and purposes, it would be a rare user that requires the ability to successfully use all of them. AssistVME's features can be generally grouped into four areas:

1. **Local Chassis VME bus exploration & configuration.**
   AssistVME offers the ability to generate and respond to just about every kind of VME interaction there is, as well as report on and offer to alter the status of major system configuration settings, such as whether the processor in question is the system controller or not. This includes the ability to offer local memory and resources to other VME bus masters, to become a bus master and access other VME resources; generate, monitor or handle interrupts and more. AssistVME allows combinations of access choices to be saved as special purpose files, which can be emailed and reloaded for diagnostic purposes. Also, AssistVME acts as the configuration area for VMEDisk, the accessory to DriveVME that makes Windows/NT "ram" disks out of VME available memory.

2. **Remote Chassis VME bus exploration & configuration.**
   AssistVME has the ability to reach out over networks to locate systems with DriveVME loaded and which have set system security in such wise as remote access is permitted. Nearly all features available to DriveVME users on the local chassis is enabled from remote AssistVME running systems (which may not necessarily even have a VME bus, such as laptop computers for diagnostics, say.)

3. **Programmer's assistance for creating VME operations specifications.**
   DriveVME offers programmers access to VME resources by reading and writing from special files. The names of these files are long and specify nearly everything about the type and nature of the VME transaction desired. DriveVME even uses this "file" method to report which memory address in the user's program space marks the beginning of an array which when

accessed will generate VME transactions of a desired type (the array being valid only so long as the related file is open). Checks for security and conflicts with other programs using the VME bus occurs at file open time.

The possibilities and variations of VME access methods are enormous. AssistVME lets the programmer "try out", save, email, reload and generally explore the results of various combinations and selections using rich "property tab" step by step graphic environment, building the specialized "file name" required by programs to repeat the successful method. AssistVME even has the ability to generate a sample program that incorporates the user's choices and exercises them, to speed development.

4. **DriveVME Kernel "C" Development Environment.**
   The DriveVME device driver, which runs at the highest available performance levels of the Windows operating system, has the ability to accept text "C" programs, compile then to a "P" code then execute them on the fly. The programs are loaded by "writing" them to a special file, then "run" by reading the same file handle. Subsequent reading and writing on the file delivers the running kernel 'C' program's "stdio" print and scan statements, respectively.

   AssistVME offers a complete development environment for this facility, with the ability to set breakpoints, monitor values and set debugging breakpoints.

   The Interpreted Kernel "C" environment makes it possible for the programmer to execute small, highly VME device specific programs at relatively high speed without writing windows device drivers. Indeed using the network features of DriveVME, remote systems which may not be Windows based at all can inject a 'C' program directly into the kernel of a processor on a VME chassis and run it and relate to it remotely.

5. **Error Log Review and Monitoring.**
   DriveVME reports errors and unusual events to the Windows NT Event Log. However, this log offers limited detail and usability. DriveVME offers its own event logging service, which offers much more detail about the sources, time and effects of unusual VME bus related events. Assist VME displays and manages this log file. Indeed this log file management ability is the only AssistVME feature enabled in the runtime version of DriveVME.

Each area outlined above is the topic of the sections to follow. The reader will find it helpful to have a copy of AssistVME running while reading the following sections.

Local Chassis VME bus exploration & configuration with AssistVME.

This chapter will outline how to use AssistVME to manage and explore all aspects of VME related operations. We presume the reader to be seated next to a running copy of AssistVME (available for free in an evaluation mode at http://www.n4comm.com).

When AssistVME first runs, it presents a menu bar, a "tool" bar, then a work area to hold VME operations windows. By default, the first window to appear is the log of recent unusual VME events in the VME log.

The VME Operations Log Window

The VME log window will appear automatically when DriveVME detects any usual event, or when the toolbar that looks like a yellow pencil on a green pad is chosen, or from the menu entry "Open Operations log" in the "file" menu section. Entries in this log appear whenever a bus error occurs, a syntax error relative to DriveVME resources is detected, and other VME related events.

The "File" menu has an entry near the bottom which will erase the contents of the operations log.

This is the only AssistVME function available in the runtime version of DriveVME.

VME and PCI chassis configuration

This selection permits the review and change of all major PCI and VME chassis-wide setup parameters. To bring up this window, choose the "gear" icon from the toolbar or the "Setup and examine chassis operating procedures" from the "Chassis selection and setup" menu.

This action brings up the "Configure Chassis Overall Setup" window, in this case for the local chassis. The window offers three general areas of detail, the first regards the VME bus and its "system controller" issues, the second applies to those implementations that also have a PCI bus, and the third for miscellaneous control items, such as lamp and switch management.

All the settings in this window can be saved in a "document", emailed to others who have this version of AssistVME and loaded there.

NOTE: Earlier versions of AssistVME had a bug which attempted to load versions of these ".vsu" chassis setup and configuration files and failed, but corrupted the AssistVME program in the process. The corruption blocked changes in further attempts to set up the chassis. If you think this is happening to you, use the "find file" utility to scan for and delete all files with the ".vsu" extension, then reboot. Then, the chassis setup options will resume normal operations.

- **The "Syscon" page: Chassis-Wide VME settings**

This page / property-tab of the chassis setup window displays all the information about the VME interface that apply equally to all programs using the VME bus on this chassis.

The primary concern of this page is the "system controller" setting. Only one card in a VME backplane can offer the "system controller" signals and services. Even though many cards equipped to do the job can be physically in the same chassis, exactly one must be enabled to do so. Furthermore, the one enabled to do so must be physically located to the left (as viewed by a person looking at the front of the chassis) of all the other VME cards that are active in sending signals on the bus. If more than one card has this facility enabled, or if no card having this facility enabled, then VME interactions will at the very least be filled with errors.

Most typically, our experience is that when the system controller setup is wrong, the first attempt to access VME resources brings the system attempting to do so to a direct and complete lock-up halt.

Various VME interfaces offer the ability to detect automatically whether it ought or ought not serve as the system controller. Depending on the engineering agreements relative to the processor card and the chassis vendor, this automatic facility may or may not work or may or may not be reliable. The section titled "Overall Chassis Control Services" offers a way to indicate to the hardware whether it ought to use its automatic system controller setup ability, or whether software should force the services to be enabled or disabled.
NOTE: Not all hardware vendors permit this to be controlled via software. Consult your processor manual.

The settings on this page are unique in that they are automatically applied every system boot.

This page also offers the administrator:

1. a way to tune how long to wait for other systems to respond when this system seeks to orchestrate which system may issue transactions on the bus [VME bus arbitration timeout];
2. how long to wait for a reply to a transaction before singaling an error [VME bus error timeout];
3. on what basis to grant control of the bus when two systems signal their request to initiate transactions at the same time. "Round Robin" grants requests to competing systems roughly on the basis that whichever waited before is first in line. "Priority" resolves control contests on the basis of cards installed closer to the system controller card first.

Clearly the choices directly affect chassis performance. Accepting the system defaults is typically the safest thing to do, but engineers who know the timing specifications of all the cards in their chassis can improve system performance considerably (particularly relative to the bus arbitration timeout setting).

The page also displays state information relative to all these choices.

- The PCI page: Details about the nature of the VME interface hardware

The majority of this page is given to the display of every available detail about the nature of the VME related interface chips, particularly in their relationship to the processor bus. Details about the processor card, the vendor and model, the revisions of the various chips, their locale on the PCI bus (if any) and more are displayed. The meanings of the technical descriptions are beyond the scope of this manual, you can get them from the PCI bus specification documents.

This page offers the ability for users to signal the VME chips that the PCI bus is capable of dealing with 64 bit VME transactions (PCI64).

Also, some vendors supply memory or otherwise arrange their hardware so that the PCI bus generates meaningless PCI bus parity error indications within the VME bridge chip logic. This page provides a mechanism for DriveVME to stop reporting those errors after the first 10 are displayed each boot up.

- **The Misc Page: Choose lamp and VME reset signaling settings.**

Many of the systems DriveVME supports offer front panel lamps and switches, others offer the ability to control what effect bus or switch signaled reset events should have. If your system supports these facilities, their default state and effect can be exercised and set here.

Generating VME cycles for response by other VME cards (other than interrupts) from the local processor

DriveVME and through it AssistVME offer two general ways to initiate VME cycles. Both involve specifying the nature of the VME transactions desired. They vary in the manner by which the actual cycles are conducted.

> This part of the manual is addressed to users of AssistVME who wish to directly work with VME resources. Users will note that beneath all the "control" areas of the windows that will arise, there is a section with a very long "file" name that is titled "copy the file name below to access the resource in programs" or similar. AssistVME users who wish to probe the VME bus can ignore that section, it is for the benefit of programmers. That section "captures" all the settings that will be chosen into a highly specialized "file" name which when opened by programs will deliver to the programs the desired VME access.

The first approach, known as "file I/O" generates VME transactions as the result of reading and writing to a file, just like any other file on the system. The benefits of this approach are that, being a file like any other, the transactions can be conducted over a network as well as from the local chassis. Also, DMA (direct memory access) transactions are available, as well as software assisted byte-swapping and other alignment and formatting options. The penalty of this approach is that because each request for VME traffic goes through the file system, there is a considerable overhead placed on each request. Using this method makes the most sense when the highest available performance relative to VME transactions is not required, when DMA access is required, when any reformatting is required, when networked operations are required, or generally when the amount of data transferred per request is 1K or more.

The second approach, known as “memory mapped IO” generates VME transactions as the result of typical program access to the contents of an array. DriveVME accepts a description of the nature of the VME transactions desired, then while a special file handle is open places an array into the memory space of the calling program or device driver. The array has the special property that when it is written to or read from, the data is not satisfied by local processor memory but instead is satisfied by VME resources on other cards in the chassis. The benefits to this approach are that it is easily and by far the fastest way to randomly access the memory on offcard VME resources. The shortcomings are that this facility, by its very nature, can not be run over a network – so programs that make use of it are forced to run on-chassis; if the processor platform provides hardware “byte-swapping” or “data-lane” swapping, then programs that make use of this feature can only be run on processors that support it (the file oriented method does software byte swapping when run on platforms that don’t support it in hardware); DMA operations are not available to users of arrays; as the program doing the I/O is typically a user program, it may be swapped out in favor of other tasks and so the transactions on the VME bus may or may not have the same timing patterns each pass through the code.

DriveVME establishes all VME resources as having an “IO” nature as opposed to a “memory” nature. The impact of this choice is that the processor will not attempt to do “cache line fills” with VME memory, and [generally speaking, subject to overriding user choices] when a VME cycle is complete from the perspective of the program it is also complete from the perspective of the VME hardware (it is not delayed). The benefit is that only the exact transactions called for on the VME bus will be done. The penalty is that the hardware will not access VME resources as efficiently as it does local RAM memory.

The AssistVME facilities for both these approaches are similar and will be treated together.

To set up VME transactions by File I/O, choose the “binoculars” icon on the toolbar or the obvious choice from the “file” menu. To setup up VME transactions by memory map, choose the “map” icon on the toolbar or the obvious choice in the “file” menu.

Note: These choices bring up similar “document” windows that are associated with files that vary in their extension. File I/O based VME transaction settings are saved in files with the extension “.vme”. Memory mapped I/O based VME transaction settings are saved in files with the extension “.vmm”. Once you have established a group of settings that are satisfactory, you can save them in a file whose name you choose using the “save” option in the file menu. If you want to revisit previously saved settings, go to the file menu and “open” the file containing the settings you like. This is a powerful diagnostic aid, as settings changes that have diagnostic meanings can be saved on one system and emailed to other locations.

Address Space Selection

Both windows first present the “Space” property page. The VME bus offers many different memory spaces. Reading address “0” in one space will not result in the same data as address “0” in another. The most typically used VME “spaces” are “A16” – offering 16 bits of address information, “A24” with 24 and “A32” with 32. These were generally historically evolved, when decoding more address lines in hardware called for additional expense, and wiring VME busses with further connectors was required to offer more addressing. The address space you require is specified by the settings on the VME resource you with wish to interact.

Technically, the “space” is signaled on the VME bus by a “user AM” code. Exotic hardware can operate in its own “space” by specifying an “AM” code which is otherwise unused. If your hardware requires such, it can be entered manually on the “space” settings page.

Address Range Selection

Both memory mapped and file I/O methods offer the “Address Range Selection” page. This page requires that you specify the lowest VME address you wish to access, then either the number of bytes which follow or the highest VME address required. Be advised that the VME interface hardware may or may not exactly be able to match your request, but DriveVME will mask the granularity of the VME interface. For efficiency in most cases, generally speaking, it is best to make a single request which specifies the largest range you know will be required for interacting with a particular VME resource using the same access parameters. This, over against opening several files that specify small subsets of the same resource.

There are VME hardware limitations, as well as security as sharing conflicts which arise when the range specified is too broad. In particular, do not specify the entire address space on the theory that you’ll rummage around and thereby “find” the resource you desire. Should any other VME program be running, such broad requests will be denied as they would conflict with sharing constraints arising from previously loaded and running VME related programs.

DriveVME has chosen 768 megabytes as the largest single range that can be mapped into a user memory array. If your need to map VME memory into the processor space is larger than that, you’ll have to open more than one file at the same time to see it. Programmers that are using AssistVME to aid their VME access specifications can cause the array to be broken into "banks" of specified size, then by issuing DeviceIO calls to the control "file" select which "bank" is available to the program- that is handled in another part of this manual.

Width per Transfer

Most take it for granted that the size of the data to be read or written will be the amount fetched. That is not the case. Some VME hardware will only respond to requests for data that deliver 8 bits at a time. Others just 16 bits, and not 8 or 32. Some VME chassis can not handle requests for data larger than 16 per request. This property page allows the user to limit the largest amount of data that can be passed per request. If you choose to limit the data to 8 bits per request, and then in your software do a 32 bit read, the hardware will generate four separate VME cycles to accomplish it. There is no way for the processor to detect the limitations of the device you wish to access. You must set this parameter with some care.

Generally it is best to select the highest width per transfer the VME resource you plan to access will permit.

Setting the character and security relative to READING from the VME resource

The "Read" property page offers the ability to set whether when initiating transactions which fetch data from other VME resources into the calling program, those transactions will be set as "Non-Privileged" or "Supervisory" VME cycles, or whether reading is disabled altogether and the resource to be made available is for writing only.

Within the typical A16, A32 and A24 address spaces, the VME bus allows transfer cycles to have either a "Supervisory" or "non Privileged" character. Hardware resources may be set to respond only to supervisory cycles, or non-privileged cycles, or both. Choose what is appropriate to your hardware.

Furthermore, DriveVME users must specify the extent to which the specified range in the specified address space may be also made available **FOR READING** to other programs on the system. You can limit the range to be available only to the thread of execution that opened it, or any thread of execution that is part of the opening thread's process, or open to any other in the system. Furthermore, you can specify that the particular range only be available to kernel mode device drivers (protecting it from all user mode programs), or only the particular kernel mode device that opened it. Also available is a diagnostic mode that bypasses security settings (except user / kernel differences). This mode allows diagnostic engineers a way to "peek" into the operations of programs under development. (Finished programs can choose to cause diagnostic access to be blocked).

Setting the character and security relative to WRITING to the VME resource

The “Write” property page offers the ability to set whether when initiating transactions which put data into other VME resources from the calling program, those transactions will be set as “Non-Privileged” or “Supervisory” VME cycles, or whether writing is disabled altogether and the resource to be made available is for reading only.

Within the typical A16, A32 and A24 address spaces, the VME bus allows transfer cycles to have either a “Supervisory” or “non Privileged” character. Hardware resources may be set to respond only to supervisory cycles, or non-privileged cycles, or both. Choose what is appropriate to your hardware.

Furthermore, DriveVME users must specify the extent to which the specified range in the specified address space may be also made available **FOR WRITING** to other programs on the system. You can limit the range to be available only to the thread of execution that opened it, or any thread of execution that is part of the opening thread’s process, or open to any other in the system. Furthermore, you can specify that the particular range only be available to kernel mode device drivers (protecting it from all user mode programs), or only the particular kernel mode device that opened it. Also available is a diagnostic mode that bypasses security settings (except user / kernel differences). This mode allows diagnostic engineers a way to “peek” into the operations of programs under development. (Finished programs can choose to cause diagnostic access to be blocked).

Notes and Read and Write Security:

The reading and writing security settings can be set separately, for example if your resource has “non destructive” reads (reading doesn’t change anything about how the resource does what it does) you might permit any thread to open the range for reading but only yours for writing.

In any event, if your AssistVME use (or calling program) attempts to access a VME resource that runs afoul of a previously open and granted request, the attempt will fail and a note as to the nature of the conflict will appear in the VME operations log window.

The VME Cycles page: Setting the timing and nature of the transactions

This page requires that you specify how the transactions, and in particular groups of transactions will be undertaken.

The VME bus separates all transactions into “program” and “data” styles. Your VME resource might be set to respond only to “program” transaction, or only “data” transactions, or possibly both. If in doubt, choose “data”. Consult your hardware documentation for details.

When writing to the VME bus, you may elect to cause your program to wait and to not return until the write cycle has been completed successfully: “coupled” cycles. Or, you may elect to have your program return immediately upon handing the data to be written off to the VME interface: “posted” writes. Posted writes are faster, coupled writes give you finer control over transaction timing.

Alignment

The cycles page gives you the ability to have the software (in file I/O mode) (or hardware if so equipped in mapped mode) “reassemble” the results of “unaligned” transfers. A transfer is said to be “aligned” if the starting address for the transaction divided by 1 for 8 bit transfers, 2 for 16 bit transfers, 3 for 32 bit transfers and so on is even. All 8 bit transfers are aligned. Some hardware resources will fail to operate correctly when transactions are not aligned. (For example reading 16 bits starting at offset 1 is not aligned). Other VME hardware may split unaligned requests for data into multiple aligned transfers.

This comes into play mostly when accessing parts of packed structures via stretches of memory maps provided by offboard resources. DriveVME’s file I/O methods offer internal realignment. The best way to really understand how this impacts your application is to experiment with it.

Chances are that if your application is the typical one which accesses particular hardware registers on VME resources, the registers themselves are set to be aligned by the hardware design engineers so this section doesn’t apply to you-choose the “hardware default” option as this is the fastest.

Multi-Byte transfers

When reading (more typically writing) consecutive addresses, the VME bus permits the address to be specified once, and following that successive data cycles. This, over against an address cycle followed by a data cycle – requiring more time. ONLY IF the VME resource to be accessed has the hardware to support and decode multibyte transfers, enable this setting.

Byte Swapping

The VME bus and Intel platforms historically chose to understand differently which byte of stored numbers that required more than one byte represented the least significant part. Intel platforms chose the lowest byte to be the least significant part, and the following bytes to be the representation of the higher values in the number. Motorola and its earlier and deeper ties to the VME world chose to have the representation of the higher parts of numbers first, then the lower.

So, when dealing with data that requires more than one byte to store, the question comes: do the bytes need to be swapped? Sometimes, your VME resource will handle this for you, most often it won't. If the data you are moving is not multibyte in character or your processor is both the source and user of the data then perhaps this issue is moot to you. Some processor vendors supply hardware that will swap the data for you so that you can access memory via memory mapped arrays and the reads and the writes will appear correctly on aligned transfers. DriveVME's File I/O methods always provide this service even if the hardware does not.

One drawback to using the hardware assisted methods in a memory mapped array is that most often the selection of hardware assisted byte-swapping requires that ALL concurrent use of the system enable that featrure also – or fail to run.

N4 suggests that users of the memory mapped feature do not request hardware assisted byte swapping and instead accomplish the needed transformations in their own logic – if they desire to have their software run generally in a platform independent way alongside other programs.

In any event, use this section to indicate if you want DriveVME to swap the bytes in 16, 32 and 64 bit transfers. Note that choosing 'yes' has interactions with software assisted unaligned transfer detangling mentioned before. Also, be advised that requiring byte swapping to occur in a memory mapped request will fail on those processing platforms that do not provide hardware assisted byte swapping features. File I/O requests for byte swapping will always succeed.

In all our experience about this tangled byte swapping issue, the best advice we can offer is that only experimentation yields the proper choice for any individual application.

VME Request Level

Most modern VME system controllers will grant requests at higher levels to devices seeking to initiate VME transactions than simultaneous requests at lower levels. However, some system controllers ignore this feature and only respond to requests at level 3. Choose according to your hardware.

VME Request Mode

In “Fair” mode, the processor will not attempt to gain control of the VME bus to initiate the cycles until the bus is free. In “demand” mode, it will assert its request immediately and let the bus priority logic in the system controller resolve conflicts.

- The Bus Settings Page

This is the last page that is really common to both the File I/O and the memory mapped access section. It’s purpose is to set how long the local processor can control the VME bus during multi-cycle transfer requests before letting others have a chance before resuming.

Bus Release and Execution Priority

The default setting will cause the local processor to release control of the VME bus when any other system asks for it, and, for File I/O requests will run the process that does the transfers at a priority that is above user mode tasks but below all other system functions. This is the default.

