VAX MP
A multiprocessor VAX simulator

OpenVMS user manual

Sergey Oboguev
oboguev@yahoo.com
September 12, 2012

Legal notice A

VAX MP is a derived work developed initially by Sergey Oboguev, based on SIMH project by Robert Supnik. As such it is covered by legal terms consisting of two parts, one covering the original SIMH codebase, and the other part covering changes to SIMH for VAX MP.

Legal terms for the original SIMH are reproduced below, as found in SIMH files:

> Copyright (c) 1993-2008, Robert M Supnik
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL ROBERT M SUPNIK BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
>
> Except as contained in this notice, the name of Robert M Supnik shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from Robert M Supnik.

Legal terms for VAX MP specific changes to SIMH codebase are similar in essence:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OR LIABILITY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL SERGEY OBOGUEV BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Legal information notice B

PLEASE DO READ CAREFULLY

VAX MP is a simulator that provides virtual multiprocessor capability and allows execution of VAX/VMS (OpenVMS VAX) operating system in symmetric multiprocessor (SMP) mode.

OpenVMS VAX is a legacy operating system and, according to the present writer's best knowledge, all VAX-related products had been classified by Hewlett-Packard as obsolete as of April 2011, and regular sales of these products had been discontinued by HP at that time.

At the time VAX/VMS (OpenVMS VAX) was sold by DEC, Compaq and HP, exact licensing terms for OpenVMS VAX SMP systems varied depending on particular CPU model. All systems required OpenVMS VAX base license (VAX-VMS). Some systems in addition required separate SMP extension license (BASE-VMS-250136) per each additional processor in an SMP system beyond first processor. Other VAX SMP systems entirely lack SMP licenses, i.e. the base VAX-VMS license would license the platform regardless of the number of the processors.

VAX MP does not constrain the user on a purely technical level from executing OpenVMS in symmetric multiprocessor mode, whether proper license is installed or not. Likewise, OpenVMS when executed on VAX MP does not constrain the user on a purely technical level from executing the system in SMP mode, logging into it and using it.

Regardless of the absence of technical constraints, legality of using OpenVMS in SMP mode is altogether another issue.

DEC/Compaq/HP have provided easily obtainable hobbyist licenses for most of OpenVMS VAX and OpenVMS Alpha products. Unfortunately, OpenVMS VAX SMP license was not among the licenses released for hobbyist use. Whether it was the intention of OpenVMS owners to release SMP capability for free hobbyist use on legacy systems is arguable: it was done on OpenVMS Alpha version, where the technical names of the base VMS license and SMP extension license coincide (OPENVMS-ALPHA) and SMP extension license simply adds units required to execute VMS on a system with multiple processors. OpenVMS Alpha hobbyist license comes with zero units, meaning unlimited number of units thus covering any Alpha configuration with any number of processors. OpenVMS VAX base hobbyist license (VAX-VMS) comes with zero units too, however in the VAX case technical name for the SMP extension license (BASE-VMS-250136) is different from the base license, therefore unlimited number of units on the hobbyist base license, strictly speaking, might be considered as not covering SMP use. While one might attempt to speculate on the ground of above (and also given that VAX is even older legacy system than Alpha) whether DEC/Compaq/HP did intend to permit unlimited hobbyist use of OpenVMS on any legacy configuration and cover hobbyist use of OpenVMS VAX on system of any capacity, just the same as for OpenVMS Alpha, there is a lack of clear-cut statement by OpenVMS owners on this matter.

Termination of sales of VAX-related products by HP also means that new BASE-VMS-250136 licenses are probably not easily obtainable from HP anymore on a commercial basis, at least not through regular sales channels. To complicate the matters, VAX MP simulates multiprocessor version of MicroVAX 3900 (or VAXserver 3900) – a machine that never historically existed in a multiprocessor configuration, and hence SMP licenses for which never existed too.

One may hope HP might eventually clarify the situation and either release OpenVMS VAX SMP licenses for hobbyist use, for VAX legacy systems too, just as it was done for Alpha, or simply state that hobbyist SMP use does not require a license separate from the base license.

In the meanwhile it might be possible to acquire VMS SMP extension licenses from resellers (either on eBay or otherwise findable online or any other way) for multiprocessor VAX systems comparable in capacity to MicroVAX 3900 or higher. While, on a purely technical level, these licenses do not need to be installed for execution of OpenVMS on VAX MP in SMP mode, they may be required to disequivocate the situation from the legal standpoint.

Please note that VAX MP is a purely technical project, and its developer or developers are unable to provide you with any legal counsel or guidance. This note is provided for informational purposes only and reflects it's author's best knowledge at the time of writing (not necessarily correct).

Please do note that **all responsibility for your use of VAX MP, including all legal responsibility in all aspects, including VMS SMP licensing aspect, is completely yours, and in no event VAX MP developer or developers assume any responsibility or liability of any kind or shall be liable for your use of VAX MP.**

Introduction

VAX MP is a variant of popular SIMH simulator that extends original SIMH to simulate multiprocessor VAX system. It allows execution of OpenVMS operating system in symmetric multiprocessor (SMP) mode on host computers that have multiple processors, such as modern multicore and hyperthreaded PCs.

VAX MP allows user to run virtual VAX systems configured with up to 32 virtual VAX processors, limited by the number of logical CPUs on the host system.

VAX MP targets primarily hobbyist use of OpenVMS. While the intent of VAX MP is to provide reasonably stable SMP VAX system simulation environment for hobbyist purposes, it cannot possibly have the same level of quality assurance or support as a commercial product. Therefore VAX MP is not recommended for production use.

If you are looking for a production solution, please examine other options, such as VAX simulator from Migration Specialties or Charon-VAX/nuVAX product ranges by Stromasys. In addition to offering support and more thorough testing, these commercial solutions offer better per-processor performance as well, since they perform on-the-fly binary JIT compilation of VAX code into host processor code[1], which neither SIMH nor VAX MP as its derivative do not perform. If you are looking for multiprocessor solution for production use, presumably your motive is increased performance, and that is where JIT simulators significantly surpass interpreting simulators such as SIMH in general and VAX MP specifically.[2]

Some of the mentioned commercial products also offer increased memory size beyond virtual MicroVAX 3900 limit of 512 MB (for example Charon-VAX 66x0 simulator offers up to 3 GB), making them altogether more suitable for production use.

VAX MP simulates a multiprocessor variant of MicroVAX 3900 or VAXserver 3900 – a machine that never historically existed. It does not simulate any historically existing VAX multiprocessor. The purpose of VAX MP is not a simulation of specific historically existing multiprocessor hardware, but of general VAX multiprocessing architecture, and execution of OpenVMS in multiprocessor mode.

VAX MP is composed of two essential pieces. One part is modified version of SIMH VAX simulator executable that supports parallel execution of multiple virtual processors. The other part is paravirtualization module (called VSMP) that is installed inside OpenVMS. During OpenVMS boot process or, at user's discretion, at any time after boot, VSMP is loaded into OpenVMS kernel and exposes VAX MP multiprocessing capability to OpenVMS.

When OpenVMS boots on VAX MP, OpenVMS initially recognizes the machine as uniprocessor MicroVAX 3900. However OpenVMS SYSGEN parameter MULTIPROCESSING is used to force-load multiprocessing version of OpenVMS kernel regardless of being started on a virtual hardware that is initially recognized

[1] Or, in nuVAX case, hardware execution or emulation of VAX code.

[2] As ballpark figures, whereas VAX MP executes on Intel 3+ GHz based PC at about 35-45 VUPS per host processor, commercial version of Charon-VAX simulating single-processor VAX executes (according to the manufacturer) at 125 VUPS per processor.

at boot time as uniprocessor. After successful bootstrap, VSMP load command is executed (this can be done either manually from DCL prompt or from SYSTARTUP_VMS.COM). Once VSMP is loaded, it reconfigures the system to multiprocessor by creating additional CPU descriptors inside VMS kernel and exposing VAX MP multiprocessing capability to VMS. After this point, system can be operated in regular way just as any VMS multiprocessor. CPUs created by VSMP are initially in inactive state, so the first step would be start them with DCL command START /CPU.

Maximum number of configurable VAX MP virtual CPUs is limited by the number of host logical CPUs, including hyperthreaded/SMT units. For example, commodity quad-core i7 based host machine allows to configure and use up to 8 VAX MP virtual CPUs. It is not possible to run more VCPUs than the number of LCPUs available on the host machine.

Maximum number of configurable VAX MP virtual CPUs is also limited by VMS maximum supported limit of 32 processors. VAX MP had been successfully tested in 32-processor configuration on host machine (a node in Amazon EC2 cloud) with two 8-core Xeon sockets, for a total of 16 cores, each multithreaded, yielding a total of 32 LCPUs, which 32 VAX VCPUs are mapped to.

Guest OS requirements

Current release of VAX MP supports multiprocessing only for OpenVMS 7.3. Some earlier versions of OpenVMS may run fine, but this had not been tested.

VAX MP dependence on OpenVMS version is mostly due to paravirtualization layer (VSMP) that is loaded into VMS as a kernel-resident module.

VSMP loads VAX MP proprietory code into VMS kernel. This kernel-resident VSMP module takes over VMS processor-specific CPULOA (SYSLOA650) routines.

VSMP LOAD command also dynamically modifies memory-resident copy of OpenVMS kernel and some drivers loaded into main memory by installing about 30 patches to the running, in-memory copy of VMS kernel and some VMS drivers. These patches are required to support multiprocessing.

VSMP does not change any files on OpenVMS system disk, it changes only in-memory copy of the loaded system. Thus, other than just installing VSMP on system disk in a separate directory, no modifications to OpenVMS system disk are required for VAX MP. OpenVMS system disk image remains intact and system files are not modified for VAX MP. Only in-memory copy of OpenVMS is patched by VSMP. Hereafter, these in-memory modifications are called “dynamic patches”, to oppose them to static patches of on-disk files that VAX MP does not perform.

There is a number of technical reasons for VSMP dynamic patches:

> Most of these patches are intended to address modern host machines’ weaker memory-consistency model compared to VAX memory model. In VAX days there were no out-of-order instruction issuing, speculative execution, write/read combining or collapsing, asynchronous multibank caches and other

high-performance designs that can reorder memory access and affect the order in which memory changes performed by one processor become visible on another. These differences do not matter as long as VAX code uses proper synchronization, such as spinlocks or interlocked instructions to communicate between parallel branches of the code, since VAX MP will issue appropriate memory barriers once VAX interlocked instruction is executed. However there are pieces of VMS code that do not issue appropriate memory barriers by executing interlocked instructions. Most importantly, these are PU and XQ drivers (drivers for UQSSP MSCP disk/tape controllers and Ethernet DEQNA/DELQA controllers) that communicate with their devices not only through CSRs but also by reading from and writing to shared control memory area – control area shared by the device and CPUs and read from and written to asynchronously by both the device and CPUs. PU and XQ driver do not (most of the time) issue memory barriers when accessing this memory area, relying instead on stronger memory-consistency model of historically existing VAX implementations that provide ordering of reads and writes to memory in the same order they were issued in VAX instruction stream. When executed on top modern host CPUs, such VAX code would not work properly and reliably since underlying memory access can be reordered by the host CPU at will, unless memory barriers are issued – and such barriers are missing in VMS PU and XQ code. Therefore VSMP installs patches in PU and XQ drivers to fix their memory accesses to the control memory regions shared by the devices and CPUs, by providing required memory barriers that force proper ordering of memory reads and writes.

Some other patches deal with the fact that VAX MP VCPU threads are idleable. On a real VAX, when VMS is in idle loop, CPU will spin in idle loop executing it over and over again, until it finds that some work to do had arrived. On VAX MP, VCPU threads do not have to spin in this situation, consuming host processor resources for nothing. When VMS enters idle loop on a given VAX MP VCPU, this VCPU's thread instead of spinning in the loop can go into hibernated state on host OS so as not to consume host processor resources while idle. Some of the patches installed by VSMP are intended to support this capability, including both going into idle sleep and then, on scheduler events, waking only the number of VCPUs actually required for pending processes in computable (COM) state. When VMS scheduler detects there are some processes in computable state, it will normally “kick” all of the idling CPUs off the idle loop and let them reschedule themselves. VSMP install patches that will wake up not all of the idling VCPUs but only actually required number.

One of the patches modifies XDELTA system debugger (if loaded) to let it properly handle multiprocessing on VAX 3900. XDELTA has CPU-specific code that gets current processor's CPU ID, and since it sees VAX MP as VAX 3900, it assumes the machine is uniprocessor and always uses CPU ID #0. The patch modifies this code in XDELTA to actually read from CPU ID register.

One of VSMP patches fixes bug in SYS$NUMTIM system service where system time location EXE$GQ_SYSTIME is accessed without properly acquiring HWCLK spinlock, which may result in incorrect value being read, including totally corrupt value. This can happen even on a real VAX, but probability of this corruption increases when VAX processor is simulated on present-day host CPUs with weaker memory-consistency model, compared to the original VAX. Patch installed by VSMP provides appropriate spinlock acquisition by SYS$NUMTIM when reading system time variable.

VSMP patches binary code of the system, and this code is of course version-dependent.

If VSMP is running on a version of OpenVMS that differs too much from 7.3 and if because of this VSMP fails to find certain target patch area and install required patch, it refuses to load. VSMP LOAD command

provides override option NOPATCH (described further in this manual) to disable loading of specific patches by their ID, but disabling patches is strongly discouraged – patches are there for a reason and disabling patches may result in unreliable system prone to crashes and/or data corruption or under-performant system.

VAX MP was designed and implemented with provisions to adapt semi-automatically across different versions of OpenVMS. In particular VSMP dynamic patches facility does not use hardwired design for patches, e.g. does not assume exact hardwired offsets and addresses, it uses instead more intelligent soft-wired design that searches for target VMS code patterns within approximate and relocatable regions of system address space where the patches are to be applied. VSMP dynamic patch facility also allows for reasonable changes in field offsets of system data structures addressed by the patched code. Thus, in theory, VSMP dynamic patch facility should be able to automatically adjust itself across OpenVMS versions as long as the changes to target VMS modules code had been moderate and not too drastic compared to 7.3 (7.3. is the reference because VSMP patches were designed based on OpenVMS 7.3). However large inter-version differences in OpenVMS modules patched by VSMP will necessitate adjustments to VSMP itself to accommodate that earlier VMS version or versions.

As of time writing this, no attempt had been made to test VSMP with versions of VMS other than 7.3. While there is a chance VSMP may run properly on versions of VMS close to 7.3, this had not been tested and is not assured. Running with even earlier versions of VMS would very likely require adjustments to VSMP source code, specifically the code installing required dynamic patches.

If you try to execute VSMP LOAD command and see error messages about certain specific “patch areas not found”, this means current release of VSMP is unable to execute on this version of VMS and has to be adjusted manually to accommodate this VMS version, either by introducing new code pattern or modifying target address range for pattern lookup.

Trying to run VSMP with versions of VMS other than 7.3 therefore is not recommended and is not currently supported in any way.

OpenVMS service patches that replace or modify VMS executive images may also potentially break VSMP dynamic patching facility and cause it to refuse loading VSMP on OpenVMS installation with executive images modified by OpenVMS service patches if those modifications had been too large for VSMP to be able to accommodate them automatically.

However VSMP had been verified and found compatible with the following most important VMS 7.3 service patches:

VAXSYS01_073
VAXMOUN01_073

VAX MP (VSMP) is compatible with these VMS service patches and can successfully execute OpenVMS 7.3 system with those service patches applied.[1]

VAX MP will also execute in uniprocessor mode any operating system that supports MicroVAX 3900 or VAXserver 3900 machines, including all versions of VMS (VAX/VMS, OpenVMS), Ultrix, NetBSD or OpenBSD that support MicroVAX 3900 or VAXserver 3900. These operating systems will recognize VAX MP as regular uniprocessor MicroVAX 3900 or VAXserver 3900 and execute in uniprocessor mode. The only operating system that will execute on VAX MP in multiprocessor mode at this time is OpenVMS 7.3 and possibly other versions of OpenVMS that VSMP may be able to cope with.

Host hardware requirements

Current release of VAX MP runs on Intel x86 and x64 processors. Total number of VAX virtual processors that can be configured in VAX MP installation cannot exceed the number of active logical processors on the host system.

Host logical processor is defined as available host execution unit, counting multiple cores and hyperthreaded/SMT execution units. For example, quad-core hyperthreaded processor has total of 8 host logical CPUs (LCPUs). Six-core processor with hyperthreading disabled will have 6 LCPUs, and so on.

> Note that performance of hyperthreaded x86/x64 processors is not 2x but typically only around 1.2-1.3x the performance of non-hyperthreaded core. For example, combined maximum performance of quad-core hyperthreaded i7 processor is not 8x of single-thread performance but rather closer to 5x. Real-world application and system performance will be further affected by workload-specific scalability characteristics.

VAX MP implements VAX virtual processors as host system threads executing in parallel.

Never configure VAX MP with number of virtual processors exceeding the number of LCPUs available on the host system. Doing so can (and likely will) cause host system lock-up. OpenVMS kernel contains sections of code that sometimes demand simultaneous execution of *all* virtual processors on the system. If they cannot execute simultaneously, OpenVMS will hang with VCPU threads spinning waiting for each other with all host CPUs 100% busy but OpenVMS unable to make any forward progress. Even if it eventually clears past this stall, slow-down will be drastic. Furthermore, threads will spin at elevated thread priority. If the number of configured VCPUs exceeds the number of host LCPUs, this will bring host system to effective lock-up, with host GUI and shell interfaces becoming unresponsive, so it may be impossible to recover from this state other than by restarting host computer via hardware reset.

Similarly, if you configure multiple instances of VAX MP running simultaneously on the same host system in multiprocessor mode, make sure that *combined* number of configured VCPUs in *all* concurrently executing instances does not exceed the number of host LCPUs available – otherwise multiple instances

[1] The fact that VSMP built for baseline OpenVMS 7.3 is able to cope successfully with modified VMS executive images provided by patch VAXSYS01_073 validates general approach of VSMP dynamic patching facility.

can and likely will enter into a livelock approximately the same way as in the case of single instance, with the same consequences.

> **Do not overcommit your system by configuring more virtual CPUs in all VAX MP concurrently running instances combined than the number of host logical CPUs available (counting host hyperthreaded units).**
>
> **Doing do will cause your host system to lock up and will require hardware reset of the host machine.**

Furthermore, depending on host operating system being used, as a practical matter we recommend to configure a *smaller* number of VAX VCPUs than the number of LCPUs available on the host system, and set a fraction of host computational capacity aside. This reserved capacity can be used both by VAX MP internal threads other than VCPU threads (such as VAX MP disk, tape and network device IO threads, virtual clock interrupt thread and console thread), by host operating systems services and other processes, and by host OS user interface, including window manager and terminal emulator windows. If VCPU threads had to compete with UI threads, this can cause system response such as terminal output to feel choppy.

The need to set LCPUs aside depends on the qualities of host OS scheduler. In our experience, Windows 7 and OS X 10.7.4 do not require any LCPUs to be set aside. With Linux 2.6.38 it is advisable to set one LCPU aside for best interactive responsiveness. With Windows XP, best interactive responsiveness is reached with two LCPUs set aside.

For example, host machine with Intel i7 quad-core hyperthreaded processor allows to configure an instance of VAX MP with up to 8 VAX virtual processors. When running Windows 7 or OS X as host operating system, all 8 can be used. When running Linux on the same host machine, it is better to configure only 7 VAX processors. If using Windows XP as host OS, it is more pragmatic to limit configuration to 6 VAX virtual processors, and to set remaining two LCPUs aside as a reserve for guest OS system functions and for VAX MP auxiliary (non-VCPU) threads. Such configuration would execute smoother and have better interactive responsiveness.

Similarly, while it is possible to configure two concurrent 4-processor VAX instances to run on the same i7 quad-core host machine under Linux, it is advisable to limit one of them to 3 VAX processors (thus totaling 7 VCPUs for two instances) and set the remaining LCPU aside for host functions.

Setting some minimum host capacity reserve aside may have also other potential benefit, especially when running VAX in production mode. In an unlikely case of OpenVMS deadlock due to some system failure, all VCPU threads may end up spinning at high host thread priority. If there were no host capacity set aside, it would be impossible for host system administrator to intervene short of hardware reset on the host machine.

VAX MP is runnable on cache-coherent SMP systems with no or very low NUMA factors. VAX/VMS and OpenVMS VAX assume flat uniform symmetric memory access for all the processors and do not

accommodate NUMA factors. Any significant NUMA factors would cause inefficient process-to-resource allocation within the guest operating system and, most importantly, jeopardize guest operating system stability; for this reason host systems with significant NUMA factors are not currently supported.[1] However it might be possible to run VAX MP on high-NUMA system as long as VAX MP is allocated to a partition with low intra-region NUMA factor.

Host software requirements

Current release of VAX MP runs on Windows, Linux and OS X. Refer to system-specific sub-sections below for information regarding particular host operating system. Note that this document does not attempt to substitute general SIMH documentation and does not seek to cover those VAX MP features and requirements that are not specific to VAX MP and are identical with general version of SIMH.

Some host operating systems allow both 32-bit and 64-bit build of VAX MP. Depending on OpenVMS workload, sometimes 32-bit build may be slightly more performant than 64-bit build and sometimes vice versa. On average, there is a preliminary feeling that 32-build when available is slightly more performant than 64-build, within few percent of difference, albeit more extensive metrics is required yet for proper comparison.

Running 32-bit build of VAX MP under 64-bit Windows is trivial.

Running 32-bit build of VAX MP under 64-bit Linux had not been attempted.

> It may be possible but would require installation and configuration of 32-bit (ia32) environment.
>
> Network-enabled version of VAX MP built with SHR option would also require 64-bit to 32-bit bridging library for `libpcap`. However if VAX MP is built with statically linked `libpcap` (build option NET), then bridging library is not needed and 32-build of VAX MP might possibly be able to run under 64-bit version of Linux. To reiterate, this have not been tried.

On OS X, VAX MP provides only 64-bit executable.

Host networking configuration

VAX MP provides the same networking support as regular SIMH, including uniprocessor SIMH VAX.

[1] Besides inefficient process-to-resource allocation high NUMA factors may distort the calibration of OpenVMS timing loops and cause OpenVMS to bugcheck because of SMP timeouts. The latter can be partially adjusted either by increasing the value of OpenVMS SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT or equivalent increase of SMT slow-down factor supplied via VAX MP SIMH console/script command CPU SMT. Timeouts in some of OpenVMS interlocked queue operations retry loops can be enlarged by increasing SYSGEN parameter LOCKRETRY. However many of OpenVMS interlocked queue retry operations use large but fixed retry count of 900,000 iterations that is not extensible by increase in LOCKRETRY.

VAX MP uses host network cards by putting them in promiscuous mode and capturing all the traffic on the wire, regardless of MAC destination address in the Ethernet packet. If there is a heavy traffic on the wire, processing all the packets (including packets not actually destined to this host or VAX MP instance running on it) can cause overhead. In this case it is recommended to connect VAX MP host to the network via a switch that filters out irrelevant traffic.

