V A X / V M S S M P

D E S I G N S P E C I F I C A T I O N

REV 2.4
May 27, 1986

VAX/VMS DEVELOPMENT GROUP

DIGITAL EQUIPMENT CORPORATION
110 Spit Brook Road
Nashua, New Hampshire 03062

F O R I N T E R N A L U S E O N L Y

F O R I N T E R N A L U S E O N L Y

Digital Equipment Corporation COMPANY CONFIDENTIAL

Title: Working Design Document for VMS Symmetric Multiprocessing

Specification Status: n/a

Architectural Status: n/a

File: DOCD$:[SMP.DOC]SMP.RNO

Design Team:

Stu Farnham

Rod Gamache

Mike Harvey

Larry Kenah

Bill Laing

Ed Los

Kathy Morse

Robert Rappaport

Authors:

Members of the Design Team

Cathy Foley - Documentation Supervisor

Abstract: Description of VMS Implementation of SMP

CONTENTS

VAX/VMS SMP - Product Overview

VAX/VMS Version 4.0 supports single CPU operation and asymmetric multiprocessing. This design specification describes how VAX/VMS will be enhanced to support symmetric multiprocessing (SMP): allowing multiple CPUs to execute in parallel almost all functions; for example, executing streams of I/O, kernel mode code, processes, and so on.

This specification documents a working design of the VAX/VMS enhancements to support symmetric multiprocessing. It will include the pros and cons of various designs that were considered during the design process and then rejected; thereby, providing a historical record for future maintenance of or enhancements to the symmetric multiprocessing capabilities.

1.1 Product Abstract

SMP will execute on VAX multiprocessors that have the following characteristics:

- All memory is common to all CPUs.
- One copy of VAX/VMS is in memory.
- Most (if not all) devices can handle requests from all CPUs and direct an interrupt to any CPU.
- Cache coherence is implemented in the hardware.
- All CPUs are of the same model and are at the same hardware and firmware ECO level.
- All CPUs can perform all, or almost all, functions (for example, I/O initiation and post-processing, kernel mode code execution, and so on). (System time keeping is an example of a function limited to one CPU.)
- Tightly-coupled operation (that is, CPUs that crash together)
- The boot CPU should have the capability of starting other CPUs. This function is usually performed through the console subsystem.

The VAX/VMS SMP product will not include fault-tolerant features, beyond those currently available in VAX/VMS.

1.2 References

- VAX/VMS Internals and Data Structures, written by Larry Kenah and Simon Bate
- VAX/VMS Device Driver Writer's Manual (Version 4.4)
- DEC SID 032 VAX Architecture Standard (hereafter referred to as DEC STD 032)
- Guide to Multiprocessing on VAX/VMS (Version 4.4)

1.3 Glossary

ASMP - asymmetric multiprocessing occurs in a computer system with multiple CPUs, where one CPU is the master and the other CPUs are the slaves. The slave CPUs are limited to doing only certain types of work, while the master CPU does any kind of work.

BOOT CPU - is the CPU in an SMP system that controls the boot. It has responsibility for initializing memory, loading VAX/VMS, and so on. The boot CPU is also the CPU to which the console terminal is (physically or logically) connected. Thus, it is responsible for the output of all software-generated messages to the console terminal from any CPU (until this function is switched to another CPU, using software mechanisms). By default, the boot CPU is also responsible for keeping system time and is also called the Time Keeper CPU.

Not all console output is generated by a VAX running in program mode. The console subsystem is capable of generating output directly to a physical device. For example, the console subsystem will report the occurrence of a microcode detected error halt condition directly to the physical console terminal or to a logical console interface. Until a mechanism exists which switches the physical console connection to another CPU, the BOOT CPU must remain bound to the CPU that is physically connected to the console.

CRASH CPU - is the CPU in an SMP system that, if possible, takes control when any CPU requests a system bugcheck or crash. It may be preferred that the CPU requesting the crash not perform the bugcheck, because that CPU may be having problems and, therefore, may be suspected of contributing to the crash. The crash CPU will take over the console terminal, write the console output, write the dump file, and cause a reboot, if appropriate. The crash CPU becomes the boot CPU for the reboot of the system.