The next setting will not release the VME bus until the transfer is done, but nevertheless will run the task at a priority above user mode threads but below system functions and other hardware events. (The execution priority only applies to the File/IO requests once control has been passed to DriveVME via a file read or write call, memory mapped transfers happen at the priority of the user’s or kernel thread).

The next setting locks the VME bus to the calling program for the entire duration of the transfer, and conducts the transfer at a priority that can only be minimally interrupted long enough to handle other higher priority system hardware interrupts in the most minimal way possible (no post processing of the interrupt implications).

The next setting is like the previous but does “Address on Hold” cycles. If you don’t know what this rare and complex use of the VME bus means, it is nearly certain this facility will be of no importance to you.

The next setting locks the VME bus to the caller, and virtually dedicates the local processor to accomplishing the VME transaction (File I/O mode only). No interrupts are acknowledged (except NMI) and essentially everything is put on hold until the transfer is complete, then the bus is released.

The last setting is like the above but with ADOH cycles.

VME Cycles Per I/O

This has the most meaning to File I/O and particularly DMA operations, but can apply to block-copy posted writes from memory too. Here, the user can set how many cycles the VME controller will execute before turning control over to any other VME device requesting control. Whereafer, the local hardware will resume the transfer.

Direct Memory Access

This feature is only available in File I/O mode and only has meaning when the program is running on hardware platforms that permit the VME interface chip to generate memory cycles on the processor side and ferry them over to the VME side without software intervention each cycle. On systems so equipped, this is the fastest way to move blocks of data across the VME bus. It is also known to have the most problems from a hardware perspective, early Universe chip based system should be very certain of the hardware errata and limitations associated with this feature. The benefit to using this method is the speed of the transfer, the penalty is the amount of time it takes to set up the chip to do it. Short transfers almost never benefit. Note: Some VME interfaces feature DMA that does not alter the VME address during DMA transfers (the VME source/destination location does not change, but the processor side destination/source does). As of V3.12b DriveVME does support this optional feature.

Note: DriveVME does not make any attempt to mask hardware errors in DMA operations. If the hardware DMA logic is known to transfer other than the intended data to the intended destination at the intended time, DriveVME assumes that the user is aware of the issues and will program their device accordingly.

Block Debuggers Access

Earlier it was noted that a person could specify their use of AssistVME as being in the role of a debugger. This would allow a user to manually "peek" into the operations and address spaces and ranges of programs that specifically requested such operations be blocked. Should a programmer become confident enough about the operations of his logic so that even those specifying that they are debuggers be also not permitted to override their sharing selections, this option permits that choice to be expressed. Note: Kernel mode threads specifying they are debuggers are always granted access to any user mode thread.

For those generating VME cycles via File I/O operations Only: Read-Modify-Write Settings.

The VME bus provides in hardware a method for establishing multiprocessing sequencing called “read modify write” cycles. This amounts to a fast way for a processor to make a change in a memory value and to signal to other processors that it has done so and to know that during the attempt to so signal whether it was or was not successful over against other processors or processes on the same processor attempting the same thing at the same time. This is one of those occasions that if you need these feature and know what it means, the screen and option available to file I/O mode users will explain itself, and others need not concern themselves about it.

Options exist to do ADOH cycles on writes and RMW on reads, or RMWs in both cases. Also retention of the swap value is available.

Note: On hardware platforms where Read-Modify-Write cycles are not available in Hardware, DriveVME locks out all other processing on the platform, locks the VME bus to the local processor, accomplishes in software the hardware effect, then releases the VME bus, then resumes normal processing.

VME BUS ERROR HANDLING

Both File I/O users and memory mapped users may encounter VME bus errors. A bus error occurs when the calling computer generates a cycle and there is no response within an amount of time set on the configuration page – or the response is muddled by hardware conflicts or other problems.

The most frequent cause of VME bus errors is an attempt to access a VME resource using incorrect address space, data width or other settings.

The second most frequent cause of VME bus errors occurs on chassis that require manual jumper settings on the VME backplane that do not match the installed hardware.

DriveVME offers various methods of reporting and handling bus errors for their transactions, including ignoring them. Users can cause a message to appear in the VME operations log, send messages to their windows programs, abort the File I/O based transfers and report error conditions in the typical way that errors reading and writing files are reported, make programmatic calls to count how many errors have occurred globally on the system as of the time of the call or do none of the above.

Note that not all methods work on all VME interface implementations, and some indications of errors that work only some time after the failed cycle in question occur. DriveVME exploits in a maximum way all available error detection and reporting logic offered by the hardware vendor. Consult your hardware vendor for VME error detection logic supported by your hardware.

VME Transactions Via File IO Only: VME "Seek" Control

When generating VME transactions by reading and writing a file, it is possible to "seek" to any offset in the file so that the subsequent read or write will occur at the desired offset from the VME start address offered by the chosen and opened file. Some applications use files in such a way that they always issue seek requests of the same sort after every transaction. This section automates that procedure—saving the time required for the program to carry out "seek" operations. The options are:

1. To treat the "VME file" normally, causing an "EOF" end of file indication at the end of the range (the default).
2. To automatically act as if a "seek" to the beginning of the file was issued after every transaction. Useful when the range specifies a set of registers always read or written as a block from the beginning.
3. To make the file effectively endless, by "seeking" to the beginning automatically when the highest transfer in the range occurs. Useful when the range is read and written in the same pattern but a single transaction ought not cover the entire range.

VME DIRET DATA FORMATTING FOR WORD PROCESSORS, SPREADSHEETS AND DATABASE PROGRAMS (File I/O VME Transactions Only)

This page offers the user a way to cause the raw VME data to be automatically formatted to the specs needed for easy viewing or use by spreadsheets, databases and word processors. Once all the other information is correct, the user can set this feature to cause the data from the "file", the name of which is being built in the window below all the selections mentioned above, to provide VME information exactly in a format readable by high-level user programs.

This feature is very useful for data logging applications and others that want to use a Windows processor to "peer" into and collect information about ongoing VME operations. Or, any other application that calls for quick and ready access to VME operations without the need for systems level programming.

The first choice in this window is to choose the general data format desired. The default is one-to-one binary format (a byte read from the file is exactly the byte which comes from the VME resource at the given offset into the file plus the start address of the file name range (see above).

Other choices include a read-only "pretty print" format for debugging, showing the range, the values in hex and their ASCII printable equivalents; signed and unsigned decimal values, and hexidecimal.

The next section allows the user to specify how wide the data is. The user can choose to have one, two, four or eight bytes of VME data to be combined into a single value.

The user can choose to separate data with a space, a comma then a space, or to run it all together.
The user can choose whether to end the line after a set number of characters per line with an LF/CR combination or to let it go on with no line breaks.

Files so specified can be both read and written, except for the pretty print format that is read-only. Some experimentation to see which mode suits your application usually results in the shortest time to a working project.

TESTING YOUR VME TRANSACTION CHOICES

This page permits you to explore the accuracy and effects of all your prior choices by generating VME transactions.

Note that if your system does not have exactly one properly functioning system controller installed in or to the left as you look at the front of the chassis of your processor card, it is likely that the first test you do will cause your system to simply lock up until the power is cycled.

The left hand top box on this page permits you to specify a VME start and stop address, and if writing a single value to the entire range what that value is. To the right is a button which reads the range from VME, depositing the results in the edit window pane beneath the page. Below that is a button which attempts to read the values in the edit window pane and write them to VME. The fill button writes the value in the "fill" box to the entire range without changing the edit window pane's contents.

Below these buttons is a section which causes the system to act as if the read or write button, or both, were pushed repeatedly at an intervals you specify. This makes testing easier for those who have bus analyzers and in other situations.

To the right of this VME range oriented read/write section is a section which offers the ability to generate exactly one VME cycle at the address you specify, depositing the read result in to the little box in that window or writing to VME the editable result held there. This facility is usually the first stop when testing a VME range selection.

DISPLAY-ONLY SOFTWARE BYTE SWAPPING VME ACCESS VIA MEMORY MAP USERS ONLY: The "software byte swapping" button will cause the information in the window displayed to be "byte swapped" just prior to display and “byte swapped” back just prior to writing to VME, for diagnostic purposes. This only has meaning when doing 16, 32 or 64 bit transfers.

“NOTEPAD” ICON VME ACCESS VIA FILE I/O USERS ONLY: The “Make Icon” button will create a Windows “shortcut” file which when chosen will cause the windows “notepad” application to display the content of the pseudo “file” which really is the VME range chosen according to the selected formatting and display. (Be sure not to have selected the “endless seek” option or the application load will fail as the “file” has near infinite size).

Benchmarks and Testing

Assist VME has built in routines to time the reading and writing of data according to the selections chosen and to display the results. A great many choices go into how the system performs, many of which can be specified in DriveVME, but many of which can only be specified in the BIOS relative to how long the VME interface chip can use the PCI bus at a go. This section permits the user to specify whether reading, writing or both should be tested, and the various data widths to use. The results are displayed in the editable window pane beneath.

Note: in order for these tests to be meaningful, there must be hardware supporting registers at the VME addresses specified in the range that can be read and written without harmful effect, and if both read and write tests are enabled, the data read must faithfully reflect the data written.

ASSIST-VME SOFTWARE CREATION ASSISTANCE PAGE

Once you have made and tested your choices for initiating VME transactions, the options on this page are designed to make it easy for you to see how these choices can be incorporated into programs by actually writing a sample working program that uses your choices for you. These programs function without using AssistVME, indeed AssistVME doesn’t even need to be included on systems for these programs to work (except for error log reporting).

You can choose the button marked to have AssistVME write a program for you that stands entirely alone, making use of nothing except the DriveVME device driver to effect your choices.

Also, you can choose to store your choices under a “key” whose name you provide, in the registry. Different implementations of the same VME program on different chassis might require VME resources to be located and addressed differently. Programmers can specify their resources only as named “keys” so that field installation staff might be able to reconfigure chassis and update appropriate key values without forcing changes to compiled programs (possibly from previously stored AssistVME .vme or .vmm files created using the above information, saved then emailed for the purpose). Lastly, programmers can cause AssistVME to create a program that demonstrates how the named key might be used.

VME resource backed, network shareable "RAM" Disk Drives

The controls on the right of the software page permits the user to cause the specified VME range to be treated as memory-- causing a standard, network sharable Windows disk drive to be created that uses that range, accessed in the method specified earlier, as the off card memory store that supports it. This uses the VMEDISK.sys device driver once sold separately but now supplied standard with DriveVME. Many such "drives" can be created and be functioning at the same time.

To review and manage which local RAM disks are associated with what memory resources, choose the blueish toolbar icon to the right with an image of a disk and the letters VME on it.

-- This ends the section describing how to use AssistVME to generate all allowed VME transactions other than interrupts. The sections which follow describe how to offer local processor board memory and other local processor resources as targets for cycles generated by hardware on the VME bus other than the local processor; how to generate, handle and monitor interrupts, and how to develop 'C' programs for VME operations that run in the kernel of the Windows OS. --

Offering Processor Card Resources to Other VME Bus Master-Capable Cards With AssistVME

You can offer the resources of the VME chassis processor to other VME resources by choosing the toolbar icon that has an arrow pointing to "VME" or by the obvious choice in the "File" menu.

ADVISORY NOTE: Certain hardware platforms that include the Universe PCI-VME interface chip fail to concurrently handle VME – to – PCI and PCI – to – VME transactions in very special occasions. The problem arises when a PCI bus owner requests a coupled VME transaction while a different card owns the VME bus and is requesting resources from the PCI side. Typical conflict resolution logic fails on these systems and the usual result is that the VME transaction ends in a bus error and the PCI side ends in an abort. This problem can be avoided by:

1. Not offering local processor resources to VME bus masters.
2. Not initiating VME cycles while offering local resources.
3. Only during the times the local processor is offering VME resources, locking the VME bus to the local processor before initiating VME cycles, then releasing it.

By the time you read this, other solutions and hardware upgrades may have occurred. Contact your processor vendor for details. DriveVME's file I/O initiated VME cycles are sensitive to this problem and only on susceptible platforms lock the VME bus to the processor during transactions when local resources are exposed to off-card VME bus masters.

Resource sharing specifications generated in this section can be saved in files with the extension ".vsl". These files can be sent to other systems and used for diagnostic procedures. See the "Open" and "Save" and "Save as" entries in the "File" menu.

The usage of this window is a matter of working through the property tabs and making specifications from left to right. The process ends in a resource being available to other processors on the chassis in question. During the process of specifying a resource to share, you will note a long file name being built in the window below. This is used by programmers to cause the resource to be available under programmatic control, mapping the resource into the user's memory space as well as being able to read and write to it by the usual file handle methods (the consequence being that network remote offcard computers can cause DriveVME enabled processors to offer VME resources, fully supporting networked applications).

The property pages to consider when offering VME resources are:

Sharing Address Space Selection

The first page is the "Space" property. The VME bus offers many different memory spaces. Offering a resource at address "0" in one space will not result in the same data as address "0" in another. The most typically used VME "spaces" are "A16" – offering 16 bits of address information, "A24" with 24 and "A32" with 32. These were generally historically evolved, when decoding more address lines in hardware called for additional expense, and wiring VME busses with further connectors was required to offer more addressing. The address space you offer is specified by the settings on the VME resource you with wish to interact.

Technically, the "space" is signaled on the VME bus by a "user AM" code. Exotic hardware can operate in its own "space" by specifying an "AM" code which is otherwise unused. If your hardware requires such, it can be entered manually on the "space" settings page.

VME Address – where your resource is to be offered in the selected space

Whatever resource you choose to use, you must locate it somewhere in the VME address space you have chosen. This page allows you to specify whether you are

1. Indifferent to the VME address chosen (allowing the computer to choose it will assure it does not conflict with any other resource in use on the processor, the system offers a way to detect what VME address it selected).
2. Wish the VME address to start at any multiple of the local processor's memory plus an offset you choose (compatible with older designs).
3. Wish the VME address to start exactly at the VME offset you specify (least compatible, generally safe with modern designs).

Note that the system may indeed be forced to offer more or consume more VME address space than you specify here if the hardware's ability to offer resources is not able to align itself evenly with your choice. For example, if you wish to offer VME resources at VME offset 7, it is almost a certainty that the system will assign resources starting at VME offset 0 – while reporting to your applications the proper local address which cooresponds to VME offset 7. The actual addresses assigned and used are also available.

Local (usually PCI) address specification page: naming the resource to be offered.

This page allows the user to specify what local resource is being offered. Note that not all hardware platforms can offer this wide a variety of resources. Nearly all can offer local processor memory, beyond that you must contact your processor vendor. Available resources are:

1. Addresses in the local processor's I/O or input – output space (that is, not memory. This allows VME resources to directly control local ISA and PCI devices such as serial ports, etc. Use with great caution.)
2. Actual, specific PCI memory addresses, above the range used by Windows. This feature makes it possible to offer the memory windows of memory mapped PCI devices. This is usually not used for sharing local processor RAM.
3. Windows offers the ability to require it to operate in less than the total available local memory. The physical local memory above that limit can be set aside for DriveVME's sharing with VME cards. The advantage is that Windows does not maintain large stretches of consecutive non-paged memory suitable for sharing with VME within its own managed space (it does maintain a small pool of such, however). This option allows the user to specify particular addresses in the local memory range above where Windows is permitted to operate. As there is usually no advantage to specifying these addresses, it is best to not choose this option without good cause.
4. This option is the preferred option. DriveVME manages the physical memory beyond the assignment of Windows, and will assign a free address block of suitable alignment and size to meet the need of the request, reporting the base address to the user at file open time. If there is no memory set aside for DriveVME above Windows, then the system will attempt to use memory from small NT "nonpaged system memory pool". Sharing requests in the low hundreds of kilobytes or less are usually successful, beyond that you should limit Windows and assign free memory above it to DriveVME (see below for how).
5. This option exposes the entire local VME interface control registers to the VME bus. Essentially it offers VME users the ability to completely assume control of local operations. Use with caution.
6. Some VME interfaces offer a "Location monitor" – VME addresses which when accessed by offcard VME bus masters generate a local interrupt – only on the local system and not bus wide. This offers those addresses to offcard VME masters. (See the interrupt handling section for more detail on using this facility).

To the right of the specification control is the section for specifying how many bytes are to be offered to VME, and if applicable to the type of resource being offered, a place to specify the start address.

Reserving Large Amounts (>1MB) of local RAM for VME Sharing.

Windows can be instructed to operate in less than the total available RAM, the "top end" remainder can be offered to VME resources. The section on the "Local Address" page lower to the right allows you to specify how much memory the system has altogether, and how much ought to be reserved for DriveVME. This makes changes to the windows "Boot.INI" file on the Windows startup drive, the settings don't take effect until after the system has been restarted.

The Offered Resource Security Page

The VME bus offers a distinction between "Non-privileged" and "Supervisory" requests. You can choose to have the region previously offered respond to either one or the other or both of these types of requests. If in doubt, choose both.

Furthermore, the resource you offer to share to VME bus requestors your system also can access by both the convention of a locally mapped memory array, or by reading and writing the file handle associated with making the request to share the resource. This page offers the ability to limit what other processes and threads might also be allowed to share the same VME range and space and / or the same specific local addressing. The resource can be set to be shared by none but the opening thread, any thread in the opening process, any process in the system, or only kernel-mode device drivers.

Two special options exist, one is claiming that the opening requester is a "debugger" and therefore asking DriveVME to overlook usual conflict security issues and permit "peeking" into presumably the ongoing operation of another program. The other reserves what ought to be an entire range of addresses in one block, so that the physical resources that manage the VME sharing will be set to offer the entire scope, in anticipation of smaller requests for subsets of the same scope to follow.

DriveVME does its best to coalesce like requests for sharing resources (and initiating cycles too, for that matter) in ways that most aggressively conserve the VME interface's ability to satisfy a broad range of mixed requests. This last mentioned facility amounts to little more than a "hint" by the user that many small requests for sharing a larger range of like kind will be forthcoming. Making requests of this kind, when it is known that smaller requests inside the same scope will follow, make it more likely that the physical limited hardware that makes the resource sharing possible might more likely be able to meet the need of the application.

Lastly, offering local resources has many stability and security implications. System administrators may wish to limit the ability to offer local resources to kernel mode device drivers, or to disable any such doing whatsoever. It is possible to signal this desire by making the choices available on this page. So long as the file handle containing those choices is open, no other will succeed.

Choosing To Which VME Cycle Types to Respond

The "Cycles" page permits the specification of whether the previously identified resource should respond to VME "program" or "data" style cycles, or both. The VME bus maintains a distinction within the typical address spaces (A16, A32, A24) of requests with a "program" attribute and others with a "data" attribute.

This page also offers the ability to instruct the system to signal to the VME bus that writes to the local resource have completed when in fact they are in a queue waiting to actually occur on the local side, or whether the VME resource be made to wait until the local side has actually completed the physical write.

Some systems offer hardware byte swapping "on the way out" on offered resources, if your system affords this option and your system requires you may enable it on this screen. Read the treatment in the Initiating VME cycles section far above for more detail on the causes and implications of byte swapping. Note that this has no effect on how the local resource behaves locally, just how the VME requestors see the local data.

The local system can respond to VME requests by anticipation, prefetching them from local memory and making them available more quickly to anticipated VME read requests. This makes for the possibility that the local processor might write a value to a register before a VME device asks to read it but after it was prefetched into the VME interface queue. On the other hand, the VME reading operations go faster when prefetching is enabled. (Note: If the resource being shared is anything other than RAM memory—DO NOT enable prefetching).

When an offboard resource seeks to execute a VME "Read-Modify-Write" cycle, it does so because it wants to be assured that no other device whatsoever can slip and to access the location being updated during the process. If the local processor is not part of the critical timing issues being arbitrated, it is a good idea to not force the local PCI or system memory bus to also be locked during the operation. If, however the local processor is a party to the arbitration consideration, then you should choose to lock the local PCI bus during VME Read-Modify-Write cycles so that the local processor does not execute cycles at the same time as VME resources resulting in both concluding a resource is either unavailable to both or available to both at once.

Last on this page, when your program is well and truly debugged, and if you are worried about other processes or AssistVME users wandering around your offered area posing as debuggers, you can specify that debug attempts to co-offer the same region ought to be allowed only if any other request would also be allowed.

The "Bus" page- How to handle local accesses to the shared resource via I/O to the open specification file handle.

As a matter of procedure, AssistVME actually offers the resource in the same fashion as any other program using DriveVME would offer one, by opening a file with a special name on the "V" drive. When that occurs, reading and writing to the file update the contents of the shared resource, seeking in the file positions the offset into the shared resource of the subsequent read or write.

At the same time, once a file handle specifying a sharing is open, the resource is mapped as an array into the local memory space of the calling thread (or kernel memory space if the caller is another device driver).

While only limited restrictions discussed above can be placed on how offcard VME resources access the resource, and none can be placed on how the local resource is affected by access to the local array, DriveVME offers the ability to impose some consistency synchronization between local and offcard VME requests.