For instructions regarding host networking software configuration refer to system-specific sub-section below for particular host operating system. Note that in most cases you would want to configure network setup *after* performing basic OpenVMS installation first. However you want to consult system-specific section on network configuration *before* doing VAX MP installation to make sure you host system has `pcap` (`libpcap`) installed.

Installation of `pcap` is optional for execution of VAX MP, but is required for networking with VAX MP, i.e. in order to be able to run TCP/IP Services for OpenVMS, VAXcluster (VMScluster), LAT or DECnet over Ethernet for connecting either to other physical or virtual machines over the physical network or via virtual Ethernet to the host machine itself or to other instances of virtual machines running on the host.

Operation of system-specific network stack differs across various host operating systems, with significant implications for VAX MP networking configuration set-up.

Windows network stack implements full interchange of packets between host side and virtual side. This normally allows VAX MP user to set up very simple *single-interface* networking configuration with just one interface (one DELQA or DEQNA controller bound to host NIC[1]) used for communication between VAX and remote nodes and also between VAX and host node itself and also between VAX and other virtual machines running on the host. Under Windows, just one single interface normally suffices for all categories of communications. However even under Windows this is not always the case and more complex set up may be required in some cases: see notes further down in this chapter on TOE (TCP offload engine) adapters and Gigabit Ethernet jumbo frames.

Under Linux and OS X, `pcap` bound to host NIC allows VAX MP to communicate only with remote nodes, but not with the host computer itself. The reason is that while `pcap` does capture Ethernet packets arriving to the NIC from external nodes and forwards them to both VAX MP and the host machine, but it does not fully interchange NIC packets between the host machine and VAX MP.

> Packets sent by the host machine generally *are* visible by the virtual side. However packets sent by the virtual side generally are *not* received by the Linux or OS X host network stack since it is not listening in promiscuous mode – unlike on Windows where network stack does deliver transmitted packets to the whole stack when the interface is in promiscuous mode, such as after starting `pcap`.

Thus under Linux and OS X `pcap` bound to host machine's NIC allows VAX MP to communicate with other machines on the network but establishes no usable link between the host machine and VAX MP.

[1] NIC stands for Network Interface Card.

To address described limitation and enable packet interchange and hence network connection between the host and VAX MP (or between VAX MP instance and other virtual machines running on the local host) one can optionally configure virtual Ethernet interface on the host, such as TAP and VDE devices on Linux and OS X or Microsoft Loopback Adaptor on Windows[1].

By default this leads to a configuration (called hereafter *dual-homed set-up*) where VAX MP instance uses two virtual DEQNA or DELQA adapters: one attached to host NIC and used for communication with physically remote hosts, the other attached to virtual Ethernet interface (such as TAP device or MS Loopback Adapter) and used for communication with host machine or with other virtual machines running on the local host. Each of DEQNA/DELQA interfaces is assigned its own IP address. Likewise, host machine is assigned two IP addresses too: one is host's address on the NIC, the other is host's address on virtual Ethernet segment used to connect between the host and VAX MP. See illustration for dual-homed set-up in the section for Microsoft Windows.

Some host operating systems (including Linux and Windows but not OS X at this point) also offer an option for *bridged set-up*. In bridged set-up both VAX MP instance and host machine each use one IP address, rather than two addresses. In bridged set-up VAX MP utilizes one virtual DEQNA or DELQA interface attached to host *virtual* Ethernet interface (such as Microsoft Loopback Adapter on Windows or TAP device on Linux). Using host OS interface bridging facilities, virtual Ethernet interface is bridged with host NIC. Bridging works similarly to Ethernet hub and provides full exchange of packets between two adapters (host NIC and virtual Ethernet) thus obviating the need for two separate communication channels: (channel 1) between VAX MP instance and local host and (channel 2) between VAX MP instance and remote hosts, i.e. dual-homed set-up. Bridged set-up is illustrated and explained in the section for Linux below. Bridged set-up is equally possible for Windows as well, albeit it is of little benefit under Windows since default (on Windows) single-interface configuration already performs full packet interchange between VAX MP and the host.

Only Ethernet NICs and virtual Ethernet adapters such as Microsoft Loopback Adapter can be attached to XQ or XQB for mapping virtual DEQNA or DELQA adapter. Host WiFi interfaces might superficially look similar to Ethernet interfaces but they are actually not Ethernet and cannot be used for VAX MP or SIMH Ethernet networking.

Furthermore, some newer NICs can implement TOE (TCP off-load-engine) processing. One variant of TOE is also known as TCP Chimney Offload. TOE adaptors build IP frames within the card itself rather than within the host network stack, and therefore IP data sent by host to TOE adaptor is not captured by pcap and is not propagated to virtual machines such as VAX MP. TOE adaptors therefore cannot be used for communication between VAX instance and host machine (as long as TOE mode is enabled in

[1] To avoid confusion due to similarity in naming: Microsoft Loopback Adapter is *not* the same as TCP/IP loopback adapter with IP address 127.0.0.1. Microsoft Loopback Adapter is a virtual device that on the application's side appears as virtual Ethernet controller connected to virtual wire hooked into Windows network kernel. It thus establishes virtual subnet between the host machine and application or applications running on the host and making use of Microsoft Loopback Adapter. Microsoft Loopback Adapter is similar to Linux and OS X TAP and VDE devices.

adapter's settings), but only for communication between VAX and remote nodes. If your system uses TOE adapter and you want VAX instance to be able to communicate with the host machine, you will need either:

- Use dual-homed set-up.
- Or install additional adapter in host machine, such as perhaps USB adapter, that does not use TOE, and bind XQ to this adapter.
- Or choose to disable TOE mode on the adapter.

Another possible complication can occur with Gigabit Ethernet networks if the use of jumbo frames is enabled. Gigabit Ethernet networks may use jumbo frames of size exceeding conventional Ethernet's 1514-byte packet size limit. DEQLA and DEQNA interfaces and OpenVMS VAX software cannot handle such frames. Therefore VAX MP is unable to communicate with machines that send jumbo frames, including possibly the host machine itself. If you need to comminicate with those machines using protocols that do actually utilize large MTU, disable jumbo frame use in your network. Alternatively, install second adapter in your machine, such as perhaps USB adapter, disable use of outgoing jumbo frames on this adapter and connect it to the network via a switch that performs translation of incoming jumbo frames to regular-sized frames, then bind XQ to this adapter. However for protocols that are configured to use only smaller MTU and only send packets fitting into 1514-byte frame, operating on mixed-frame network is possible. In particular, TCP can be configured (and in home networks connected to DSL or cable modem commonly is) to use MTU of 1492 or 1500, rather than large MTU and thus limiting the size of Ethernet packets with TCP traffic to 1514 bytes.

Terminal emulation

Regular Windows, Linux and OS X console windows provide basic emulation of DEC VT 100 or VT 220 terminals – however a very basic one.

We are not aware of any quality terminal emulators for Linux or OS X. There is a large number of emulators for Windows however. While we won't try to list them here, our personal favorite is IVT VT220 available free for non-secure editions[1]. One popular commercially available option is Kermit95 by Columbia University[2].

[1] http://home.planet.nl/~ruurdb/ivt.htm
[2] http://www.columbia.edu/kermit/k95.html

Host software requirements (Windows)

VAX MP requires Windows XP SP3 or higher version of Windows, either 32-bit or 64-bit edition.

Under 64-bit editions of Windows VAX MP can be built and/or run either as 32-bit or 64-bit application.

Extensive testing of VAX MP had been performed only on Windows 7 and Windows XP. In addition limited testing of 32-bit processor VAX MP configuration had also been performed on Windows Server 2008 R2.

It is expected that VAX MP will also run under Windows Vista, but this had not been tested. It is expected that VAX MP will also run under Windows Server 2003 and later versions of Windows Server, but this had not been tested thoroughly and possibility of encountering problems although small but is higher than for desktop editions of Windows.

VAX MP pre-built executable images use statically linked C runtime library and do not require Microsoft Visual C++ redistributable runtime libraries to be installed as a prerequisite on the target system.

Host networking software configuration (Windows)

To use network with VAX MP, download and install the latest version of WinPCAP. [1] Restart the system after installing WinPCAP.

WinPCAP installation offers selectable option to autostart the *npf* driver when the host system boots; select this option during the installation, since VAX MP network layer requires WinPCAP *npf* driver to be loaded and started. If you do not select this option, VAX MP will try to load the driver, but will succeed only if running with administrator privileges.

You can see a list of host machine's network adapters in Network section of Windows Control Panel, or you can use the following VAX MP console command to list available host network interfaces:

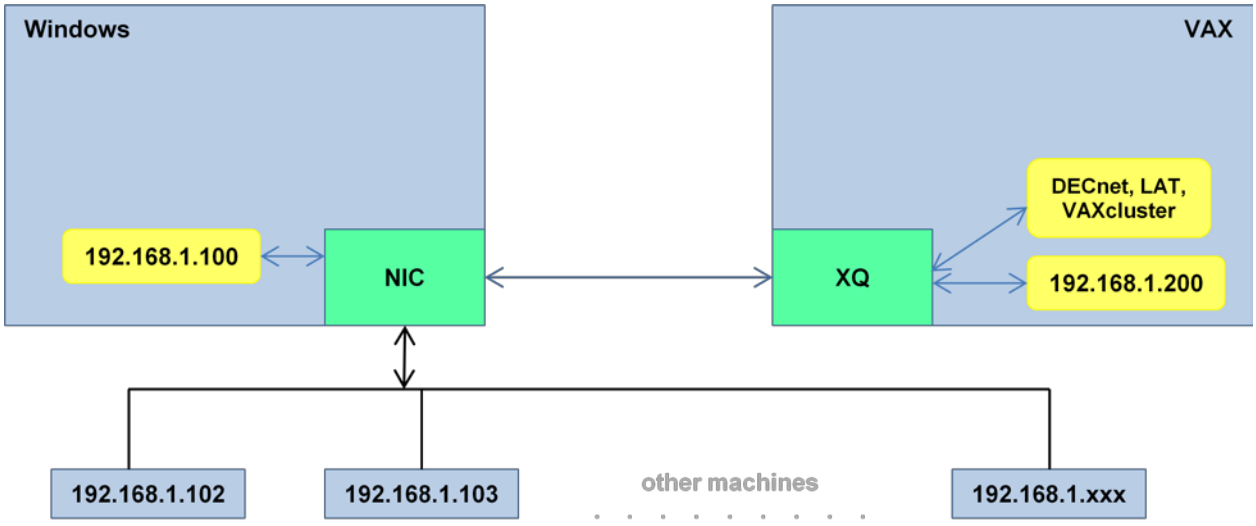
```
sim> show xq eth

ETH devices:
   0  \Device\NPF_GenericDialupAdapter                  (Adapter for generic dialup and
                                                         VPN capture)
   1  \Device\NPF_{1C501304-D702-4B2A-BACF-167D5D493025} (Marvell Yukon)
   2  \Device\NPF_{C03A487B-A0CF-4E67-BEDF-FCB945D6D13A} (Network Bridge)
   3  \Device\NPF_{E3B2F623-27E3-40DB-85D5-DE6AD8C11F7E} (VMware Network Adapter VMnet1)
   4  \Device\NPF_{C48D5C18-CB05-4E46-88D5-A04C0DBA1AED} (VMware Network Adapter VMnet8)
   5  \Device\NPF_{E389889C-A63B-4A3B-A046-D86D9C6E0101} (MS Tunnel Interface Driver)
   6  \Device\NPF_{5173795E-435D-46F0-BA13-39E49D5CFA49} (MS Loopback adaptor)
```

VAX MP can be configured under Windows either as single-interface set-up or as dual-homed or as bridged set-up.

[1] http://www.winpcap.org

(Single-interface setup)

Most usual configuration under Windows is single-interface set-up as pictured below:



In this set-up VAX instance binds virtual DELQA or DEQNA (XQ) via pcap to NIC and uses this binding to send and receive Ethernet packets both to/from remote nodes and to/from the host machine.

Some newer NICs can implement TOE (TCP off-load-engine) processing. Such adaptors cannot be used for communication between VAX instance and host machine, but only for communication between VAX and remote nodes. If your system uses TOE adapter and you want VAX to be able to communicate with he host machine, you will need to use dual-homed set-up described in the next section. Alternatively, you can install additional adapter in host machine, such as perhaps USB adapter, that does not use TOE, and bind XQ to this adapter.

Gigabit Ethernet networks may use jumbo frames of size exceeding conventional Ethernet's 1514-byte packet size limit. DEQLA and DEQNA interfaces and OpenVMS VAX software cannot handle such frames. Therefore VAX MP is unable to communicate with machines that send jumbo frames, including possibly the host machine itself. If you need to comminicate with those machines using protocols that do actually utilize large MTU, disable jumbo frame use in your network. Alternatively, install second adapter in your machine, such as perhaps USB adapter, disable use of outgoing jumbo frames on this adapter and connect it to the network via a switch that performs translation of incoming jumbo frames to regular-sized frames, then bind XQ to this adapter. However for protocols that are configured to use only smaller MTU and only send packets fitting into 1514-byte frame, operating on mixed-frame network is possible. In particular, TCP can be configured (and in home networks connected to DSL or cable modem commonly is) to use MTU of 1492 or 1500, rather than large MTU and thus limiting the size of Ethernet packets with TCP traffic to 1514 bytes.

To create single-interface set-up, edit your VAX MP startup script to attach host interface to VAX MP virtual network adaptors as needed, for example:

```
set xq enabled
set xq mac=08-00-2b-aa-bb-c1
set xq type=delqa
attach xq Marvell Yukon
```

You can also refer to host network adapters in the `attach` command by their UUID-based names, i.e. in format `\Device\NPF_{`*`UUID`*`}`.

Only Ethernet NICs and virtual Ethernet adapters such as Microsoft Loopback Adapter can be attached to XQ or XQB for mapping virtual DEQNA or DELQA adapter. Host WiFi interfaces might superficially look similar to Ethernet interfaces but they are actually not Ethernet and cannot be used for VAX MP or SIMH Ethernet networking.

By the way of example, let us say XQ is bound to the host machine's Ethernet adapter, and this host adapter is assigned IP address 192.168.1.100. Then when configuring TCP/IP Services for OpenVMS you can define interface QE0 (bound to XQ) to have IP address 192.168.1.200:

```
$ TCPIP SET INTERFACE QE0 /HOST=192.168.1.200
$ TCPIP SET CONFIGURATION INTERFACE QE0 /HOST=192.168.1.200
```

Depending on exact value of network addresses you actually use and network topology it may be necessary to additionally specify options `/NETWORK_MASK=`*`x.y.z.w`* and `/BROADCAST_MASK=`*`m.m.m.m`*.

Both host machine and VAX MP are connected to the physical wire by host's NIC they share. Host OS will generally utilize the network card to send Ethernet packets with sender's MAC address field in the packet set to the physical card's own MAC address. Whereas VAX MP will be sending Ethernet packets with sender's MAC address field set to MAC address of virtual adapter defined in startup script (e.g. 08-00-2b-aa-bb-c1 in the example above).[1] In addition, VAX MP will be sending IP packets with sender's IP address set to 192.168.1.200, whereas host OS will be sending as 192.168.1.100.

Thus even though host OS and VAX MP share the network card, they will be sending Ethernet packets to the wire as two different machines.

On the receiving side, since VAX MP puts the card is in promiscious mode, both host OS and VAX MP's virtual DELQA controller will receive all packets on the wire sent by other hosts. They will filter the packets and process only those packets that they need, i.e. host OS will generally pick up the packets destined for the card's MAC address (and for IP 192.168.1.100) whereas VAX MP will pick up the packets destined for DELQA virtual MAC address (and IP 192.168.1.200). Packets that were broadcasted to the subnet or sent to any matching multicast address will be picked up by both host OS and VAX MP.

Thus even though host OS and VAX MP share the network card, they will effectively be both sending and receiving Ethernet packets to/from the wire as two different machines.

[1] DECnet and VAXcluster protocols will further change effective MAC address to the value in range AA-00-04-00-*xx*-*yy* derived from DECnet node number and SCS ID.

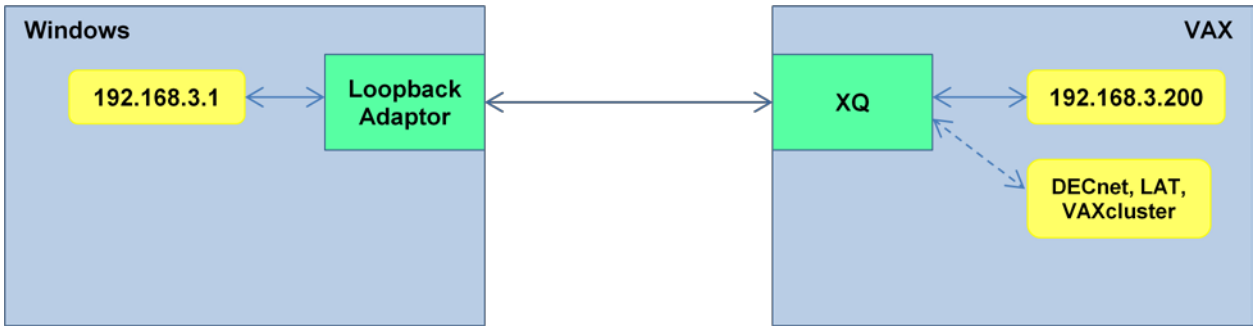
Windows network kernel will also propagate packets sent by VAX MP to the host network stack and vice versa.

If there are multiple instances of VAX MP or other SIMH simulators or other virtual machines running on the host and bound to NIC, pcap will also provide an interchange of packets between them.

If you are using a firewall, remember to configure the firewall so it allows required connections to/from addresses of the added VAX instance or instances.

(Intra-host connection set-up)

It is also possible to set up a connection between VAX instance and host only and/or other virtual machines running on the host:



Install Microsoft Loopback Adaptor[1]. Microsoft Loopback Adaptor is virtual Ethernet adapter simulating Ethernet connection (virtual wire) between host machine and applications using the adapter. Third-party virtual Ethernet adapters with similar capabilities can also be utilized instead.

Bind IP address to MS Loopback Adaptor in the adapter's properties. This address establishes host-side IP address of the adapter.

Edit VAX MP startup script to attach VAX MP DELQ/DEQNA controller to the adapter:

```
set xqb enabled
set xqb mac=08-00-2b-aa-bb-c2
set xqb type=delqa
attach xqb MS Loopback adaptor
```

By the way of example, let's assume MS Loopback Adaptor is assigned IP address 192.168.3.1 on the host's side. Then while configuring TCP/IP Services for OpenVMS you can define interface QE0 (bound to XQ) to have IP address 192.168.3.200:

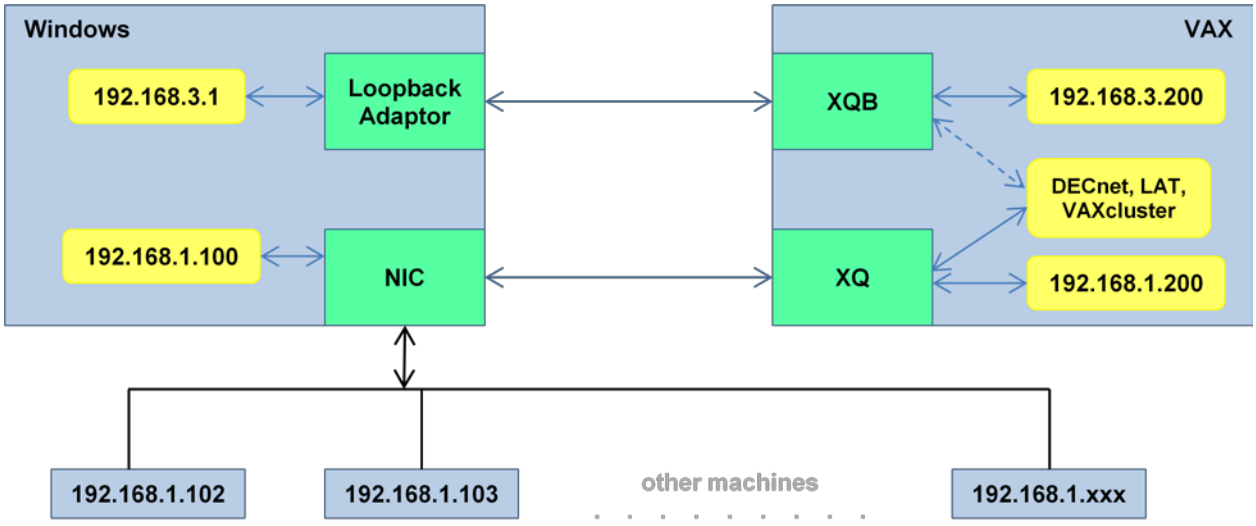
```
$ TCPIP SET INTERFACE QE0 /HOST=192.168.3.200
$ TCPIP SET CONFIGURATION INTERFACE QE0 /HOST=192.168.3.200
```

[1] As described here: http://mailman.trailing-edge.com/pipermail/simh/2011-December/006606.html

Depending on exact value of network addresses you actually decide to use it may be necessary to additionally specify options `/NETWORK_MASK=x.y.z.w` and `/BROADCAST_MASK=m.m.m.m`.

(Dual-homed setup)

It is also possible to set-up dual-homed VAX MP network configuration under Windows, whereby a separate connection is used between VAX instance and host computer alongside with second connection between VAX instance and remote computers. These two categories of connections are established via two separate host adaptors bound to two different VAX adaptors:



Such configuration may be desired in case host NIC uses TOE processing, as outlined in the section "Single-interface setup", or if it is desirable to keep VAX-host communication separate from VAX-remote nodes communication for reasons such as security.

The easiest way to establish network connection between virtual VAX and host computer itself is via Microsoft Loopback Adaptor.

Edit your VAX MP startup script to attach host interfaces to VAX MP virtual network adaptors as needed, for example:

```
set xq enabled
set xq mac=08-00-2b-aa-bb-c1
set xq type=delqa
attach xq Marvell Yukon

set xqb enabled
set xqb mac=08-00-2b-aa-bb-c2
set xqb type=delqa
attach xqb MS Loopback adaptor
```

You can also refer to host network adapters in the `attach` command by their UUID-based names, i.e. in format `\Device\NPF_{`*`UUID`*`}`.

Only Ethernet NICs and virtual Ethernet adapters such as Microsoft Loopback Adapter can be attached to XQ or XQB for mapping virtual DEQNA or DELQA adapter. Host WiFi interfaces might superficially look similar to Ethernet interfaces but they are actually not Ethernet and cannot be used for VAX MP or SIMH Ethernet networking.

By the way of example, let us say XQ is bound to the host machine's Ethernet adapter, and this host adapter is assigned IP address 192.168.1.100; and XQB is bound to MS Loopback Adapter that is assigned address 192.168.3.1. Then when configuring TCP/IP Services for OpenVMS you can define interface QE0 (bound to XQ) to have IP address 192.168.1.200; and interface QE1 (bound to XQB) to have address 192.168.3.200.