A great majority of system crashes are not due to problems with the CPU. VMS takes precautions against losing the ability to generate a crash dump. This fact, together with the lack of physical console switching support in some current multiprocessor VAXes, means that the crash CPU will more likely than not be the boot CPU in actual VMS SMP systems.

DEVICELOCK - is a dynamic spinlock that must be acquired by code that executes at device IPL (DIPL). Since devicelocks represent the lock on an individual adapter or controller, they are dynamically allocated with the device CRB by default. As synchronization mechanisms, devicelocks are replacements for the traditional VAX/VMS device IPL based techniques (DIPLs) and represent the lock on the device.

DYNAMIC SPINLOCK - is a spinlock that is created based on the configuration of a particular SMP system. It locks structures such as the I/O database. The spinlock structure itself is created by allocating it from nonpaged pool, rather than by being assembled into the system in the manner that a static spinlock is created. An example of a dynamic spinlock is a DEVICELOCK.

ENQ/DEQ - is a method for user-level application programs to synchronize accesses to a shared resource. This functionality is provided by the $ENQ and $DEQ system services. This is the preferred method for user processes to synchronize access. This method works in a uniprocessor, a multiprocessor, and in a VAXcluster environment.

FORKLOCK - is a static spinlock that must be acquired by code that may fork or stall and that must be automatically reacquired when the fork thread resumes. All code that must synchronize with such threads must also acquire the same forklock. As synchronization mechanisms, forklocks are replacements for the traditional VAX/VMS fork IPL based techniques (FIPLs).

LOCK - is a generic term for any kind of synchronization mechanism. The lock may be a mutex, a spinlock, or an ENQ/DEQ structure.

MUTEX - is a synchronization method in which a lock is taken out on a resource or database under process context, and in which the IPL is below 3 (the scheduling IPL) or may be lowered to a level below IPL 3. (That is, a mutex is used for synchronization when the 'owner' of the lock can be rescheduled.) Spin-waiting is not allowed when a process is attempting to acquire a mutex. A process is put into a wait state when it fails to immediately acquire a mutex. Occasionally, system level code may want to acquire a mutex without process context. Therefore, the code must have its own method of attempting to retry access to the mutex, if it fails to initially acquire the mutex. Mutexes may be held at any IPL above IPL 1. Another characteristic of a

mutex is that it will allow either one writer or multiple readers.

OTHER CPUs - include all CPUs in the SMP system, except for the BOOT CPU.

SEMAPHORE - is a lock that allows access to only one consumer at a time. Other locks may allow multiple consumers, as in the case of multiple readers on a mutex.

SMP - symmetric multiprocessing occurs in a computer system with multiple CPUs, where all CPUs are equal members and can perform all types of work.

SPINLOCK - is a semaphore. It is a synchronization method in which there can only be one processor 'owning' the lock and that 'owner' must be running above IPL 2 (that is, no reschedule can occur). Spin-waiting is allowed on spinlocks. Also, there is no concept of multiple readers on spinlocks; that is, you either have the resource or you do not. The IPL level may be raised above, but never lowered below, the IPL at which the spinlock was acquired. Spinlocks are acquired and released using interlocked memory operations.

SPIN-WAITING - occurs when a CPU attempts to acquire a spinlock, but the spinlock is already 'owned' by another CPU. The CPU must execute in a loop, waiting for the spinlock to be released.

STATIC SPINLOCK - is a spinlock, the data structure for which is permanently assembled into the system. Its existence and definition are fixed from one system to another. A static spinlock is also called a system spinlock.

Time Keeper CPU - is the CPU in an SMP system that keeps the system time and checks for timer queue element expirations. In systems that implement the TOY clock in the console subsystem, the Time Keeper CPU must be a CPU with access to the console, access which does not depend upon the availability of another CPU.