Sometimes it is necessary to know that during the period of a read or a write to a local resource, no VME device will be also able to read or write that resource. (That is why under the file I/O method for initiating VME cycles to offcard resources it is possible to lock the VME bus to the local processor for the duration of a transfer.) This page lets the user specify at what priority reads and writes to the local device occur, and whether the VME bus ought to be locked entirely during that time so that no offcard updates can occur during that period.

Note that locking a bus is a serious business, directly affecting at least VME performance. No transactions can occur on a locked bus – none at all, between any two cards. Generally, if it is known that only one process on the local computer concerns itself with a particular resource, it is better to update the resource as much as possible using the direct array, and then just updating critical synchronization sections using the File/IO method with the bus locks.

Handling Local Bus Errors Caused By Offcard VME requests

The "errors" page allows the user to specify what to do in the event an offcard VME request generates a local system bus error. This almost never arises when the shared resource is local memory. However, when sharing physical devices which may or may not respond to the VME chip's request for local traffic, or local requests to access the shared resource, local bus errors may arise. The handling options presented on this page mirror the options discussed in the "errors" section relative to generating VME cycles for other resources to handle as discussed above.

The Testing Page – Actually Offering the Local Resource

This page functions identically with the testing page as described in the section relative to generating VME transactions in the previous section, with one difference. The specified resource is shared only when the "file" whose name all the other choices have been specifying, is open. This page presents the ability via a button to hold the file open, and thereby actually share the resource, for any desired duration. During the time the special file is held open, the user may schedule periodic reading and writing, do occasional probing of contents, and read /write and accuracy benchmarks.

Automatically Generating Software that Shares a Resource

The last property tab in this section is exactly like the matching tab discussed under the section relative to initiating access to offcard VME resources. Buttons are available to cause AssistVME to generate example software in various modes that exercises the particular choices set forth previously as an example to further programming by the users.

Creating a "RAM" Local Disk Drive that is also shared with VME users.

Like the 'software' page under initiating VME cycles, it is possible to cause the locally provided memory resource to be used by the operating system as a complete disk drive, and so sharable over a network like any other, while at the same time offering access to the memory shared to offcard VME resources. To review and manage which local RAM disks are associated with what memory resources, choose the blueish toolbar icon to the right with an image of a disk and the letters VME on it.

Except for using local resources instead of offcard VME resources, the operation is the same as described in the cycle initiating section described above.

This ends the section discussing how to offer local resources, other than interrupts, to offcard VME resources. The next sections take up generating and responding to interrupts, and following that a 'C' language development environment for generating VME related programs that run in the kernel mode.

Responding to VME Interrupts through simple File Operations.

DriveVME makes it possible for user programs to avoid the overhead of running an entire dynamic linked library to handle as many as a few dozen VME interrupts per second. The method of handling interrupts involves the typical operations that are well know relative to opening, closing, reading and writing to files. As such, it is possible for programs that are running over a network to respond to interrupts and other events on network-remote processors—even network remote systems that are not even running Windows!

This method is also appropriate to users who simply want to wait for an interrupt to happen, then read a chunk of information into a spreadsheet or database program. No high performance software overhead, rapid development is a plus.

AssistVME provides a toolbar icon that has the letters "ISR", or an equivalent "file" menu entry titled "User mode interrupt and event handling". These bring up a document window that allows choices to be expressed, tested and saved with files bearing the extension ".vmh"

The first order of business is to use the pull-down box to select which VME event is of interest. Beyond the normal 7 VME interrupt levels, there are system status change events, "mailbox" events and others. The event of interest will be determined by knowledge of the hardware the program is looking to that might generate the event.

VME interrupts on levels 1 through 7 are not "bare", the device that generates them provides a value between 0 and 255 to the system that responds to the interrupt. You can choose some or all of this range to trigger your particular response. DriveVME may have been instructed to cause many events to occur when an interrupt with a particular value on a particular level happens. So long as at least one event is set to handle an interrupt on a given level, the local processor will signal its "handling" of all the values that are associated with that interrupt.

Note that this basic method of interrupt handling (DriveVME offers many others) presumes that the interrupt is of the variety that ceases signaling the interrupt when the processor responds to it, and does not continue to signal the interrupt until after such time as offcard registers are manipulated.

In any case, you can choose to arrange matters so that the request to "open" the file containing all your specifications does not complete until after the desired event occurs. Or, you can set the system to open the file right away, then delay responding to requests to read the file until either too much time passes, or the desired event occurs. Windows GUI programmers can cause DriveVME to issue a "windows message" to their process' event queue with a particular value whenever the event of interest occurs, so long as the requesting pseudo “file” is held open.

The page also provides buttons which test waiting for the desired event, and buttons to generate programs which demonstrate using the facility with the chosen options in a program (with the “dll” options just as in the previous sections to do with generating VME cycles and offering local resources to respond to other VME bus masters.)

GENERATING VME INTERRUPTS VIA ASSISTVME

The toolbar icon with the arrow above the word “int”, and the file menu option “Generate interrupts” brings up a dialog box. Unlike other options, this box does not create templates for programs nor puzzle through many VME options. It permits the user to directly cause the system to generate a VME interrupts.

The dialog box has obvious controls which permit the user to generate VME interrupts on levels 1 through 7, and to allow the user to specify a value between 0 and 255 to offer to the system which responds to the interrupt. Note that some systems ignore the lowest significant bit of the value you intend to offer, instead setting it to a 1 or 0 to signify whether it was the local system that generated the interrupt or not.

The dialog box offers the ability to manually fire off an interrupt, or to cause interrupts to be repeated at intervals you specify.

Lastly, to the right of the controls for each interrupt level is a tally of how many interrupts were generated and handled properly by the VME chassis (whether interrupt service routines on the local card or another card in the chassis). A little box describes the state of whether the current interrupt has been issued, is waiting to be issued or is expecting a reply.

Note: Some VME platforms can handle the generation of more than one interrupt on more than one VME level at the same time. Other platforms can only manage to issue interrupts serially, requiring that one be acknowledged before offering the next one. DriveVME does its best to mask these hardware differences, nevertheless users with time-critical applications ought to be aware that depending on the hardware platform, there may be a delay between the time the software signals an interrupt and the hardware actually issues it.

Note also that there is no basic file oriented facility to issue VME interrupts. Issuing interrupts can be accomplished through use of the dynamic link library for user applications, or through the “kernel ‘c’” setups discussed later.

Lastly, later sections of this manual will explain how to connect AssistVME running on one computer to other computers running “DriveVME” on a network. All facilities of AssistVME, other than relating to VME resources via local process memory arrays, are available remotely as well as locally (Presuming, of course, that the security restrictions placed on sharing the DriveVME “V” disk are met).

Monitoring VME Interrupts and other Asynchronous Events

Choosing the toolbar button labeled “Int” with an eye on top, or the “file” menu entry “Monitor Asychronous….” Will bring up a large dialogue box that displays all the events DriveVME is capable of watching for, along with counts of how often they are occuring and the total that have been detected.

You can trigger whether DriveVME ought to respond to VME interrupts and other events by depressing the button along side their labels and offering ranges of values of interest.

Using this box on one processor located in the leftmost slot of a chassis to monitor desired events, and the generation box discussed earlier in the rightmost slot to cause events, then swapping roles, is a good way to check out the validity of a chassis setup.

This is the last feature available to those casually desiring to browse around the local VME bus. The next section describes how to point AssistVME at other processors via networks. The rest of the manual is written for programmers in the order of increasing needs and skills.

ASSISTVME OPERATIONS VIA NETWORKS ON REMOTE CHASSIS' VME PROCESSORS

All of the foregoing section relative to AssistVME (and also the upcoming section relative to sending 'C' text programs into DriveVME's kernel for execution) can be accomplished by users remote to the actual via processor but nevertheless connected via a network.

To permit users remote to a local processor access, the "V" drive on the local system must be shared, and set with such security as is warranted using the typical and customary Windows tools to accomplish that result. Furthermore, the local chassis must be connected to an appropriate network, complete with all the setup and other configuration chores that entails. How to accomplish that is beyond the scope of this manual, there are many texts on Windows networking that describe the operation in detail.

The "V" pseudo-disk drive is an artifact of DriveVME, represents the system-wide access point to DriveVME operations.

AssistVME can run as a stand alone application on any computer, VME capable or laptop or otherwise, however it requires the purchase of a development kit to be licensed by N4 Communications.

For the purposes of this manual, we will assume that somewhere on a network DriveVME is installed and running, and the reader is in front of a different computer running AssistVME with network access to the DriveVME computer, and all security and sharing requirements have been met.

Check: If you can't see the "V" drive on the DriveVME computer using the "Network Neighborhood" icon and drilling down to the appropriate workgroup and so forth, AssistVME won't be able to work with it either. Be sure you can see the "V" drive of the DriveVME computer in question before seeking support from your VME supplier.

Use the Assist VME toolbar icon leftmost that looks like two monitors on a cable, or the "Select Chassis" menu item beginning "Select Chassis", to call up the Cassis Selection Dialogue Box.

A list of all available chassis visible to the system running AssistVME will be displayed. The Net Scan area permits the user to conduct a scan of the local workgroup ("Fast") , a scan of all the top level systems in the local domain ("Shallow"), a scan of all the systems on the same network as the system running AssistVME ("Deep"), and a scan that amounts to an exhaustive search through every single system possibly visible to the current system ("Full").

Alternately, if you know the path to the system you wish to access, use the "Enter Site" button to enter it manually.

To erase a system from the list, highlight it and use the "erase site" button.

To cancel changing which system the local copy of AssistVME is pointed to, use the cancel.

Otherwise, highlight the system you desire to control with AssistVME in the list then hit "OK".

From that moment onward, any new windows opened will cause their operative effects to occur on the chassis most recently selected (it will appear in the title bar of the related windows).

Note, all DriveVME features, from interrupts, to generating VME cycles and offering resources are available to remote users and remote programs. The only facility not available to remote users is generating VME transactions via accessing an array in the local machine's memory space, and reviewing the content of resources shared by the remote processor with the remote chassis through an array in the local machine's memory space. However, the information itself is available through reading and writing the open file handle associated with the resource. Full details on programming with VME files can be gleaned from reviewing the software examples AssistVME creates for you, as well as further sections of this manual.

DriveVME’s Kernel ‘C’ System and the AssistVME Development Environment

Using a special “file” on the DriveVME’s “V” drive, user’s can write a standard ‘C’ program for DriveVME to compile then run at the highest level of system performance and control, the operating system kernel.

This facility is very powerful, but also has complexities. If you do not know how to program in the ‘C’ language, or arrived at this section of the manual without first reading the section near the very beginning of the manual titled “Matching DriveVME’s Abilities With Your Requirements”, reading this section may generate more frustration than progress. This manual and facility assume that you have had some basic experience in other ‘C’ development environments, even the basic ‘C’ environments popular in the ‘1980s will be enough.

Once a kernel ‘C’ program has been written to DriveVME’s special file, further traffic on the file handle which delivered the text of the program to DriveVME delivers the ‘console’ stdin and stdout output of the running kernel DriveVME program.

DriveVME offers many specialized VME specific functions and libraries available to its kernel ‘c’ environment, and through it to user mode programs.

This facility has two equally important purposes. The first is to make it possible for off-chassis networked computers to fully access VME facilities remotely, without sacrificing the high speed operations that must be done locally, yet without requiring the remote computers to be Windows machines. So long as they can connect over the network to a Windows computer, they can cause highly sophisticated VME operations to occur without a stitch of special purpose Windows coding.

The second purpose is to allow access to kernel level VME system performance to user level programs, and to programmers who don't want to go the expense and time to climb the learning curve of Windows device driver authorship just to accomplish one or two special purpose VME effects.

The kernel 'C' facility is very powerful, allowing register level access to the VME interface and other system physical devices, while providing protection against memory reference errors and basic mistakes. In short, it is a very powerful tool one can use like any powerful tool, to good effect or by inattention to detail – catastrophe. If you are discouraged by the foregoing, perhaps you might be more comfortable using the DriveVME dynamic link library or ".dll" described later in the manual, or by using the previously described facilities in AssistVME to work with the VME bus via high level programs like visual basic or spreadsheets, or even AssistVME's ability to create sample programs that are full of safety checks based on VME access specifications you choose using AssistVME.

The AssistVME Kernel 'C' Development Environment

This writing assumes that the reader is seated next to a computer running AssistVME, and has previous working knowledge of the basic, unadorned 'C' language as specified by Kernighan and Ritchie.

To launch AssistVME's Kernel 'C' Development Environment, choose the "Interrupt, Realtime Services Processing" menu item from the "file" menu or the toolbar icon with the colored arrows arranged in a triangle.

The window presented is broken into three major parts. The top third, which holds the program text; the middle third which is a "tree and branch" setup presenting the values of variables "watched" during execution, and the lower third which can be thought of as the "console" to which the program writes and from which the program seeks console input.

There are two general kinds of DriveVME 'C' programs: those created for the purpose of providing information to people, and so run in the context of the AssistVME evironment; and those designed to be the "remote" or "high performance" or "VME specific" portions of larger programs communicated with through reads and writes to the special DriveVME file for lower priority or other non-VME related processing.

All the files in the windows\system32\drivers directory that end in ".H" (except those dedicated to the user mode .DLL and the one dedicated to kernel-mode device driver authors—see the "environment" section early on in this manual for a list) are DriveVME library files that offer specific services and are self documenting. There methods are described to access VME interface registers, generate and handle VME interrupts at high speeds, access VME resources, local resources and much more. There are not too many of these files, and the names give a good indication of their purpose and offerings. The best method is to explore them.

Most of these programs are accompanied by files ending in ".c" which serve to demonstrate the proper use and functioning of the named routines.

Use AssistVME's file "open" command to open one of these ".c" files, or create a simple "C" program yourself in the program edit window.

When you are ready to try it, click the desired command in the "run" menu. You can single step through the program using the toolbar checkmarks or the "run" menu, also breakpoints can be set and cleared with the same facilities.

Double clicking in the watch window during a break in the operations of a running program will permit the entry of variable names, which content will be displayed as they change during program execution. To delete a "watch", single click the entry then hit the delete key.

To save a program, use the "save" item in the "file" menu, you can reload it later using the "open" item of the "file" menu.

If you are developing a kernel 'C' program whose purpose is to communicate via AssistVME, then there is nothing more to be said. 'C' printf statements will go to the AssistVME console window, scanf from 'stdin' comes from the same source.

If your program has the purpose of communicating with and being a part of other software, take special notice of the "->C" toolbar item and the related "Create a C program using this program" item in the "Run" menu.

This facility will create a stand alone user mode Windows 'C' program, ready to compile with the Microsoft 'C' visual studio compiler, which incorporates the DriveVME Kernel 'C' program you've developed—it demonstrates everything necessary to have a user mode standard windows program pass a DriveVME kernel 'C' program to DriveVME, fire it up, relate to it while it is running, then close it down.

This completes the description of the services offered by DriveVME's AssistVME.

DriveVME.DLL – The DriveVME User Mode Library

DriveVME offers its services to end users and programmers with many needs and at various levels of skill. This chapter describes how to use DriveVME's user mode library. The user mode library represents a middle point on the trade-off curve between execution speed and development complexity.

If your application requires only one or two basic interactions with the VME bus, you would be better off using the direct file handling approached demonstrated in the "create program" buttons of the "software" property tabs in the appropriate AssistVME sections.

If your program needs to process more than a few dozen interrupts per second (or even into the hundreds on faster computers) you ought to consider using the Kernel "C" methods described in the previous chapter-- or for the best possible performance, creating your own device driver to link with DriveVME discussed in the following chapter. Consult the "meeting your needs" section near the beginning of this manual for further guidance.

If you are comfortable using DLL's in general and to interact with the VME bus in particular (as most company specific VME software vendors offer their services via DLL libraries), and your application calls for a mix of VME access needs, then the DriveVME library is for you.

The DriveVME DLL has two major components, the library itself, DriveVME.dll, and the associated ".exp" and ".lib" files some compilers need to be able to link with it, and the related include file, DVMEDLL.h.

The header file, DVMEDLL.h, provided for free with the evaluation version of DrieVME available at www.n4comm.com and unzipped into the C:\DriveVME directory, is extensively documented, not only providing the compiler with the definition of the function headers but also programming examples and documentation for using each one. Also, the development kit includes a DOS mode and a Windows GUI mode program and source code showing examples of using all of the DLL features and even qualifying DriveVME on your chassis. (See the catalog in the environment section at the beginning of this manual for details).

The list of functions and facilities in the DLL is extensive, and most easily understood in the context of a development environment. **In short, the DriveVME DLL offers a function call oriented method to accomplish nearly every feature DriveVME affords the user, supporting control of remote VME chassis via networks with the same library calls used as in controlling the local VME processor.**

The DriveVME DLL can be used by any program capable of relating to dynamic link libraries, including Visual Basic, the Microsoft Office suite of products, and nearly all programming languages and database systems qualified to run in a Windows environment.

Writing Windows Device Drivers that use VME resources

For the highest possible level of interrupt handling performance, handling interrupts in the many hundreds or thousands per second, or to integrate VME hardware as native windows devices, it is necessary to create a Windows device driver.

DriveVME offers a .zip file in its evaluation directory that explains and demonstrates this task. Look for DDKDemo.zip, and within it heavily comme nted dvmeddk.c, a sample kernel mode device driver that relates to DriveVME, and also DVMEDDK.h, the include file for the device driver sample.

New in version 4: Support for 16 and 32 bit IACK cycles for VME interrupts 1..7 on most TSI148 based PCIx – VME platforms. See DVMEddk.h for details.

It is far beyond the scope of this manual to provide instructions about the business of writing Windows NT device drivers generally, consult Microsoft.com looking for "DDK" to get full details. It is a significant task, requiring a substantial investment in Microsoft specific methods and practices. Unless your application demands this approach, fully explore alternatives before trying this method.

Generally speaking, anything that can be accomplished by user mode facilities can be accomplished by NT device drivers. Device drivers have the advantage of being much closer to the source of interrupt processing, and don't impose the processing penalty of switching operating contexts from system to user mode and back again each interrupt or VME access call. Furthermore, device drivers can control how much of the system's processing ability they use to a much finer degree than user mode programs.

Be advised that the general device driver development cycle calls for "ironing out" the particulars of the VME accesses required using AssistVME, then capturing the parameters in embodied in the DriveVME "filenames" from AssistVME for use by the kernel mode driver. The only feature unique to the kernel mode driver is the nature of interrupt callbacks, a facility demonstrated in the examples provided.

Directly Initiating VME Transactions via Memory Map or File I/O

This section documents the particular details of the syntax of the DriveVME "file" strings associated with generating VME cycles. The basic information surrounding the meaning embedded in the creation of these filename strings is described in the very early chapter regarding AssistVME. AssistVME also has a facility that creates a fully functional program showing how these specially named files result in generating VME cycles.

This section is for those who seek full technical detail about this subject, and who have already mastered the basics presented in the "AssistVME" chapter.

Most Drive/VME, AssistVME users do not need to understand this material, as AssistVME automates the creation of these specifications. However, if detailed systems operation and understanding is required, this document does supply added detail particularly in the area of alignments and swapping.

This Drive/VME facility offers rich access to VME bus resources by ordinary file open, close, read and write requests. This same interface also can "map" sections of VME memory into a user-mode process, and return the "base address" of that array. This file oriented approach is the simplest method Drive/VME offers to programs and kernel mode drivers to provide nearly all types of VME bus-master transfer needs, such as for security-transparent debugging, bus locking and even uninterruptible single or burst transfers, and it fully supports normal, asynchronous and DMA reading and writing operations. The only sort of master mode VME access not suited to this mode of operation are those requiring VME read-modify-write cycles when there is possibly contention among processes or other VME resources for a particular VME resource. For that, and high speed interrupt processing and related needs, other Drive/VME facilities are available.

To make it easier to review and manipulate VME resources with commercial, off-the-shelf software and other high-end user application programs, even popular editors, databases, and spreadsheets, this Drive/VME access method also provides ASCII text formatting and translations of VME binary data. For example, you can choose to have an 8 bit binary register containing the value 0x8f when read, not to deliver not the default single byte of 0x8f, but instead the two ASCII characters "8" then "f", or the three character value in decimal "143". Also, there are ways to read and write ranges of VME values in the popular "comma separated variable" interchange format, or combination binary/ASCII debug formats.

VME data can be accessed just by opening the device "DriveVME" as a file, optionally with VME access options appended after "DriveVME", then doing ordinary file read and write operations. Drive/VME provides higher priority, and even uninterruptible ways to access VME resources (through this facility and also interpreted user "mini-I/O" programs run by Drive/VME in kernel mode- programs which work in concert with memory shared by the calling user mode process). But, this file oriented method is the easiest approach to use, and is widely adaptable to most needs, is the most portable across CPU boards, and to be preferred whenever possible.

For example, opening the file "V:\M\sSa(100#4)rSf(H4)t(4)" (or, opening only "DriveVME:" or "\\.\DriveVME", then immediately writing the entire previous filespec string) would result in a file which when read would return the results of a single supervisory cycle 4 byte (D32) read at offset 100 hex of the VME standard address space in the form of 8 ASCII hex characters, which is appropriate for a 32 bit value.