This set-up will configure two subnets.

One subnet is private between VAX MP and the host machine and runs virtual wire from the host's MS Loopback Adapter (192.168.3.1) to VAX MP's XQB (192.168.3.200). Conversely, you can access VAX MP from the host at address 192.168.3.200. You can access the host from inside VAX MP at address 192.168.3.1. There can be more than one instance of VAX MP running on the local machine and connected to this subnet (with addresses let us say 192.168.3.201, 192.168.3.202 and so on), likewise there may be other virtual machines (such as other simulators or VMware instances) running on the local host and connected to this private subnet, but this subnet is totally inside the host, unless you set up a routing between this inner subnet connected to loopback adapter and an exterior subnet.

The other subnet is on the physical wire. Both host machine and VAX MP are connected to the physical wire by host's NIC they share. Host OS will generally utilize the network card to send Ethernet packets with sender's MAC address field in the packet set to the physical card's own MAC address. Whereas VAX MP will be sending Ethernet packets with sender's MAC address field set to MAC address of virtual adapter defined in startup script (e.g. 08-00-2b-aa-bb-c1 in the example above).[1] In addition, VAX MP will be sending IP packets with sender's IP address set to 192.168.1.200, whereas host OS will be sending as 192.168.1.100.

Thus even though host OS and VAX MP share the network card, they will be sending Ethernet packets to the wire as two different machines.

On the receiving side, since VAX MP puts the card is in promiscious mode, both host OS and VAX MP's virtual DELQA controller will receive all packets on the wire sent by other hosts. They will filter the packets and process only those packets that they need, i.e. host OS will generally pick up the packets destined for the card's MAC address (and for IP 192.168.1.100) whereas VAX MP will pick up the packets destined for DELQA virtual MAC address (and IP 192.168.1.200). Packets that were broadcasted to the subnet or sent to any matching multicast address will be picked up by both host OS and VAX MP.

[1] DECnet and VAXcluster protocols will further change effective MAC address to the value in range AA-00-04-00-*xx*-*yy* derived from DECnet node number and SCS ID.

Thus even though host OS and VAX MP share the network card, they will be both sending and receiving Ethernet packets to/from the wire as two different machines.

Using TCP/IP Services for OpenVMS, adapters in the above example will be bound IP addresses the following way, for XQA and XQB correspondingly:

```
$ TCPIP SET INTERFACE QE0 /HOST=192.168.1.200
$ TCPIP SET CONFIGURATION INTERFACE QE0 /HOST=192.168.1.200

$ TCPIP SET INTERFACE QE1 /HOST=192.168.3.200
$ TCPIP SET CONFIGURATION INTERFACE QE1 /HOST=192.168.3.200
```

Depending on exact value of network addresses being used and network topology it may be necessary to additionally specify options `/NETWORK_MASK=`*`x.y.z.w`* and `/BROADCAST_MASK=`*`m.m.m.m`*.

(Bridged set-up)

It is possible to bridge host NIC and MS Loopback interfaces in order to eliminate dual network addressing per VAX MP instance and instead have only one IP address for host machine and only IP address for VAX instance. There is little benefit in bridged set-up under Windows since Windows supports single-interface setup that provides all the benefits of bridged set-up at a simpler configuration.[1] Nevertheless technically bridged set-up is possible under Windows.

Bridging can be easily performed with standard Windows interface bridging facility. In a bridged set-up VAX MP will be using only MS Loopback Adapter bound typically to XQ and bridged in Windows to host NIC.

To create a bridge on Windows XP, open Control Panel's section Network Connections, right-click on MS Loopback Adaptor and select "Add to Bridge", then do the same with Ethernet adapter. Then right-click on "Network Bridge" interface and configure it as desired. You can add interfaces to the bridge and remove interfaces from the bridge via Network Bridge properties dialog.

To create a bridge on Windows 7, open Control Panel's section Change Adapters Settings, select adapters to be bridged, right-click on any of selected adapters and select "Bridge Connections".

Once the bridge is created, you need to configure VAX instance with only single XQ adaptor attached to MS Loopback Adaptor and utilizing single IP address for VAX instance. XQB does not need to be enabled in VAX MP bridged configuration script or attached to any host interface.

[1] Bridged set-up would have offered benefits beyond a single-interface set-up in case of Gigabit Ethernet NICs if Windows bridging performed translation of jumbo frames to regular-sized frames. At this time we are not aware whether it does.

Host software requirements (Linux)

VAX MP requires Linux kernel version 2.6.38 or later. You can check the version of your system's kernel using command `uname -r`.

VAX MP also requires version of GLIBC with NPTL threads. We recommend using GLIBC version 2.13 or later. The way to check the version of GLIBC currently installed on the system depends on a flavor your Linux distribution. One common way is to execute command `ldd --version`. Another way is to locate your GLIBC shared library and simply execute it as any executable file, for example as

```
bash$ /lib/libc.so.6
```

and to look for the information in produced output.

When executing VAX user-mode code, VAX MP virtual CPU threads run at low Linux priority corresponding to `nice +10`. However critical sections of OpenVMS are executed at elevated VCPU thread priorities and VAX MP will use real-time priority range for its threads to execute OpenVMS critical sections. Highest priority used by VAX MP is displayed at VAX MP startup; currently it is 37. It is important that Linux account used to run VAX MP has proper permissions to elevate thread priority to this value, otherwise VAX MP and guest OS (OpenVMS) may run poorly. If login account does not provide adequate permissions for VAX MP to elevate its threads to required priority level, VAX MP will display a warning message at startup.

Unless you use superuser/root account or `sudo` command to execute VAX MP, the easiest way to allow priority elevation for regular account is via file `/etc/security/limits.conf`. Add the following lines to `limits.conf`, where *vaxuser* is the username of the account used to run VAX MP:

```
vaxuser        hard    nice           -20
vaxuser        soft    nice           -10
vaxuser        hard    rtprio         37
vaxuser        soft    rtprio         10
```

The changes to `/etc/security/limits.conf` become effective after login, so if you performed them while logged into `vaxuser` account, then you need to log off and log on back again.

Alternatively, you can grant the account capability CAP_SYS_NICE and use Linux command `chrt` to launch VAX MP.

Host networking software configuration (Linux)

Use of VAX MP networking features requires the latest version of `libpcap` package be installed. At the time of writing, this is package `libpcap0.8` with actual version 1.1.1-2; you can check for latest available version with tools like Synaptic Package Manager or equivalent. For example:

```
sudo apt-get install libpcap0.8
```

Like other recent versions of SIMH, VAX MP also supports TAP and VDE interfaces. VDE option requires VDE libraries installed. For more information on building SIMH images with VDE refer to appropriate general SIMH documentation (file `0readme_ethernet.txt`).[1]

In order to access network interfaces VAX MP needs Linux capabilities (privileges) CAP_NET_RAW and CAP_NET_ADMIN. To be able to run VAX MP with networking option using account other than root, these capabilities have to be assigned to VAX MP executable fle itself. To assign these capabilities to VAX MP executable file, first make sure you have `setcap` command available by installing packages `libcap2` and `libcap2-bin`, for example:

```
sudo apt-get install libcap2
sudo apt-get install libcap2-bin
```

Then grant required capabilities to VAX MP executable:

```
sudo setcap cap_net_raw,cap_net_admin=eip /path/vax_mp
```

VAX MP (and, more generally, SIMH) networking can be configured under Linux either as dual-homed or as bridged setup. Bridged setup is most usual under Linux and will be explained in detail in this section, however it is also possible to configure dual-homed setup or setup that uses only one interface linking VAX instance either to remote nodes only (via NIC) or only to the host machine itself (via TAP interface).

Network connection between VAX MP and host machine requires the use of TAP/TUN devices. For bridged setup, TAP devices are bridged with physical network interface. First make sure you have pre-requisite packages installed:

```
apt-get install bridge-utils
apt-get install uml-utilities
```

Make sure you also have gawk installed. Type gawk at the shell prompt and if the shell displays a message that gawk is not installed, then install the package suggested by shell.

Ensure you have default route defined. Issue command `route -n` and look for entry 0.0.0.0 (last line in the example below). If you do not have default route defined on your system, define it before proceeding.

```
$ route -n

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.21.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eth0
0.0.0.0         192.168.21.2    0.0.0.0         UG    100    0        0 eth0
```

[1] Also see Jordi Guillaumes i Pons, "SIMH 3.9 – Using VDE for fun and flexibility", http://ancientbits.blogspot.com.es/2012/06/simh-39-using-vde-for-fun-and.html

Then take a note of your interface and route properties in case you may want to roll back the bridging without rebooting the system:[1]

```
ifconfig >old-ifconfig.txt
route –n >old-route.txt
```

Make script file `linux-tap.sh` executable

```
chmod a+x linux-tap.sh
```

and execute it as root passing parameters for the name of tap interface to create, existing physical Ethernet interface to use, bridge interface to create and userid of the account to own the created tap interface and that will be used to run VAX MP, for example:

```
sudo ./linux-tap.sh create br0 eth0 tap0 vaxuser
```

Only Ethernet NICs and virtual Ethernet adapters such as TAP adapters can be attached to XQ or XQB for mapping virtual DEQNA or DELQA adapter. Host WiFi interfaces might superficially look similar to Ethernet interfaces but they are actually not Ethernet and cannot be used for VAX MP or SIMH Ethernet networking.

This will create a bridge interface (br0) connecting two “slave” interfaces (tap0 and eth0). Original IP address of Linux machine initially bound to eth0 will now be bound to br0, and Linux transport protocol layer will be talking to br0 instead of eth0.

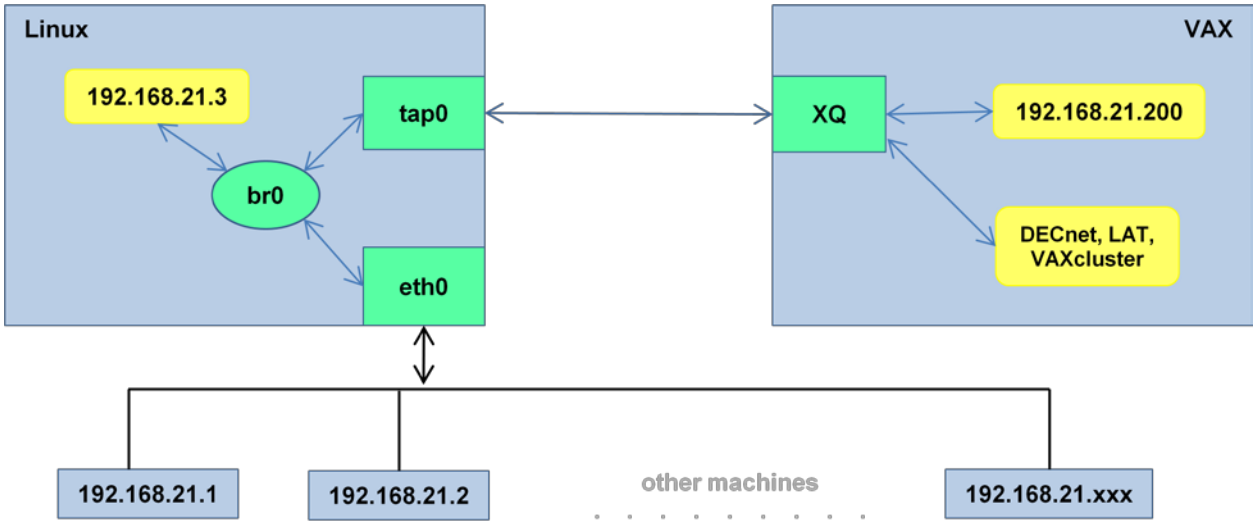
Attach created tap0 adapter to XQ in VAX MP startup script:

```
set xq enabled
set xq mac=08-00-2b-aa-bb-c1
set xq type=delqa
attach xq tap:tap0
```

The bridge is three-way and connects together VAX MP (via tap0), host protocol layer (via br0) and host NIC (eth0). All Ethernet packets arriving from any one of these sources will be propagated to the other two.

Assuming for the sake of example host IP address is 192.168.21.3, you can configure OpenVMS TCP/IP Services interface QE0 (bound to virtual adaptor XQ) to have IP address let us say 192.168.21.200. Both host and VAX MP can communicate with each other using target’s assigned address. They are also able to communicate with external computers linked to eth0 (and also nodes routed to further) by these nodes’ addresses. External hosts in turn are able to communicate with both the host machine and VAX MP by addresses assigned to those.

[1] Bridging association and created tap interface are lost when host system reboots. Instructions on how to roll back the bridging without rebooting can be found further down in this section.



If you intend to use multiple Linux accounts to run VAX MP, you will need multiple tap interfaces with different security settings. If you intend to run multiple instances of VAX MP and/or other SIMH simulators within the same account, you will also need to create multiple tap interfaces owned by that account.

Note that `linux-tap.sh create` command assumes the bridge does not exist yet. If the bridge already exists, it is possible to create additional *tapX* devices (such as perhaps to run multiple instances of VAX MP and/or other SIMH simulators concurrently, or by different users) and link them into the existing bridge with command:

```
sudo ./linux-tap.sh addtap br0 tapX vaxuser
```

For instance:

```
sudo ./linux-tap.sh addtap br0 tap2 vaxuser2
sudo ./linux-tap.sh addtap br0 tap9 vaxuser9
```

Note that interfaces and bridging created by script `linux-tap.sh` are not persistent and need to be re-created each time host system restarts, i.e. `linux-tap.sh create` and `addtap` commands should be added to Linux system startup script or other procedure executed each time at Linux restart or otherwise executed after Linux startup and before VAX MP use.

Created additional tap interfaces (beyond the last remaining one) can be destroyed with command

```
sudo ./linux-tap.sh deltap br0 tapX
```

The final (or the only) interface and the bridge itself can be destroyed with command

```
sudo ./linux-tap.sh destroy br0 eth0 tapX
```

or by rebooting Linux. You can invoke help on available `linux-tap.sh` options with

```
./linux-tap.sh help
```

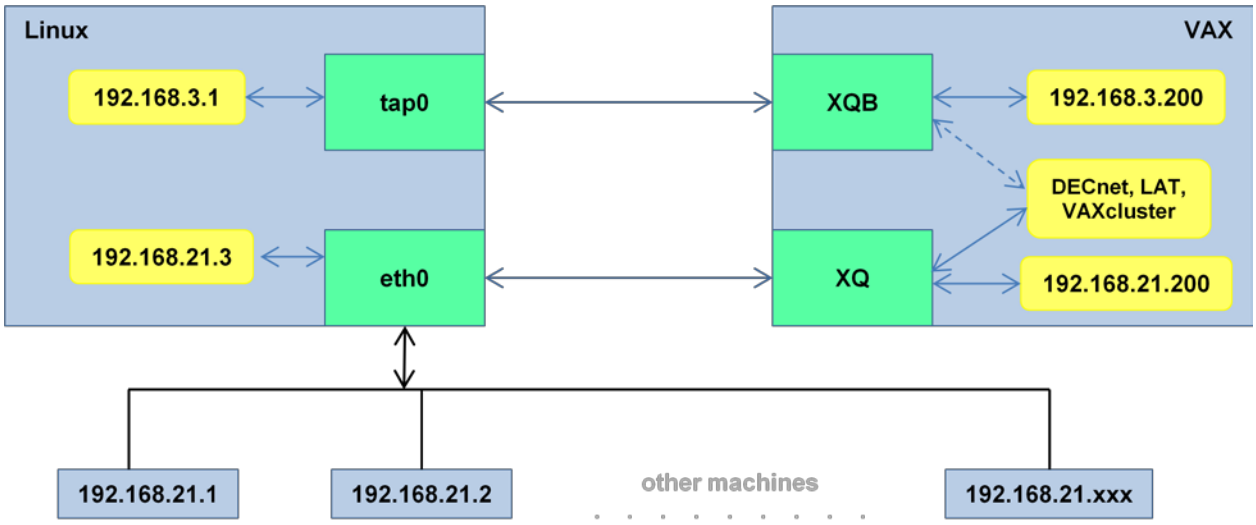
Using TCP/IP Services for OpenVMS, XQA adapter in the above example will be bound IP address the following way:

```
$ TCPIP SET INTERFACE QE0 /HOST=192.168.21.200
$ TCPIP SET CONFIGURATION INTERFACE QE0 /HOST=192.168.21.200
```

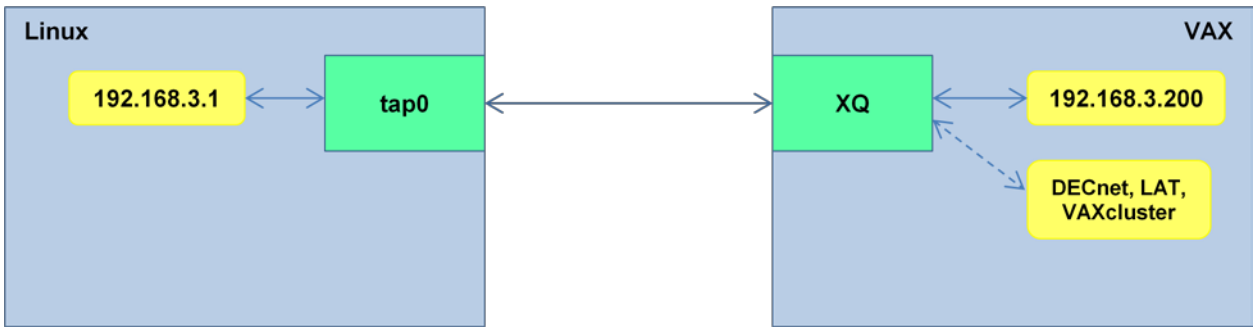
Depending on exact value of network addresses being used and network topology it may be necessary to additionally specify options `/NETWORK_MASK=`*`x.y.z.w`* and `/BROADCAST_MASK=`*`m.m.m.m`*.

If you are using a firewall, remember to configure the firewall so it allows required connections to/from addresses of the added VAX instance or instances.

It is also possible to set up VAX MP dual-homed configuration where XQ is attached to host NIC (such as eth0) and XQB is attached to tap*X*:



It is also possible to set up VAX MP network configuration with only single network interface – either attached to NIC (for communications only with other hosts) or attached to tap*X* (for communications only with the local host or other virtual machines running on it). For example:



Host software requirements (OS X)

VAX MP had been tested on OS X 10.7.4 (both server and workstation editions) on a range of computers including quad-core i7 Mac Mini Server (configured up to 8 VAX CPUs), quad-core i7 MacBook Pro (configured up to 8 VAX CPUs) and older CoreDuo based MacBook (configured to 2 VAX CPUs) and is expected to run on other Mac multiprocessor models as well. VAX MP may also run on slightly earlier versions of OS X than 10.7.4, but this had not been tested.

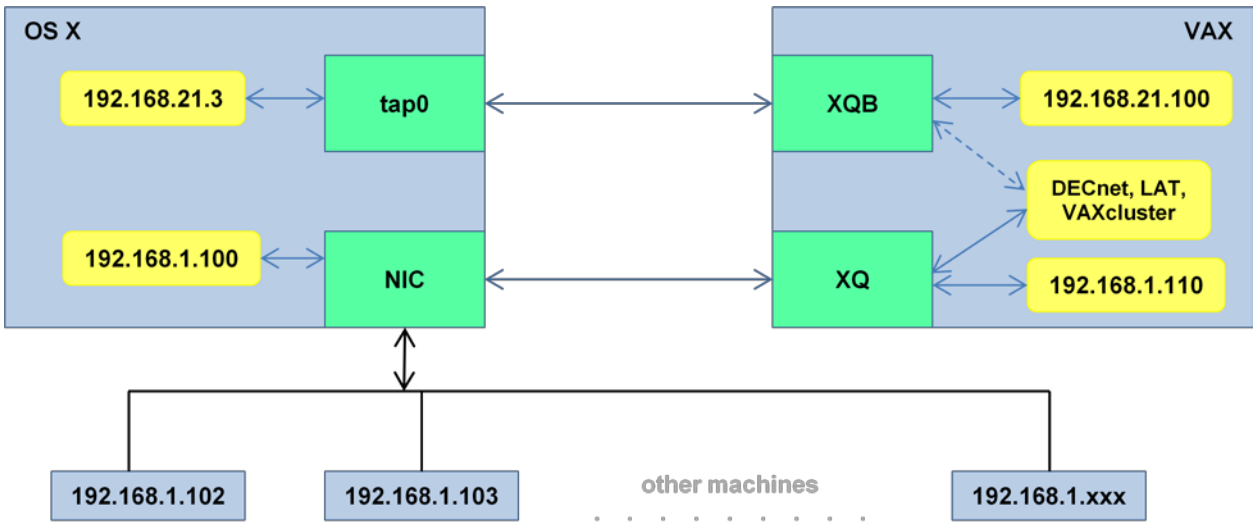
You can find the details of your computer configuration with OS X System Information utility and Intel MacCPUID utility[1].

You can find helpful OS X specific instructions on how to disable Time Machine for large files (such as VAX MP disk image fles), and on virtualizing OpenVMS installation CD to an ISO image file at Hoffman Labs web site: http://labs.hoffmanlabs.com/node/922.

Host networking software configuration (OS X)

VAX MP networking is configurable on OS X either as (a) dual-homed set-up or (b) using only one network interface mapped either to NIC to communicate only with remote host machines or mapped to TAP to communicate only with the local host. There is no easy or tested bridged set-up available for OS X that bridges TAP to NIC (however see the note at the end of this section for one perhaps possible, but untested bridged set-up).

Accordingly, this section describes only dual-homed configuration similar to pictured below:



It is also possible to run just one of pictured connections: either connection to host machine (via tap0) or connection to remote machines (via NIC) instead of running both connections.

[1] http://software.intel.com/en-us/articles/download-maccpuid

Running a connection from VAX MP to remote hosts via NIC is possible with non-root account, however use of TAP connection from VAX MP to local host or other virtual machines running on the local host requires running VAX MP as root (via `sudo`). Therefore if you intend to be able to make virtual network connection between VAX MP and local host, as you likely will, you would have to run VAX MP as root (via `sudo`) anyway. In this case you can disregard interface security related guidelines in set-up instructions below.

OS X comes pre-packaged with `libpcap` that is used by VAX MP for networking. Therefore no separate installation of `libpcap` is required. However `libpcap` requires read-write access to devices `/dev/bpf*`. By default the access to these devices is limited to root only.

If you want to run VAX MP under the account or accounts other than root, you should enable access to "`bpf*`" devices for the account that will be used to run VAX MP. Note that `bpf` devices are re-created each time the system boots, therefore permissions on them should be adjusted each time after a reboot, ideally by providing a script that does it at boot time.