1.4 Internal Interfaces Under ECO Control

This working design document will be modified as the design for SMP evolves. The design is expected to evolve from a simple to a complex set of locking mechanisms. Each phase of the evolution will be aimed at increasing the functions that can be executed simultaneously on all of the CPUs. The intended goal is that of enhancing the performance of the entire system. Obviously, the first phase must be to implement a correctly working SMP system.

All interfaces should remain internal for at least two releases of SMP. This will protect the customer base from any changes that are necessary, based on experience and extended use of the SMP system. However, some documentation detailing how to write device drivers and user-written system services must be provided with the first release of SMP.

1.5 Markets

The SMP system will enhance the performance of isolated, networked, and clustered VAX/VMS systems. It will be a desirable price/performance enhancement to VAX/VMS for all markets in which VAXes are currently sold. SMP is also expected to open new markets. These are markets where higher absolute performance is needed and can be provided by SMP than what is currently provided by a single VAX CPU.

1.6 Competitive Analysis

This product will compete directly with the other vendors' multiprocessing systems currently available on the market. SMP will be attractive due to its capability of load-leveling, which is currently only partially available in VAXclusters (through the use of batch queues).

New applications, which can apply multiple processes to the same problem, will be able to exploit the presence of multiple processors. This is an important new development in the scientific processing market.

1.7 Product Audience

This working design document is intended to support the efforts of the design team as they develop SMP. The document will be valuable to future VMS developers, whose responsibility it will be to maintain and enhance the SMP functionality. In addition, organizations within Digital that train or support field personnel may want to utilize this design specification, to prepare the field for the SMP product when it becomes available.

This document must remain internal to Digital and company confidential.

VAX/VMS SMP - Product Goals

The goals for the VAX/VMS SMP product are as follows:

- o A correctly working SMP system
- o Performance acceptable to ship as a product
- o Continued shipment of only one version of VMS

To meet these goals, a correctly working SMP system must first be designed. This design must include an evolutionary plan, so that the performance of the system can be guaranteed to be of product quality.

2.1 Design Goals

The ability to change the granularity of the system locking mechanisms is a goal for the SMP design. It is viewed as one of the best methods for improving performance.

Another goal is to build design consistency checks into the SMP product. This should reduce the amount of time spent debugging interactions between the various CPUs. These consistency checks should be designed so that they can remain in the code that is shipped to customers and later turned on at a customer site that is having problems. Providing these consistency checks removes the need for VMS developers to build special "patched-in" checks whenever a site has difficulties that are not reproducible on other machines.

The SMP system will not be guaranteed to run the N-highest priority processes on an N-processor SMP system. This would contribute to increased overhead in interprocessor communications, rescheduling, and context switching. Therefore, a design goal is for the SMP system to behave more like a uniprocessor and guarantee only execution of the highest priority process at any given time. However, the SMP system will attempt to run the N-highest priority processes. Also, idle processors will continually look for work and, therefore, help reduce the scheduling latency time.

The design should have the capability for 'limited' device affinity. Device affinity will be required to support console terminals and console block storage devices. However, the design is not intended to support a wide variety of devices connected to only one CPU.

2.2 Performance Goals

The SMP system must have acceptable performance under a wide range of workloads, both commercial and scientific. Interrupt latency may be affected by one CPU having the capability to lock out another CPU. In general, interrupt latency should work as well on an SMP system as it does on a single CPU system or, perhaps, even better.

Performance may be enhanced because each controller or adapter has a separate lock. Therefore, if there are a sufficient number of controllers or adapters in an SMP system, then multiple interrupts may execute simultaneously on different CPUs.

A single CPU system should be able to execute the same VMS code as a multi-CPU system, with only minor performance degradations over current VMS operating system performance. VMS applications that are written to take full advantage of the additional CPUs, by definition, will be impacted the most. This impact will vary according to the contention for system resources, rather than according to the amount of overhead caused by the VAX/VMS operating system.

For performance reasons, the design consistency checks may be executed only during field test and special situations, instead of all the time.