In another example involving any plain text editor, such as the Windows "notepad", passing: "V:\\M\a(100#40)f(P)t(1)wZp(D)" as a "filename" to be edited would display a pretty printed 64 VME bytes starting at VME short-io (a16) address 0x100 using non-privileged cycles transferred 8 bits at a time on four display rows of 78 characters each, then report an End-Of-File. The p(D) parameter enables the quick-and-simple debug aid, and as such this request does not interfere with address range exclusivity requests of other Drive/VME using software under test also using that range. (Therefore, the wZ parameter blocks writes to the named range.)

All in all, this is a very quick way to access, and even "pretty print" the status of a device with no programming required.

Last, the this interface of Drive/VME is fully networkable. By designating the "V" drive as a shared network" drive, remote users, even dial-in users, can be granted access to local VME resources.

Detailed DriveVME Filename Specification

Note: The "AssistVME" program automates the process of making these choices and composing these strings, as well as testing the chosen VME resources and help autogenerating proper source code and other abilities.

Using any standard and ordinary means to open a device file named:

V:\M\<VME space><read cycle types><write cycle types><address range><special cycles><max transfer width, byte-swapping, alignment><bus access><error handling><permissions><priority><formatting options>

will, if successful, make ordinary or asynchronous reads or writes to the file generate VME master mode access transfers of the specified sort. When the file is closed, all claims to the resources it names on the VME bus will be dropped. The available options offer full access to all VME resources in either binary or various text formats and levels of interruptibility, exclusivity, transfer width options and so forth. If the "V" drive is shared, then these resources are available to network users, so remote requests to open such DriveVME files will succeed security permitting. Note this is the best way to transfer large amounts of data, and the only way to do so over a network, and the only way to do so without regard to processor platforms differences, the nature and status of other VME users, and the abilities of VME interfaces.

V:\V\<VME space><read cycle types><write cycle types><address range><map bank size><max transfer width, byte-swapping, alignment><bus access><error handling><permissions><priority>
will, if successful, map the requested VME space into the virtual memory space of the caller's process until the file is closed. To obtain the address in the user mode process of the first byte of the VME space in the given range, the user must execute an ordinary 4 byte read. The structure of the available data read is:

```
struct data_returned_from_mapped_VME_file {
  void *first_byte_of_map_in_callers_memory_space;
  unsigned long read_write_register_value;  //only useful to kernel drivers
  unsigned long bytecount_of_hardware_largest_mapped_transfer_width; //used by assistVME
  unsigned long current_bank; //x times 64K + VME start address is the
                      //first_byte_xxx from start points to.
};
```

Whatever amount of the above structure is read must be read in a single read request, as all read requests are presumed to reset to fetch the first byte of the above structure.

Note: The second 4 bytes read (which if desired must be read along with the native mapped pointer in the first four bytes done together in a single 8 byte read) is the value which when passed in kernel mode to READ_REGISTER_xxx routines will always deliver VME transfers of the desired width on all processor platforms. For example, the DEC Alpha will return 0 as the first parameter should the transfer width specified be less than D32. This is because the DEC Alpha platform can't support D8 or D16 cycles outside of special kernel routines. User mode programs can access these D8 or D16 routines (for which DriveVME qualifies the parameters) by writing an array of structures of this format of long values to the open file handle simply using the provided DriveVME.DLL or by file writes as follows:

```
struct mapped_io_instruction {
  unsigned long zero_for_VME_read_not_zero_for_VME_write;
  unsigned long transfer_width;
          //0->transfer bytes using D8's,
          //1->transfer words using D16's,
          //2->transfer DWords using D32's,
          //3->transfer QWords using D64's.
  unsigned long cycle_count;     //The number of consecutive cycles
          //So if transfer_width is 2 and this is 1, then 2 D32 cycles
          //will move 8 bytes.  An entry of zero will end the processing.
  void *value;          //The address of the first byte in user space for the transfer.
};
```

DriveVME will report as the result of the write the number of mapped_io_instruction structures processed without error times the size of the mapped_io_instruction structure. The value array must point to valid user memory if called in by a user process. These DriveVME files should be opened with the "FILE_FLAG_WRITE_THROUGH" option set to avoid system file data caching. Kernel mode routines must insure the 'value' parameters passed are valid. DriveVME checks the memory access security for user mode routines.

Note that this can be done more simply by using the provided calls in the DriveVME.DLL and as described in the DriveVME.H file.

The advantage to using these calls over accessing VME memory using File I/O is that the setup overhead to map the desired VME space is already established once the file is open, and the requests for VME access are not serialized but run directly and in the context and at the priority of the calling program. Therefore, these remain the fastest way to accomplish short transfers when networking or guaranteed access independent of VME interface is not required.

Note that to date all X86 DriveVME processors support mapping D8, D16 and D32 transfers directly as ordinary pointers in both user mode and kernel mode programs and so do not need this additional facility, but do need it for D64 transfers. However DEC Alpha platforms need to use this mechanism to accomplish D8 and D16 cycles from mapped memory but do not need it for D32 and potentially D64 cycles.

The first four bytes returned by the read of the Drive/VME file may be zero. In this case, the specified transfer width can only be accomplished using the above mechanism (or READ/WRITE_REGISTER_xxx routines in kernel mode using the second 4 bytes). If the first four bytes are nonzero, then the only DriveVME guarantee is that transfers of the specified width will occur without further system software overhead using the provided pointer as a base address.

> **WARNING:** Smaller or larger width transfers using the provided pointer may or may not use smaller or larger cycles on the bus depending on the processor platform and VME interface. Doing transfers using this pointer other than at the requested size may be possible but the results are processor platform specific and VME interface hardware specific. The same program executing such cycles may not accomplish the same set of VME transfers on different systems.

The resulting pointer is the base address the user should reference to cause a master cycle to occur. This mapping is dropped once the file is closed. Note that many processors do not support mapped access in all modes, and often there is a limit to how many spaces of which size need be accessed. For example, Xycom 486 class processors can only map 64K at any one moment of the extended VME addresses below 0x40000000 or above 0xfbffffff, but can map the entire remaining VME spaces. So, the first file open request to map the extended space in that region sets what is possible for the others until it is closed. Clearly, programs should not use this feature if conflict-free portability is of prime importance. Consult the implementation notes online and the DriveVMEReadme.txt file included with DriveVME for your processor to determine what VME modes can be used for mapping. Typically non-supervisory, non-byte-swapped, release-on-request modes for all spaces are supported. Newbridge/Tundra Universe chip processors are restricted to mapping only 768 megabytes of the extended space in any one map request.

Note: If a kernel mode device driver opens a VME memory map file (using the Zwxxx calls), the desired VME space is mapped into memory accessible by all kernel mode device drivers and processes (in the upper half of the x86 memory range). Kernel mode device drivers can fully access the Drive/VME functionality. It is intended that third party device drivers layer atop DriveVME to offer native NT services for VME devices.

Note: Any VME permissions set for "\m" file-io access and "\v" virtual memory mapped access apply equally to both sorts of future accesses. Blocking a virtually mapped range against subsequent access will also block that range for file-io access.

The options listed above after the V:\M\ or V:\V\ can be placed in any order.

There are defaults for all of them, typically only the <address range> need be specified. The parameter options are not case sensitive, the casing shown below is suggested for clarity. In order to be accepted as valid, at least one parameter must be specified. Also, the "extension" convention (the . CSV or .TXT or .BIN) is ignored. Actually, all characters including and after the first '.' are ignored, any comments or "extension" conventions may be noted there.

Like the other interfaces, the master mode interface appears through the file system as a sub-directory, usually on the "V" drive (use the "registry" or similar environment tracking system device configuration parameters (typically the "DriveLetter" parameter) to select other "drive" letters). Note that during directory scans, only "open" or "in use" filenames will appear, but all "open" requests will succeed if formatted properly and if access is not blocked by other VME resource using programs.

Some operating systems do not permit passing arbitrary and lengthy strings through the filename convention to devices. If the device file "DriveVME" is opened with no arguments (usually \\.\DriveVME or DRIVEVME:), the desired parameters can be established by passing them in the first write call to the newly opened file handle. Just make the entire string listed above from the root (with or without a trailing 0, but without the "drive" letter) the first write to the newly opened "DriveVME" device file. The two specification methods thereafter give identical results. (So: "\M\...options") If there are errors (permission conflicts, etc.), the open or write request will fail.

> Note: When a VME space is opened using the disk drive letter method "V:\M\..." the system does transfers using "Direct I/O". Direct IO uses memory management to "page in and lock down" in physical memory the entire program's data buffer related to the PC side of the VME transfer. This method has the advantage of not making copies of the data from the user space to non-paged system memory for writes to VME addresses, (and likewise not getting VME data into system memory, then copying it to user memory for reads). As physical memory can be a scarce resource, the overhead involved it setting up all these mappings is a good choice when transferring large blocks of data. As the "disk drive letter" oriented method is usually used for debugging and readable displays, not for dedicated programs, the "Direct I/O" method is used to memory and disk-paging time for these typically somewhat larger-per-transfer needs.

When the VME space is opened by using the DriveVME device, then writing a configuration string, the system does transfers using buffered IO. In this process, a copy of the data headed toward the VME bus is made from pageable user memory into non-pageable system space, and likewise data from the VME bus comes into a reserved non-pageable system space buffer then is copied into possibly pageable user space. This has the time expense of some copy operations, but does not otherwise consume the overhead involved in memory mappings. This is the best speed choice for repeated transfers of under a few dozen or so bytes per transfer. As most programs which use the "device" choice are dedicated to accessing typically small numbers of registers, the buffered approach was selected.

Note: It may be on some systems that certain address ranges or privilege levels or other types of VME resources may be marked as blocked or otherwise strangely unavailable. This may be due to conflicts with other programs which claimed exclusive use of the same space, or a system setup which marked certain types of VME access as restricted to kernel mode or supervisory programs only. DriveVME as configured provides full access to all VME resources, but system administrators can reserve and restrict access by command.

If a bus error occurs during reading or writing VME bus through a DriveVME file, a Device Data Error appropriate for your operating system will be returned, and any attempt to transfer remaining bytes may or may not occur after that transfer. Subsequent transfers will make fresh attempts, there is no need to close and re-open the file after a failed transfer. Indications of this error are controlled by the error reporting choices (windows message notice, DriveVME event log posting). In any event, the .DLL calls providing the total number of bus errors detected will be updated.
Should any errors occur when opening a Drive/VME file, the open request will fail, and a status message your operating system will be returned. Typical errors relate to syntax errors in the parameter field (STATUS_INVALID_PARAMETER), attempts to use facilities not supported by the current CPU, attempts to use facilities already in restricted use, and so on (STATUS_ACCESS_DENIED). In any event, the open request succeeds without error only if the requests are granted, and the file transfers end without error only if the desired transfer is complete.

MASTER MODE DRIVE/VME OPEN FILE PARAMETER REFERENCE

<VME space>

This selects which of the various VME memory spaces will be accessed. <space> may be one of: sI, sS or sE.

sI - Short I/O Space (A16). Address range 0- 0xffff (Default)

sS - Standard Memory Space (A24). Address range 0 - 0xffffff

sE - Extended Memory Space (A32). Address range 0 - 0xffffffff

s6 – VME 64 Extension A64 Memory Space. Address range 0 - 0xffffffffffffffff (new in version 4)

sC – CR/CSR Modified A24 Memory Space. Address range 0 - 0xffffff (new in version 4) sU(xx) - User set VME address modifier, not available on all CPU boards. Sets the read/write cycle and access type (AM code) to the hex code in place of 'xx', above. Use sU{xx} for decimal xx.

If no <space> entry is specified, the default is the short I/O (64K) address space (sI).

<read cycle types>

This describes the privilege level setting to use for these VME master mode read cycles. Some VME hardware and other resources require supervisory mode cycles, but most use non-privileged mode cycles. <read cycle> may be one of:

rN - Non-Privileged cycles (default if term is omitted).

rS - Supervisory cycles.

rZ - No reads will occur. This is handy for setting blanket protections without blocking permitted access. For example, a device with an exclusive read handle already opened might permit new "write only" access, so those new "write only" access files must specify "rZ" to signal they will not issue read cycles to this range.

If no <read cycle> entry is specified, non-privileged data cycles (rN) are used.

<write cycle types>

This describes the privilege level setting to use for these VME the master mode write cycles. Some VME hardware and other resources require supervisory mode cycles, but most use non-privileged mode cycles. <write cycle> may be one of:

wN - Non-Privileged cycles (default if term is omitted).

wS - Supervisory cycles.

wZ - No writes will occur. This is handy for setting blanket protections without blocking permitted access. For example, exclusive writing to a register could be confined to a single program, while permitting any others non-exclusive read access. This parameter would signal no VME write cycles will be attempted through this handle.

If no <write cycle> entry is specified, non-privileged data cycles (wN) are used. Currently, all accesses are "data", not "program" mode transfers.

<address range>

This describes the range of addresses, accessing the named <space> using the named <cycle> type, and so on, to be made available to the file. The address range to be used is specified by the letter 'a' followed by a two numbers as follows:

a([<Memory>]<Start offset><# for count or - for end offset><Count or end offset>)

For example: a(1ba-1bd) specifies the desired range to be from offset 0x1ba, through 0x1bd. Using the - as a separator implies a starting/ending address combination. Alternately using a # as a separator specifies a starting address - number of bytes following combination. So, a(1ba#5) specifies offset 1ba, for 5 bytes.

> If no address range term is specified, then DriveVME assumes the program is using Drive/VME to test for particular CPU VME capabilities (transfer widths, DMA, etc) without running the risk of address conflicts with already claimed VME addresses. All reads/writes will report EOF in this case. Attempts to open Drive/VME with unsupported options (DMA transfers, or VME64, for example) will fail.

If no separator character is found, "#1" is assumed. So: a(20-20) = a(20#1) = a(20) = a{32}.

Note: using {} instead of () implies decimal instead of hex numbers, so a(10#10) is the same as a{16-32}.

Note: the <Memory> parameter is either the letter 'z' or 'w' after the a{ or a(, or nothing. If no 'z' or 'w' are present, transfers behave like disk files, using the concept of a file position and "end of file" reported at the end of the range. Using 'z' causes each read and write to begin at the offset 0, no concept of a file position is kept (this is handy for using a file to refer to a single register or block of registers, it avoid the need to "reposition" the file after each transfer). Using 'w' is like the simulated file mode above in all respects, except the "end of file" never occurs-- reading past the end of the range wraps back to the beginning of the range.

So if the range was a(w10#6), a 3 byte read, followed by a three byte write would read VME addresses 10-12, then write VME addresses 13-15(hex), the next read or write would start at "file offset" 0 (VME address 0x10, in this case), and an EOF would never occur. This mode is best for application programs which repeatedly read or write registers of different content stored sequentially. For example, four banks of registers representing four copies of the same type of hardware could be read (if sequentially addressed) with four read commands which send the results to different structures. The cycle could be repeated indefinitely without the need for intervening "seek" or "set file position" commands.

If the range above was a(10#6), then the next transfer after 6th byte was transferred would report an EOF. Programs using this or the 'w' option must keep track of the current "file" position, and set it as needed. This mode is best for "user application" type programs which expect to read "files" which have a beginning and and end, and which have no trouble with concepts like setting file positions or "seeking" to selected areas. For example, it is appropriate to use this mode with the "notepad" simple editing program using DriveVME with ASCII formatting options to quickly review VME registers.

If the range was a(10#6) then each transfer occurs at the beginning of the range, so a 3 byte read then a 3 byte write above would affect bytes 10-12 only. It is as if an automatic "set file position" to zero occurs **just before** each transfer. This mode is the one to choose when Drive VME is essentially equating a file handle with a particular hardware register, or group of sequential registers which are always read in order, and together.

<Special RMW/ADOH Cycles>

This option is only valid when initiating VME cycles via file I/O. It has no effect for VME spaces mapped into caller's contexts. When it is present, the behaviour of the open DriveVME file changes in many ways from the default. Reads and writes do not operate the same way. This is a highly technical advanced option which can be safely ignored. It provides for advanced synchronization techniques for multiprocessor chassis. This is the method for executing Read-Modify-Write cycles and "one of" ADOH cycles on the VME bus. If you aren't familiar with these terms, it is very likely you do not need to read this section further. The command 'V(<enable>,<compare>,<swap><swap-retain>)' when present in the file open string will enable these special understandings of reads and writes. When the V(....) parameters is absent DriveVME operates normally. Examples:

V(0,0,0) - good for ADOH writes, nearly useless for RMW reads.
V(2,2,0R) - If any read returns a value such that the value "and 2" is "2", then always write back the same value with the 2's bit off and don't allow any VME transactions after the read until the write of the updated value is done (note that the return from the read is always the initial value, the one with the 2's bit on).
V(2,2,0) - Operate the first read just like the above, but then update the swap value as though the value passed to DriveVME was V(2,2,2) - which will only change the stored value in the two's position if some other agency writes a 0 in the 2's position.

Note that some users will need to open the same range with more than one file handle when using these methods, so be sure to consider the reading and writing thread and process security concerns. The following is a step by step discussion of what occurs during these read transactions.

If the hardware supports the generation of ADOH cycles and the V(...) is present, writes of any size to the open file handle execute exactly one ADOH cycle at the current VME offset, which then updates the file pointer as it would if one cycle occurred (including considering the parameters for file position updating below). If the hardware does not offer the native ability to execute ADOH cycles, the write will fail. ADOH cycles do not use the values passed in to the open file call's V parameter. This option is not the typical or most useful way to generate ADOH writes, see the i(....) priority command below.

Each attempt to read the file handle will attempt to execute exactly one RMW VME cycle at the the first address in the file open range. Each read request will execute one cycle at the current VME address (base address specified plus current file offset), no more than that will be read from VME no matter the amount asked for. The operations of this special cycle occur in order as follows:

1. Read Modify Write cycles are so named because they prevent other cycles from occurring on the VME bus until the whole RMW cycle completes. RMW cycles are like ordinary read cycles, except right after the read some work is done based on the value read and then a value is written to the read location and nothing can happen on the bus until that whole process completes. The idea is to ensure in a multiprocessor safe way a common setting for signal values. Most bridge chips and other hardware which is the target of a RMW transaction will ensure that the local hardware will lock the local bus to the RMW initiator to maintain data consistency across not only VME processors but also the VME target if it too happens to be a processor which acts based on the results of that memory. Not all VME interface boards offer RMW cycles, and some that do don't do so in all cases work without error. If any given VME chip doesn't fully offer RMW cycles in the particular way desired, then DriveVME will not attempt a RMW cycle. Instead, it will lock the VME bus to the local chassis, emulate the RMW software operations at the highest possible processor priority (above all interrupts), then release the bus (if it wasn't previously locked to this chassis). The risk is when this occurs, there is no special indication at the target that the cycle is different than a closely spaced read/write cycle set, so the target system may not lock the local bus or memory against local updates which may occur during the transaction. Consult the hardware vendor (including any errata offered) regarding the performance of your hardware. (Note: one way around this is to cause the target

memory to reside on a VME board which never changes the memory except as a target of VME cycles).

2. The data at the desired address will be read from the VME bus target and stored for satisfying the caller's read request without any change.
3. A computation to determine what value to write back occurs. For each bit position in the 32 bit enable word, if the enable bit is NOT set (is 0) the result written back for that bit position is the same as the result read and nothing else happens regarding that bit position.
4. If the enable bit for a bit position is set, the result read is compared with the same bit position in the "compare" value passed. If the result read for a bit position (for which the enable bit position is set) does not match the bit stored in the "compare" value at that position, then the value written back in that bit position is the same as the result read and nothing else happens regarding that bit position.
5. If the enable bit for a bit position is set, and the data read from VME matches the compare value for that bit position, then the value written back for that position is the value stored in the "swap" parameter for that bit position, without regard to what the original value read was.
6. In the event that bits from the swap register are written, the user may specify a further choice. If the 'R' (for "retain") parameter is specified after the swap parameter, then this step is skipped. If it is not specified, then in the event a bit position from the swap register is written, after it is written, the value originally read from VME in that bit position is saved into the swap register in that bit position. In such events, the next time such a cycle occurs, should the situation again call for writing the content of the swap register, the value read which triggered the previous swap will be written back, and NOT the original value passed in as the swap parameter.
7. Once the RMW cycle completes (or the software emulation of it completes and the bus lock is released if it wasn't already locked to this chassis at the onset of the cycle, and the processor execution priority is returned to the normal values for the calling thread), the first value returned to the caller is the original value read from the VME target. If the caller's read request is large enough to hold two values, the second value returned is the current content of the swap parameter. The width of the values returned is controlled by the cycle size selection (1, 2, 4 or 8 bytes).