Libpcap site does provide such script and instructions on how to install it:

```
http://cvs.tcpdump.org/cgi-bin/cvsweb/cvs/libpcap/chmod_bpf?rev=1.1.2.1
http://cvs.tcpdump.org/cgi-bin/cvsweb/cvs/libpcap/README.macosx?rev=1.2

http://cvs.tcpdump.org/cgi-bin/cvsweb/cvs/libpcap/ChmodBPF/ChmodBPF?rev=1.1
http://cvs.tcpdump.org/cgi-
bin/cvsweb/cvs/libpcap/ChmodBPF/StartupParameters.plist?rev=1.1
```

Script `chmod_bpf` as provided by tcpdump.org assumes that all users running VAX MP (or other SIMH based simulators with network access) will belong to group `admin`.[1]

Instead of making each user needing to run VAX MP a member of `admin` group, it is possible to define a special group that controls access to `bpf` devices and nothing else, let us say the group named `pcaprw`. You can create this group using OS X Server Admin tool or `dscl` commands. Once the group is created, add all users who need to run VAX MP as members of this group. You can easily check the list of groups a user belongs to with OS X command

```
id username
```

Now modify file `chmod_bpf` to set `pcaprw` as the owning group of `bpf` devices, instead of `admin`:

```
chgrp pcaprw /dev/bpf*
chmod g+rw /dev/bpf*
```

Execute this file:

```
sudo /usr/local/bin/chmod_bpf
```

and ensure device protection is set correctly:

[1] It might be tempting to replace it with ACL based access, however unfortunately OS X `devfs` does not support ACLs, so access control has to be group based.

```
$ ls –l /dev/bpf*
crw-rw----  1 root  pcaprw   23,   0 Jul 10 09:37 /dev/bpf0
crw-rw----  1 root  pcaprw   23,   1 Jul  9 21:37 /dev/bpf1
crw-rw----  1 root  pcaprw   23,   2 Jul  9 21:37 /dev/bpf2
crw-rw----  1 root  pcaprw   23,   3 Jul  9 21:37 /dev/bpf3
```

Then follow instructions in tcpdump.org documents at links mentioned above to set up `chmod_bpf` for automatic execution each time the system restarts.

Assuming your host machine's Ethernet interface is en0, add the following lines to your VAX MP startup script:

```
set xq enabled
set xq mac=08-00-2b-aa-bb-c1
set xq type=delqa
attach xq en0
```

Only Ethernet NICs and virtual Ethernet adapters such as TAP adapters can be attached to XQ or XQB for mapping virtual DEQNA or DELQA adapter. Host WiFi interfaces might superficially look similar to Ethernet interfaces but they are actually not Ethernet and cannot be used for VAX MP or SIMH Ethernet networking.

Assuming for the sake of example that your host machine's IP address is 192.168.1.100, you can configure OpenVMS TCP/IP Services QE0 interface with IP address 192.168.1.110 or any other address either in the same subnet or in another subnet, for instance:

```
$ TCPIP SET INTERFACE QE0 /HOST=192.168.1.110
$ TCPIP SET CONFIGURATION INTERFACE QE0 /HOST=192.168.1.110
```

Depending on exact value of network addresses being used and network topology it may be necessary to additionally specify options `/NETWORK_MASK=`*`x.y.z.w`* and `/BROADCAST_MASK=`*`m.m.m.m`*.

Under OS X `libpcap` bound to host NIC allows VAX MP to communicate only with other computers, but not with the host computer itself. `Libpcap` captures Ethernet packets arriving to the NIC from external nodes and forwards them to both VAX MP and the host machine, but OS X network kernel does not fully interchange packets between the host machine and VAX MP. Packets sent by VAX MP via NIC are not made visible to the host machine by OS X network stack. Thus `libpcap` bound to host machine's NIC allows VAX MP to communicate with other machines on the network but establishes no usable link between host machine and VAX MP.

To enable network connection between the host machine and VAX MP one can optionally configure TAP interface. This requires installation of open-source 3rd party component: `tuntaposx` by Mattias Nissler. The use of this component requires running VAX MP as root.

To install `tuntaposx` navigate to http://tuntaposx.sourceforge.net, click "Download" and then click the link to download current release. As of time writing this, this link leads to http://downloads.sourceforge.net/tuntaposx/tuntap_20111101.tar.gz. After downloading the tar file, decompress and expand tar file to a temporary directory. Double-click on file tuntap*_yyyymmdd*.pkg

and follow the installer's screens. After installing `tuntaposx`, optionally execute command `kextstat` to display a list of all loaded kernel extensions and make sure that tap and tun extensions had been successfully loaded.

After installing tap interface, add the following lines to your VAX MP startup script:

```
set xqb enabled
set xqb mac=08-00-2b-aa-bb-c2
set xqb type=delqa
attach xqb tap:tun0
! ifconfig tap0 192.168.21.3 up
```

Note the space in the last line after "!".

The latter line is executed by VAX MP as shell command and causes interface tap0 to be configured with specified address. You can then configure OpenVMS TCP/IP Services interface QE1 (mapped to XQB) with IP address let us say 192.168.21.100:

```
$ TCPIP SET INTERFACE QE1 /HOST=192.168.21.100
$ TCPIP SET CONFIGURATION INTERFACE QE1 /HOST=192.168.21.100
```

Host and VAX MP can cross-reference each other using these addresses, but they are not visible outside of the host machine unless you set up a routing between this internal subnet and exterior network. Described set-up follows the scheme described in network setup chapter for Windows, and the chart from Windows chapter equally applies to the described OS X setup.

Remember that the use of TAP/TUN on OS X requires VAX MP to be run as root (e.g. via `sudo`).

OS X currently does not have builtin or 3rd party open-source freeware facilities for Ethernet bridging, like in the setup suggested for Linux above. However there is a commercial product that provides Ethernet bridging under OS X: IPNetRouterX.[1] It may allow to implement network set-up with single network address for each of the host machine and VAX MP, rather than two addresses for each, but we have not tested neither this product nor the possibility of using it for such setup under OS X.

Finally, if you are using a firewall, remember to configure the firewall so it allows required connections to/from addresses of the added VAX instance or instances.

[1] http://www.sustworks.com/site/prod_ipnrx_overview.html

Quick VAX MP Installation Guide

First make sure you have VAX MP installed and have met requirements described above in section "Host software requirements" appropriate for your host operating system.

The easiest way to install VAX MP is by downloading pre-built executable image for your host OS. If you want to build VAX MP yourself, refer to Addendum A of this document ("Building VAX MP").

> For running VAX MP on 64-bit edition of Windows, we recommend to use 32-bit build of VAX MP, since it is preliminary believed to have slightly better overall performance than x64 build, by few percent.
>
> Default suggested mode for 64-bit Linux is the use of 64-bit VAX MP build. Use of 32-bit build of VAX MP on 64-bit Linux is possible[1] and is straightforward for builds with NONET or NET options and without VDE. However we have not benchmarked yet 32-bit build of VAX MP vs. 64-bit build when both are executed under 64-bit Linux and cannot at this point definitively recommend using 32-bit build on 64-bit Linux.
>
> Installation of 32-bit build of VAX MP built with SHR or VDE options (i.e. making use of 32-bit `libpcap` and/or VDE shared libraries) under 64-bit Linix may be possible as well, but is much more complicated and had not been attempted or tested and is not recommended, and its description is beyond the scope of this document.
>
> For OS X only 64-bit build of VAX MP is provided.

You can either move your pre-existing OpenVMS installation to VAX MP or install OpenVMS under VAX MP from scratch. No modifications to existing system disk image are required other than the installation of VSMP tool in a separate directory on the OpenVMS system disk.

However we most strongly recommend using only MSCP disks (SIMH device name RQ*xxx*) with VAX MP instance intended to run in multiprocessor mode. Other types of virtual disks (RL*xxx* devices) can be used in multiprocessor mode, but their performance will be inferior and they can also induce "choppy" system behavior and poor responsiveness.[2] Therefore the use of non-MSCP disks for any significant IO loads is strongly discouraged. If you have an existing system disk image on a non-MSCP device, transfer it to an MSCP device first; and similarly for all data files on other disks you are planning to use. If you plan on installing OpenVMS from scratch, install it only on MSCP (RQ) disk device.

If you have an existing system disk image you want to use with VAX MP, proceed to step 2. For new installations, proceed to step 1.

---

[1] After installing ia32 compatibility package, such as `ia32-libs` for Ubuntu or other Debian or Debian-like editions of Linux. Refer to http://www.debian-administration.org/articles/531 and http://www.debian-administration.org/articles/534.

[2] Non-MSCP disk devices will have device affinity set to primary processor only and thus reduce IO scalability. They also do not support asynchronous IO execution on separate IO threads. Instead host IO operations for non-MSCP storage devices will be executed in the context of VCPU thread, possibly causing a spinlock or mutex held by VCPU for a long time while blocking host IO operation is in progress and bringing other VCPUs to a stall, should they need the resource protected by this spinlock or mutex. We recommend therefore limiting the use of non-MSCP disk devices only for minor, light IO loads, and if possible better avoiding them altogether.

Step 1. Perform new OpenVMS installation and create system disk.

Initial installation of OpenVMS under VAX MP is identical to installation under regular uniprocessor SIMH VAX. It is beyond the scope of this document to describe how to perform such installation. Refer to existing SIMH documents, for example Phillip Wherry's "Running VAX/VMS Under Linux Using SIMH"[1] or Stephen Hoffman's "SIMH tips: VAX emulation on Mac OS X"[2] or Lennert Van Album's "Installing VMS in SIMH on Linux"[3].

You can use either of the simulators – VAX MP or uniprocessor SIMH VAX – to create OpenVMS bootable disk image and perform initial customization, such as running AUTOGEN or installing and configuring layered products (such as TCP/IP services etc.) and loading licenses.

Step 2. Load VSMP tool distribution.

Obtain VSMP tool distribution. VAX MP provides it as either virtual tape or virtual disk image. VSMP tool can be installed in any directory of your choosing. In this document and its examples we will assume VSMP will be installed into SYS$COMMON:[VSMP].

To install VSMP, log in to OpenVMS as SYSTEM.

If you obtained VSMP distribution on virtual disk, simply mount it and copy VSMP directory to target location.

If you obtained VSMP distribution on virtual tape image, attach it to SIMH or VAX MP virtual tape drive device, for example `attach -r tq vsmp.vtape`, then mount it and restore contents to the target directory:

```
$ ALLOCATE MUA0:
$ MOUNT /FOREIGN MUA0: VSMP
$ CREATE/DIRECTORY SYS$COMMON:[VSMP]
$ BACKUP MUA0:VSMP/SAVE_SET SYS$COMMON:[000000…] /LOG
```

Step 3. Build VSMP tool.

VSMP tool distribution already includes pre-built VSMP binary for OpenVMS 7.3. If you are installing under OpenVMS 7.3, skip to step 4. Otherwise you will need to build VSMP for your version of OpenVMS or VAX/VMS. Install C compiler. VSMP had been tested with Compaq C compiler version 6.4. You can check version of the compiler installed on your system with DCL command `CC /VERSION`. To build VSMP, execute command `@BUILD` from VSMP directory. This command creates file VSMP.EXE for the installed version of VMS (as well as listing files, map file and various scratch files such as temporary OBJ and MLB for the compiled modules). Files are created in the current (VSMP) directory. You can execute PURGE after @BUILD completes.

[1] http://www.wherry.com/gadgets/retrocomputing/vax-simh.html
[2] http://labs.hoffmanlabs.com/node/922
[3] https://vanalboom.org/node/18

If you are building under OpenVMS version other than 7.3, we strongly urge you to take a backup of your system disk and data disks before proceeding to the next step.

Step 4. Set SYSGEN parameters.

Set SYSGEN parameters on your system as required for multiprocessor operations under VAX MP.

```
$ SET DEFAULT SYS$SYSTEM
$ RUN SYSGEN
SYSGEN> USE CURRENT
SYSGEN> WRITE UNI-SAFE.PAR
SYSGEN> SET MULTIPROCESSING 2
SYSGEN> SET TIME_CONTROL 6
SYSGEN> SET SMP_LNGSPINWAIT 950000
SYSGEN> WRITE CURRENT
SYSGEN> EXIT
```

If you intend to create large number of virtual processors in your VAX configuration, additionally increase SYSGEN parameter SPTREQ by (NCPUS – 1) * (2 * INTSTKPAGES + 12). For example, assuming value of of INTSTKPAGES of 18 and initial value of SPTREQ of 7000, to create configuration with 12 VAX processors enlarge SPTREQ by (12 – 1) * (2 * 18 + 12) = 11 * 48 = 528 to 7528:

```
$ RUN SYSGEN
SYSGEN> USE CURRENT
SYSGEN> SHOW INTSTKPAGES
Parameter Name            Current    Default     Min.     Max.     Unit  Dynamic
--------------            -------    -------    -------  -------   ----  -------
INTSTKPAGES                    18          6         4     65535   Pages
SYSGEN>  SHOW SPTREQ
Parameter Name            Current    Default     Min.     Max.     Unit  Dynamic
--------------            -------    -------    -------  -------   ----  -------
SPTREQ                       7000       3900      3000        -1   Pages
SYSGEN> SET SPTREQ 7528
SYSGEN> WRITE CURRENT
SYSGEN> EXIT
```

If you housekeep your OpenVMS system with AUTOGEN, take a note to reflect these changes also in MODPARAMS.DAT – not just yet, but after you decide you are satisfied with VAX MP performance and stability.

Shutdown the system (do not reboot it just yet !):

```
$ SHUTDOWN
```

Step 5. Edit VAX MP SIMH startup script.

Below is a sample SIMH startup script with specific additions for VAX MP.

The script starts with command `set asynch`. This command enables asynchronous IO mode for all devices defined subsequently in the script that support asynchronous IO, including RQ devices (MSCP disks), TQ (MSCP tape drives) and XQ (Ethernet DEQNA/DELQA controllers). It is highly recommended to enable asynchronous IO mode for multiprocessing.

> If asynchronous IO is not enabled, VAX MP will be executing host IO requests in the context of VCPU threads, possibly causing a spinlock or mutex held by VCPU for a long time while synchronous host IO operation is in progress and VCPU thread is blocked waiting for this operation to complete. This may bring other VCPUs to a stall, should they need the resource protected by this spinlock or mutex.
>
> Synchronous IO on VCPU thread (as opposed to relegating IO operations to IO threads and performing them asynchronously of VCPUs execution) also distorts VCPU calibration.
>
> For these reasons we highly recommend using asynchronous IO and enabling it with `set asynch`.

Note that this configuration script disables RL disks, as had been advised before, and uses RQ (MSCP) disks instead.

It also disables DZ multiplexor and uses VH multiplexor instead. DZ can be used with VAX MP, however use of DZ is not recommended if you want to move your configuration back and forth between VAX MP and older versions of SIMH (3.7 and earlier, and early builds of 3.8) because of the bug in these older SIMH versions. See details in the notes later in this document.

The script also uses TQ instead of TS for tape drive (i.e. MSCP tape drive instead of TS11), as advised in the notes below.

Value of `host_turbo` should be set to the ratio of maximum host CPU frequency to minimum host CPU frequency. Modern CPUs can dynamically change their operating frequency depending on thermal conditions and workload. For example, when only one or two cores are loaded on a 4-core chip, the chip with assistance of operating system can shut down unused cores and boost frequency on active cores. OpenVMS timeout and busy-wait loops need to be adjusted for this possible change in frequency. The adjustment is performed via `host_tubro` setting. Use BIOS or host OS system information utilities to determine exact model of CPU chip in your host computer, then look up chip manufacturer's site for CPU data. For example, suppose your computer is equipped with Intel i7-3770T processor. Intel online data sheet for this processor indicates:

> Clock Speed: 2.5 GHz
> Max Turbo Frequency: 3.7 GHz

Accordingly, turbo factor for this processor will be 3.7 / 2.5 = 148%. This is `host_turbo` value setting you should enter for the given host processor:

```
deposit host_turbo 148          ; enter as decimal
```

By default `host_turbo` is set to 120%, which is sufficient for many modern processors, but not for all of them. You should enter the value that is correct for your particular computer. Current value of `host_turbo` setting can be displayed with SIMH console command `examine host_turbo` (it displays the value as decimal) or from inside OpenVMS using `VSMP QUERY` command.

Value of `ws_lock` is set to 1 to cause pre-paging of VAX memory and simulator's own code and data into VAX MP working set.

```
set asynch                    ; offload disk, tape and network IO to threads
cpu multi 6                   ; configure VAX with six virtual CPUs
set cpu 512m                  ; configure RAM = 512 MB

dep host_turbo 120            ; host cpu max freq / min freq ratio (decimal percentage)

load -r \myvax\ka655x.bin     ; load MicroVAX ROM content
attach NVR \myvax\ka655.nvr   ; attach MicroVAX NVRAM storage

set rl disable                ; do not use non-MSCP disks
set cr disable                ; who cares for card reader?

set ts disable                ; use MSCP tape instead of TS11
set tq enable                 ; …

set dz disable                ; do not use DZV11, use DHV instead
set vh enable                 ; …
attach vh 10025               ; activate multiplexor on this port

set rq0 ra92
attach rq0 \myvax\vms-sysdisk.vdisk

set rq1 cdrom
attach -r rq1 \myvax\vms-hobbyist-cd.iso

set rqb enable
set rqb0 rd54
attach rqb0 \myvax\vms-pageswap.vdisk

show xq eth

set xq enabled
set xq mac=08-00-2b-aa-bb-cc
set xq type=delqa
attach xq Marvell Yukon

set xqb enabled
set xqb mac=08-00-2b-aa-bb-dd
set xqb type=delqa
attach xqb0 MS Loopback adaptor

set cpu idle=VMS
set cpu conhalt
set console brk=10
dep ws_lock 1                 ; pre-page memory into VAX MP working set
set console telnet=10023
boot cpu
```

Step 6. Boot OpenVMS, execute VSMP to create VAX processors in OpenVMS and start the processors.

Start VAX MP with configuration script you just created, passing script's file path as command line argument to `vax_mp`. Perform regular VMS bootstrap and log into the system:

```
VAX MP simulator V3.8-3
vSMP revision: 1, primary interlock: portable/native
Asynchronous I/O enabled
NVR: buffering file in memory
RQ1: unit is read only
Listening on port 10025 (socket 668)
WinPcap version 4.1.1 (packet.dll version 4.1.0.1753), based on libpcap version
1.0 branch 1_0_rel0b (20091008)
ETH devices:
   0  \Device\NPF_{1A5BAE1C-656A-49B4-88C5-7A60B15340D6} (WIFI)
   1  \Device\NPF_{FA7E7B05-AB9C-4A07-AF9D-2ADC15C9F35D} (MS Loopback Adaptor)
   2  \Device\NPF_{45D07CBE-215C-4F04-ADA1-13D4BF5D0B98} (VMware Network Adapter VMnet8)
   3  \Device\NPF_{3923F443-FD3C-4D4D-A3C7-7B65C42218AB} (VMware Network Adapter VMnet1)
   4  \Device\NPF_{BAA69382-B5FE-43B9-931D-816AA152F991} (Local Area Connection)
```

```
Eth: opened OS device \Device\NPF_{BAA69382-B5FE-43B9-931D-816AA152F991}
Eth: opened OS device \Device\NPF_{FA7E7B05-AB9C-4A07-AF9D-2ADC15C9F35D}

KA655X-B V5.3, VMB 2.7
Performing normal system tests.
40..39..38..37..36..35..34..33..32..31..30..29..28..27..26..25..
24..23..22..21..20..19..18..17..16..15..14..13..12..11..10..09..
08..07..06..05..04..03..
Tests completed.
>>>B
(BOOT/R5:80 DUA0

  2..
-DUA0
  1..0..

%SYSBOOT-I-SYSBOOT Mapping the SYSDUMP.DMP on the System Disk
%SYSBOOT-I-SYSBOOT SYSDUMP.DMP on System Disk successfully mapped
%SYSBOOT-I-SYSBOOT Mapping PAGEFILE.SYS on the System Disk
%SYSBOOT-I-SYSBOOT SAVEDUMP parameter not set to protect the PAGEFILE.SYS
   OpenVMS (TM) VAX Version V7.3     Major version id = 1 Minor version id = 0
%WBM-I-WBMINFO Write Bitmap has successfully completed initialization.
$!  Copyright 2001 Compaq Computer Corporation.

%STDRV-I-STARTUP, OpenVMS startup begun at 23-APR-2012 20:47:57.18
%RUN-S-PROC_ID, identification of created process is 00000206
%DCL-S-SPAWNED, process SYSTEM_1 spawned

%%%%%%%%%%%  OPCOM  23-APR-2012 20:48:06.65  %%%%%%%%%%%
Operator _MYVAX$OPA0: has been enabled, username SYSTEM

[...]

%SET-I-INTSET, login interactive limit = 64, current interactive value = 0
  SYSTEM       job terminated at 23-APR-2012 20:48:24.86

  Accounting information:
  Buffered I/O count:            2443         Peak working set size:    1672
  Direct I/O count:               872         Peak page file size:      8424
  Page faults:                  12617         Mounted volumes:             1
  Charged CPU time:           0 00:00:17.91   Elapsed time:     0 00:00:27.84

 Welcome to OpenVMS (TM) VAX Operating System, Version V7.3

Username: SYSTEM
Password:

 Welcome to OpenVMS (TM) VAX Operating System, Version V7.3
    Last interactive login on Wednesday, 18-APR-2012 14:12
    Last non-interactive login on Sunday, 21-AUG-2011 19:13
```

At this point VMS recognizes the virtual machine as plain uniprocessor MicroVAX 3900 or VAXserver 3900:

```
$ SHOW CPU

MYVAX, a VAXserver 3900 Series
Multiprocessing is ENABLED. Full checking synchronization image loaded.

PRIMARY CPU = 00
Active CPUs:      00
Configured CPUs:  00
```

Let's load VSMP:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD
%VSMP-I-CALIBRATING, Calibrating OpenVMS timing loops...
%VSMP-I-CALIBRATED, Calibration completed at 25 samples
%VSMP-I-LOADED, VSMP resident image was loaded into the kernel and activated
$
$ SHOW CPU

MYVAX, a VAXserver 3900 Series
Multiprocessing is ENABLED. Full checking synchronization image loaded.

PRIMARY CPU = 00
Active CPUs:      00
Configured CPUs:  00 01 02 03 04 05
```

Now VMS sees the machine as multiprocessor with 6 CPUs installed in the chassis. Only the primary processor (CPU 00) is active at the moment. Let's start other processors:

```
$ START /CPU /ALL
%SMP-I-CPUBOOTED, CPU #01 has joined the PRIMARY CPU in multiprocessor operation
%SYSTEM-I-CPUSTARTD, CPU 1 started
%SMP-I-CPUBOOTED, CPU #02 has joined the PRIMARY CPU in multiprocessor operation
%SYSTEM-I-CPUSTARTD, CPU 2 started
%SMP-I-CPUBOOTED, CPU #03 has joined the PRIMARY CPU in multiprocessor operation
%SYSTEM-I-CPUSTARTD, CPU 3 started
%SMP-I-CPUBOOTED, CPU #04 has joined the PRIMARY CPU in multiprocessor operation
%SYSTEM-I-CPUSTARTD, CPU 4 started
%SMP-I-CPUBOOTED, CPU #05 has joined the PRIMARY CPU in multiprocessor operation
%SYSTEM-I-CPUSTARTD, CPU 5 started
$
$ SHOW CPU

MYVAX, a VAXserver 3900 Series
Multiprocessing is ENABLED. Full checking synchronization image loaded.