VMS is aware of the problems caused by using large caches on VAXes. Such problems include the refilling of the cache by lots of cache misses when a process is started on a processor that is different from the one the process last ran on. Although VMS does not have a definitive solution for this problem, the design team is investigating alternatives that might alleviate the severity of the situation. For example, one alternative is for a VMS SMP system to attempt to run a process on the same processor that the process last ran on. There are also a number of other alternatives based on this idea. However, the overriding goal to always run the highest priority process will not be altered.

2.3 Support Objectives

The SMP enhancements to VMS must not result in a separate VMS product. The enhancements must run on all VAX systems, that is, uniprocessors and multiprocessors that have been designed for symmetric operation. The methodology used for synchronization must be the same for all VMS systems.

In summary, the VMS development group, layered product groups, and support organizations will not have to learn and maintain two different VMS strategies.

2.4 Environments

The VMS SMP system is expected to function as a member of a VAXcluster, a node in a network, or as an isolated entity. The same software that runs on a true SMP system should also execute on a single CPU system. However, a true SMP system will only run on selected processors in the VAX family.

Note that all processors in a multi-CPU SMP system will be required to be at the same hardware and firmware ECO level. The intent of this requirement is to guarantee, for example, the ability of any given processor to resume the execution thread of a process that had been executing previously on a different processor.

2.5 Reliability, Availability, Supportablity (RAS) Goals

The SMP system should have the same RAS features that currently exist in VMS.

2.6 Non Goals

It is not a goal of this project to provide a fault-tolerant system, wherein one CPU could fail and the others would continue to execute, without losing the ability to do work. It is expected, however, that one CPU could be shut down in a controlled manner and the system would continue to execute correctly, without losing any ongoing work. It will also be possible to exclude a processor from running when booting or rebooting an SMP system.

It is not a goal of this project to design an SMP system to execute on VAX systems that do not provide common access to all (or most) peripherals and memory from all CPUs. For example, SMP will not be expected to run on the VAX 11/782.

NOTE

This raises the question of supporting Asymmetric Multiprocessing (ASMP) under VAX/VMS in general. There are two aspects of the problem that must be considered. The first question that must be answered is whether or not ASMP will continue to be offered on multiprocessors that are capable of running symmetrically. It is expected that SMP will entirely replace ASMP on all processors that are capable as such. The second part of this concerns the VAX-11/782. Will VMS have to keep ASMP in operation forever to ensure that new features of future versions of VAX/VMS continue

to be available to the VAX-11/782 customer base? Or will ASMP be decommitted in some planned future release of VAX/VMS, the final release of VAX/VMS that these customers will be able to acquire as long as they are using VAX-11/782 processors? There aren't many VAX-11/782 systems left. It might behoove us to explore alternatives to continuing support of ASMP for these systems forever. This seems to be a product management decision.

VAX/VMS SMP - Functional Description

Currently, in VMS Version 4.0, IPLs are used for synchronization and prioritization. When a CPU needs to synchronize access to a system structure, the processor IPL is raised to lock out certain activity to allow controlled access to the structure. However, raising the IPL on an SMP system would only lock out activity on the currently executing CPU; all other CPUs could do whatever they wanted, including executing the same code thread. This could cause race conditions and unpredictable crashes. Therefore, to run multiple CPUs, a new locking mechanism must be designed to synchronize all CPUs in an SMP system.

Different CPUs in the VMS SMP system should be able to handle interrupts from different hardware devices, execute kernel mode code that requires different system locks, and execute user applications simultaneously. In general, there should be no imposed binding between processes or system events and CPUs.

Situations will arise where one CPU executes a code thread that requires a lock that is being held by another CPU, executing the same or different code. The design must include a mechanism for blocking execution of the second thread until after the first has released the lock, and then executing the second thread.

3.1 Alternative Designs

Several alternatives have been proposed and reviewed as a basic design for the first phase of implementation. The design team has seriously reviewed these alternatives, searching for a correct and evolutionary design. A description of the various alternatives that were rejected for one reason or another is included in an appendix to this document, along with the pros and cons attendant to each approach.