<Bank Size>

This option is only valid when mapping VME memory into locally accessible memory space. It should only be considered by those who need to map more than 768 megabytes of VME space into local memory at one time. Others can safely ignore this term. The maximum size of any one memory map is 768 megabytes. This limitation has to do with the physical memory overhead involved in keeping track of page table entries for each virtual memory page mapped into VME space. However, DriveVME will permit memory maps of any size to be allocated and reserved in one call. If the memory size allocated is greater than the parameter specified here, (or 768 meg if this parameter is omitted), then only the first 768 meg will actually be visible in the user process. Access to higher multiples must be achieved by making a DeviceIoControl call which specifies the number of the desired multiple of the bank size to "bank in" or make available. This process of switching banks causes a different section of the reserved VME target memory to be available to the caller's code. Previously, this bank swapping could be achieved by simply closing one map handle and opening another with the new desired range. But, the security limitations involved in not reserving all the associated device memory as well as the overhead associated with opening and closing files prompted this enhancement. If the total memory desired is less than the map size (which defaults to 768 megabytes), then this parameter has no practical effect. To set the bank size to a desired value, use:

m(<bank size in hex bytes) or m{<bank size in decimal bytes>}

Note that currently the bank size must be an even multiple of 64 kilobytes, that is: 65536 or (10000 hex), and if it is specified at all, less than the overall map length. The bank can be thought of as a sliding window which can be positioned to start at any 64K boundary in the requested VME space. Note that to employ the banking scheme, the VME start address must be on a 64K boundary, as well.

To change which bank of the whole mapped space is visible to the process, the caller must execute a DeviceIoControl call as follows:

```
#include <dvmeddk.h>  //This holds the definition of the
IOCTL_DRIVE_MAP_BANK_SET constant, below.
BOOL SetBank(HANDLE drive_vme_open_mapped_file_handle, int banknum,void
**base_address) {
  //banknum is an integer which when multiplied by 64K and added to the VME start address
will
  //yield the VME address the first byte of the returned array will reference.
  BOOL ret;
  ret =
DeviceIoControl(drive_vme_open_mapped_file_handle,IOCTL_DRIVEVME_MAP_BAN
K_SET,
    &banknum,sizeof(int),base_address,sizeof(void *),&got,NULL);
  if (ret) { if (got!=sizeof(void *)) ret = FALSE; }
  return ret;
}
```

If the call returns TRUE, then the base_address will point to the location in local memory which if accessed will initiate a VME cycle at the VME address which is the start of the VME range in the a(..) parameter above plus the result of multiplying the passed in bank number with the bank size specified in the m(..) parameter (or 768 MB if no m(..) parameter was included. The address will point to VME for m(...) number of bytes. Actually the structure returned is the same in all respects as that returned by reading the file handle. This call combines the "set" with the "read" for efficiency.

Note that the call may or may not return the same value as previous bank requests. It is an error to presume that the virtual address returned will not change as the bank requests occur. Although on many platforms the result will not change (most of the time), on others it will change every time. So this architecture is required for best portability.

<General cycle choice: program/data, coupled/posted writes, single/block transfer,bus grant,arbitration>

This describes the method by which these transfers will share and access the VME bus. The share and access specifiers are C(...), such as C(WSD) for coupled writes, data cycles, single transfers.

c(<writes coupled/posted><program/data cycles><single/block transfers><bus grant level><bus sharing mode>) where:

C

Do coupled writes. When a VME write occurs, wait until the transfer on the VME bus is complete before moving on to the next instruction. This is slower, but assures that all the other VME hardware will read the written data at the moment the instruction to write it completes. Also, the cycle width requested will be the cycle width executed, there will be no automatic performance improving assembly. (For example, a chain of byte writes will all do D8 cycles, even though a D16 or larger ability was approved and selected). This is the default.

W

Do posted writes. When a VME write occurs, put it in a DMA FIFO then return immediately. This is much faster, and makes it possible for the DMA hardware to assemble back to back transfers of shorter width into fewer transfers of larger width (in particular, back to back 32 bit writes may be assembled into single D64 transfers assuming the other parameters are chosen properly on suitable hardware). Note that all local CPU VME reads will be held up until the last of the posted writes has completed. Warning: This means that although the local CPU has written the data, the data may not be immediately available to the other VME cards in the chassis. If sequencing is essential, do a read after a chain of writes. The read will not complete until all the pending writes have completed. Note that popular VME interfaces can only do coupled writes when the VME bus is locked to the local chassis. To insure you are getting posted writes, set the "release on request" or "release when done" bus priorities. Note also that if the VME hardware design suffers from the deadlock which can occur when the local chassis memory is the target of VME cycles at the moment the local system is attempting to initiate VME cycles, DriveVME will automatically attempt to lock the VME bus before initiating cycles via file I/O (to prevent the deadlock). This has the side effect of converting posted writes to coupled writes.

S

For every VME address cycle, do only one data cycle. These are "single" transfers. Some VME hardware accepts a single address cycle, then a series of data cycles with the assumption that the data is for sequential addresses. This improves the VME performance by nearly 100% on large transfers. However, not all hardware can accept such burst traffic. To accommodate these one address for one data cycle devices, use this "single" flag. This is the default.

B

Permit Block transfers, replacing S, above. Should multiple back-to-back transfers of the VME bus be requested to consecutive addresses, do only one address cycle, then data cycles. This improves performance nearly 100%, but requires hardware capable of accepting this mode. This is because the time necessary to execute an address cycle between each and every data cycle may be saved. Note: Not compatible with non-incrementing DMA.

L

Do multi-block transfers, replacing B above. MBLT Multiblock transfer types will be attempted. New for version 4.

E

Do 2eVME transfers, replacing L above. Double edged higher performance VME 64 extension transfer type. Usually 160MB/s. New for version 4.

T

Do 2eSST transfers., replacing T above. Requires following (decimal) or {hex} desired maximum transfer rate. So T(160) -> attempt 2eSST cycles at 160MB/sec max. Other popular rates are 267 and 320. 0-> lowest available. New for version 4.

R

Do 2eSST Broadcast Transfers. Replacing T above. Requires following (decimal) or {hex} desired maximum transfer rate, followed by (decimal) or {hex} broadcast destination slot bitfield, where value & (1<<(slot-1)) -> send 2eSST cycle to card in this slot. So R(160){a} -> attempt to broadcast 2eSST cycles at 160MB/sec max to slots 2 and 4 . Other popular rates are 267 and 320. 0-> lowest available. New for version 4.

D

Execute Data cycles. This is the default. The VME bus permits cycles to the same addresses to be distinguished as either for program or data. Some VME processors set this election automatically depending on whether the access is an instruction fetch or a register load. However, other processors require this be set by the user for all transfers.

P

Execute Program cycles. Should the nature of the data to be transferred be program instructions, (or for other addressing reasons) program cycles are available. Note that most VME interfaces do not permit more than one of these elections to be made for the entire interface chip, so memory mapping requests should all specify one or the other (preferably D). If this is not the case, whichever memory request comes second will be denied.

0 or *1* or *2* or *3*

Request the VME bus on at this request priority level. Nearly every application calls for the default, 3. Note that only one such setting may be permitted for all memory mapped resources in most cases. Also some platforms can only support requests for the bus on level 3. Finally, there are some VME "slot 0" arbitration cards which only provide arbitration services for level 3. In short, use the default unless you are very sure your application reasonably requires something else and does not need to be too portable or have third party VME software running.

F

Request VME bus access in "fair" mode. This mode will not request the VME bus on the chosen level a second time until no other requests for the bus at the chosen level are outstanding. This provides essentially round robin access to the bus, with the longest waiting request at the highest level being granted the bus first. However, this mode is not the tradition in VME applications, and is not the default. Note also that only one such election can be made for all mapped resources per CPU on most platforms-- whichever application makes the first mapped memory request will set what is permitted until the last application making this request is closed. (Same goes for program/data cycles, bus request mode, bus release mode (when done/on request), bus request level).

M

Request the VME bus access in "demand" mode. This mode will grant the card closest to slot 0 access to the VME bus at any given request level. Higher levels are always granted access first. Demand mode guarantees performance and implies a "priority sensitive" installation of bus mastering cards in the chassis. Cards in higher numbered slots may not be able to access the bus in a timely fashion -- be sure to consider this requirement in your designs. This is the default.

<max VME transfer width, byte swapping, alignment selection, prefetching>

This describes how much data, at most, to transfer per VME I/O cycle, and choices about how to handle it during the transfer. For VME access through File I/O, the system will use exactly the width specified (or highest available if t(0) or t(0s)). When accessing VME memory mapped into user process or kernel space the returned pointers may be different depending on the ability of the processor to access memory at the desired widths and lower widths.

(In particular, the DEC Alpha computer will always return 0 as the first long pointer as the mapped address if the transfer size requested is less than D32, since the processor can only directly address values using D32 cycles (even programs which only refer to bytes or 16 bit values using this pointer will execute D32 cycles to get these). To access VME memory as 32 and/or 16 and/or 8 bit values, pass the second 4 byte pointer read from the map file in kernel mode to the READ_REGISTER_xx routines-- on X86 platforms the first and second values are the same. In user mode, see the DriveVME.DLL documentation or see the DVMEDDK IOCTL_DRIVEVME_MAPPEDMEM_IO call)

For example, a read request of two bytes can be read using two single byte cycles "t(1)" , or one 16 byte cycle with or without byte-swapping ("t(2s)" or "t(2)"). The default for this term is t(2s). The width specifier is the letter 't' followed by maximum number as follows:

t(<max>[<swap>][<align>][<prefetch>]) where <max> is one of:

0

Best available transfer width for the current CPU. This is guaranteed never to be less than sixteen bits per transfer (t2). Note that some VME configurations do not come with a P2 backplane, but use VME processors which support 32 bit and higher bits-per-transfer cycles. So, "best available" will execute 32 bit cycles the upper 16 bits of which will be wrong unless a P2 VME backplane is installed. If the application does not gain a speed benefit by using 32 bit or higher bits per transfer cycles, consider using a "2" instead for better portability. Note that the Pentium only executes 32 bit external bus signals, but these are automatically assembled by hardware into 64bit VME transactions for speed if t(8..) is chosen on some systems.

1,2,4 or 8

Transfer at most one, two or four or eight bytes per cycle. Note that smaller transfers are always possible, and sometimes even necessary. This just sets the maximum. An explicit request for a particular maximum transfer width (like 64 bit maximum "t(8)") will return an error if the host CPU does not support that option (usually VME64).

<swap> is the letter 's' or nothing. If the letter 's' is appended to the width, then the data bytes are swapped on multi-byte transfers (so t(1) and t(1s) are the same). For example, if 16 or 32 bit quantities are to be read, and the quantities are each separate numbers, then no swapping need occur between big-endian (Motorola, most VME peripherals) and little endian (Intel) architectures, as the data is placed correctly on the bus. But if these are byte quantities (text) being read in multi-byte transfers, then byte swapping should be selected so that the string "fast" saved in a big-endian way does not come out as "tsaf" on a little-endian machine. The swap is for the full width of the transfer, so for 16 bit transfers bytes 1 and 2 come over 2,1, 32 bit transfers bytes 1,2,3,4 as 4,3,2,1, and so on. Some processors do this at high speed in hardware, others require slower software emulation. Note: If the 't' term is specified, then you must still add 's' for swapping if swapping is desired, even though t(2s) is the default.

<align> is the letter 'a' or nothing. This option is important when doing multibyte transfers per cycle when the VME side of the transfer is not word aligned for 16 bit transfers, or double word aligned for 32 bit transfers, or quad-word aligned for 64 bit transfers. The 'a' option forces execution of multibyte transfers starting with the first specified byte for the requested width yielding the result as though it were aligned and transferred with one cycle. With this feature enabled, the appearance of single-cycle alignment blind transfers occurs wha tever the VME address and cycle size, even if it is not aligned on the boundary needed to accomplish the transfer in reality in one VME cycle. Without this option, a request to transfer words starting on an odd VME address would transfer one byte, then words. A request to transfer 32 bits starting at VME address 1 would transfer the byte at 1, the word at 2, then quadwords from then to the end, etc. So, there is a VME performance penalty associated with the freedom to execute unaligned transfers. Leaving this option off is best for speed, if one knows that no transfers of unaligned multibyte quantities can occur. Note that many bus interfaces will not execute unaligned transfers, so forcing the system to operate using the 'align' flag may cause more transfers to read the same data. Also note: If all the transfers are aligned on the cycle boundary appropriate for their width, then this flag is essentially a "no op", as it's services will never be engaged.

No VME memory mapped into a process can be unaligned. This parameter is ignored when mapping memory into a process, it is up to the programmer to only execute aligned requests. Or, to purposefully execute unaligned requests which does have merit on rare occasions. The results of executing an unaligned request in mapped memory are undefined, most processors will execute two of the nearest aligned requests, then assemble the data -- but this should not be depended upon.

The easiest way to understand these effects is to see the effects of the various operations. The table below shows how data stored at the given VME address where there data equals the address would be read under the various options. The information listed is shown as the "little endian" (Intel) processor would store the data bytewise in its local memory. Note that just because the data appears to have been stored the same way in some cases, it may have gotten stored there with more or fewer VME cycles. The more bytes per transfer, the fewer the total cycles. Also note that using the 'A' flag to force the data to appear aligned when accessed in an unaligned way may result in cycles smaller than the stated size.

For example, accessing a 32 bit value declared as t(8a) at but at an offset where "address AND 3 == 1" one will cause an 8 bit VME cycle, then a 16 bit VME cycle, then another 8 bit VME cycle. The result will be stored as though a single 32 bit unaligned cycles was supported, however instead of taking 1 cycle to access the data, three were needed. This also makes the assumption that the relevant device can support cycles smaller than the named one. This is because the VME bus does not support unaligned transfers at the physical level.

The table below shows transfers which start at different offsets. Where a list ends with ..., the implication is that the pattern repeats. Entries without ... show the result of fixed 16 byte reads, typically on partial transfersized unaligned end bytes.

```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10 11 12 13 14 15 16 17 18 19
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10 11 12 13 14 15 16 17 18 19

:(1sa),t(1a),


 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10 ...


    1  2  3  4  5  6 ...

       2  3  4  5  6  7 ...

          3  4  5  6  7  8  9  ...
             4  5  6  7  8  9  A ...
                5  6  7  8  9  A  B...
                   6  7  8  9  A  B  C...
                      7  8  9  A  B  C  D...

 1  0  3  2  5  4 ...
    1  3  2  5  4  7  6 ...
    2  1  4  3  6  5 ...

 3  2  1  0  7  6  5  4  B  A  9  8 ...
    1  3  2  7  6  5  4  B  A  9  8  F  E  D  C  10
    4  3  2  1  8  7  6  5  C  B  A  9 ...
       3  2  7  6  5  4  B  8  9  8  F  E  D  C  11 10
```

This whole issue of alignment and byte-swapping arises because the VME bus allows for processors which use different data storage schemes to operate together. As the bus can't know the content of the data, it is up to the software and each processor to correctly align and swap bytes as needed to preserve the meaning in the data transferred.

Note that if the last few bytes of a transfer are smaller than the word size selected, the system will attempt to complete the transfer by choosing one transfer at each of the next smaller transfer sizes in order to the byte level if needed, using the same swapping choices and alignment as it falls.

<prefetch> sets how many bytes of VME data are read upon an initial read to speed up delivery of following, presumably sequential, reads. This is new in version 4. If omitted, no prefetching is done. The choices are:

N

No prefetching. The safest choice particularly if reading io registers that might be destructively read, or to avoid bus errors when VME targets don't offer memory that doesn't end on a cache line boundary (32 bytes typically), or when reads are rarely sequential.

B

Best prefetch. Prefetch as much as the VME interface chip allows. At this writing that is 16 lines of 32 bytes.

1,2,3,4

2^^(choice) = number of 32 byte cache lines to prefetch.

<error handling>

This term determines what will occur in the event a VME bus read or write or other request from this CPU results in a bus error. The choices are included in e(...). The default is to return the bus default value, usually -1, otherwise taking no action. This choice is taken as advisory; meaning that the system will comply if the hardware permits, but will not fail the open request if it does not.

e(N) Causes no special action to occur in the event of a VME bus error, other than returning the VME bus default value. This is typically -1. Note that many systems can only support this mode, which is the default. The AssistVME log will not be updated to report anything, even if a bus error is detected.

e(S) Stop any I/O in progress, up to but not including the byte at which the error occurred. The net effect of this is that read or write requests will return with fewer than the requested number of bytes read or written. This option only has meaning for file mapped VME memory (\M\....). This option is not available on all platforms. The AssistVME drive operations log will be updated with as much details as the system can provide. It is ignored on mapped regions (other than enabling logging to AssistVME).

e(Mxxxx) Send windows message xxxx (always in decimal) to the calling process, with the VME address which failed as the 32 bit LParam value. Not all platforms record accurately which VME request it was which generated the error, only that the error did occur in the context of processing a request from the calling (file) or current (mapped) process. If the address causing the error is unknown or indeterminate, the resulting value is 0. If the S parameter is also specifed (e(SMxxx)) then the AssistVME drive operations log will be updated with as much details as the system can provide. Note that AssistVME must be running in its default "service" mode for this feature to have meaning. (AssistVME is always installed with Drive/VME, unless special action is taken to disable or stop it.)

<permissions>

This term determines which other threads and processes can access this range. The permissions allowed follow the letter 'p'. Unless qualified, the permissions apply to both reading and writing permissions sought. See below.

p(A) Allows all others non-exclusive access. (default if no p term specified). Has the effect of blocking future exclusivity requests.

p(P) Allows access to this space and range by any other thread in this process, but no other processes.

p(T) Allows this thread to have added access, but no other threads.

p(K) Allows only kernel debuggers shared access. Can't be issued in user mode. Used by kernel mode device drivers layered atop Drive/VME. Will fail if a user-mode debugger holds part of the requested space.

p(N) Blocks all reads and writes from this handle for the given space (meaningless if rZ and wZ), but allows other kernel mode only, processes access to the range. Handy for reserving the entire supervisory space to kernel processes, without blocking the reservation of any particular subspace by a kernel process. Can be issued in user mode.

p(D) Identifies this process as a debugger. This will allow access typically denied by p(P), p(T) requests if made by a user mode process, and will grant any access if this is a kernel mode process. Any ranges granted will not block any exclusivity requests by other processes, including other debuggers.

p(....F) Identifies this software as "finished" software, and in particular essentially disables the 'D' Debugger parameter of any user mode request to prevail against local register access exclusivity. So, an address selected with p(TF) would block all other threads, even debugger threads, from getting granted access to the named range, except: any kernel mode debugger will be granted access.

The permissions for reading and writing a space can be set separately as p(w?r?) where the ? can be replaced with A,T,P,K,N,D above, the F for finished software (to block user mode debuggers) may be added. the W option sets the write permissions, the r sets read permissions. If a one read or write permission is set, but not the other, the unset one remains at 'A' . (The default of p(A) is the same as p(wArA).) Note that p(AF) "permit all, finished code" will not block debuggers as a special case, permit all includes debuggers.

Note that if the read/write access type is set to 'none' (rZ or wZ) then the permission sought is ignored for that type of operation-- except for the p(N) parameter, which permits other kernel mode processes exclusive access to a range, but no user mode processes. Permissions are granted in a first come, first served order, so administrators who seek to reserve certain VME access types should open the relevant file handles during system startup operations. This means that even kernel mode requests for exclusivity which occur after established but conflicting user mode requests will be denied.

Also, consider how overlapping ranges work. For example, if thread 1 does a "permit thread" to for addresses 10-12, then another thread 1 "permit all" for ranges 1-100 would succeed. Other threads would not be able to access 10-12, but would be able to get "permit all" access to 1-9 and 13-100.

<priority>

This term defines whether other processes may interrupt transfers using this handle, and whether to use DMA modes, whether to attempt an ADOH cycle and local bus lock before file I/O transfers, and other transfer bus related options. The optional substitutions for the ? term and the xxxx term in the commands are explained later.

i(?Sxxxx) Generally "Release on Request" (ROR) (i(S0) is the default). Does processing at dispatch level priority. The CPU may not execute transfers while the bus is locked elsewhere. This does the transfer above all user-mode threads, but doesn't lock the VME bus. This is the safest mode, but transfers that take more than one bus cycle to complete may be interrupted by other bus activity. This may be a problem if other VME resources may access the same registers this one is changing, or if there is critical time-sensitivity to setting these registers. Note that if transferring multi-byte data in a single cycle, this mode does just as well as any more restrictive, higher-priority mode. If i(S) is not available on this CPU, i(D) below is substituted. This retains the VME bus unless some other resource requests it, done or not, saving bus-acquisition time on subsequent transfers.   (Timeour or Done) and Request.

i(?Dxxxx) Exactly like i(S), except releases the VME bus when done (RWD) with the transfer, whether other requests for it are waiting or not. This CPU must re-acquire the bus before the next transfer can occur. If not available on this CPU, i(S) is automatically substituted. (Timeout or Done).

i(?Qxxxx) Exactly like i(S), except releases the VME bus when done or time expires and any other bus request exists. not available on this CPU, i(S) is automatically substituted. (Timeout and Request) or Done.

i(?Bxxxx) Exactly like i(S), except releases the VME bus when done or a higher priority request for the bus occurs. If not available on this CPU, i(S) is automatically substituted. (Timeout and BCLR) or Done.

i(?Lxxxx) This locks the VME bus during transfers. The transfers occur at DPC priority level, that is, above all user threads but just below local CPU interrupt processing. The transfer will not commence until the bus is locked to this system. The bus is released when the transfer completes. No other transfers can occur on the VME bus once this starts, until it finishes (or this segment finishes if using a nonzero xxxx number).

i(?Rxxxx) This is the realtime option. Like i(L), but the transfer happens at the highest possible NT priority after the bus is locked. The priority returns to normal after the transfer completes and the bus is released. The transferring process essentially owns the entire system for the duration of the transfer, it locks out everything but catastrophic failures, including the system clock, so keep these short. Some systems may impose a 1024 transfer-per-segment limit on this option. Instead of this option, consider using the ML option, that is, locking the VME bus but using DMA for large transfers. This will avoid locking out interrupt processing for this CPU.