PRIMARY CPU = 00
Active CPUs:      00 01 02 03 04 05
Configured CPUs:  00 01 02 03 04 05
```

All processors had been started and are able to execute workload. You can also display detailed information about the processors:

```
$ SHOW CPU /FULL

MYVAX, a VAXserver 3900 Series
Multiprocessing is ENABLED. Full checking synchronization image loaded.

PRIMARY CPU = 00

CPU 00 is in RUN state
Current Process: SYSTEM          PID = 00000214
Capabilities of this CPU:
        PRIMARY QUORUM RUN
Processes which can only execute on this CPU:
        *** None ***

[...]

CPU 05 is in RUN state
Current Process: *** None ***
Capabilities of this CPU:
        QUORUM RUN
Processes which can only execute on this CPU:
        *** None ***
```

Now you can place load on the system by starting parallel processes and observe system status:

```
$ SHOW SYSTEM
OpenVMS V7.3  on node MYVAX  23-APR-2012 20:54:37.41  Uptime  0 00:08:36
  Pid    Process Name    State  Pri      I/O       CPU       Page flts  Pages
00000201 SWAPPER         HIB     16        0   0 00:00:00.61         0      0
00000205 CONFIGURE       HIB      8        5   0 00:00:00.20       111    197
00000206 LANACP          HIB     13       54   0 00:00:00.71       343    918
00000208 IPCACP          HIB     10        6   0 00:00:00.17        96    197
00000209 ERRFMT          HIB      9       21   0 00:00:00.25       129    233
0000020B OPCOM           HIB      6       59   0 00:00:00.56       261    137
0000020C AUDIT_SERVER    HIB      8       49   0 00:00:01.00       539    812
0000020D JOB_CONTROL     HIB     10       31   0 00:00:00.34       201    375
0000020E SECURITY_SERVER HIB     10       22   0 00:00:01.21       726   1374
0000020F TP_SERVER       HIB     10       41   0 00:00:00.39       202    337
00000210 LATACP          HIB     14        9   0 00:00:00.43       271    275
00000211 TCPIP$INETACP   HIB      9       47   0 00:00:00.69       256    232
00000212 TCPIP$FTP_1     LEF     10     1171   0 00:00:03.77      1979   1382  N
00000213 SYSTEM          CUR  0  11     1741   0 00:01:19.63     27391    538
00000216 _TXA0:          LEF     10      634   0 00:00:08.91      4861    375
00000217 TCPIP$FTPC00003 HIB      6      146   0 00:00:02.47      1434   1174  N
0000021A CPU_HOG_1       CUR  2   2       50   0 00:00:07.70       405    348  S
0000021B CPU_HOG_2       COM      2       50   0 00:00:07.79       428    333  S
0000021C CPU_HOG_3       CUR  1   2       50   0 00:00:07.29       405    348  S
0000021D CPU_HOG_4       COM      2       50   0 00:00:06.67       405    348  S
0000021E CPU_HOG_5       CUR  5   2       50   0 00:00:06.97       405    348  S
0000021F CPU_HOG_6       CUR  4   2       50   0 00:00:06.28       405    348  S
00000220 CPU_HOG_7       CUR  3   2       50   0 00:00:06.06       405    348  S
```

```
$ MONITOR/INT=1 MODES

                            OpenVMS Monitor Utility
            +-----+         TIME IN PROCESSOR MODES
            | CUR |              on node MYVAX
            +-----+         23-APR-2012 20:54:57.00

Combined for  6 CPUs                 0         150       300       450      600
                                     + - - - - + - - - - + - - - - + - - - - -+
 Interrupt Stack                     |
                                     |         |         |         |          |
 MP Synchronization                  |
                                     |         |         |         |          |
 Kernel Mode                         |
                                     |         |         |         |          |
 Executive Mode                      |
                                     |         |         |         |          |
 Supervisor Mode                     |
                                     |         |         |         |          |
 User Mode                       600 |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
                                     |         |         |         |          |
 Compatibility Mode                  |
                                     |         |         |         |          |
 Idle Time                           |
                                     + - - - - + - - - - + - - - - + - - - - -+
```

```
$ MONITOR/INT=1 SYSTEM

Node: MYVAX                 OpenVMS Monitor Utility     23-APR-2012 20:56:15
Statistic: CURRENT             SYSTEM STATISTICS
                                                          Process States
          + CPU Busy (600)          -+         LEF:     2      LEFO:     0
          |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒|         HIB:    13      HIBO:     0
CPU     0 |--------------------------| 600     COM:     2      COMO:     0
          |▒▒▒▒                      |         PFW:     0      Other:    6
          +--------------------------+         MWAIT:   0
          Cur Top: CPU_HOG_5 (92)                          Total: 23

          + Page Fault Rate (5)     -+         + Free List Size (461067)  +
          ||                         |         |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  | 470K
MEMORY  0 |--------------------------| 500   0 |--------------------------|
          |                          |         |                          | 51K
          +--------------------------+         + Mod List Size (1831)     +
          Cur Top: SYSTEM (5)

          + Direct I/O Rate (0)     -+         + Buffered I/O Rate (2)   -+
          |                          |         |                          |
I/O     0 |--------------------------| 500   0 |--------------------------| 500
          |                          |         |                          |
          +--------------------------+         +--------------------------+
          Cur Top:  (0)                        Cur Top: SYSTEM (2)
```

You can stop secondary CPUs manually if needed. Note that all active secondary CPUs, i.e. all active CPUs except the primary, will be stopped automatically during system shutdown.

```
$ STOP /CPU /ALL
%SMP-I-STOPPED, CPU #01 has been stopped.
%SYSTEM-I-CPUSTOPPING, CPU 1 stopping
%SMP-I-STOPPED, CPU #02 has been stopped.
%SYSTEM-I-CPUSTOPPING, CPU 2 stopping
%SMP-I-STOPPED, CPU #03 has been stopped.
%SYSTEM-I-CPUSTOPPING, CPU 3 stopping
%SMP-I-STOPPED, CPU #04 has been stopped.
%SYSTEM-I-CPUSTOPPING, CPU 4 stopping
%SMP-I-STOPPED, CPU #05 has been stopped.
%SYSTEM-I-CPUSTOPPING, CPU 5 stopping
```

VSMP tool also provides a function to display VAX MP configuration details from inside VMS:

```
$ VSMP :== $SYS$SYSROOT:[VSMP]VSMP
$ VSMP QUERY
VSMP tool version: 1.0
SIMH VAX MP simulator API version: 1
Number of virtual processors configured: 6
Processor IDs mask: 0000003F
Advised host SMT slow-down factor: 1.8
Advised host turbo factor: 120%
vSMP revision level: 1
vSMP options: 3
     portable interlock
     native interlock
VAX virtual machine provider: SIMH VAX MP 3.8.3
Synchronization window(s) active: none
```

VMSP tool has a number of other functions and options that are described below.

Step 7. Invoke VSMP in SYSTARTUP_VMS.COM.

Once you are satisfied with stability of your VAX MP setup, you can incorporate commands to automatically enable multiprocessing into your SYSTARTUP_VMS.COM file. You will need to include two commands:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD
$ IF F$GETSYI("AVAILCPU_CNT") .GT. 1 THEN START /CPU/ALL
```

Note that if system disk image with the above commands in SYSTARTUP_VMS is used to boot on a uniprocessor SIMH that does not implement VAX MP API or on real hardware VAX or with SYSGEN settings to load uniprocessor rather than multiprocessor OpenVMS kernel image, VSMP$LOAD.COM will display a message and exit gracefully, i.e. will not abort the calling SYSTARTUP_VMS.COM.

If desired, you can move execution of VSMP$LOAD.COM and starting the CPUs to an earlier stage in startup process, to speed up the startup by allowing parallel execution of the processes, however VSMP$LOAD.COM should be invoked *after* startup executes SYSGEN AUTOCONFIGURE ALL.

VSMP commands
description and reference

VAX MP OpenVMS multiprocessing is enabled and controlled primarily by VSMP tool. In most cases VSMP need not be and is not invoked by OpenVMS system administator directly. Rather, all that system administrator needs to do is to invoke command file VSMP$LOAD.COM either from SYSTARTUP_VMS.COM or interactively after system reboot:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD
```

or

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD  [optional parameters for VSMP LOAD command]
```

Nevertheless VSMP commands are described further for completeness and reference.

To invoke VSMP tool commands define DCL symbol for the tool:

```
$ VSMP :== $SYS$SYSROOT:[VSMP]VSMP
$ VSMP command
```

Brief synopsis of VSMP commands is:

```
VSMP QUERY

VSMP LOAD [IDLE=ON|OFF|NEVER] [TIMESYNC=ON|OFF]
          [NOPATCH=(list)]
          [XQTIMEOUT=<nsec>] [INTERLOCK=PORTABLE|NATIVE]
          [SYNCW=(SYS|ILK|SYS,ILK|ALL|NONE)]
          [SYNCW_SYS=pct] [SYNCW_ILK=pct]

VSMP SET [IDLE=ON|OFF] [TIMESYNC=ON|OFF]

VSMP SHOW [IDLE] [TIMESYNC]

VSMP SET AFFINITY { device | device-pattern }
                  /CPU={PRIMARY|ALL}

VSMP SHOW AFFINITY { device | device-pattern | /ALL }
```

VSMP QUERY displays information about VAX MP multiprocessor configuration.

VSMP LOAD loads VSMP kernel-resident module into OpenVMS kernel and sets up OpenVMS ready for multiprocessing.

VSMP SET and VSMP SHOW modify and display some of VSMP settings.

VSMP SET AFFINITY and VSMP SHOW AFFINITY set and display affinity of IO devices to CPUs.

Besides enabling and controlling multiprocessing, VSMP also provides useful auxiliary feature called TIMESYNC that synchronizes OpenVMS system time with host time. TIMESYNC works effectively as NTP client, except that it queries time not over the network from time server but rather directly from host. Use of TIMESYNC is optional and can be disabled, but it is enabled by default. TIMESYNC addresses the

tendency of SIMH idea of system time (such as OpenVMS system time, when OpenVMS is executed on SIMH) to fall behind wall clock time.

> This "falling behind" happens because of a number of technical and semi-technical reasons: host timer events are fired later then requested; it takes time for the hibernating thread to resume and start tracking time; accumulating errors in rounding time data obtained from the host, SIMH timing mechanisms dependence on VCPU calibration (i.e. instructions per second rate) which is subject to variations because of concurrent host load and also inherently – because of inherent variations in the composition of executed instruction set, page faults, synchronous IO operations executed on VCPU thread etc.; because of OpenVMS suspension while SIMH console is in use or ROM console is in use; because of OpenVMS suspension while host system was suspended and similar reasons.

VAX MP eliminates some of these sources of guest OS timing distortion[1], but not all of them, and OpenVMS system time drift vs. wall clock is still possible. TIMESYNC is intended to fix this drift and to keep OpenVMS time in sync with host time.

TIMESYNC runs in OpenVMS kernel, as a part of kernel-resident VSMP module.

Every 5 seconds TIMESYNC compares OpenVMS system time with host time. If a divergence of more than 1 second is detected, then TIMESYNC initiates adjustment process. Whenever possible, TIMESYNC adjusts OpenVMS time gradually by increasing or reducing the size of OpenVMS system clock tick and spreading system time adjustment over the next 5-second interval (thus accelerating or slowing down virtual OpenVMS time flow over this interval), rather than performing the adjustment in one big abrupt jump. Gradual adjustment is possible for OpenVMS time falling behind the host time by up to approximately 50 hours or running ahead of host time by up to 5 seconds. Divergences within this range will be handled by TIMSYNC by performing gradual adjustment over next 5-second interval. Divergences outside of this range will be handled by TIMSYNC by performing one big abrupt adjustment, rather than gradually.

TIMESYNC is enabled by VSMP LOAD command by default, however it is possible to disable it. In most cases OpenVMS system administrator will want to have TIMESYNC enabled. However in case OpenVMS installation runs NTP client obtaining time from another source, such as from NTP server over the network, then TIMESYNC should be disabled since TIMESYNC and NTP client will interfere and be at conflict with each other. Alternatively, TIMESYNC can be left enabled and NTP client be disabled instead, leaving time adjustment job to TIMESYNC, assuming host time itself is synchronized from time server or believed to be of satisfactory precision and TIMESYNC 1-second drift tolerance is also deemed to be sufficient.

The rest of this section describes individual VSMP commands.

[1] E.g. by using dedicated clock strobing thread rather than regular SIMH timing mechanism based on counting VCPU cycles.

VSMP QUERY

VSMP QUERY command displays information about VAX MP configuration. It can be executed at any time, either before or after VSMP LOAD.

For example:

```
$ VSMP :== $SYS$SYSROOT:[VSMP]VSMP
$ VSMP QUERY
VSMP tool version: 1.0
SIMH VAX MP simulator API version: 1
Number of virtual processors configured: 6
Processor IDs mask: 0000003F
Advised host SMT slow-down factor: 1.8
Advised host turbo factor: 120%
vSMP revision level: 1
vSMP options: 3
     portable interlock
     native interlock
VAX virtual machine provider: SIMH VAX MP 3.8.3
Synchronization window(s) active: none
```

If VSMP QUERY is executed on a system that does not support VAX MP API, such as regular SIMH VAX simulator or hardware VAX, it will display a message and exit with status code VSMP_MSG_SYS_NOT_VAXMP (%XF1B800C).

VSMP LOAD

VSMP LOAD loads VSMP kernel-resident module into OpenVMS kernel and sets up the system ready for multiprocessing. In particular it creates descriptors for additional virtual processors in the OpenVMS system and exposes VAX MP multiprocessing capability to OpenVMS. Once VSMP LOAD is executed, created virtual processors are ready to be started with OpenVMS DCL command START /CPU.

System administator normally does not invoke VSMP LOAD command directly, rather this command is invoked by executing command file VSMP$LOAD.COM

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD  [optional parameters for VSMP LOAD command]
```

VSMP$LOAD.COM performs three actions:

- It checks whether it is being executed on VAX MP rather than system that does not provide VAX MP API, such as hardware VAX or regular SIMH VAX simulator or other VAX simulator. If VAX MP API is not available, VSMP$LOAD will display a message and exit gracefully with exit status “success” and thus not aborting the calling command file such as SYSTARTUP_VMS.COM

- VSMP$LOAD then sets up appropriate affnity for IO devices on the system by calling file DEVICES.COM that executes VSMP SET AFFINITY commands for various devices on the system. DEVICES.COM is derived from knowledge about particular OpenVMS device drivers[1] and should not be modified except by VAX MP developers.

- VSMP$LOAD then issues VSMP LOAD command, passing it any (optional) arguments that were specified on VSMP$LOAD command line. If VSMP LOAD fails, VSMP$LOAD will display a message and exit gracefully with exit status “success” and thus not aborting the calling command file such as SYSTARTUP_VMS.COM.

If arguments to be passed to VSMP LOAD via VSMP$LOAD command line are too long or exceed 8 arguments, additional arguments can also be passed via DCL symbol VSMP$LOAD_OPTIONS.

Synopsis of VSMP LOAD command is:

```
VSMP LOAD [IDLE=ON|OFF|NEVER]
          [TIMESYNC=ON|OFF]
          [NOPATCH=(list)]
          [XQTIMEOUT=<nsec>]
          [INTERLOCK=PORTABLE|NATIVE]
          [SYNCW=(SYS|ILK|SYS,ILK|ALL|NONE)]
          [SYNCW_SYS=pct]
          [SYNCW_ILK=pct]
```

All arguments are optional.

[1] As described in section “Device timeouts” in “VAX MP Technical Overview”.

IDLE sets whether VCPU threads should hibernate at host OS level when VMS is executing idle loop. IDLE=ON enables hibernation. IDLE=OFF disables hibernation and causes threads to spin in OpenVMS idle loop without attempting to hibernate. IDLE=NEVER disables hibernation and instructs VSMP even not to hook into OpenVMS idle loop. If IDLE setting is specified as ON or OFF for VSMP LOAD command, its value can be later dynamically changed with VSMP SET command. If it is specified as NEVER, its value cannot be subsequently changed until the system is rebooted.

If ILK synchronization window is used (SYNCW=ILK or SYNW=ALL), IDLE cannot be specified as NEVER, and value of NEVER in this case will be overriden to OFF.

TIMESYNC setting enables or disables operation of TIMESYNC feature. TIMESYNC=ON enables it, TIMESYNC=OFF disables it. The value of this setting can subsequently be changed dynamically with VSMP SET command.

INTERLOCK setting selects whether VAX MP will be using portable implementation of VAX interlocked instructions based on thread-level locking or native implementation, utilizing host-specific interlocked instructions and not using locking thread-level wherever possible and is thus potentially more performant and scalable. Portable mode is selected with INTERLOCK=PORTABLE. Native mode is selected with INTERLOCK=NATIVE. See section “Interprocessor synchronization window” for more detailed description of this feature. Use of native interlock mode requires also the use of synchronization window, with SYNCW=ILK or SYNCW=(SYS,ILK). Default it INTERLOCK=PORTABLE.

SYNCW setting selects synchronization windows to be used. See section “Interprocessor synchronization window” for more detailed description of this feature. Default is SYNCW=NONE. SYNCW=ILK selects ILK window. SYNCW=SYS selects SYS window. SYNCW=(SYS,ILK) or SYNCW=ALL select both windows.

SYNCW_SYS and SYNCW_ILK define a percentage of SYS and ILK windows size that can be allocated by VAX MP to permitted interprocessor drift. These parameters are intended for use by VAX MP developers only and their use requires thorough understanding of VAX MP interprocessor synchronization window logics. Default value for both is 66, valid range is 10 to 90.

XQTIMEOUT changes DEQNA and DELQA transmit timeout from default of 5 seconds to a specified higher value. Default is 255 seconds (XQTIMEOUT=255), allowed range is 5 to 255 seconds. If XQ transmit timeout is set or left too short, involuntary VAX MP thread preemption by host OS may cause this timeout to expire and OpenVMS falsely decide that XQ controller failed and shut the controller down. It is recommended to use XQTIMEOUT default of 255 seconds.

NOPATCH=(*list*) instructs VSMP LOAD *not* to install specified patches to OpenVMS kernel and/or drivers. *List* argument is the list of VSMP patch IDs *not* to install, for example: NOPATCH=RESCHED or NOPATCH=(XDELTA, CHSEP). Disabling patches is strongly discouraged – patches are there for a reason and disabling patches may result in unreliable system prone to crashes and/or data corruption or under-performant system. For a list of VSMP patch IDs see the table below.

VSMP dynamic patches

inhibitable with NOPATCH option

XDELTA	This patch is required when using XDelta debugger. XDelta by itself is not aware of multiprocessing capabilities of VAX 3900 and does not know how to store XDelta per-CPU context on this machine. This patch provides XDelta with such knowledge. Without this patch XDelta cannot be properly operated when VAX MP system has more than one processor in active set. This patch is automatically disabled if VSMP LOAD detects that XDelta is not loaded. Delta debugger does not require this patch and, unlike XDelta, can be used without it.
CHSEP	This patch modifies behavior of VMS executive SCH$CHSEP routine to wake up just one rather than all idle CPUs when a process becomes computable. This patch removes overhead of waking up all idle virtual processor on every scheduling event. It can be disabled at the cost of increasing this overhead.
RESCHED	This patch modifies behavior of VMS executive SCH$CHSEP routine in a manner similar to CHSEP patch. Other descriptions from CHSEP patch apply.
LOCKRTRY	This patch modifies the value of SYSGEN parameter LOCKRETRY to enlarge it to 900,000 (unless it was already set higher) to make it consistent with hardwired value used by VMS interlocked queue operations macros. Disabling this patch to use lower value (such as default value 100,000) will tighten inter-CPU synchronization window and degrade VAX MP system scalability.
NUMTIM	This patch corrects SYS$NUMTIM system service. In regular OpenVMS this service (SYS$NUMTIM) retrieves current system time from EXE$GQ_SYSTIME location without acquiring HWCLK spinlock, which may result in incorrect value being read, including totally corrupt value. Probability of this corruption increases when VAX processor is simulated on present-day CPUs, compared to the original VAX. NUMTIM patch provides appropriate spinlock acquisition by SYS$NUMTIM when reading system time variable.
MFYCAP	This patch modifies executive routines SCH$ADD_CPU_CAP and SCH$REMOVE_CPU_CAP to ensure that idle virtual CPUs are waken up after CPU capability modification. In practical terms it ensures that START /CPU and STOP /CPU commands are always executed promptly and shutdown of virtual CPU in idle sleep state is not delayed by a fraction of a second.
UCBTMO, CRBTMO	These patches are currently not in use and can be disabled if requried. They are meant to support extension of timeouts on virtual devices to counteract thread preemption, however at the time no VAX MP virtual device relies on this option.
PU1 – PU7	These patches apply to PUDRIVER (UQSSP port driver) and are used if SIMH MSCP disk (RQ) or MSCP tape (TQ) controllers are configured. These patches must be enabled for MSCP storage port to operate properly in a multiprocessor configuration, otherwise memory and data corruption can result. Patch PU6 can fail to install if MSCP tracing is enabled on the port (typically using some diagnostic tool). In this case stop tracing, run VSMP LOAD to install the patches, then restart the tracing.
XQTIMXMT	This patch extends transmit timeout period on DEQNA and DELQA controllers from the

	XQDRIVER's default of 5 seconds to the value specified by VSMP option XQTIMEOUT (default is 255). If this patch is not applied, and simulator thread operating DEQNA is preempted for more than few seconds, OpenVMS will falsely detect timeout on the controller and shut it down. It is not advisable to operate DEQNA or DELQA without this patch enabled. However if VAX MP is operated in strictly uniprocessor mode, with just primary CPU active, and ASYNCIO on XQ devices is disabled too, then this patch makes no difference. This patch is automatically disabled if VSMP LOAD detects that XQDRIVER is not loaded and there are no XQ devices configured.
XQTX1 – XQTX10, XQRX1 – XQRX4	These patches are required for proper memory synchronization when using DEQNA or DELQA Ethernet controller. They must be enabled when DEQNA or DELQA is in use and XQDRIVER is loaded. Without these patches Ethernet controller cannot be operated when running system in multiprocessor mode (with more than one CPU in active set) and will produce data corruption and other controller failures. These patches are automatically disabled if VSMP LOAD detects that XQDRIVER is not loaded and there are no XQ devices configured.

VSMP SET

VSMP SET command changes the values of VSMP dynamic settings from their initial values established by VSMP LOAD or previous VSMP SET command.

The syntax is:

```
VSMP SET [IDLE=ON|OFF]
         [TIMESYNC=ON|OFF]
```

For example:

```
VSMP SET IDLE=ON
VSMP SET TIMESYNC=OFF
VSMP SET IDLE=ON TIMESYNC=OFF
```

VSMP SHOW

VSMP SHOW displays current values of VSMP dynamic settings.

The syntax is:

```
VSMP SHOW [IDLE] [TIMESYNC]
```

For example:

```
VSMP SHOW
VSMP SHOW IDLE
VSMP SHOW TIMESYNC
```

VSMP SET AFFINITY

VSMP SET AFFINITY changes affinity of specified device or devices to either all CPUs on the system or only to primary processor.

This command is intended for use by VAX MP developers only.
Change of device affinity requires thorough understanding of particular device driver internals.

The syntax is:

```
VSMP SET AFFINITY { device | device-pattern }
                  /CPU={PRIMARY|ALL}
```

Examples of use:

```
VSMP SET AFFINITY TXA7 /CPU=PRIMARY
VSMP SET AFFINITY TXA* /CPU=ALL
VSMP SET AFFINITY T* /CPU=ALL
VSMP SET AFFINITY * /CPU=ALL
```

Note that semicolon is not used in device name or name pattern syntax, i.e. the following is not a valid syntax:

```
VSMP SET AFFINITY TXA7: /CPU=ALL
```
(invalid syntax!)

VSMP SHOW AFFINITY

VSMP SHOW AFFINITY displays affnity of specified device or devices to particular CPUs in the system.

The syntax is:

```
VSMP SHOW AFFINITY { device | device-pattern | /ALL }
```

Examples of use:

```
VSMP SHOW AFFINITY TXA7
VSMP SHOW AFFINITY TXA*
VSMP SHOW AFFINITY TXA
VSMP SHOW AFFINITY T*
VSMP SHOW AFFINITY T
VSMP SHOW AFFINITY *
VSMP SHOW AFFINITY /ALL
VSMP SHOW AFFINITY
```

For example:

```
$ VSMP SHOW AFFINITY D
Local device affinity to CPUs:

  DUA0          ALL
  DUA1          ALL
  DUA2          ALL
  DYA0          PRIMARY
  DYA1          PRIMARY
  DUB0          ALL
  DUB1          ALL
  DUB2          ALL
  DUB3          ALL
```

Note that semicolon is not used in device name or name pattern syntax, i.e. the following is not a valid syntax:

`VSMP SHOW AFFINITY TXA7:` (invalid syntax!)

Interprocessor synchronization window

Interprocessor synchronization window is experimental feature of VAX MP. While it is believed to be stable, it has not been tested yet for VAX MP initial release as thoroughly as running VAX MP with synchronization window turned off. It is recommended therefore for now for most installations and use purposes to use VAX MP without synchronization window enabled, until more testing had been performed and more benchmarking data is available confirming expected benefits of synchronization window.

Interprocessor synchronization window is VAX MP mechanism that constrains the drift of VCPUs relatively to each other to a certain maximum number of instruction cycles.

Without synchronization window enabled VCPUs may “drift” relative to each other due to host scheduler preemption, simulator page faults, host antivirus interventions and other factors outside of VAX MP control. Such drift may cause OpenVMS to signal false timeouts and cause OpenVMS to believe that one of the processors is hung (whereas in fact its VCPU thread had been temporarily preempted by the host scheduler) and cause OpenVMS to bugcheck. For this reason, when synchronization window is not in use, OpenVMS SYSGEN parameter TIME_CONTROL must be set to 6 to disable bugchecks because of SMP timeouts.

Use of synchronizaton window constrains relative drift between the VCPUs. If some VCPU is falling too much behind compared to other VCPUs forward progress, other VCPUs will temporary pause their execution and wait for the lagging VCPU to “catch up”. To minimize impact on scalability, VAX MP synchronization window only constrains the execution of high-IPL code that is likely to designate processors as performing inter-CPU interaction or activities relevant for inter-CPU interaction. Synchronization window will not constrain low-IPL code, in particular user-mode code.

> The only exception is that execution of interlocked queue instructions will momentarily enter VCPU into constrained mode for the duration of the instruction regardless of current IPL or CPU mode, even when interlocked queue instruction is executed in user mode.

Detailed technical explanation of synchronizaton window mechanism and logics is beyond the scope of this document.

> For detailed description of synchronization window refer to “VAX MP Technical Overview”, chapters “Interprocessor synchronization window” and “Implementation of VAX interlocked instructions”.

Very roughly, synchronization window consists of two subcomponents: SYS and ILK. SYS is responsible for constraining the drift between processors holding a spinlock or trying to acquire a spinlock or running other high-IPL code. SYS component constrains the drift between processors in described states to a fraction of a threshold controlled by SYSGEN parameter SMP_SPINWAIT and prevents CPUSPINWAT bugchecks. ILK component is responsible for limiting relative drift between processors executing certain interlocked instruction patterns, such as trying to perform operations on interlocked queues, and is associated with SYSGEN parameter LOCKETRY and prevention of BADQHDR bugchecks.

Use of synchronization window allows TIME_CONTROL to be relaxed from 6 to 2. Full relaxation of TIME_CONTROL to 0 is not supported yet by current release of VAX MP. Relaxation of TIME_CONTROL from 6 to 2 allows OpenVMS to generate bugchecks and restart the system in case one (or some) of the processors lock up while holding a spinlock or stop responding to interprocessor interrupt requests within reasonable amount of time. With OpenVMS being mature and stable, these are unlikely events.

True benefits of synchronization window use lie elsewhere.

* * *

One other benefit of synchronization window is that with synchronization window enabled VAX MP provides greater compatibility with mal-written code that can fail even on real hardware VAXen but is more likely to fail on a simulator where VCPUs are implemented as preemptable threads that can “drift” relative to each other.

We are not aware of any actually existing code of this nature, neither in baseline OpenVMS VAX 7.3, nor in any layered or 3rd party components, however theoretically it can be imagined.

> Consider for example privileged-mode code one section of which executes on VCPU1 at IPL RESCHED (3) and locks certain interlock queue header with BBSSI instruction in order to quickly scan the queue, relying on elevates IPL as a protection from being preempted. Another section of the code, executed on VCPU2, performs $INSQHI on the queue, i.e. INSQHI operation with finite number of retry attempts (limited either to 900,000 retries as hardwired into $INSQHI and associated macros, or by SYSGEN LOCKRETRY parameter, or limited to any other finite number of retries).
>
> Such code can fail even on a real hardware VAX. (And in case of $INSQHI macro generate bugcheck BADQHDR.)
>
> Indeed, although the code on VCPU1 is running at IPL 3, it can still be interrupted by higher-priority interrupt and it might spend some extended time inside processing this interrupt. Default SYSGEN parameters would allow such interrupt handler to last without system crash for up to 1 second. On fast machines, such as NVAX/NVAX+ based machines, both VCPU1 and VCPU2 are legitimately allowed by default SYSGEN parameters to execute about 50 mln. instructions before raising SMP timeout condition – a number well in excess of 2.7 mln. instruction limit hardcoded into $INSQHI and far in excess of much smaller limit controlled by default value of LOCKRETRY. Thus $INSQHI based spinning in the described code can timeout even on a real VAX, within default values of system parameters.
>
> Such code would therefore be mal-written and we are not aware of its actual existence.

Even though such mal-written code is incorrect and prone to failure even on a real VAX, probability of such failure may be small and the code may “pass” on a real VAX due to low probability of factors required to cause its failure, but it may fail with greater probability when executed on a simulator, because VCPU thread preemption on the simulator is much more likely event than long high-priority interrupt processing on a real VAX.

We are not aware of any such code actually existing. If such code were to happen nevertheless, synchronization window comes to the rescue. Synchronization window will not necessarily spare this

code from failing altogether (although in some cases it may), but it will reduce probability of its failure to a level comparable with real hardware VAXen or less.

It must be reiterated that described problem remedied by the use of synchronization window would be caused only by the code that was mal-written in the first place and that to our current knowledge does not actually exist in baseline VMS, and we are also not presently aware of any such code actually existing in any layered or 3rd party products as well.

* * *

But the primary benefit of synchronizaton window is that with synchronization window enabled (more specifically, with its ILK sub-option enabled) VAX MP allows to use host "native" mode to implement VAX interlocked instructions.

VAX MP provides internally two algorithms to implement VAX interlocked instruction set. Default algorithm uses "portable interlock". Another algorithm uses "native-mode interlock" and is expected to be more performant and more scalable on workloads that use high rate of BBSSI, BBCCI and ADAWI instructions, such as workloads that perform a lot of interprocessor synchronization activity, e.g. spinlock acquisition. This includes workflows that induce high rate of IO operations, high paging rate or intensive inter-process synchronization.

> The difference in performance and scalability between native and portable interlock modes is likely to be small when running VAX MP on a dedicated host machine with HOST_DEDICATED set to 1. Native mode is expected to outperform portable interlock mode when running workload with high rate of interlocked instructions on non-dedicated host with HOST_DEDICATED left at 0.

However the use of native mode requires the use of synchronizaton window, more specifically ILK sub-option of synchronization window.

To start VAX MP multiprocessing with native-mode interlock, load VSMP with one of the following option sets:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD INTERLOCK=NATIVE SYNCW=ILK
```

or:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD INTERLOCK=NATIVE SYNCW=ALL
```

Table below summarizes valid combinations of parameters most relevant for synchronization window.

Three most relevant parameters are:

- VSMP LOAD command option INTERLOCK=*value*, where *value* can be PORTABLE or NATIVE.

  Default is PORTABLE. On some host systems that do not provide adequate instruction set for native-mode emulation of VAX interlocked instructions, setting to NATIVE can be impossible. It

is possible for the ubiquitous case of Intel x86/x64 host processors.

- VSMP LOAD command option SYNCW that can be any combination of SYS and ILK: ILK, SYS, (SYS, ILK) also expressible as ALL, and NONE.

- OpenVMS SYSGEN parameter TIME_CONTROL.

Before VSMP tool starts multiprocessing, it performs a check that combination of configuration parameters being used is valid and will refuse any incoherent combination with appropriate diagnostic message.

Valid combinations of these parameters are:

INTERLOCK = PORTABLE SYNCW = NONE, TIME_CONTROL = 6	This is the default setting. Synchronization window is not used. This setting provides the best scalability in portable mode. It is safe with base OpenVMS, but may potentially cause OpenVMS to crash under some code sequences that do not exist in base OpenVMS but might exist in layered or third-party privileged products.
INTERLOCK=PORTABLE SYNCW = ILK, TIME_CONTROL = 6 SYNCW = SYS, TIME_CONTROL = 2 or 6 SYNCW = (SYS,ILK), TIME_CONTROL = 2 or 6	These are the safest settings. These settings engage one or both components of synchronization window. They may slightly reduce scalability and performance compared to the default setting. Use these settings only when running layered or third-party privileged components that are known to crash for SMP-timing related reasons (such as bugchecks CPUSPINWAIT or BADQHDR) under the default configurations. Currently no such components are known.
INTERLOCK=NATIVE SYNCW = ILK, TIME_CONTROL = 6 SYNCW = (SYS, ILK), TIME_CONTROL = 2 or 6	Using native mode. When using native mode, ILK is a must. Native mode is experimental at this point and only limited testing had been performed for it. Native mode is expected to offer less overhead than portable mode when system makes heavy use of BBSSI and BBCCI interlocked instructions, such as heavily using spinlocks, which in turn may occur during heavy paging or intense IO. Actual performance benefits of native mode have not been assessed yet. Native mode may be faster than portable mode on some workloads.

While native mode is expected to offer performance and scalability benefits for workloads generating high rate of BBSSI and BBCCI instructions, the use of native mode may destabilize some user-mode application and third-party privileged components compared to the use of portable interlock mode. Applications and components may fail to work properly under native-mode interlock under any of the following five conditions:

- If the component contains preemptable code (user-mode code or privileged mode code executing at IPL < 3) that uses retry loop count on interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) smaller than min(900000, LOCKRETRY). [1]

- If preemptable code uses finite retry count on interlocked queue instructions *and* queue header also gets locked with BBSSI instruction.

- If preemptable code tries to lock interlocked queue header with BBSSI instruction and uses finite retry count for this BBSSI.

- If non-preemptable code (kernel-mode code executing at IPL >= 3) uses smaller retry count for interlocked queue instructions than is hardwired in VMS macros $REMQHI, $REMQTI, $INSQHI or $INSQTI, i.e. 900,000 times, or set by SYSGEN parameter LOCKRETRY, whichever is less, i.e. smaller than min(900000, LOCKRETRY).

- If non-preemptable code tries to lock interlocked queue header with BBSSI instruction and uses retry count yielding total number of instructions executed in retry loop lesser than 3 * min(900000, LOCKRETRY).

Conversely, to execute properly under native-mode interlock, application should:

- Preemptable code running at IPL < 3 should use retry loop count of min(900000, LOCKRETRY) for interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) executed at IPL < 3.

- Preemptable code should use infinite retry loop for any interlocked queue instructions *if* this queue header also gets locked with BBSSI instruction.

- Preemptable code should use infinite retry loop count when trying to lock interlocked queue header with BBSSI instruction.

- Non-preemptable code running at IPL >= 3 should use retry loop count of min(900000, LOCKRETRY) or more for interlocked queue instructions executed at executed at IPL >= 3.

[1] VSMP LOAD command normally elevates SYSGEN parameter LOCKRETRY to 900000 too, unless prohibited from doing so with command option NOPATCH=LOCKRETRY.

- Non-preemptable code trying to lock interlocked queue header with BBSSI instruction while executing at IPL >= 3 must use BBSSI retry count yielding total number of instructions in retry loop not less than 3 * min(900000, LOCKRETRY). If retry loop drops IPL below 3 in the interim, retry count should be restarted.

Any application or component failing to meet these requirements can only be executed in multiprocessor mode on VAX MP if portable interlock is selected, which is the default setting.

If an application or component not meeting listed requirements is executed on VAX MP with native interlock mode enabled, described retry loops can timeout and cause component or application to fail. They will not timeout when executed on VAX MP using portable interlock mode.

We are not aware at present time of any application or component that fails to meet these requirements. Base OpenVMS system itself meets them. If you are sure all of your VAX software executed under VAX MP meets them too, you may be able to increase system performance by enabling native-mode interlock with the following VSMP LOAD options:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD INTERLOCK=NATIVE SYNCW=ILK
```

or:

```
$ @SYS$SYSROOT:[VSMP]VSMP$LOAD INTERLOCK=NATIVE SYNCW=ALL
```

Note that described limitations of native mode are not “failures” of synchronization window mechanism, but rather reflect its design goals. Synchronization window is designed to provide highest possible scalability of multiprocessor system and impose least constraining synchronization compatible with the needs of operating system. Therefore it constrains only the patterns that OpenVMS is known to actually use or that OpenVMS and well-written applications can reasonably be expected to use. Furthermore, synchronization window constrains these patterns only to an extent (synchronization window size) they need to be constrained to in order to assure their stable execution. Synchronization window by design avoids over-constraining the system and hampering its scalability for the sake of supporting any and all conceivable patterns.

If widest range of synchronized patterns is a desired goal, it is recommended to use portable interlock mode, or better yet, portable interlock mode in conjunction with synchronization window. The latter combination may produce slightly less scalable configuration, but will provide support for a wider set of synchronization patterns.

VAX MP SIMH console command set changes and extensions

VAX MP introduces a number of extensions and changes to SIMH console command set.

CPU MULTIPROCESSOR

Command **CPU MULTI{PROCESSOR} <n>** creates virtual processors in SIMH. For example the following console commands create 6 virtual processors on the system, i.e. extra five CPUs in addition to the primary CPU #0, and start the primary processor:

```
sim> CPU MULTI 6
sim-cpu0> BOOT CPU
```

CPU ID

Command **CPU ID <n>** sets default context for execution of other console commands. Console user can switch between processor instances while examining and modifying their state:

```
sim-cpu0> EXAMINE PSL
[cpu0]  PSL:    041F0000
sim-cpu0> CPU ID 2
sim-cpu2> EXAMINE PSL
[cpu2]  PSL:    03C00009
```

Thus, for example, commands

```
CPU ID 2
EXAMINE STATE
DEPOSIT PSL 041F0004
STEP
```

will display the state of CPU2, change PSL of CPU2 and execute single instruction on CPU 2. Explicit CPU unit specification in a command, such as for example `EXAMINE CPU3 STATE` overrides default context.

Reference to device “CPU” without specifying explicitly its unit number means a reference to currently selected CPU, for example

```
CPU ID 2
EXAMINE CPU STATE
```

means a reference to the state of CPU2, not CPU0.

Device and register *names* do not depend on selected CPU and are the same for all CPUs. Command SHOW DEVICES will list the same device names set for every CPU, regardless of currently selected CPU context. However depending on the context, name can refer to different device or register *instances*.

Some of devices or their registers are system-wide (such as for example RQ or TS devices), and listings appearing in the context of different current CPUs will refer to the same global, system-wide instance of the device, whereas other devices (such as TLB or CLK) will be per-CPU and any references to such devices mean a reference to device *per-CPU instance*.

To help user easily distinguish between per-CPU registers and system-wide registers, console commands such as EXAMINE had been modified to display CPU ID tag when displaying a register value, for example:

```
sim-cpu0> CPU ID 2
sim-cpu2> EXAMINE QBA STATE
[cpu2]  SCR:    8000
[cpu2]  DSER:   00
[cpu2]  MEAR:   0000
[cpu2]  SEAR:   00000
        MBR:    00000000
[cpu2]  IPC:    0000
[cpu2]  IPL17:  00000000
[cpu2]  IPL16:  00000000
[cpu2]  IPL15:  00000000
[cpu2]  IPL14:  00000000
```

In this example, MBR register is system-wide, while other displayed registers are per-CPU and the tag displayed in front of them indicates ID of the processor the value is displayed for.

Console command **CPU ID** without a parameter displays currently selected default processor for other console commands context:

```
sim-cpu2> CPU ID
Current default CPU: cpu2
```

Instruction history recording

Instruction history recording had been modified to support simultaneous history recording for multiple virtual processors.

VAX MP provides SIMH console user with two recording options.

One option is unsynchronized recording where each processor records the history of its instruction stream independently from other processors. Recorded histories are strictly per-CPU and are uncorrelated with each other and there is no telling about relative real-time order of executions of instructions on different processors, i.e. whether certain instruction had executed on CPU A before or after certain other instruction executed on CPU B. Recorded histories are locally sequenced for each processor, but they are not globally sequenced.

The other option provided by VAX MP is to record globally sequenced history of instructions executed by all the CPUs. Globally sequenced recording is somewhat slower than uncorrelated recording, but it allows one to examine global order of execution and inter-processor interaction.

Desired recording method is selected by user via VAX MP SIMH console command, for example

```
SET CPU HISTORY=200/SYNC
```

selects recording of globally sequenced history, while

```
SET CPU HISTORY=200/UNSYNC
```

selects recording of locally sequenced per-CPU histories that are not synchronized across the processors. Default is SYNC.

Command SHOW CPU HISTORY by default displays recorded history for current default processor, i.e. one selected by command CPU ID. To display globally sequenced instruction streams history for all processors, rather than just currently selected processor, use console command

```
SHOW CPU HISTORY=SYNC
```

or

```
SHOW CPU HISTORY=10/SYNC
```

to display ten last commands from the whole system-wide history of instruction streams.

Without SYNC option, SHOW CPU HISTORY command applies to currently selected processor's instruction stream only.

VAX MP also provides a setting that allows instruction history recording be stopped when processor executes BUGW or BUGL pseudo-instruction. This is helpful for debugging VMS system crashes, since recording gets stopped at the point bugcheck was generated, and history buffer is not flushed by subsequent instructions. To activate this setting, execute SIMH console command

```
DEPOSIT  BUGCHECK_CTRL  1
```

Breakpoints

SIMH breakpoint facility had been extended for VAX MP to support multiprocessing.

Multiple nearly-simultaneous breakpoint hits on multiples processors are supported. During execution of breakpoint actions, CPU context is set to the CPU that hit a particular breakpoint. If multiple CPUs hit multiple breakpoints almost simultaneously, all triggered breakpoint actions will be invoked in their appropriate CPU contexts.

The logic for breakpoint handling in multiprocessor case is as follows. Once a breakpoint is triggered on one or more processors, all processors are stopped and control is transferred to the console thread. Console thread examines processors for triggered breakpoints and builds in-

memory script with the following content (assuming, by the way of example, that processors 3 and 4 triggered breakpoints and console's previous default context was CPU0):

```
CPU ID 3
… breakpoint action commands triggered by CPU3 …
CPU ID 4
… breakpoint action commands triggered by CPU4 …
CPU ID 0
```

Console thread then executes this script. If any of the invoked actions contained CONT(INUE), it will not be executed immediately, but recorded as a flag till all other commands specified in the actions complete. If "continue" flag is set after the execution of the script, console thread will execute CONT(INUE) after the whole script had been performed.

STEP, on the other hand, is executed immediately, in-sequence with respect to other breakpoint action commands. This allows to define breakpoint handling scripts that examine the state, execute STEP on the paused processor and examine the state again after the step.

If execution of STEP triggers another breakpoint, console thread will build another script and execute it as a nested script. There can be a stack of nested scripts for breakpoint invocations. As a safety measure against runaway scripts, script nesting depth is limited.

Commands RUN, GO and BOOT cannot be executed directly or indirectly (in the invoked script files) from a breakpoint action context. Attempt to execute them will be rejected with diagnostic message.

If breakpoint action invokes STEP command, it will be executed for the processor that triggered the breakpoint. If multiple processors triggered breakpoints with STEP in the bound actions, STEP will be executed for all of them.

If breakpoint action invokes CONT(INUE), this command will be executed after all other commands listed in all triggered actions complete.

WS_LOCK, WS_MIN, WS_MAX

Significant paging of simulator's code or data[1] during multiprocessor execution may negatively affect performance of the system. If VCPU thread experiences page fault while holding a resource lock, other VCPU threads may have to wait till page fault is resolved. It is therefore

[1] Including simulator's own code and data, code and data of runtime libraries used by the simulator and host memory region that holds VAX "physical" memory.

desirable to keep simulator's paging rate to a minimum. Note that this is not required for correctness of operation, but only for better scalability and performance.

VAX MP provides user-controllable mechanisms intended to reduce paging. These mechanisms are specific to host operating system.

Note that these mechanisms are not intended to prevent paging altogether, but merely to reduce paging rate to a level that does not cause any significant degradation of system scalability and responsiveness. Some paging will always happen – at a minimum of pageable operating system code and paged system pool accessed on behalf of the simulator's process.