3.2 Final Design

The design team believes that the final SMP design consists of the following:

- Spinlocks:
  - There are a limited number of static spinlocks (perhaps a maximum of a dozen).
  - Each spinlock is acquired and released at a particular IPL, associated with a specific spinlock.

  - Spinlocks are acquired in a particular order, defined by their rank. However, they may be released in any order, as long as a particular spinlock is never reacquired until all higher-ranked spinlocks are released.

  - Spinlocks are owned by CPUs, not processes.

  - Other CPUs "busy wait" when they need a spinlock that is already owned.

  - Spinlocks control access to data structures contained within the VMS operating system.

- Mutexes:

  - Mutexes are owned by processes, not CPUs.

  - Mutexes are used instead of spinlocks, whenever the lock will be held for a long period of time and the IPL must be lowered.

- Device interrupts:

  - Multiple CPUs should be able to handle device interrupts simultaneously, by locking the appropriate controller (or adapter) devicelock.

- System time:

  - Time must be maintained by a single CPU.

  - Separate 10 millisecond timers will be used on each CPU to handle quantum end and statistics gathering.

- Console terminal:

  - CPUs should be able to share one console terminal, or have separate consoles (useful in debugging).

- Debugging tools:

  - XDELTA must be enhanced for debugging SMP.

  - Macros will be used to generate the locking and unlocking code, and to enable the easy inclusion of design consistency checking.

- Per-CPU database:

  - A new data structure called the per-CPU database that contains data relevant to each CPU must be created.

  - Space for interrupt stack operations that a CPU may require must be allocated for each CPU. It is a goal that no one stack may be used simultaneously by more than one CPU.

  - It must be easy and efficient for a CPU to access data that is specific to that CPU.

3.2.1 Design Approach

Much VMS code must be read and understood to implement SMP. The design team has decided to identify the specifics of the final design now, without continuing to consider various alternative proposals. Thus, when the VMS code is changed, the design team hopes it will only have to be changed once if the macros have sufficient arguments to implement several of the previously discussed alternatives (essentially by changing the macros and rebuilding VMS). This may prove to be a useful debugging tool, if the design team is unable to identify an insidious race condition.

The design team must identify each resource being locked by IPLs (in VMS Version 4.0) and define either a mutex, a spinlock, or another method for locking that resource. Any queue that is not protected by a spinlock, must become an interlocked queue. Any threads of code that manipulate a data structure at different IPLs, must do so with interlocked instructions or use CPU-specific queues (for example, the pool look-aside lists will use interlocked queues, while the fork queues will become CPU-specific).

The final design approach is to make one pass through the VMS sources, inserting what is expected to be the final set of lock and unlock macros. These macros could later be changed to implement some of the previously discussed alternatives, should the full implementation encounter some debugging difficulties.

3.2.2 Hardware Interval Timer Interrupt

The 10 millisecond interval timer causes a hardware interrupt at IPL 22 or 24 (depending upon the VAX CPU type). Each CPU in the SMP system has its own interval timer and,

therefore, takes these interrupts. Some of the work that must be done in this interrupt handler must be done on every CPU. A few work items must be done on only one CPU.

One CPU in the SMP system must be designated the maintainer of the system time. It is the job of this CPU (hereafter called the Time Keeper) to update the system data structures that keep track of the system time. (Note: If all CPUs were allowed to do this work, time would be incremented in a random fashion, since the interval timers cannot be synchronized.)

By default the boot CPU is the Time Keeper CPU when the system is first booted. It also owns the console terminal for the SMP system. These duties are bound together; that is, the CPU that owns the console terminal also maintains the system time. (The system manager can switch the console terminal to a different CPU with a DCL command, STOP /CPU=n; provided that the SMP hardware supports that functionality.)

The Time Keeper CPU must also check the timer queue. This queue holds timer queue entries (TQEs) for system-wide periodic activities. If a TQE has expired, an IPL 7 software interrupt is requested. It is not necessary for other CPUs to perform this check since the time keeper updates the system time.