The xxxx in all the above is a hexadecimal number (if i{...} then xxxx is a decimal number). If non-zero, this is the number of transfers (not necessarily bytes, if t(4) then 1 transfer is 4 bytes) to complete before releasing the VME bus to other bus masters and waiting local processes. After the specified number of transfers, the bus is released and the transfer scheduled to re-commence in the system queue- but after any other processes scheduled at this priority have had a chance to run.

The ? in all the above is one or both of the optional letters 'A' and 'M' and (new November 2000 v3.12 on Universe IIB equipped boards) ‘N’. If missing, the transfers happen using CPU programmed I/O with no ADOH cycle and local bus lock before each transfer request. if the letter 'A' is present, before each File/IO transfer request, and only when parameters L or R (which lock the bus) are specified, the processor will execute an ADOH cycle on the VME bus, and a LOCK on the local bus before commencing the transfer (but only if the interface supports it). So i(AL200) would, upon each transfer request, raise the processor priority above task switching (but not interrupt handling), lock the VME bus to the local chassis, execute an ADOH cycle on the VME bus, do a LOCK command on the processor bus, do at most 200 transfers, release the VME lock, restore the processor priority, then return.

If the letter 'M' is present for ?, where swapping modes, hardware capability and priorities permit, the transfer happens using DMA transfers. So, i{ML5000} would use DMA at just below interrupt priority in 5000 transfer cycle maximum (at however many bytes per transfer) chunks. Leaving the "M" out of the above results in CPU managed transfers. Note that for DMA transfers to occur, the low-order 3 address bits of both the source and destination addresses must match, and no software-assisted alignment can be selected.

If the letter ‘N’ is present for ?, in every respect the system behaves as though ‘M’, the DMA choice was made—except the VME address is not incremented during the transfer. Note that this feature has all the restrictions of the DMA feature, but neither software assisted alignments nor multi-byte transfers can be selected. If set up improperly, normal DMA or programmed IO transfers occur without warning.

Note: The DMA option is best suited to asynchronous VME read and write operations of large (over 125 or so) transfer operations. The time involved to set up DMA requests must outweigh the performance gains due to the parallel execution nature of such transfers.

<formatting options>

Typically, VME reads and writes return the actual binary values without interpretation. This formatting code is f(B), unformatted binary transfers, the default. No parameters are accepted with the binary format.

In general the syntax is: f(<general format type like binary, signed decimal...>[<bytes per item, 1, 2, 4, or 8>[<item separator style>[<items per row for non-binary formats>]]])

f(H1) interprets the results in ASCII hexadecimal, 2 bytes transferred via read/write for every one VME byte transferred, with no spaces between the values.

Note: for any many-ascii-to-one-byte options, if the program does a series of single byte transfers, the VME data is read once to deliver the most significant digit, the remaining digits are cached for subsequent reads. Any seek or Set File Position to any but the next character in the cache, dumps the cache.

f(H1s) puts a space between each two hex digit output (3 ASCII for 1 binary), f(H1c) puts a comma, then a space between each (4 ASCII for one binary). Appending an 'r72' after the s or c will substitute the first space or comma/space which would have fallen *after* the 72nd column with a newline character f(H1sr72) or a CR, LF f(H1cr72). The substitution of this 'end-line' sequence will always take place if the next element will not entirely fit on or before the given column number. You can replace the 72 with any decimal value. The 'r' parameter retains constant output-bytes-per-line to binary-bytes ratio by replacing, not appending, the " " or ", " with either a linefeed or return-linefeed combination respectively. You can effectively create comma-separated-variable (CSV) formatted files this way, thereby loading VME data directly into spreadsheets and databases. Note: This assumes columns are numbered starting with 1, and the column numbers are figured based on the VME address offset of the item relative to the beginning of the requested VME address space, not the beginning address of any single transfer.

f(H2[s[rxx],c[rxx]]) does the same but transfers 16 bits and transfers them through 4 ASCII hex characters, with the same optional r, s and c qualifiers.

f(D[1,2,4,8][s[rxx],c[rxx]]) does the same as the H counterpart, but in unsigned decimal, using three spaces for 1 byte, five for 2, 10 for 4, right justified, space padded. f(S...) is the same as f(D..) but signed, with one more space to allow for the possible minus sign for each byte width (4 for 1, 6 for 2, 11 for 4.).

f(P) pretty prints the reads, and assumes binary writes. This feature makes for quick debug displays. It is possible to create a "program manager" entry passing a Drive/VME "file" to the notepad editor, which when run will take a snapshot of a selected VME resource. Each read should be in multiples of 81 characters, and prints a line showing 16 values starting at the current offset like: 0x<offset>: x x x x x x x x | x x x x x x x x *........|........*<CR,LF> where the .'s are the ASCII if the binary is printable, and the xx are the hex representations of the data. If the range does not extend to fill the line, blanks are padded. No parameters are accepted with the pretty-printing format.

For example opening : "V:\M\a(100#20)f(P)t(1)wZ" would pretty print the 32 bytes starting at VME address 0x100 in the short-io space using standard cycles transferred 8 bits at a time on four rows, then report an EOF. It would also block any attempts to write the file. The display would be truncated upon the first bus error.

Offering Local Resources to Other VME Bus Masters

This section contains the full technical details relative to offering local resources to other VME bus mastering hardware. Note that the section on AssistVME at the front of this manual outlines these functions in basic detail, and includes an automatic and relatively painless method for creating the detailed VME file “names” required by DriveVME, as well as an automated method for generating a sample program that actually offers the local resource.

Although this section does not require that the user try AssistVME section first, it might represent a great time savings to do so.

This section is meant for those users who want or need to understand all the technical details behind the choices.

This Drive/VME facility allows the user to open a file whose name specifies how regions of local PC memory should be offered to the VME bus. Drive/VME allocates these regions, removes them from the virtual memory pool or from other conflicts or use by the rest of the operating system, then provides the physical address of the slave memory for use by other processes (if it was unspecified, initially), and a virtual address for the same space for access within the caller's process. Security is enabled, is it possible to reserve regions of memory so that they may or may not be shared by other processes.

For example, while the file "V:\S\a(100#5)sSb(100000)" was open, 5 bytes of memory starting at local physical address 100 would be available to VME users for both program and data, normal and supervisory cycle access in the standard VME space at offset 0x100100. Reading four bytes from this file would give an address in the callers process which serve as the base address for the first byte at the local physical address 0x100. When the file is closed, the memory set aside for VME slave operations is returned to the local operating pool. When the last file offering slave memory is closed, the slave memory interface to the VME bus is disabled (which is the default state until the first request occurs).

Note: Many systems limit the number of access modes and spaces in which slave memory can be simultaneously offered. Users should select the extended VME space, permitting all cycle types when possible. Also: For security, if the first file opened is "\S\X" then the slave interface is disabled, and all further attempts to grant slave access to local memory will be denied.

Last, the this interface of Drive/VME is fully networkable. By designating the "V" drive as a shared "network" drive, remote users, even dial-in users, can be granted access to local memory slaved to other VME resources.

Major Features:

1. Ability to map different VME spaces to some or all of the same local physical memory.
2. Ability to share local physical memory exposed to the VME bus among more than one local thread, process, or kernel mode driver.
3. Allows NT kernel mode drivers to directly offer any local physical memory (including memory mapped devices) and the local I/O space to one or more VME address ranges.
4. Ability to set aside an arbitrary amount of fixed, non paged, system memory for Drive/VME offering to the VME bus.
5. Ability to read and write the VME shared local memory region as a file, so (if desired) remote users via networks can examine and/or alter the content of shared memory regions, or share new regions using ordinary file I/O.
6. The system security features of Windows/NT are preserved.
7. Multiprocessor NT and later systems supported.
8. Full resource sharing arbitration among unrelated applications running on the same CPU.
9. Ability to specify or let the system choose particular VME characteristics and addresses, local memory characteristics and addresses, or any combination.
10. Ability to restrict or disable VME memory sharing.
11. Support for local program and VME debuggers.

Using any standard and ordinary means to open a device file named:
V:\S\<VME space><slave cycle types><PC address (opt), length><VME Offset (opt)><Slave Bus options><Slave Transfer Priority><Select Slave Byte Swapping><permissions><Disable or Limit Slave Interface><error handling>

For example, while the file "V:\S\a(#6)sSb(1000)" was open, 6 bytes of memory starting at a local physical address picked by DriveVME would be available to VME users for both program and data, normal and supervisory cycle access in the standard VME space at offset 0x1000. Reading four bytes from this file would give an address in the caller's process which serve as the virtual memory process-relative base address for the first byte of an array at least 6 bytes long starting at the local physical address which corresponds to VME A24 address 0x1000. The next four bytes yield the actual address other VME bus masters should use to access the reserved space-- 0x1000. Once the file is closed (and no other local request exists to hold the range open) the VME space corresponding to the request will no longer be decoded-- requests for the space will result in bus errors.

Please read about the p(N) parameter, which allows a range of VME addresses to be reserved for subsequent multiple local slave allocations. The conserves limited VME hardware mapping resources. It also makes it possible to have many local or off-card threads/processes access the same patch of shared memory. If many applications which offer slave resources are running, they may exhaust the available VME slave range decoding registers when none would do so if run individually. Using the p(N) parameter and AssistVME, it may be possible to create one overall slave reservation which combines the compatible spaces needed by the requests of the individual programs-- allowing them to use the same VME slave map resource.

Detailed discussion:

Once the desired file containing the specifications for the desired space to share has been opened, the very first read from the file will deliver the following information. As much of the information listed below as desired should be read in one request, as subsequent requests for reading or writing will deliver the contents of the shared memory (beginning at offset 0), just as though it was a file. This makes it possible for remote network users to offer and read/write slave space on a distant NT VME chassis by doing ordinary file I/O.   Starting in Version 4, Note the first structure applies only in spaces other than A64, the second applies when exposing local memory to the VME64 extensions A64 space.

struct VME_Slave_Allocation_Results {

void *local_memory_slave_base; //32 bit pointer to slaved memory. Null for p(N) calls.

unsigned long vme_memory_base_offset; //where in VME space *local_memory_slave starts.

volatile unsigned long vme_memory_low_decode; //first VME address decoded to support the given request

volatile unsigned long vme_memory_high_decode; //last VME address decoded to support the given request

//Note: vme_memory_low_decode <= vme_memory_base_offset.

//Note vme_memory_base_offset+requested_length-1<=vme_memory_high_decode.

```
};

struct VME_A64_Slave_Allocation_Results {

    void *local_memory_slave_base; //32 bit pointer to slaved memory. Null for
    p(N) calls.

    unsigned __int64 vme_memory_base_offset; //where in VME space
    *local_memory_slave starts.

    volatile unsigned __int64 vme_memory_low_decode; //first VME address
    decoded to support the given request

    volatile unsigned __int64 vme_memory_high_decode; //last VME address
    decoded to support the given request

    //Note: vme_memory_low_decode <= vme_memory_base_offset.

    //Note vme_memory_base_offset+requested_length-
    1<=vme_memory_high_decode.

};
```

1. The first 4 bytes read from the open VME memory slave file handle are a pointer the local caller should use to access the first byte of the slaved memory which corresponds to the VME space-- the VME address of which is in the next 4 bytes. These are the offset into the selected VME space other bus-masters should use to get the first byte of memory referred to in the first four bytes. If the caller is a kernel mode caller, the addresses given will be valid in any process/thread context. If a user mode caller (the majority), the virtual addresses given will only be valid in the context of the calling process. This value has no meaning for requests made from processors connected via network. For requests to reserve space (p(N) parameter), this value is NULL, no space is mapped.

2. The next four bytes (8 if A64 space) are the VME address of the byte in the desired spa ce which corresponds to the first byte offered by this request. Note that VME addresses below this value may be decoded, this just corresponds to the address which if modified by other VME bus masters will be readable as the first byte of the file or at the address in the user process mentioned by the first 4 bytes read. This value is usually the <VME offset> b(xxx) or b{xxx} parameter, which shouldn't be specified explicitly unless it is clearly necessary. (See the discussion under the p(N) parameter).

3. The next four bytes (8 if A64 space) refer to the lowest address in the chosen VME space that is decoded by (as thus allocated to) this processor to support the named request. This may be lower, perhaps depending on the hardware much lower, than the desired VME offset. These addresses are always backed by physical memory which is always available, whether the calling process is "swapped in" or not. Note that large requests for slave memory may not succeed if the system doesn't have enough "non paged" memory set aside. Windows/NT users can avoid this constraint for large slave space sharing requests (>64K roughly depending on the installed memory of the system), see the note below.

4. The next four bytes (8 if A64 space) are the highest VME address used by this request, which may be well beyond the desired number of bytes and starting offset requested. The size of the decoded space may be larger than the requested space due to hardware constraints which usually limit the minimum decodable size to 4096 bytes, but possibly many megabytes on some processors.

   Note: On some VME interfaces, local memory outside the ranges desired and returned will be exposed to the VME bus which is subject to "paging" under the operating system. VME bus masters which change this memory can cause system crashes and unpredictable behavior. On systems using the Newbridge/Tundra Universe chip, only memory specifically allocated for VME access is made available, which maintains local integrity. But, this is not the case for all interfaces.

   So, beyond 4K address boundaries on non-Newbridge systems, the decoded space may refer to physical memory subject to paging, and should not be used by other VME resources, even though it is decoded by the local hardware. To save physical memory, DriveVME may not be able to allocate to this request all the memory needed to allow the hardware to offer just the desired addresses. It will always fix in a non-paged way and allocate enough for the desired request, however, and is little more as physically possible. Be sure only to refer to the requested memory both locally and over the bus, as even though other physically nearby memory may be fixed and allocated, it may not be mapped into the virtual space of the calling process. Changing that reserved memory will likely cause random system failures.

5. For example, to share 16 bytes at VME offset 40, it is likely the system will have to decode 4096 bytes starting at VME offset 0. The system allocates and sets aside whatever physical memory is needed so that requests to the

decoded space do not result in bus errors, but access is granted to the calling process of only the requested length beginning at the requested offset. Drive/VME attempts to consolidate nearby requests for slave VME memory, to save the scarce resource of VME slave memory decoding registers.

6. To conserve VME slave hardware decoding resources, it is the best policy to open as few of these files as possible, requesting as much memory as will be needed by the entire effort in one handle, then sharing that within the driver or application. Generally speaking, opening several file handles requesting memory in the same space but offset more than 4K from another request, or using different sharing terms generally consumes a VME slave interface hardware resource.

7. Most CPU-VME interfaces have minimum numbers of bytes that must be decoded from the VME space. For example, if a request asks only for 16 bytes to be shared between VME and local space, it might be that 4096 bytes of VME memory must be decoded to offer access to the 16 desired. This has no effect upon the calling program, but may conflict with other VME cards.

8. Note: Many systems limit the number of access modes and spaces in which slave memory can be simultaneously offered, sometimes to just one. Users should allow for as many default parameters as possible for compatibility. Also: For security, if the only file opened is "\S\X" then the slave interface is disabled, and all further attempts to grant slave access to local memory will be denied.

Slave Interface Filename Oriented Command Parameters:

The options listed above after the V:\S\ can be placed in any order. There are defaults for all of them, typically only the <address range> need be specified. The parameter options are not case sensitive, the casing shown below is suggested for clarity. In order to be accepted as valid, at least one parameter must be specified. Also, the "extension" convention (the . CSV or .TXT or .BIN) is ignored.

Actually, all characters including and after the first '.' are ignored, any comments or "extension" conventions may be noted there.

Like the other DriveVME interfaces, the slave mode interface appears through the file system as a sub-directory, usually on the "V" drive (use the "registry" or similar environment tracking system device configuration parameters (typically the "DriveLetter" parameter) to select other "drive" letters). Note that during directory scans, only "open" or "in use" filenames will appear, but all "open" requests will succeed if formatted properly and if access is not blocked by other VME resource using programs.

Some operating systems do not permit passing arbitrary and lengthy strings through the filename convention to devices. If the device file "DriveVME" is opened with no arguments (usually \\.\DriveVME or DRIVEVME:), the desired parameters can be established by passing them in the first write call to the newly opened file handle. Just make the entire string listed above from the root (with or without a trailing 0, but without the "drive" letter) the first write to the newly opened "DriveVME" device file. The two specification methods thereafter give identical results. (So: "\S\...options") If there are errors (permission conflicts, etc.), the open or write request will fail.

Should any errors occur when opening a Drive/VME file (or writing to \\.\DriveVME the first time), the open request will fail, and a status message your operating system will be returned. Typical errors relate to syntax errors in the parameter field (STATUS_INVALID_PARAMETER), attempts to use facilities not supported by the current CPU, attempts to use facilities already in restricted use, and so on (STATUS_ACCESS_DENIED). In any event, the open request succeeds without error only if the requests are granted, and the file transfers end without error only if the desired transfer is complete.

Setting aside more than 64K of slave memory under Windows/NT

WINDOWS/NT Note: Microsoft NT does not provide a supported way to automatically specifically take an arbitrary fixed system physical address, remove it from the virtual memory scheme and make it free for slave use. This note describes a setup method which through extra setup makes this possible. Without any special intervention, NT can provide non-cached, cache-aligned addresses *which it picks* for slave memory operations, which are stable, but not repeatable boot to boot. In the slave memory parameters (\s\...) the address term will always succeed if the physical start address is omitted, and only the desired length is provided.

For example: \S\a(#100) would assign 256 fixed bytes from the non-paged memory pool at an address convenient to Windows/NT, which is not reported to the user until after the call succeeds. This is perfectly acceptable to any application which can communicate to the other bus-master devices the address to use. This method is by far to be preferred as it is more portable.

However, some devices require that the physical address used, coupled with a known VME address offset, be known in advance to offer PC memory at a known fixed VME space memory address, which is always the same every time the system boots. Other requirements call for a larger amount of memory to be slaved than the non-paged memory pool under

Windows NT provides. There is a safe way to satisfy both these needs under Windows NT.

The Windows NT memory manager can be instructed to use less than the total memory available on the processor for paging, essentially ignoring the remaining memory. Drive/VME can cause this "unavailable" memory to nevertheless be assigned into the slave memory task in a fully supported, if not documented, way to the virtual memory space. To make such fixed memory available, you must be sure you need it, then edit the "boot.ini" file setting the MAXMEM parameter. Find the line that boots the system, then reduce the amount of memory available to Windows NT virtual memory management by the number of megabytes you wish to offer for known-in-advance fixed memory physical addressing. For example, on a 16 megabyte machine, BOOT.INI entry:

```
multi(0)disk(0)rdisk(0)partition(1)\NT="Windows NT" /MAXMEM=15
```

would reduce the amount of memory Windows/NT uses to 15 of 16 megabytes. You can then specify that the physical address to use be assigned in that last megabyte, for example:
\S\a(f00000#1000)b(20000000) would allocate 64K bytes of memory starting at fixed VME address 0x20f00000 in the VME extended space. The virtual address of this 64K chunk mapped into the calling process may vary from boot to boot, but the actual memory allocated will not change. Remember that the first four bytes read from the file returns the address in the caller's space to use for local access to the shared pool, and that the next four bytes is the base address from the VME perspective to use.

In order for this to succeed, you must edit the "Registry" entry for the DriveVME device as follows:

1. Launch the "Windows NT Diagnostics" program, choose under the "tools" option the "registry editor".

2. Under the "HKEY LOCAL MACHINE" entry, open "System" "Current Control Set" "Services" open "DriveVME".

3. Open the "Parameters" sub-entry. There you will see an entry 'SlaveRamBase' and 'SlaveRamLength'. Set these to the lowest address DriveVME can assume is both present and assignable as slave memory, in hex, but on or above the value used for the "MAXMEM" parameter discussed above. Set 'SlaveRamLength' for the total number of bytes starting at "SlaveRamBase" you wish to reserve for slave memory operations. DriveVME will claim these resources on startup, and will not start if another device previously claims that space. If you change these parameters, you must reboot before the changes will be noted.

Once accepted, you may specify ranges between SlaveRamBase through SlaveRamBase+SlaveRamLength-1 as the first parameter in the a(xxx#yyy) term, which will then give an address in VME space which does not change each boot.