Under Microsoft Windows, VAX MP provides a way to increase working set size of the simulator process. Working set size is controllable with two configuration variables accessed via SIMH console commands DEPOSIT and EXAMINE. These variables, named WS_MIN and WS_MAX, define minimum and maximum size of VAX MP process working set expressed in megabytes. As a rule of thumb, it is recommended to set WS_MIN to the size of VAX "core memory" as defined by SET CPU command (e.g. SET CPU 256M) plus extra 40 MB for WS_MIN and extra 100 MB for WS_MAX. For example, for 256 MB VAX memory configuration, perform

```
sim> DEP  WS_MIN  300
sim> DEP  WS_MAX  360
```

Alternatively and easier, setting variable WS_LOCK to 1 causes both WS_MIN and WS_MAX being set to appropriate values:

```
sim> DEP  WS_LOCK  1
```

Setting WS_LOCK to 1 will also automatically set WS_MIN and WS_MAX to heuristically reasonable values, as described above. However if you want fine-grain control, you can set WS_MIN and WS_MAX directly instead. Note that setting either WS_MIN or WS_MAX to non-zero value will also set WS_LOCK to 1.

Linux does not have a notion of per-process working sets. It has one system-global working set for all the processes in the system and the operating system itself. LRU-like page replacement is performed on this system-wide working set as a whole. It is possible to lock process pages into this working set and thus effectively in memory. This is achieved by "DEPOSIT WS_LOCK 1" as described above. For locking to work on Linux, caller's account must have high RLIMIT_MEMLOCK (definable on per-account basis in `/etc/security/limits.conf`), or VAX MP executable file should be granted privilege CAP_IPC_LOCK.

If you want to have a cross-platform VAX MP configuration script file usable both on Windows and Linux machines, you can define either WS_MIN/WS_MAX or WS_LOCK in it. Setting WS_MIN or WS_MAX to non-zero value causes WS_LOCK to be set to 1, and conversely, setting WS_LOCK to 1 causes WS_MIN and WS_MAX being set to a heuristically calculated values.

On OS X setting either of WS_MIN, WS_MAX or WS_LOCK to non-zero value causes VAX core memory block to be pre-paged when VAX VCPUs execution starts or resumes, but no memory locking or system-level alterations to working set size are performed.

If VAX MP is unable to perform extending working set or locking code and data in memory, it will display warning message at first BOOT CPU command, or first RUN, CONTINUE, GO and STEP command.

CPU SMT

Command CPU SMT establishes override value of SMT/hyperthreaded slow-down factor that is reported by VAX MP to VSMP and is used by VSMP to adjust calibration of OpenVMS timing loops. See section “Miscellaneous notes” later in this document, subsection “Using VMware, VirtualBox and other virtual environments”.

CPU INFO

Command CPU INFO displays VAX MP state information of use for a developer, including summary of current VCPU modes, IPL levels and pending interrupts, synchronization window etc. For example:

```
sim-cpu0> cpu info

CP                   MO IP LI  THRD  ACLK
U#    PC      PSL    DE L# RQ  PRIO  AIPI
-- -------- -------- -- -- -- ------ ----
00 8E9CC856 04080008 IK  8  0 OS  42
01 8E93A566 00C20008 KU  2  0 RUN 42
02 8E9576AF 04C80000 IU  8  0 OS  42
03 00000E7A 03C00001 UU  0  0 RUN 42
04 (STANDBY)
05 (STANDBY)

CP  External pending interrupts
U#  Local and buffered interrupts
-- -------------------------------------
00 XT: NEW SYNCLK ASYNC_IO
   LC: (none)
01 XT: NEW SYNCLK
   LC: (none)
02 XT: NEW SYNCLK
   LC: (none)
03 XT: NEW SYNCLK
   LC: (none)

Interlock mode: native
Active synchronization windows: SYS,ILK
SYS and ILK check interval / window size: 89100 / 891000
Synchronization window quant / maxdrift:  89100 / 712797

CPUs active in synchronization window:

CP  active    rel       wait   wt
U#    in      pos       seq#   on         waited by
-- ------- ---------- -------- -- ---------------------------
00     SYS          0
02 SYS,ILK          0
```

PERF

VAX MP adds console command to monitor VAX MP performance counters. This information is valuable for tuning VAX MP and is intended mostly for VAX MP developers. Format of the command and its options is:

```
PERF [ON / OFF / SHOW / RESET] [counter-id]
```

ON enables specified counter or counters.
OFF disables it.
RESET resets accumulated counter data.
SHOW displays accumulated values.

PERF with no arguments is equivalent to PERF SHOW.

If *counter-id* is omitted, operation is performed on all counters defined in the simulator.

Currently performance counters are implemented for VAX MP internal locks.

Miscellaneous notes

Do not set SYSGEN MULTIPROCESSING = 4

Do not set SYSGEN parameter MULTIPROCESSING to 4. OpenVMS VAX 7.3 and possibly some earlier versions have a known problem manifesting itself when the system is booted with MULTIPROCESSING set to 4. The problem is unrelated to VAX MP and manifests itself even on a regular uniprocessor SIMH. One manifestation of the problem is system crash triggered when a disk is dismounted, often resulting also in volume's home block being wiped out and total disk data loss.

Always use MULTIPROCESSING = 2 for multiprocessor execution or MULTIPROCESSING = 3 or 0 for uniprocessor execution.

Use of DZ multiplexor

If you have a system disk with SYSGEN parameter MULTIPROCESSING set to 2 and want to boot it on regular uniprocessor SIMH VAX simulator version 3.7.x or earlier, disable DZ multiplexor device using SIMH console command "set dz disable" or, if you leave it enabled, do not try actually using DZ device.

Older versions of SIMH VAX incorrectly define DZ interrupt level. This will case OpenVMS to crash if a use of DZ use is attempted when MULTIPROCESSING is set to 2, once the device actually fields an interrupt to the processor. The problem is fixed both in VAX MP and regular uniprocessor SIMH version 3.9 and late 3.8-2 builds, which are safe to interchange system disk images with VAX MP. If you want to use VAX MP disk image with early 3.8 SIMH VAX or earlier versions, make sure either to disable DZ or boot with MULTIPROCESSING set to 0 or 3. The problem is specific to DZ and does not affect VH multiplexor. It is safe to boot OpenVMS disk image with MULTIPROCESSING=2 on older SIMH VAX versions that have VH, but not DZ configured.

Using VMware, VirtualBox and other virtual environments

VAX MP can be executed under VMware and similar virtualization hypervisors running on top of host operating system, however this configuration is not recommended since hypervisor will be unable to effectively propagate VAX MP threads priority settings and dynamic priority changes to the host system and thus host system in combination with the hypervisor will be unable to schedule VAX MP virtual processors properly. This may result in losses of CPU resources to unnecessary lock-holder busy-waits and other spin-waits and thus lower overall performance and scalability of the system, slowdown of critical operations and poorer interactive response compared to VAX MP execution directly on the host system. Furthermore, SMP hypervisors introduce overhead related to virtual processor synchronization and thus total overhead will be higher when running VAX MP inside the hypervisor than directly on the host system.

These issues do not apply to VAX MP executing under a hypervisor running directly on top of hardware, rather than on top of host operating system.

Nevertheless it is possible to run VAX MP under a hypervisor executed on top of host operating system.

Some hypervisors may report incorrect host processor topology to guest operating system and misrepresent hyperthreaded/SMT units as separate cores. For example VMware 7.x executed under Windows misrepresents hyperthreaded units to Linux guest as separate cores. If left uncorrected this could cause OpenVMS to undercalibrate its timing loops. VAX MP allows to check for this situation and correct it. When running VAX MP under host OS based hypervisor, before starting VSMP (i.e. before VSMP LOAD command or executing VSMP$LOAD.COM) execute command VSMP QUERY and check for reported value of “Advised host SMT slow-down factor”:

```
$ VSMP QUERY
[…]
Advised host SMT slow-down factor: 1.8
[…]
```

On processor with non-hyperthreaded cores, reported value should be 1.0. On 2-way hyperthreaded processor reported value should be 1.8, consistent with expectation that hyperthreaded core normally operates not at 2x speed of an independent core but only at 1.2-1.3x its speed and thus effective slow-down of hyperthreaded core compared to two independent cores is 1/1.2-1.3 or approximately 1.8.

If you are executing VAX MP under a hypervisor on a 2-way hyperthreaded host processor and see slow-down factor reported as 1.0, this means that hypervisor misrepresents processor topology to a guest operating system executed inside the hypervisor and on top of which VAX MP is being executed.

To correct this situation add the following command to your VAX MP startup script that overrides the value implied by incorrect processor topology exposed by the hypervisor:

```
CPU SMT 1.8
```

Console ROM prompt

MicroVAX 3900 console ROM is not SMP-aware and can wreak havoc on the system if executed while multiprocessing is active. For this reason VAX MP disables access to console ROM (>>> prompt) when more than one virtual processor is currently active. It is possible to access console ROM when only one processor is active, even after SMP had been enabled; however if multiple processors are currently active, BREAK action (such as telnet BREAK command or console WRU character such as Ctrl/P) will be ignored. Access to console ROM prompt is reenabled when primary VCPU is the only CPU in the active set, such as before START /CPU command, after STOP /CPU/ALL command, after system shutdown or during and after system bugcheck.

One firmware console function that is sometimes needed during system operation is the access to VAX SIRR register (software interrupt request register) that is used to request VMS to cancel mount verification on a device or trigger XDELTA debugger.

Normally these requests would be triggered by using console ROM to write to SIRR, such as:

```
>>> D/I  14  C
```

Since firmware console is inaccessible when multiprocessing is active on VAX MP, SIRR access functionality had been replicated in VAX MP SIMH console.

To trigger XDELTA during multiprocessor execution of OpenVMS (and assuming XDELTA is loaded), bring up VAX MP SIMH console using Ctrl-E, then perform:

```
sim-cpu0> DEP SIRR E
sim-cpu0> CONT

1 BRK AT 800027A8
800027A8/NOP
```

To access mount verification cancellation prompt, do:

```
sim-cpu0> DEP SIRR C
sim-cpu0> CONT

IPC> C DUA1
```

XQDRIVER

DEQNA and DELQA Ethernet controllers require a modified version of their OpenVMS driver (XQDRIVER) for proper execution in multiprocessor mode. VSMP LOAD command modifies in-memory active copy of XQDRIVER, without modifying it on disk.

If DEQNA or DELQA Ethernet controllers are present on the system, XQDRIVER must be loaded before VSMP LOAD command is executed, otherwise required corrections won't be applied to the driver. Don't load or reload XQDRIVER after executing VSMP LOAD, otherwise reloaded original version of the driver will miss required corrections. Under OpenVMS 7.3 XQDRIVER is not unloadable (reloadable) anyway, however it may be reloadable or unloadable in earlier versions of OpenVMS if controller initialization failed. Do not attempt to unload or reload XQDRIVER after executing VSMP LOAD.

MONITOR

If MONITOR utility is executing while VSMP should be loaded, stop MONITOR before performing VSMP LOAD and restart MONITOR after completing VSMP LOAD. Otherwise MONITOR will not

see additional processors added to the system and will under-collect performance data, at best, or may even crash.[1]

On a 32-processor system MONITOR utility collects and displays load information only for first 31 processors. While 32nd processor is there and is performing system workload, MONITOR will not see it and will not collect load information for it.

ANALYZE /SYSTEM

Similarly, if SDA utility is running on a live system while VSMP LOAD command is being executed, restart SDA after completing VSMP LOAD to let it see additional CPUs added to the system.

Watchpoint utility and WPDRIVER

Use of WPDRIVER is unsupported in the current release of VAX MP when multiple processors are active. It is safe to use WPDRIVER on VAX MP when only one processor is active, but not when multiple processors are active.

WPDRIVER references WHAMI macro. This macro is not aware of VAX MP multiprocessing capabilities and will incorrectly treat it as uniprocessor MicroVAX 3900, resulting in improper operation and possible system crashes. It may be possible in the future to provide a patch for WPDRIVER enabling its use in multiprocessor mode on VAX MP, should there be a need for such use that justifies required effort.

Running cluster

When running VAX software simulator, whether uniprocessor or multiprocessor, as a member of a cluster, it is prudent to adjust the values of SYSGEN parameters TIMVCFAIL and RECNXINTERVAL across the cluster. TIMVCFAIL governs timeout interval for cluster virtual circuits. If a cluster member does not receive acknowledgement from another member within TIMVCFAIL interval, it considers the circuit broken and initiates reconnection attempt that lasts up to maximum interval indicated by RECNXINTERVAL. Once RECNXINTERVAL expires, remaining nodes abandon attempt to reestablish connection with the lost node and proceed to rebuild the cluster without the lost node or nodes.

Unlike with silicon VAX, simulator-based VAX (or Alpha) node response can be delayed by simulator's thread preemption due to host system load, spin-up of disks previously spun down according to host power saving policy, intervention of antivirus software and so on. To prevent false circuit breakdown induced by these factors, it is prudent to increase the value of TIMVCFAIL from default of 1600 (corresponding to 16 seconds) to 3000 (corresponding to 30 seconds). There is no point in further increasing TIMVCFAIL beyond the value 3000 as NI port

[1] This issue can potentially be fixed later by converting resident part of VSMP to SYSMAN SYS_LOADABLE image, however that would present its own inconveniences.

driver has hard maximum limit of 30 seconds hardcoded. RECNXINTERVAL can be set to any practical value.

These guidelines apply regardless of whether any simulator-based cluster node is running in multiprocessor or uniprocessor mode.

Volume shadowing

We strongly caution against the use of host-based volume shadowing in OpenVMS 7.2 and 7.3. Shadowing code has severe bugs that were introduced in OpenVMS VAX 7.2 and are present in 7.3 and may also be present in service patches for OpenVMS VAX 7.1 (but not in base OpenVMS 7.1). These VAX-specific bugs (OpenVMS Alpha is not affected) are in volume shadowing code source module [SYS.SRC]WBM_RTNS.C that makes multiple read references to system time cell (EXE$GQ_SYSTIME) not properly protecting these references by HWCLK spinlock. These bugs can lead to data corruption and system crashes. The opportunity for these bugs to be triggered occurs on each overflow of low 32-bit word containing system time and carry over into high 32-bit word, which happens approximately every 6 minutes.[1]

These bugs can be triggered even on a uniprocessor system, including both simulator and real silicon VAXen, but are more likely to show up on a multiprocessor system, and are even more likely to show up when OpenVMS is executed on a simulator running on a modern host machine with weaker memory consistency model than silicon VAX.

It might be possible to provide a static patch or dynamic patch for correction of these bugs in a subsequent release of VAX MP. However for now we strongly suggest to avoid using volume shadowing when running OpenVMS VAX 7.2 or 7.3. If you need data redundancy, rely on host based functions such as RAID and/or SAN and host based multi-pathing.

Disk data integrity

OpenVMS does not provide proper means of flushing disk cache. OpenVMS VAX system services, including SYS$FLUSH and non-cached disk QIOs, do not implement an explicit fence causing disk-level flush command to be emitted. MSCP protocol provides FLUSH command, but it is not called by OpenVMS VAX itself and is not exposed to higher levels of application software.

VMS assumes instead that *any* disk write completed at QIO level had been physically committed to disk – not just disk controller and controller's cache, but actual persistent storage on the hard

[1] WBM_RTNS is not the only VMS module that accesses EXE$GQ_SYSTTIME in unprotected fashion. Although most references to EXE$GQ_SYSTTIME in VMS code had been converted to READ_SYSTIME macro that acquires proper spinlock for reading EXE$GQ_SYSTIME, some secondary routines still do use unguarded direct access to EXE$GQ_SYSTIME. Luckily most of these remaining accesses are relatively benign, such as in accounting code, and at worst can distort accounting data. However WMB_RTNS.C makes a non-benign use of unprotected access to system time cell. Another system component that is making potentially harmful use of unprotected access is SYS$NUMTIM system service, which gets dyn-patched by VSMP tool to correct the problem.

drive. In other words, OpenVMS and VMS applications assume full write-through semantics of *all* disk writes and rely on this semantics for data integrity.

SIMH by default uses write-back disk caching semantics, in order to improve virtual disk IO performance. This means that if host system crashes or loses power, guest OS data and applications data can be left corrupted.

If you need to run applications that require high level of data integrity, such as databases, you should explicitly put disks in write-through mode. This feature is available in VAX MP as well as regular uniprocessor SIMH in versions coming after 3.9.

Disks with write-through semantics have lower IO performance than disks with write-back semantics, therefore to maintain general system performance, it is recommended to keep data requiring high integrity on separate virtual drive(s) and put only those drive(s) in write-through mode, while leaving system disk and disk with paging and swap files in write-back mode (but do keep system disk standby backup copy).

To put disk drive in write-through mode, attach it as “raw” format.
For example:

```
set rqb0 format=raw
attach rqb0 mydb.vdisk
```

or as a single command (note exact order of arguments!):

```
attach rqb0 –f raw mydb.vdisk
```

Note that write-through operations with RAW format are reliable under Windows, however under Linux they are subject to deficiencies in Linux implementation of write-through semantics: some (many/most) Linux implementations will only flush the write to the controller, but will not wait for the writes being flushed from the controller cache to persistent media.[1]

Scalability

Multiprocessor scalability of an application or application mix depends to a great extent on the application and host system properties. In any event, scalability will be limited by Amdahl law. Compute-bound applications with little or no dependency between processes can be expected to scale close to linear, subject to the bandwidth of host machine’s memory bus and locality of application memory references enabling or disabling efficient caching. Heavy paging however would introduce implicit dependency between processes contending for virtual memory system (for structures protected by serialization on MMG spinlock and for disk IO to or from paging file).

[1] http://www.mail-archive.com/linux-kernel@vger.kernel.org/msg272085.html
http://www.spinics.net/lists/linux-scsi/msg38592.html
http://milek.blogspot.com/2010/12/linux-osync-and-write-barriers.html

IO bound application cannot be expected to scale, since they will be largely limited by the bandwidth of disk IO subsystem and by serialization on the IO device, including serialization within the device itself and serialization by system locks associated with the device (device spinlock and fork queue lock) held during IO request processing and also by OpenVMS processing of interrupts and kernel-mode IO postprocessing that happens on the primary processor only.

Our IO test application (ST_FILE) performing disk file reading and writing in 512-byte blocks levels off at maximum scalability level of 1.8 over single processor; whereas for more realistic applications, reading and writing in longer blocks, saturation will be reached even earlier.

We do not expect VAX MP itself to limit scalability to a significant extent (beyond limitations of host OS and host machine, as well as inherent serialization within a workflow), however it must be borne in mind that historical VAX machines existed in configurations with only up to 6 CPUs maximum, and therefore OpenVMS VAX never historically ran on configurations beyond 6 processors. It was not designed and tuned for scalability to much larger number of processors.

> For example, OpenVMS VAX scheduler uses global locking and does not feature per-CPU scheduling queues with inter-CPU balancing. IO interrupts are processed on primary processor only. Timer expiration is processed on primary processor only. And so on.

We have not performed VAX MP scalability benchmarking yet. It is nevertheless a preliminary expectation that for compute-bound workloads with moderate inter-process and inter-processor interaction VAX MP itself won't be a significant limiting factor (apart from scalability limitations of underlying host system).

It should be also borne in mind when running VAX MP on system with hyperthreaded cores that hyperthreaded core is equivalent in performance not to 2x of non-hyperthreaded core, but approximately 1.2-1.3x. Thus, for example, assuming single-processor performance estimate of 40 VUPS, rough estimate for cumulative performance of virtual VAX system executed on quad-core hyperthreaded host processor would be not 320 VUPS, but around 200 VUPS.

Virtual disk file extension (Microsoft Windows)

On Windows, avoid using extension DSK for virtual disk container files and also using any other file name extensions monitored by Windows System Restore facility. Full list of monitored extensions can be found in article "Monitored File Name Extensions".[1] If System Restore is enabled, Windows may try to create backup copies of these files in Restore area, causing simulator to freeze for a long time while the file is being copied by Windows. The easiest way to avoid the problem is to use extensions such as VDISK, VDSK or VDK that are not on the list of monitored extensions.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/aa378870%28v=vs.85%29.aspx

Time Machine (OS X)

We recommend disabling Time Machine automatic copying of VAX MP disk image files. If Time Machine starts copying these files while VAX MP is running, this can cause slowdown of VAX MP or cause it to freeze for a long time while the file is being copied.

Refer to "Time Machine" section in article http://labs.hoffmanlabs.com/node/922 for references on how to disable copying of disk image files by Time Machine.

OpenVMS bugcheck CPUSANITY

If you observe OpenVMS bugcheck with code CPUSANITY ("CPU sanity timer expired") make sure you have SYSGEN parameter TIME_CONTROL set to values either 2 or 6. Bit 1 *must* be set in the parameter. Current release of VAX MP does not support OpenVMS execution with TIME_CONTROL set to 0.

OpenVMS bugcheck CPUSPINWAIT

If you observe OpenVMS bugcheck with code CPUSPINWAIT ("CPU spinwait timer expired") this may indicate that you have values of SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT set too low, in this case reset them to default. If parameters are at their default value[1] or higher, this may indicate a possible bug in VAX MP.

In the latter case please report the crash to SIMH mailing list ideally submitting on request details of your VAX MP SIMH set up, including: simulator startup script file, exact VAX MP version, console output during early startup and before the crash, OpenVMS version, SYSGEN parameter file (typically VAXVMSSYS.PAR unless used different file during SYSBOOT) and OpenVMS crash dump file (SYSDUMP.DMP).

Increasing values of SMP_SPINWAIT and SMP_LNGSPINWAIT may resolve the problem, however under most circumstances default values[2] should be amply sufficient.

As a sure-fire temporary fix, setting SYSGEN parameter TIME_CONTROL to 6 should disable OpenVMS SMP timeout bugchecks.

If you are using native interlock mode (VSMP INTERLOCK=NATIVE), falling back to portable interlock mode might also provide a temporary fix.

[1] 100,000 (1 second) for SMP_SPINWAIT; 950,000 (9.5 seconds) for SMP_LNGSPINWAIT. Note that default value of SMP_LNGSPINWAIT suggested by VAX MP is less than standard OpenVMS default value of 30 seconds. VAX MP uses smaller value to avoid busy-wait loop iteration counter overflow problem mentioned further. 9.5 seconds must be amply (indeed, over-amply) sufficient for any legitimate SMP synchronizaton.

[2] SMP_SPINWAIT set to 100000 or 1 second.
SMP_LNGSPINWAIT set to 950000 or 9.5 seconds.

In certain cases CPUSPINWAIT bugcheck can be triggered because the value of SMP timeout is set too high. If processor calibrates as fast too, computed counter of maximum spin-wait loop iterations can overflow.

> Counter of maximum spin-wait loop iterations is computed as
>
> ```
> EXE$GL_TENUSEC * EXE$GL_UBDELAY * timeout in 10-usec units
> ```
>
> Counter is 32-bit *signed* integer, i.e. the product of three parameters in the formula must not exceed 0x7FFFFFFF. If it exceeds this maximum value, OpenVMS is likely to falsely signal SMP timeout and raise CPUSPINWAIT bugcheck.