All CPUs must perform two activities in their timer interrupt handlers: 1) quantum end, and 2) statistics gathering. If the current process executing on a CPU has exceeded its quantum of CPU time, then that CPU requests an IPL 7 software interrupt. All CPUs must also gather statistics on the current mode, on a per-CPU basis. This information is displayed by the MONITOR MODE command.

NOTE

> The mode times and null time are kept in the per-CPU database. The kernel mode time is correct and does not need to be adjusted by the null time. This is different from previous versions of VMS (where the null time was counted as kernel mode time).

3.2.3 IPL 7 Software Interrupt Handler (IPL Timerfork)

Any CPU may request an IPL 7 software interrupt. No distinction is made by the IPL 7 software interrupt handler to indicate whether the CPU that is executing is the Time Keeper CPU. The two major activities are 1) quantum end expiration, and 2) executing TQEs that have expired. The first entry in the timer queue is checked by every CPU executing the IPL 7 software interrupt handler. Therefore, TQEs may execute on any CPU.

3.2.4 Timer Queue

The timer queue is ordered according to time. When a code thread has to be executed at a particular time (or after a given amount of time has elapsed), an entry referred to as a timer queue entry (TQE) is placed in this queue.

3.2.4.1 Timer Queue Examination

There are three instances when the timer queue is examined:

- When the hardware clock interrupt occurs
- In the IPL 7 software interrupt handler
- When a TQE is added or removed from the queue

In the hardware clock interrupt service routine, the Time Keeper CPU must examine the first entry in the queue (as it does in VMS Version 4.0) to determine if any TQEs have expired and to request an IPL 7 software interrupt if one has expired. Because the queue is ordered by time, the Time Keeper CPU need only examine the first entry. In the IPL 7 software interrupt handler, TQEs are removed from the head of the timer queue until all due TQEs have been dispatched. Finally, any CPU that has to insert or delete a TQE must search some portion of the timer queue.

3.2.4.2 Timer Queue Synchronization

Synchronization of the timer queue could be done in a variety of ways: by interlocked queues, spinlocks, and so forth. It is not desirable to block the Time Keeper's access to the queue; therefore, a spinlock on the entire queue is not appropriate (because searching the queue could take a long time).

The design team has decided to create two spinlocks for the timer queue to address synchronization issues:

- HWCLK - hardware clock database spinlock
- TIMER - software timer queue database spinlock

A new system quadword will be added, called TQE$GQ_1ST_TIME. It will contain the quadword of time for the first TQE in the timer queue. Access to this quadword and to the first TQE in the timer queue is controlled by the spinlock, HWCLK. A second spinlock, TIMER, controls access to the rest of the timer queue.

Because TQEs may be placed at any position in the timer queue (not just at the head or the tail of the queue), the timer queue will not be an interlocked queue (that is, accessed by INSQTI/REMQTI instructions). Rather, the timer queue will be a standard queue (that is, accessed by INSQUE/REMQUE instructions).

The following example describes how these two spinlocks are used:

- Time Keeper's IPL 24 (or 22) thread:

  1. Execute the LOCK macro on the HWCLK spinlock (thus locking the hardware clock database)
  2. Advance the system time in EXE$GQ_SYSTIME by 10 milliseconds
  3. Compare the quadwords TQE$GQ_1ST_TIME and EXE$GQ_SYSTIME to determine if the first TQE in the timer queue has expired
  4. Cause a SOFTINT at IPL$_TIMERFORK, if the first TQE has expired
  5. Execute the UNLOCK macro on the HWCLK spinlock (thus unlocking the hardware clock database)

- SOFTINT TIMER thread (can run on all CPUs as a SOFTINT is generated at the time of process quantum end and the first TQE expiral):

  1. Execute the LOCK macro on the TIMER spinlock (thus locking the software timer database)
  2. Execute the LOCK macro on the HWCLK spinlock (thus locking the hardware clock database)
  3. Check the first entry in the timer queue
  4. If the first entry is due, remove it and reset the quadword of time for the new first TQE
  5. Execute the UNLOCK macro on the HWCLK spinlock (thus unlocking the hardware clock database)
  6. Execute the UNLOCK macro on the TIMER spinlock (thus unlocking the software timer database)