Note: The value "SlaveRamBase+SlaveRamLength" is used to set the highest possible address that might be offered as slave memory for this system. The default is SlaveRamBase=32Meg, SlaveRamLength=0. It is important that you set this parameter accurately to within a power of 2 for the amount of memory in your system. So if your system is 16MB, 8MB<=SlaveRamBase+SlaveRamLength<=16MB. If this parameter is set higher than need be, your system may deny requests for particular VME addresses actually available

For example, if the VME address you desire starts at 16Meg in the extended space, and due to un-updated SlaveRamBase values the system thinks you have 32Meg of memory, but you only really have 12, it will only accept b(xxx) parameters that are multiples of 32Meg on some CPU's. Once you update SlaveRamBase, the system will allow slave mappings to occur every 16Meg, permitting the use of the desired address.

Note: Remember to read the address information bytes regarding the slave configuration in a single read, not multiple smaller reads.

Slave Mode DRIVE/VME Open File Parameter Reference

<VME space>

This selects which of the various VME memory spaces will be accessed. <space> may be one of: sI, sS or sE.

sI - Short I/O Space (A16). Address range 0- 0xffff (not often supported)

sS - Standard Memory Space (A24). Address range 0 - 0xffffff

sE - Extended Memory Space (A32). Address range 0 - 0xffffffff (Default)

s6 – VME Extensions A64 Memory Space. Address range 0 - 0xffffffffffffffff

sC – CR/CSR Space. Special register sharing version in A24. 512K bytes/image. Address range 0 - 0xffffff (Nearly always only used for mapping VME bridge device interface registers to the VME bus.) NOTE: This expects VME address setting of a(#80000) which will always choose as the 0..31 <<19 A24 value the value stored in the CBAR / CR-CSR base address register at system powerup time. Specifying a VME address is possible (e.g. …a(100000#80000)…), that will set CBAR.

sU(xx) - User set VME address modifier, not available on all CPU boards. Sets the read/write cycle and access type (AM code) to the hex code in place of 'xx', above. Use sU{xx} for decimal xx. Some interfaces do not support this.

If no <space> entry is specified, the default is the extended (4Gb) address space (sE).

<Blanket Slave Interface Disable>

One of: X, K. The letter 'X' blocks all slave interface open attempts of any sort. The letter 'K' blocks any user mode process from offering slave memory, but not kernel mode processes. These calls can only succeed when there are no otherwise prohibited existing slave mappings in effect. If the X option is included, no other parameters have meaning or need be specified. A request to offer memory can proceed AFTER one with the K option, but only from a kernel mode process. Neither of these is selected by default. Any parameters specified with the K option are ignored, and no space is offered.

<Slave Response cycle types>

This describes the privilege level setting to use for these VME master mode read cycles. Some VME hardware and other resources require supervisory mode cycles, but most use non-privileged mode cycles. <Slave cycle> may be one of:

cN - Respond to Non-Privileged cycles and Supervisory cycles (default)

cP - Respond to Program and Data access cycles (default)

cD - Respond to Data access cycles, but not program cycles.

cG - Respond to Program access cycles, but not Data cycles.

cS - Respond to Supervisory cycles, but not Non-Privileged cycles.

cV - Respond to Non-Privileged cycles, but not Supervisory cycles.

New in version 4: All modes respond to Single cycle transactions. In addition:

cB – Respond to BLT cycles
cM – Respond to MBLT cycles
cE – Respond to 2eVME cycles
cT – Respond to 2eSST cycles
cR – Respond to 2eSST Broadcast cycles (Board ID #1..21 set at boot time in the registry, DriveVME DWORD parameter RXBROADCASTID. Missing or 0 means no 2eSST broadcasts will be received. Configure using AssistVME's 'Syscon' general controls setup page. Changes only take effect after a reboot.)

c1 – only for cT and cR: Respond only when speeds are 160MB/s or less.
c2 – only for cT and cR: Respond only when speeds are 267MB/s or less.
c3 – only for cT and cR: Respond only when speeds are 320MB/s or less.

<Slave Transfer Priority Options>

L0 - When slave transfers are being done via File I/O to local memory, this command (the default) will execute the transfer at the thread's normal execution priority and without first locking the VME bus.
L1 - This will raise the processor priority above user threads lock the VME bus, execute the transfer, flush the processor caches, then release the VME bus.
L2 - Same as L1 but the transfer happens at a priority above all VME interrupts.
L3 - Same as L2 but the transfer happens above all but multiprocessor, clock and power fail interrupts.
L4 - Same as L3 but accepting no interrupts.

<Slave Bus Options>

uP - VME writes are "posted" - they are marked complete before the local write completes, for performance. This is ignored for I/O space writes

uC - VME writes are "coupled" - they do not complete until the associated local write completes, for interlocking. This is the default.

uR - VME reads are anticipated and prefetched on the local bus, to speed up sequential VME reads. This is ignored on I/O space reads.

uI - VME reads occur on the local bus one for one, for interlocking. This is the default.

uL - VME Read-Modify-Write cycles lock the local bus during the VME RMW transfer, for full atomic operations throughout. This is not the default as every non-block transfer is slowed by this choice, RMW or not.

uN - VME Read-Modify-Write cycles are broken into non-atomic back to back read/writes on the local bus. This is faster, and should be used for all slave regions not involved in RMW operations. It is the default.

> Note: The standard driver assumes the local bus is a 32 bit bus. If a 64 bit local bus is implemented (e.g. PCI64) Contact N4 for a PCI64 enabled DriveVME driver.

You may specify more than one parameter, like: uPuR for posted writes and prefetched reads.

<PC local memory address range>

This describes the range of local, physical addresses, to be offered under the named <space> and so on, to be made available to the VME bus. Some local resources have local addresses and ranges that are fixed by hardware design. These follow. Later the methods for specifying stretches of local memory are defined. The predefined resources are:

M --
this causes the broadcast location monitor detectors to be mapped beginning at the chosen VME space and address. This is a method for many systems to receive an interrupt at the same moment caused by one other system, or to detect offboard activity at a particular VME address. There are usually 8 bytes per location monitor of which only the lsb is actually watched, and there are usually 4 monitors. Any VME traffic at all to any board on the chassis that includes the address of a location monitor will, if the associated interrupt is enabled via other mechanisms, cause an interrupt to occur on the system with this file open whether it was the target of the VME cycle or not. The system will reserve the minimum space, 4K, though the actual range monitored is much less than that. Do not read or write to this file once opened, its purpose is only to establish VME location monitor addresses, active only while the file is held open. To generate location monitor interrupts, generate VME write cycles to the monitored addresses via other mechanisms. As the location monitors decode VME space, the 4K range starting with the monitor address cannot be otherwise used. Systems monitoring a location will acknowledge any master cycle initiated for that space. Do not overlap the same space with a local memory based share. *Note: there are usually VME device register specific registers that allow for checking location monitor flags / status if using the interrupt system is problematic. See <VME bridge chip name>check .c and .h for details. Use the interrupt facility if board version independence is important.

F –
this causes the entire VME bridge device control interface register set to be offered to the VME bus. If the address space chosen is the CR/CSR space then the system will allocate 512K. Most bridge devices will decode the first 4K to the device's interface registers, while the further 508K will decode to local memory. Other VME spaces will decode only the registers – the first 4K only. Some interfaces allow one CR/CSR map and a concurrent single other map to another VME space. Local system access to this range will appear to offer only memory, bridge interface register access is not decoded on the PCI side via this mechanism. This feature should be used with great caution as contention for register settings between VME masters and DriveVME on the local system is a possibility. DriveVME resets all registers upon loading and unloading, then only as necessary during operations.

G --
New in version 4. This causes the global control and status registers (semaphores, mailboxes, and other general status and control information, generally a subset of the entire bridge device interface range designed for inter-board communications), to be offered starting at the named VME space. Due to OS granularity limitations, the system will show a 4K reservation, however the VME bus typically only decodes the first 32 bytes. Local accesses to this space will not decode the registers but only 4K of local memory, use the better other mechanisms provided for direct individual multi-process safe local access to these registers. For user space examples see the 'Kernel C' program <bridge device id>Check.c. Kernel device drivers can access the register map directly, see dvmeddk for details. The GSCR space is a subset of the entire register range. It allows for commonly used shared functions to be isolated and offered separately from the entire bridge device register set. Consult the manual for your bridge device to determine the layout and functions of this register block. Note that mailbox interrupts require both the mailbox locations to be offered (this function or F above), and the related mailbox interrupt to be enabled. Otherwise the status bits will operate but no interrupts will occur.

To offer local physical memory or IO spaces, use one of the letters a, r or i followed by a two numbers as follows:

a([<Start offset>]<# for count or - for end offset><Count or end offset>) -- this offers local memory addresses.

r([<Start offset>]<# for count or - for end offset><Count or end offset>) -- this offers local memory addresses specified plus the value of SlaveRamBase (the beginning of an optional large area reserved for DriveVME slave memory)

i([<Start offset>]<# for count or - for end offset><Count or end offset>) -- this offers local I/O addresses. This makes it possible for VME masters to directly control I/O devices on the local processor bus. (Not all VME interfaces support this. The Universe chip does support this, the TSI148 Tempe does not).

For example: a(1ba-1bd) specifies the desired range to be from offset 0x1ba, through 0x1bd. Using the - as a separator implies a starting/ending address combination. Alternately using a # as a separator specifies a starting address - number of bytes following combination. So, a(1ba#5) specifies offset 1ba, for 5 bytes.

If the letter "a" above is replaced with "r", i.e. r(0-10), then the value of "SlaveRamBase" is added to the range. In this way, the user may specify a

Drive/VME slave address to start at some known position each boot, without having to know for each processor where the space reserved for slave memory begins.

Unless it is absolutely necessary, do not specify a physical absolute starting address to offer. Let the system choose one. Specifying a fixed starting memory address forces many added compatibility checks which are otherwise avoidable, these can cause the request to be denied.

1. Only kernel mode drivers may offer slave access to the local I/O spaces.

2. Only kernel mode drivers may specify absolute physical memory addresses below SlaveRamBase which are not already set aside for slave operations with acceptable permissions.

3. A request to offer slave memory will yield a particular range of physical addresses, which can then be explicitly specified in another request to offer the same space to another VME range. The second and further requests to slave a particular space may succeed if the security parameters on the first successful request permit it. Usually this can make memory sharable between a kernel mode driver, the VME bus, and one or more user mode programs all at once-- so long as the kernel mode driver initiates the share request. That is, the request would have been denied had a user mode process first explicitly asked for a particular range of physical addresses below SlaveRamBase.

4. It is permitted for two processes to request to offer the same local memory to the same VME space. In this case, both (without regard to whether they are user or kernel mode) will have local process access to the same stretch of local memory, as well as offering that memory to VME.

5. If memory is set aside using the SlaveRamBase/SlaveRamLength registry keys, and the /Maxmem= boot.ini parameter, then the system will attempt to allocate unspecified start-address requests for slave memory from the top of this space downward, avoiding all previously allocated regions. If there is no such space, or none remains free, the system will attempt to

allocate memory from the non-paged Windows/NT memory pool.

6. The mechanisms for exposing local memory to VME users requires hardware resources which typically have one or more basic limitations:

1. The number of modes in which memory may be shared is usually limited to somewhere between 1 and 4, and is hardware limited. The first request which uses the hardware sets the mode for the following requests. Ending sharing frees the resource for other use.

2. The minimum amount of memory which actually gets shared is somewhere between 4K (A16 space) to 64K (all others), but could be as high as 16MB on some hardware. However, all memory reported by Drive/VME as satisfying a sharing request is guaranteed to be fixed, non-pageable memory, though the sharing process on non-Universe chip VME access designs may also expose other memory as a side-effect.

3. The exact same VME space can't be exposed to two different local physical addresses. But, access to the local physical addresses shared on VME may be shared among different processes. And, the same local physical addresses may be appear at the same moment as different VME spaces.

The range of VME addresses offered can only be offered in chunks of 4K for the VME A16 space, and 64K for all other spaces. Requests for less than this succeed, but memory is actually allocated for the entire size of the minimum chunk with whatever alignment limitations exist, further requests for the same sort of space get tucked into gaps left by earlier requests. Some hardware only offers megabytes of memory to be shared only one way in the A24 or A32 space, and A16 capabilities. Note that this means a request to share 1 byte may consume 16MB of A32 space on some designs, 64K on others.

If no address range term is specified, then DriveVME assumes a(#1).

If no separator character is found, "#1" is assumed. So: a(20-20) = a(20#1) = a(20) = a{32}.

Note: using {} instead of () implies decimal instead of hex numbers, so a(10#10) is the same as a{16-32}.

Note: Some systems will not let you specify a particular physical address to share, or do not offer the ability to share all legal physical addresses. If your system does not let you specify a particular starting physical memory address, then you can let the system determine a legal physical address for your process/driver by only specifying an amount of memory desired to share as follows: a(#<count>) or a{#<count>} for a decimal count. An eight byte read of the open file will yield first the virtual address (legal address mapped into the virtual space of your software) of the shared space, followed by the actual starting physical VME (not local) address shared. For example: a(#100) would allocate and fix a range 256 bytes long somewhere in physical memory, which it will report in the file read.

> It is always safer and more portable to let the system choose the physical start address whenever possible. Some systems permit preassigned fixed slave addressing, see the Windows/NT note above.

<VME Address 0 offset>

NOTE: The meaning of this term has changed from pre-release versions.

z(xxxx), or z{dddd} for decimal, instructs the system to make place the first byte of the shared physical memory at the VME address given in this command in the VME space, or fail the request.

b(xxxx), or b{dddd} for decimal, to attempt the "z" interpretation above, or failing that, where R = number of bytes of physical local memory: z & ~(r-1) <= assigned VME base address <= (z+r) & ~(r-1)., or fail the request if that isn't possible. In prose, the system will try to pick the given address, and failing that pick an otherwise unused VME address somewhere between the next lower and next higher even multiple of the amount of system memory nearest the desired value. If you must specify a VME address in your request, this is by far the most compatible.

The default if this term is omitted is that this is system assigned. This is best for compatibility, as some designs can not support two processes with different values. (Note however, the use of the p(N) parameter below, which sets aside a large chunk of memory for later allocation into smaller parts). Drive/VME will attempt coalesce nearby requests for eqivalent VME modes into the use of one hardware VME slave memory access resource.

The local virtual address and actual VME address assigned can be read in an 8 byte chunk from the open file handle as the first read. If the first process to open a slave memory region does not specify this parameter, then the system will use the values established in the ROM BIOS setup for this processor, or if no such value is present, 0. If you are using Net/VME or some other device driver that uses slave memory before your process can load, set the desired VME slave offset using the ROM/BIOS, so the drivers that load before yours establish operations using a VME base offset address acceptable to your process.

Note: For some hardware, the acceptable values here depend on the amount of system memory you have (see the introductory text about SlaveRamBase and SlaveRamLength.) In particular, for portability, do not specify offsets with low order bits set beneath the next higher power of 2 of the highest legal physical memory address your system supports.

For example, for best compatibility, if you have a 32 MB system, be sure for b(xxxx) xxxx & (32MB-1) ==0.

Note: Note: to see what choices are in effect, list the "S" (for slave) subdirectory of the "V" (DriveVME) drive. All currently in-effect choices (currently open files) will be listed there.

<Slave byte swapping>

This describes how to handle data during the transfer.

t([<swap>], one of: t(S) -the default, or t(N).

<swap> is the letter 'S' or 'N'. If the letter 's' is appended, then the data bytes are swapped on multi-byte transfers t(s). For example, if 16 or 32 bit transfer cycles are received, and the quantities are each separate numbers, then no swapping need occur between big-endian (Motorola, most VME peripherals) and little endian (Intel) architectures, as the data is placed correctly on the bus. But if these are byte quantities (text) being read in multi-byte transfers, then byte swapping should be selected so that the string "fast" saved in a big-endian way does not come out as "tsaf" on a little-endian machine. The swap is for the full width of the transfer, so for 16 bit transfers bytes 1 and 2 come over 2,1, 32 bit transfers bytes 1,2,3,4 as 4,3,2,1, and so on. This can only be done on the hardware that supports it, no byte swapping software emulation is available. Use t(N) to cause data to be moved "straight through".

This whole issue of alignment and byte-swapping arises because the VME bus allows for processors which use different data storage schemes to operate together. As the bus can't know the content of the data, it is up to the software and each processor to correctly align and swap bytes as needed to preserve the meaning in the data transferred.

Note: Slave byte swapping is not supported by all slave VME interfaces (current versions of the Tundra/Newbridge Universe chip do not, for example).

<error handling>

This permits the caller to receive a windows message, or to have an event sent to the Drive/VME operations log, or both, whenever a write posted by VME request for local resources fails. Coupled requests are terminated with VME BERR cycles, but posted requests may fail after the VME process which initiated them complete. NOTE: Typically any writes which have been posted are flushed upon the detection of an error on the local side.

e(N) Causes no special action to occur in the event of a local bus error in reply to a VME request, other than returning the bus default value. This is typically -1. Note that many systems can only support this mode, which is the default. The AssistVME log will not be updated to report anything, even if a bus error is detected.

e(S) The AssistVME drive operations log will be updated with as much detail as the system can provide.

e(Mxxxx) Send windows message xxxx (always in decimal) to the calling process, with the local address which failed as the 32 bit LParam value. Not all platforms record accurately which VME request it was which generated the error, only that the error did occur in the context of handling a coupled local slave VME initiated posted write request. If the address causing the error is unknown or indeterminate, the resulting value is 0. The AssistVME drive operations log will be updated with as much details as the system can provide. Note that AssistVME must be running in its default "service" mode for this feature to have meaning. (AssistVME is always installed with Drive/VME, unless special action is taken to disable or stop it.)

<permissions>

This term determines which other threads and processes can also offer this range. The permissions allowed follow the letter 'p'. These permissions apply to VME spaces taken as a whole (io, standard, extended) without regard to cycle type (supervisory, non_privileged, user_am).

p(N) Allows all others any level of access to the given range, including exclusive ones. However, it does not offer the caller any access to the named range. This makes it possible to pre-declare a large range of VME addresses which are to be used in shared or even process/thread exclusive ways specified by calls yet to be made. This allocates local memory to support the entire request, then allows future requests with compatible options to be autoassigned into some or all of this range. This includes exclusive requests for non-overlapping sub-elements.

Whenever Drive/VME gets a request to open slave memory where the VME base and/or physical base are left for DriveVME to choose, DriveVME will first search all the available already allocated spaces for those with compatible choices, and try to set the discretionary values so as to be a part of the pre-existing request. Even when all parameters are specified but still any new request can be logically contained within another, it will be. This allows more calls for slave resource offering to succeed. Only when calls arrive that can't be merged due to address or access mode conflicts will a set of VME mapping registers be comsumed, if any are available.

The purpose of this is to make good use of the limited hardware resource which decodes the VME requests for local access. Pre-declaring larger ranges, which then can be sub-allocated to any number of particular threads, uses typically only one set of VME decoding hardware resources. Were this feature not employed, making several independent requests for nearby ranges would use one resource per request, quickly exhausting VME hardware mapping resources.

Note that this request can be made using the AssistVME program in advance of running other programs which offer slave memory. The request should be big enough to encompass all the VME-alike slave offering needs of the group of programs to be run. In this way, programs which failed due to lack of resources allocated separately can succeed.

The resources for the entire range will be released when the last open file referring to any part of the range is closed.

p(A) Allows all others non-exclusive access. (default if no p term specified). Has the effect of blocking future exclusivity requests.

p(P) Allows shared offering and access to this space and range by any other thread in this process, but no other processes. Note that if a kernel mode driver specifies this option, it will not block overlapping user mode requests-- however overlapping user mode requests of this sort from different processes will block each other-- So basically the first process claiming all or part of a space marked in this way by a kernel mode driver (which are typically run first) will deny access to the extent of the processes's claim from other processes. Kernel mode requests to regions marked this way by threads will succeed.

p(T) Allows this thread to have added access, but no other threads. Note that if a kernel mode driver specifies this option, it will not block overlapping user mode requests-- however overlapping user mode requests of this sort from different threads will block each other-- So basically the first thread claiming all or part of a space marked in this way by a kernel mode driver (which are typically run first) will deny access to the extent of the threads claim from other threads. Kernel mode requests to regions marked this way by threads will succeed.

p(K) Allows only kernel debuggers shared access. Can't be issued in user mode. Used by kernel mode device drivers layered atop Drive/VME. Will fail if a user-mode debugger holds part of the requested space.

p(D) Identifies this process as a debugger. This will allow access typically denied by p(P), p(T) requests if made by a user mode process, and will grant any access if this is a kernel mode process. Any ranges granted will not block any exclusivity requests by other processes, including other debuggers.

p(....F) Identifies this software as "finished" software, and in particular essentially disables the 'D' Debugger parameter of any user mode request to prevail against local register access exclusivity. So, an address selected with p(TF) would block all other threads, even debugger threads, from getting granted access to the named range, except: any kernel mode debugger will be granted access.