In fact, the reason for SMP_LNGSPINWAIT reduction from 30 seconds to 9.5 seconds is exactly that absurdly high value of 30 seconds causes overflow of spin-wait cycles counter on sufficiently fast host machines.

> In general, there should be no legitimate reason for SMP timeouts to be longer than 1 second (and that's still with very wide safety margin), even for "long" timeouts with for low-IPL spinlocks that may allow nested spinlock acquisitions, and even much less for high-IPL spinlocks.

VSMP LOAD command will check values of SMP_SPINWAT and SMP_LNGSPINWAIT to ensure they do not cause overflow on given host. If they do, VSMP will display advisory message to reduce their values, including advised new values, and refuse to load and enable multiprocessing.

It might be possible (though exceedingly unlikely) for layered privileged components, possibly including 3rd party products, to utilize private values for their SMP timeouts, such as for private spinlocks. If these values are set too high, they can cause spin-wait counter overflow and trigger CPUSPINWAIT bugcheck. If this unlikely situation is ever encountered, SMP timeouts for such product would need to be adjusted.

OpenVMS bugcheck BADQHDR

If you observe OpenVMS bugcheck with code BADQHDR ("Interlocked queue header corrupted") this may indicate either a possible bug in VAX MP or possible OpenVMS bug, perhaps made more prone to being triggered when OpenVMS is executed on a simulator. Please report the crash as described in the entry for CPUSPINWAIT bugcheck above.

The reasons for particular bugcheck need to be investigated on case-by-case basis, however the following remedies may be available against conditions that lead to BADQHDR.

If you are executing VAX MP using native interlock mode (@VSMP$LOAD INTERLOCK=NATIVE) try switching either to portable mode or portable mode with synchronization window additionally enabled.

BADQHDR can be generated in three principal cases:

- Interlocked queue header is genuinely corrupt, for example was overwritten by memory access to incorrect address. No immediate fix is possible. Reason for memory corruption need to be investigated and resolved.

- Interlocked queue operation retry loop uses fixed hardwired retry count, such as 900,000, and queue interlock holder takes abnormally long time to release it. Albeit some limited recourses may be possible, such as tightening SYNCW_ILK window, normally queue interlock holder is not expected to hold it for that long. Switching from native interlock mode to portable mode or to portable mode in conjunction with synchronization window are possible temporary ad hoc emergency recourses, however the underlying reason for abnormally long interlock holding need to be investigated and resolved.

- Interlocked queue operation retry loop uses retry count controlled by SYSGEN parameter LOCKRERTY. In this case additional ad hoc emergency recourse is possible: increasing the value of LOCKRETRY. However LOCKRETRY should not be increased excessively high since long spinning time waiting for queue header interlock can lead to low responsiveness of spinning processor and to CPUSPINWAIT bugcheck being triggered by other processors. If the latter happens, threshold for CPUSPINWAIT timeout (SYSGEN parameter SMP_SPINWAIT) may also be increased somewhat as emergency override measure, such as from 1 second to 2-3 seconds. However the underlying reason for abnormally long queue holding time should be identified and resolved as soon as possible.

CPUs do not start

If you execute START / CPU/ALL command but some CPUs won't start, check if you have properly increased the value of SYSGEN parameter SPTREQ as described in section "Quick VAX MP Installation Guide". If SPTREQ is insufficient to allocate per-CPU data structures and map them, START /CPU/ALL will fail to start some or all of additional processors, but it won't display error message. If you try to start a specific processor with START / CPU <n> command and SPTREQ is insufficient, the command will display generic error NOSUCHCPU. Unfortunately DCL START /CPU command ignores actual diagnostic code coming from SYSLOA/VSMP modules and does not display it properly.

Message "Warning: Unable to lock VAX MP in memory (not critical)" – Linux only

This message may be displayed by VAX MP on Linux only in response to startup script command `DEPOSIT WS_LOCK 1` or commands that set non-zero values of `WS_MIN` or `WS_MAX`. When these values are set, VAX MP tries to lock all of the simulator's code and data in memory to reduce paging. If page faults happen during the execution of simulator while VCPU thread that incurred the fault is holding a lock or OpenVMS level spinlock or needs to process time-critical operation such as responding to interprocessor interrupt, waiting for the fault to complete may

temporarily serialize the system until the fault is resolved. It is not critically important to avoid the faults, but it is desirable from the performance standpoint. VAX MP therefore tries to lock all of its code and data in memory, including code and data of the simulator itself, code and data of all loaded runtime libraries, thread stacks etc. VAX MP uses Linux system call `mlockall(MCL_CURRENT)` to perform such locking. Depending on Linux version, this call requires either process privilege (running as root or having capability CAP_IPC_LOCK) or increasing RLIMIT_MEMLOCK for the account via /etc/security/limits.conf. If VAX MP does not have sufficient privileges or resource allocation limits expanded, it will display this message.

On Linux host systems with plenty of available RAM where VAX MP does not incur paging at run time other than initial page-in at boot time, this message is unimportant in practical terms and does not really need a corrective action.

If VAX MP incurs paging at run time, as may be indicated by Linux system monitoring utilities, it is desirable to enable VAX MP full working set locking in memory. To enable memory locking privileges do either of the following:

- run VAX MP as root (e.g. via `sudo`);
- grant CAP_IPC_LOCK capability to VAX MP executable file with Linux `setcap` command:

```
sudo setcap cap_ipc_lock+eip /path/vax
```

- grant capability CAP_IPC_LOCK to the account or accounts used to run VAX MP and use Linux command `chrt` to launch VAX MP;

- on Linux 2.6.9 and later, add the following lines to file /etc/security/limits.conf for each account that will be used to run VAX MP and that you want to be able to lock VAX MP data and code in memory:

```
vaxuser   soft   memlock   xxxx
vaxuser   hard   memlock   xxxx
```

  where *xxxx* denotes the amount of code and data applications executed by the account are allowed to lock in memory, expressed in kilobytes. Note that this authorization is per-account and allows memory locking not only by VAX MP, but also by any application that issues memlock system calls and is executed by the account.

LOCKRETRY

VSMP LOAD command will set SYSGEN parameter LOCKRETRY to 900000 unless command option NOPATCH=LOCKRTRY is used. Do not decrease the value of LOCKRETRY after executing VSMP LOAD. VSMP LOAD passes the value of LOCKRETRY to the simulator's SIMH layer which uses passed value to compute internal control variables such as synchronization window size. Decreasing LOCKRETRY on top of the running simulator after VSMP LOAD command will make these pre-calculated controls invalid and can cause OpenVMS to bugcheck with BADQHDR code.

TIME_CONTROL

SYSGEN parameter TIME_CONTROL should be set to 6 when running VAX MP with synchronization window disabled. With synchronization window enabled, it can be set either to 6 or 2. See section “Interprocessor synchronization window” in the present document for valid sets of parameter values. VSMP will refuse to load if incoherent parameter set is used.

Use of dedicated host machines

If host machine is dedicated to run VAX MP with no competing host workload, it is possible to set configuration parameter `host_dedicated` that may increase performance. This parameter disables VAX MP VCPU priority elevation while executing system-critical code and saves the cost of frequent “set thread priority” system calls.

Note that this setting affects the priority of VAX MP VCPU threads on the host system, it does not affect relative priority of OpenVMS processes and OpenVMS scheduler.

Since the calls to elevate and downgrade VCPU threads priority are executed frequently by VAX MP when performing system-intensive workload, saving can be noticeable under workloads that execute a lot of interlocked instructions, such as workloads that perform heavy IO, heavy paging or other activities resulting in high rate of spinlock and mutex acquisitions, manipulating interlocked queues or performing other interlocked operations. By the way of example, performance of a workload similar to OpenVMS boot sequence can be boosted by about 10-15% if VCPU thread priority control is disabled.[1]

For workloads that are mostly computational in nature with limited dependency and interaction between the processes that execute on different processors , i.e. workloads that do not produce high rate of interlocked instructions, disabling VAX MP thread priority control will not result in noticeable performance improvement.

Thread priority control should be disabled only if there is no competing host workload and the machine is dedicated to executing VAX MP.

Furthermore, since some host workload is also created by host OS system processes such as system services and daemons, terminal emulator and window manager as well as VAX MP auxiliary non-VCPU threads (IO threads and clock strobe thread), we advise to reserve one host LCPU for these purposes if disabling VAX MP thread priority control and configure VAX MP with the number of processors at least one less than the number of LCPUs on the host system.

Since reserving host LCPU reduces configured VAX system throughput, disabling thread priority control only makes sense if the increase in throughput due to performance gain achieved by

[1] Performance of actual OpenVMS bootstrap cannot be boosted by using HOST_DEDICATED. When VAX MP operates in uniprocessor mode, as would be the case during initial boot phases, it does not control thread priority and avoids most other overhead associated with multiprocessor synchronization.

disabling thread priority control and eliminating its overhead outweighs the loss of throughput due to smaller number of processors, i.e. if dedicated host system has large number of LCPUs *and* VAX instance executes a workload with very high rate of interlocked instructions.

To disable VAX MP VCPU threads priority control, add the following command to VAX MP startup script:

```
DEPOSIT HOST_DEDIDCATED 1
```

TS vs. TQ

If you intend to make heavy use of tape drive, it is better to use MSCP drive (SIMH TQ device) rather than TS-11 or TSV-05 (TS device). TQ performs better under multiprocessor load than TS. For occassional or light use TS is fine.

Boot ROM failures

Very occasionally boot ROM initialization diagnostic may fail with messages like:

```
sim-cpu0> boot cpu

?C6 2 01 FF 00 0000

P1=00000000  P2=00000000  P3=00000000  P4=00000000  P5=00000000
P6=00000000  P7=00000000  P8=00000000  P9=00000000 P10=00000000
r0=00000080  r1=00000020  r2=00000000  r3=201407A4  r4=2004D2F0
r5=2004D310  r6=2004D312  r7=00000000  r8=00000000 ERF=80000000

KA655X-B V5.3, VMB 2.7
Performing normal system tests.
40..39..38..37..36..35..34..33..32..31..30..29..28..27..26..25..
24..23..22..21..20..19..18..17..16..15..14..13..12..11..10..09..
08..07..06..05..04..03..
Normal operation not possible.
```

This problem is not specific to VAX MP but occurs with uniprocessor SIMH VAX as well. It is safe to disregard this message and proceed with booting operating system.

SHOW CLUSTER may intermittently fail when used in native interlock mode

SHOW CLUSTER /CONTINUOUS command has somewhat hidden feature whereby user by pressing SELECT key is allowed to enter command console mode to add extra infromation classes to default SHOW CLUSTER display.

When using VAX MP native interlock mode (VSMP LOAD INTERLOCK=NATIVE) this display may intermittently fail for an update cycle and display question marks in some data fields (such as queue lengths) instead of values.

Probability of such failure is very low, but it is possible.

If the failure happens nevertheless, data items can be expected to be updated on the next screen refresh cycle. (Probability of two or more failures in a row is exceedingly low.) Thus the impact of the problem is that for a second or few seconds, depending on selected screen update frequency, one or few data items can be displayed as question marks rather than valid values.

The failure cannot happen if VAX MP portable interlock mode is used (VSMP LOAD INTERLOCK=PORTABLE, which is the default setting).

> The underlying reason is that module [CLUTL.SRC]SHWCL_SNAP.MAR uses interlock queue locking loop limited by LOCKRETRY with only 2 instructions in the loop, rather than usual 3 or more as the rest of VMS does. VAX MP synchronization window size is tuned for 3+ instruction loops, so 2-instruction loop can timeout under adverse host load conditions and return status SS$_INTERLOCK. This status will then be picked by routine DISPLAY_QUEUE.

If this issue is deemed worth fixing, it should be addressed in subsequent release of VAX MP, perhaps by statically patching SHWCLSTR.EXE to increase its busy-wait loop size or by recognizing BBSSI-AOBLSS/SOBGTR loops inside VAX MP.

Validation tests
(stress tests)

VAX MP source distribution provides a number of stress tests to verify the validity and integrity of VAX MP multiprocessor operations. Supplied stress tests create parallel processes that perform various activities stressing the system and verifying continued integrity of its operations while under stress conditions.

These tests are contained in VAX MP distribution's subdirectory VSMP_STRESS and are intended to be deployed on OpenVMS to any installation directory of system administrator's choosing.

To execute any of these tests, first ensure that system is configured with sufficient page and swap space to support running multiple test processes. VMS command SHOW MEMORY displays the size of swap and page files installed. We recommend minimum page file size of 500000 blocks and swap file size 100000 blocks for stress testing. Page and swap files can be enlarged with procedure @SYS$UPDATE:SWAPFILES.COM, or by installing additional files either on the system disk or secondary disk.

Tests should be executed under an account that has sufficient quotas. Command file CREATE_STRESS_USER.COM creates user STRESS that has required quotas.

Executable images for the tests can be built by command file BUILD_ALL.COM.

You will typically want to use two or multiple terminals for testing and log in as STRESS on a secondary terminal. At the same time you will increase priority of the process on your primary terminal to a high level (such as SET PROCESS/PRIO=16) so you could run MONITOR and other tools on this terminal unimpeded, even while stress tests launched on secondary terminal or terminals put very high load on the system.

Some of the tests, such as disk IO tests can (and uncached IO test and possibly paging test will) generate very high rate of disk IO with long disk queues creating a situation similar to denial-of-disk-service for other processes. When running these tests you should expect that response time for actions relying on disk operations, such as executing a command or launching an image, will be high on the primary console regardless of its process priority. This is expected and "normal"[1].

All the tests are launched with command file LAUNCH.COM file in corresponding test's directory. Parameters to LAUNCH.COM are specific to the test and are described below. Tests are terminated by pressing Ctrl/Y on the terminal used to run the test. Pressing Ctrl/Y will abort all launched test sub-processes and delete temporary work files.

[1] I.e. to whatever extent it represents a problem, this problem exists in OpenVMS itself, rather than in VAX MP.

The following stress tests are provided:

CPU_HOG Creates compute-bound processes that execute tight infinite loop. To start the test, run command file

```
@[.CPU_HOG] LAUNCH <nprocesses>
```

CC Creates subprocesses that repeatedly compile “Hello World” C program, execute it and purge files in the subprocess private directory. To start the test, run command file

```
@[.CC] LAUNCH <nprocesses>
```

FILE Creates subprocesses that repeatedly write generated data to a temporary file on the disk and read it back. This stress test exercises disk susbsystem, file system and caches. To start the test, run command file

```
@[.FILE] LAUNCH <nprocesses> RMS [inst-id] [npasses]
```

or

```
@[.FILE] LAUNCH <nprocesses> QIO [inst-id] [npasses]
```

Scratch files are created in a temporary subdirectory of LAUNCH.COM location.

RMS mode of the test exercises FCP cache, RMS buffering and disk extent/VIOC cache.

In QIO mode RMS is not involved (except for opening files) and disk IO bypasses caches via IO$M_NOVCACHE option. Notice that if your goal is to exercise disk system (such as UQSSP port and its driver) you generally want to use QIO mode in order to make sure the data is actually read back from the drive, and not from the extent cache.

You can observe cache hit rate with command
SHOW MEMORY /CACHE / FULL.

It is also possible to disable VIOC cache (even for RMS mode) by booting with SYSGEN parameter VBN_CACHE_S to 0 (note: this disables the cache throughout the cluster).

Optional parameter “inst-id” is a string, typically character A to Z, used in

naming creating processes. This option allows multiple instances of FILE stress test to be ran in parallel (typically from parallel stress login sessions) on multiple disk drives and/or controllers, such as DUA0, DUB0, DUC0 or VAXA$DUA0 and VAXB$DUA0. To run the test on multiple drives, replicate [STRESS.FILE] directory to each of the drives, log in at multiple terminals (or create batch jobs), and execute LAUNCH.COM off corresponding locations.

Optional Parameter “npasses” if specified causes testing image to be restarted every specified number of file write/readback passes, thus creating extra load on the system.

Like with all other stress tests, it is recommended to have separate stress-test login session or sessions and also main console session that can be used for system management and monitoring system state, and to keep the latter at elevated process priority, such as 15 or even higher. Nevertheless be advised that running FILE tests can generate very high disk activity with long disk driver queues, so console process response even at this high session priority can be sluggish when activating new images or performing other actions relying on disk IO.

PAGE This test exercises virtual memory paging system. Each subprocess launched by this test creates virtual pages and then at random fills them with data and eventually verifies its content. This causes heavy paging and stresses paging system. The test is intended to verify the reliability of the paging system and its ability to maintain integrity of data. To start the test, run command line

```
@[.PAGE] LAUNCH <nprocesses> <npages>
```

This test can be executed in two modes.

One is large physical memory configuration. In this case paging happens mostly between process memory space and VMS modified and free lists. Pages get written to paging file, but are not actually read back and mostly served to the process from either modified or free list caches.

The other testing mode uses reduced physical memory configuration. In this mode pages actually get cycled back to the process via paging file, thus stressing also the integrity of IO system and interaction of paging system to IO system. To activate this mode follow instructions in the header of file LAUNCH.COM.

Proper functioning of network interfaces can be tested in two ways. Smaller test involves copying files

via FTP to VAX system and back utilizing multiple simultaneous connections, and comparing the resultant copies of the files with the originals. More thorough test involves running FILE QIO stress test over the VAXcluster, to remote disks. Our personal preference in the latter case is to set-up two-node VAXcluster with two disks per each node and run criss-crossed remote disk IO traffic between the nodes by starting two FILE QIO sessions on each node to remote disks of other node and vice versa.

Addendum A

Building VAX MP

Building VAX MP on Windows

Download and install the latest version of WninPCAP (verson 4.0 or older is required).
Restart Windows to activate WinPCAP.

Download WinPCAP Development Pack and unzip it alongside VAX MP project so file tree structure looks like this:

```
{ROOTDIR}\simh-vax-mp\scp.h
{ROOTDIR}\windows-build\winpcap\WpdPack\Include
{ROOTDIR}\windows-build\winpcap\WpdPack\Lib
```

Use MSDEV 2008 Professional or higher to open project file `VAX_MP\VAX_MP.sln`.
Select target configuration = Release.
Select target platform = Win32 (32-bit version of VAX MP) or x64 (64-bit version of VAX MP). We recommend using Win32 target even when running on 64-bit version of Windows, since pending results of more detailed benchmarking it is tentatively believed that 32-bit version of VAX MP has slightly better performance than x64 version (by few percent).

Build the project and copy produced VAX_MP.EXE to deployment directory.

Building VAX MP on Linux

gcc 4.5.2 or higher version is required and glibc 2.13 or higher. Versions can be checked with `gcc --version` and `ldd --version`.
Earlier versions of gcc may work but have not been checked with.
Earlier versions of glibc may not work properly with VAX MP if they do not contain NPTL version of PTHREADS.

Unless you already have `make` installed, install it using package management facility for particular Linux flavor, for example:

```
apt-get install make
```

Download and install the latest versions of libpcap and pcap development pack, for example:

```
apt-get install libpcap0.8
apt-get install libpcap0.8-dev
```

To include VDE (Virtual Distributed Ethernet) support, additionally download and install VDE development pack.

Convert project files from DOS (CR-LF) format to Unix (LF) format:

```
dos2unix d2u.sh
chmod a+x d2u.sh
./d2u.sh
```

Newer versions of GCC version support compilation option `-Wno-unused-but-set-variable`. This option suppresses some false warning messages emitted by GCC when building VAX MP. This option is present in GCC 4.6.3 onward but absent in GCC 4.4/4.5. When compiled with versions of GCC that do not have this option, build process will *not* produce false warning messages. When compiled with versions of GCC that do have this option, build will produce false warning messages unless the mentioned option is enabled in `Makefile2`.

`Makefile2` as currently distributed does not set this option for Linux.

If you are using version of GCC that does have this option, such as GCC 4.6.3 or higher, edit your your local copy of `Makefile2` in VAX MP build tree to change fragment

```
ifneq (,$(findstring darwin,$(OSTYPE)))
    CFLAGS_W += -Wno-unused-but-set-variable
endif
```

to

```
# ifneq (,$(findstring darwin,$(OSTYPE)))
    CFLAGS_W += -Wno-unused-but-set-variable
# endif
```

To build VAX MP, execute in the project's directory:

```
chmod a+x *.sh
export OSTYPE
./mk.sh {x86|x64} {dbg|rel} {net|shr|nonet} [tap] [vde] rebuild
```

For example:

```
./mk.sh x86 rel shr tap rebuild
```

Options meaning is as follows:

`x86|x64` selects target processor architecture

`dbg|rel` selects debug or release build

`net` selects network support using statically linked libpcap.a
`shr` selects network support using dynamically linked libpcap.so
`nonet` selects no network support

`tap` selects support for TAP/TUN virtual network devices

`vde` selects support for VDE virtual network devices (VDE package must be installed both on build and target machines)

Cross-building x86 version of VAX MP with no network support on x64 Linux systems is possible, but requires installation of 32-bit version of *glibc*, with appropriate development header files (package *libcx-dev* or *glibc-devel,* depending on your edition of Linux). For simplicity, we suggest to build 32-bit version of VAX MP under 32-bit Linux and then copy produced executable file to 64-bit Linux system.

Building VAX MP on OS X

XCode comes with GCC-LLVM compiler. We advise against the use of GCC-LLVM since simulator produced by it is about 2x less performant than compiled with regular GCC compiler. Accordingly, VAX MP build files are not even set up for LLVM. Unfortunately OS X or XCode do not ship with regular GCC included, so to build VAX MP under OS X you need either to obtain existing prebuilt GCC or build and install GCC on your machine.

VAX MP had been tested under OS X with GCC 4.7.0. We recommend using either this or higher version of the compiler.

Convert project files from DOS (CR-LF) format to Unix (LF) format by executing the following commands. The first two commands in the sequence (perl and mv) can be skipped if file `d2u.sh` is intact and are required only if `d2u.sh` had been inadvertently converted during project transfer (such as FTP) to DOS format and fails to run. You can copy these commands from the comment at the top of `d2u.sh` and paste them into Terminal shell prompt.

```
perl -p -e 's/(\r\n|\n|\r)/\n/g' d2u.sh >d2u.tmp

mv -f d2u.tmp d2u.sh

chmod a+x d2u.sh
./d2u.sh
```

To build VAX MP, execute in the project's directory (assuming g++-470 is the name of your private-built GCC executable):

```
chmod a+x *.sh
export OSTYPE
CXX=g++-470
export CXX
./mk.sh x64 {dbg|rel} {nonet|shr} [tap] [vde] rebuild
```

For example:

```
./mk.sh x64 rel shr tap rebuild
```

Options meaning is as follows:

x64 selects target processor architecture (32-bit x86 mode is not supported)

`dbg|rel` selects debug or release build

`nonet` selects no network support

`shr` selects network support using dynamically linked libpcap library;
note that statically linked libpcap (“`net`” option) is not available on OS X

`tap` selects support for TAP/TUN virtual network devices

vde selects support for VDE virtual network devices (VDE package must be installed both on build and target machines)