Note that p(AF) "permit all, finished code" will not block debuggers as a special case, permit all includes debuggers.

Also, consider how overlapping ranges work. For example, if thread 1 does a "permit thread" to for addresses 10-12, then another thread 1 "permit all" for ranges 1-100 would succeed. Other threads would not be able to access 10-12, but would be able to get "permit all" access to 1-9 and 13-100.

DriveVME User Mode Interrupt Handling Reference via File I/O

This chapter describes in great technical detail the topics largely automated by the related section in the AssistVME section. It is for those who seek greater technical understanding of the details and structure of DriveVME's user mode interrupt handling operations.

This chapter describes how to detect and handle interrupts using Drive/VME from typical "user mode" programs. Writers of kernel mode device drivers should use a higher speed method described in the DriveVME DDK. This method is somewhat slower, but works over networks on remote chassis as well as locally, and offers more flexibility. In particular, any program or method by which one can open and close files can be employed to respond to VME events of choice.

The ability to read and write open files adds flexibility of handling VME interrupts from high level "easier to write" user mode programs such as from database systems, spreadsheets, visual basic, etc. DriveVME's "kernel 'C' Java" interface also offers a full featured realtime operating system independant network enabled interrupt handling facility, including time stamping, queueing and other features when higher performance is needed. Drive/VME's Windows/NT .DLL interface also encapsulates this functionality for Windows/NT users familiar with .DLL style interfacing. See the .DLL "include file" DriveVME.H for programming details.

Note that Drive/VME prioritizes interrupt servicing in the order VME 7 first then in order to 1.

The choices regarding which interrupts to handle and what should occur when the desired interrupt occurs are specified by parameters encoded into the name of a typical file. Note: the "AssistVME" program automates the creation and testing of these file names in a user-friendly graphical way. The easiest approach is to use AssistVME to create the specifications desired, then "cut and paste" the filename created there into the application or program of your choice.

To handle VME interrupts, open a file on the Drive/VME pseudo-drive (V: on local systems, or via a network) using these file name parameters:

V:\H\ <VME Interrupt Level><Low Vector Limit><High Vector Limit><Issue Windows Message><Timeout Selection><Open Wait>

Some operating systems do not permit passing arbitrary and lengthy strings through the filename convention to devices. If the device file "DriveVME" is opened with no arguments (usually \\.\DriveVME or DRIVEVME:), the desired parameters can be established by passing them in the first write call to the newly opened file handle. Just make the entire string listed above from the root (with or without a trailing 0, but without the "drive" letter) the first write to the newly opened "DriveVME" device file. The two specification methods thereafter give identical results. (So: "\H\...options") If there are errors (permission conflicts, etc.), the open or write request will fail.

For example, opening the file "V:\H\I(1)L(10)H(20)M(30000)T(10000000)W" would cause windows message 30000 to be issued to the opening thread every time VME interrupt 1 happened but only if there was a bus error during the acknowledgement cycle, or the vector returned by the interrupting device during the acknowledgement cycle was not less than 10 and not more than 20. These messages would be sent so long as the file was open. Also, while the file was open, read requests to the open file handle would complete with a timeout indication one second after they were issued, or sooner if a device issued VME interrupt 1 with an acknowledgement vector between 10 and 20 decimal, inclusive. The format of the data returned is shown below. Also, due to the 'W' parameter, the file open request itself would not complete until the first interrupt event happened, and would complete showing an error if the event did not happen within one second (the T value) of attempting to open the file.

The interrupt handling process works as follows:

1. The user opens a DriveVME interrupt handling file choosing the parameters of interest. Interrupts are shared among all callers, there is no security needed.

2. Once the file is opened, if the Windows message option is selected and the chassis the file is opened on is the local chassis (not a remote one via a network), windows messages will be issued when the event occurs until the file handle is closed or the events stop. This operates in conjunction with AssistVME. If a higher interrupt processing rate is needed, try the kernel "C" routines from user mode programs or the DriveVME DDK kernel mode interrupt processing facility for highest performance by your own native NT kernel mode driver interfaced to DriveVME.

3. Once the file is opened, make read requests to detect the event(s). Using the overlapped I/O feature, many read requests may be pending, each event instance will trigger the completion of only one read event (the oldest). In applications where there is some question about whether the processing of an event will complete before the next occurs, using overlapped I/O to have two such events outstanding is a good idea. Since the information read includes a data about which event caused it, one might establish a routine that does nothing but keep a couple of reads per desired interrupt always pending, and queuing the completed structures. Should the timeout feature be used, read events may be completed due to timeouts instead of the actual event.

4. Should the event occur when a read request is available to complete, or a timeout occur, the information returned will be at most 20 bytes per read request. The data returned uses the 'C' structure:

```
struct DriveVME_Async_Event_Read {
  unsigned short status; //This value is ascii 'OK', with the letter 'O' in the low byte if the event
     //completed normally (whether a bus error or not), or 'TO' with a 'T' in the low byte
    //if completed due to a timeout.  The ascii is to make it easier for high level applications
     //to use this facility, a simple one byte character read will get the most important information.
  unsigned short event;   //This is the xx of the I(xx) parameter which opened this event file (16 bits)
  unsigned long number_of_events; //total number of times the event occurred without error
    //Note the above is incremented whether there was a read pending to record the event
    //or not.  If 100 events happened and only one read request was made just before the
    //100th event happened, the value returned here would be 100, not 1.
  unsigned long information; //the last acknowledgment IACK value, -1 if bus error during iack.
  __int64 timestamp;  //64 bits, the loc al system time in 100ns units since bootup when the
   // last event happened.
};
```

Later versions of DriveVME may complete each read request with more information, **but it is never an error to request only the amount of the read structure desired, even if it is less than the total available**. Although the file offset is ignored, good practice (and any network use requires) that the file offset for the read to be 0 for each request.

> Therefore, it is acceptable for applications to read just one byte per request, which will complete and return the letter 'O' after the event happened, or will return the letter 'T' if a timeout happened before the event did.

Note also that setting the timeout value to be 0 has a useful effect when all that is needed is a count of interrupts of a particular sort. The call will return immediately with a timeout, but will fill in the total event count information. This is a very low overhead interrupt handling mechanism.

5. When the file is closed (either by call or by program termination), the interrupt handling of the event will end, any pending read requests will be terminated with no data transferred, and windows messages upon event notice will stop.

6. The system will continue to generate IACK cycles for any interrupt level for which there is at least one outstanding request for processing. So only after no applications seek interrupt service on a particular level will the processor stop generating IACK cycles in response to interrupt requests on that level.

7. ***NOTE:*** These files should be opened with the flag "FILE_FLAG_WRITE_THROUGH" or other necessary indication to cause all requests for data to go to the physical device and not be cached by the operating system. See the supplied testdrivevme.c routines for examples of how to use the CreateFile function with this flag. There is also a programming example there of how to translate this sort of open file into the form used by "C" functions "read write open" and also stream functions "fread" "fwrite" and "fscanf". Programs not capable of flushing file buffers between calls should consider using the 'W' or 'wait' parameter, discussed below. This parameter will not permit the file open

request to complete until the desired event of interest has occured or a timeout has happened. Because every application on the operating system is capable of opening files, even the highest level programs can suspend operations until an interrupt of choice happens and detect an error if it doesn't happen within a given amount of time.

Parameter Specifications:

<VME Interrupt Level>

This selects the VME interrupt level to be handled. One file handle can handle only one interrupt level. All VME interrupts are shared, and handled in the priority of highest numerically first, when the hardware does not supply this VME prioritization DriveVME implements it in software. These values are one of I(0) I(1) ... Where:

I(1) I(2) .. I(7) - VME Interrupt Levels 1 through 7

I(8) - Power failure. Called as soon as a power failure is detected. The application should immediately save any critical information then exit. Not all chassis detect this or provide for a safe power-down interval.

I(9) - Abort Switch Press. Some processors provide an external button or switch, this level will detect when that has been pushed or pressed. Note that there is limited "debounce" support in some cases, so many interrupts may occur within a fraction of a second and correspond to a single press.

I(10) - VME Bus Error. Some processors generate an interrupt any time a VME cycle initiated by the processor (NOT initiated by other bus masters) ends in a bus error. Other VME interfaces do so only in certain modes (like posted instead of coupled transfers). This interrupt is generated by those processors on each such occasion.

I(11) - VME SysFail signal. This interrupt is called when any device on a chassis asserts the sysfail signal. Note that some VME interfaces can not clear this assertion except by hardware reset. This interrupt is issued upon the first such detection and thereafter about once per second until the signal is cleared. If the local hardware does not support software clearing this signal it is generated until power-down once per second.

I(12) - Software initiated local interrupt. This signal is generated internally by DriveVME, it has little practical value to others. Consult the Tundra/Newbridge manual for details.

I(13) - Interrupt Acknowledge Interrupt. This signal is generated (on some processors, check your manufacturer for details) whenever another VME resource issues an interrupt acknowledge cycle for ANY interrupt level issued but not yet serviced by the local chassis. DriveVME handles these interrupts internally as it sequences and in some cases serializes local interrupt generation. This signal is available as a trigger for user programs for completeness, however we do not see heavy practical application by most.

I(14) - Local Bus Error. This interrupt occurs whever a VME cycle can't be serviced by the local bus. This should never happen unless DriveVME's control of the VME interface is corrupted or there is a physical fault.

I(15) - DMA Complete. Drive/VME offers DMA oriented data transfers on interfaces that support it. This interrupt is issued whenever an ongoing transfer has completed.

I(16) - VME Bus Ownership. DriveVME's VME master interface can upon command lock the VME bus for exclusive use by this processor. The request to lock the bus may not be competed right away, this interrupt is generated when the bus is granted to the local system. As most locked bus transfers happen within the context of another DriveVME facility, we do not foresee much application for users by this facility.

I(17) - "No Drift" Clock interrupt. Drive/VME offers an interrupt on clock timeout based on a fine grained timer which can be set to trigger at fixed intervals and retrigger only at fixed intervals without regard to processing time or clock resetting. That is, the timer does not drift beyond the limits imposed by the driving system timer clock. This enables wakeup on those intervals. See the parameters for setting the wakeup interval.

This value has no default and must be supplied with every interrupt handling file opened.

<Low Vector Limit>

This parameter takes the form L(<integer>), like L(12). This parameter has meaning for those interrupts which supply information with the interrupt, and for the timer.

In the case of detecting VME interrupts (in conjunction with parameter I(1).. I(7)), should the interrupt vector supplied by the interrupting source during the IACK cycle issued by this processor be below the value in L(xx) then this handler will ignore the interrupt. (Remember, Drive/VME interrupts are shared, so other handlers may process the interrupt. These choices determine only what this handler will do and have no effect on the other handlers). Note that bus errors during the IACK cycle will trigger this interrupt, supplying a vector of -1, without regard to this value.

In the case of I(17), the clock, this value is an amount of time expressed in units of 100 nanoseconds for the timer interval. The value is stored internally as a 64 bit integer.

This value if omitted is presumed to be L(0).

<High Vector Limit>

This parameter takes the form H(<integer>), like H(255) or H(-1). This value if omitted is presumed to be L(255).

In the case of handling VME interrupts, (in conjunction with parameter I(1)..I(7)), this value describes the highest interrupt acknowledge vector which can trigger this handling routine. Should the IACK value supplied by the interrupting source be higher than this value, the routine will not be triggered. In the event there is a bus error while reading the interrupt vector, the routine will be triggered with an IACK value of -1.

In the case of I(17), the clock, should this value be zero or above, then the clock will act as a "one shot" clock, triggering the interrupt just once upon completion. Note that the "no drift" feature in this case does not have meaning because the interrupt will only happen once. In the case this value is -1, the clock will retrigger each count down, and will trigger an interrupt to occur at the next even interval measured from the base time measured from when the file was opened. Events will only be triggered at the time intervals established, should the processing of an the event timeout take longer than the interval (when uninterruptible code runs-- blocking the timer, say), when it eventually does finish the next wakeup will not occur until the next even multiple of the base time plus interval, skipping the missing ones. To restate: the timebase is absolute marked from the time the file is opened.

<Issue Windows Message>

If operations are occurring on the local chassis (this interrupt handling file is not opened over a network), then it is possible to cause Drive/VME to issue a windows message to the calling thread whenever the chosen interrupt event occurs. The interrupt acknowledge vector or other information is supplied as the "long" parameter. If there was a bus error then the "word" parameter will be 1 (and the "long" parameter -1), otherwise it will be zero.

The parameter is specified M(xxx), like M(32000). The windows message value is a decimal integer corresponding to the windows message you wish DriveVME to send to the calling process. This value is determined by the needs of each application, typically it is on or above the value WM_USER. Note: AssistVME must be running to provide this facility. This uses the same mechanism as the bus error interrupt alerts, see the file DvmeMFCDemo for source code and executable programming examples.

If this parameter is omitted, then no Windows message will be issued. This parameter will not generate windows messages on the local system when the interrupt file is opened to handle interrupts on a remote chassis via a network.

<Timeout Selection>

When file read request arrives to wait for the desired event, typically the request will not be satisfied until either the file handle is closed or the event occurs or 10 seconds elapse. Specifying this value of T(xxx) where xxx is a time in 100 ns units will cause any request pending for this file handle to be cancelled with a timeout data indication sometime shortly after this amount of time has expired starting when each request is issued. This option has no meaning or effect on the Windows Message feature. This value can be set to at most 10 seconds, and if omitted is defaulted to 10 seconds. If a value higher than 10 seconds is requested, the value is shortened without notice to 10 seconds. Programmers must always check to see whether the result of a read was due to a timeout or the desired event. The reason is that pending reads consume system resources, particularly over networks, and so potential indefinite pending reads are not good progamming practice. Note that the information transferred in a timeout is accurate as of the last recorded event. Also by checking the first 2 bytes of the returned information the user can determine whether the read event was the result of a timeout or an event occurance.

<OpenWait>

This feature is enabled by including the single letter 'W' in the file open string. It is by default disabled. When present, the request to open the event processing file will not complete until either the timeout specified has occured or at least one of the desired events itself has occured. If this parameter is present and no timeout parameter is present or if the timeout parameter is over 10 seconds, the system will insert a timeout parameter of ten seconds. If the event desired did not occur within the desired interval, the attempt to open the file will fail with an "access denied" error message. If the event occured, the file will open normally. Once open, the file will behave normally (however the default upper timeout limit of 10 seconds will still be in effect if no lower timeout limit was requested). This feature makes it possible to detect an event just by noting being able to open a file. If the file opens (even via a network) then the event happened. If the file doesn't open, then the event didn't happen.

DriveVME License Agreement

Note: DriveVME is sold only to VME processor manufacturers and embedded systems manufacturers, not to end users. If you wish to obtain DriveVME and your processor or embedded system manufacturer doesn't carry DriveVME, contact N4 Communcations at www.n4comm.com

N4 Communications DriveVME LICENSE AGREEMENT

This is a legal agreement between you, the licensee, and N4 Communications.
ANY SOFTWARE INTERNALLY MARKED "EVALUATION", SUCH AS THAT DOWNLOADED FROM THE INTERNET, IS NOT LICENSED FOR ANY PURPOSE OTHER THAN EVALUATION AND INITIAL DEVELOPMENT EFFORTS FOR A PERIOD OF 30 DAYS FROM THE DATE IT WAS DOWNLOADED. IN PARTICULAR ANY COMMERCIAL USE OTHER THAN FOR FUNCTIONAL EVALUATION, INCLUDING ALTERATION OF THE SOFTWARE TO DEFEAT NOTIFICATION OF THE EVALUATION STATUS OF THE SOFTWARE, IS UNLICENSED AND A IS A VIOLATION OF COPYRIGHT. NO WARRANTY OF ANY KIND IS OFFERED RELATING TO EVALUATION SOFTWARE.

REGARDING SOFTWARE FOR WHICH A FEE HAS BEEN PAID TO N4 COMMUNICATIONS: BY OPENING THIS SEALED DISK PACKAGE, OR ORDERING THIS SOFTWARE BY ELECTRONIC MEANS AFTER BEING NOTIFIED OF THIS AGREEMENT- YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, TERMINATE THE ELECTRONIC ORDERING PROCESS NOW OR PROMPTLY RETURN THE **UNOPENED** DISK PACKAGE AND THE ACCOMPANYING ITEMS (including written materials and binders or other containers) TO THE PLACE WHERE YOU OBTAINED THEM FOR A FULL REFUND.

Products:

DVMEDDK - Drive/VME Developers Kit.. Kit includes source code to all programming examples, source code to DriveVME.DLL. Complied programs AssistVME, Drive/VME runtime for one CPU, demo versions of other N4 products. License is the typical "development end user" type license where the license attaches to one CPU in the end-user's organization. If there are two developers, two DDK's must be purchased, etc. One DDK must be purchased per customer before any DVMERUN's may be licensed.

DVMERUN - Drive/VME Runtime. Kit includes Drive/VME runtime for one CPU, minimal installation software. The license attaches to the CPU the product is first installed on, and moves with the ownership of the CPU.

GRANT OF LICENSE. Development Kits: N4 Communications grants to you the right to use one copy of DVMEDDK in particular and any other N4 product whose order code ends in "DDK" (driver development kits) license purchased on a single terminal connected to a single computer (i.e. with a single CPU). You may not network the SOFTWARE or otherwise cause it to be loaded on more than one computer or computer terminal at the same time within the purchasing organiztion.

GRANT OF LICENSE. Runtime Versions: N4 grants the licensee of DVMERUN in particular and all other N4 software whose product code name ends in "RUN" the right to install it on a single processor and no other, but with the license automatically transferring to the any new owner of that particular processor.

COPYRIGHT. The SOFTWARE is owned by N4 Communications and is protected by United States copyright laws and international treaty provision. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g. a book or musical recording) except that you may either (a) make one copy of the SOFTWARE solely for backup or archival purposes, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may not copy the written materials.

OTHER RESTRICTIONS. You may not rent or lease the SOFTWARE, but you may transfer the SOFTWARE and written materials on a permanent basis provided you retain no copies and the recipient agrees to the terms of this Agreement. You may not reverse engineer, decompile or disassemble the SOFTWARE.

AUDITING RIGHTS. You agree to instruct the vendor of your VME or other processors and also you yourself provide N4 or its agents reasonable access during business days upon demand by N4 to facilities and records and to assist N4 and it's agents in the task of demonstrating compliance with this license agreement. You understand that in the event noncompliance with this agreement is discovered the purchaser and end users will be responsible for all agency fees, expenses, penalties and damages provided by law relating to the discovery and remedy of the noncompliance, which fees are far in excess of the license price.

DUAL MEDIA SOFTWARE. If the SOFTWARE package contains more than one form of electronic media (3-1/2" disk, CD-ROM, etc.) then you may use only the form appropriate for your single-user computer. You may not use the other form on another computer or loan, rent, lease, or transfer it to another user except as part of the permanent transfer (as provided above) of all SOFTWARE and written materials.

Transfer N4 may assign it's rights and obligations to any under this agreement to any party upon 30 days notice to any active Plan B or Plan C licensee.

LIMITED WARRANTY

LIMITED WARRANTY. N4 Communications warrants that the SOFTWARE will perform substantially in accordance with the performance demonstrated by the freely available evaluation versions of the products for a period of 90 days from the date of receipt. Any implied warranties on the SOFTWARE are limited to 90 days. Some states do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you. N4 provides extensive and freely available over the internet "demonstration" versions of all these products for review. N4 warrants that the delivered "resale" versions are the same as the freely available versions except for the removal of the "demo software" functionality. Given the ability for customers to review the product's performance in advance, N4 does not provide any further warranties of any kind regarding the functions or appropriateness of N4 for any particular purpose. Prospective Licensees and Licensees of older versions are encouraged to determine the suitability of N4's products for their purposes in advance of license purchase and deployment.

CUSTOMER REMEDIES. N4 Communications's entire liability and your exclusive remedy shall be, at N4 Communications's option, either (a) return of the price paid, or (b) repair or replacement of the SOFTWARE that does not meet N4 Communications's Limited Warranty and which is returned to the place of purchase. This Limited Warranty is void if failure of the SOFTWARE or hardware has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warranty period or 30 days, whichever is longer.

NO OTHER WARRANTIES. N4 Communications AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE ACCOMPANYING WRITTEN MATERIALS, AND ANY ACCOMPANYING HARDWARE. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE TO STATE.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. IN NO EVENT SHALL N4 Communications OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING WITHOUT LIMITATION DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR THE INABILITY TO USE THIS N4 Communications PRODUCT EVEN IF N4 Communications HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

U.S. GOVERNMENT RESTRICTED RIGHTS

The SOFTWARE and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Contractor/manufacturer is N4 Communications / N4 Communications / 2118 Lundy Lane / Bettendorf, Iowa 52722.

This Agreement is governed by the laws of the State of Iowa.

Should you have any questions concerning this Agreement, or if you wish to contact N4 Communications for any reason, please write or call: N4 Communications / 2118 Lundy Lane / Bettendorf, Iowa 52722 / Tel : (563) 650-7800 / Fax: (563) 332-9633