- Any CPU inserting a TQE in the timer queue:

1. Execute the LOCK macro on the TIMER spinlock (thus locking the software timer database)

2. Search the timer queue to locate its position in the timer queue

3. If the new TQE goes on the head of the queue, then execute the LOCK macro on the HWCLK spinlock

4. Insert the TQE in the timer queue

5. If the new TQE went on the head of the queue, then set the new quadword of time

6. If the new TQE went on the head of the queue, then execute the UNLOCK macro on the HWCLK spinlock

7. Execute the UNLOCK macro on the TIMER spinlock (thus unlocking the software timer database)

NOTE

The system time quadword (EXE$GQ_SYSTIME) is protected by the HWCLK spinlock, because system time can only be changed or compared with multiple instructions. Thus, any CPU that reads or writes (EXE$GQ_SYSTIME), must own the HWCLK spinlock to obtain a consistent copy of the quadword.

3.2.5 IPL Usage

IPLs are currently used in VMS to perform both synchronization and prioritization. A brief synopsis of what the different IPLs achieve is described in this subsection.

IPL 0 is used by kernel mode programs that need to access data structures that are only accessible in kernel mode. For example, MONITOR goes into kernel mode for some of its data collection. If the program is sensitive to the content of data structures that may be modified above IPL 0, then it must not execute at IPL 0. Rather, the program must raise IPL and acquire the appropriate spinlock.

IPL 1 is not used.

IPL 2 is currently used in VMS to examine the current PCB. Process deletion is also prevented by raising to IPL 2, since the deletion AST cannot be delivered. (Note: Certain code sequences

in VMS incorrectly raise to IPL 2 to "lock" a shared data structure such as a Job Information Block, JIB. These locking sequences should be removed and replaced with another mechanism that will work in an SMP system such as a mutex to lock a JIB.)

Pagefaults may be taken by code executing at IPL 2; however, this is not the case for code that executes at higher IPLs. Therefore, programs that are sensitive to the content of pageable data structures run at IPL 2 to take pagefaults. For example, the allocation of paged pool is one such structure; it is protected by a mutex.

IPL 3 is used to request a SCHEDULE/RESCHEDULE of a process. This code immediately raises to IPL SYNCH (8) to perform all synchronization (in VMS Version 4.0). IPL 3 interrupts are used only to prioritize events, not synchronize them. With SMP systems, this code will use the SCHED spinlock to perform multiprocessor synchronization. The SCHED spinlock is acquired and released at IPL SYNCH (8).

IPL 4 is used to dispatch I/O completion requests that no longer require access to the UCB. Therefore, these requests do not have to synchronize at the fork IPL of the device. The IPL 4 post processing queue will be CPU-specific, this will help reduce contention on an interlocked queue and enhance the ability to permit device affinity.

IPL 5 is currently reserved for future use in VAX/VMS SMP.

IPLs 6 through 15 represent the remaining software IPLs. Except for IPL 15, this is where the synchronization will be expanded for other processors. The new synchronization techniques, called spinlocks, are discussed in the next section. The fork IPLs (6,8,9,10 and 11) may be be reached by forking. When a fork thread is executed, a spinlock is automatically acquired to synchronize that fork thread across the SMP system. These spinlocks are called forklocks.

IPL 15 is used to enter XDELTA. This IPL is not otherwise significant to the operation of an SMP system and, therefore, will not be discussed.

IPLs 16 to 31 represent the hardware IPLs. For these IPLs the SMP system will allocate separate dynamic spinlocks called devicelocks.

3.2.6 Spinlocks

Spinlocks are one of the synchronization methods used by the SMP system to ensure that multiple CPUs are accessing shared system data structures in a consistent and controlled manner.

There are two kinds of spinlocks:

- Static spinlocks - assembled into the VMS executive
- Dynamic spinlocks - created as required

Static spinlocks are the same from system to system; they are created by the assembling of VMS. Their names are predefined and they are the most commonly used spinlocks. Static spinlocks will lock the following types of structures:

- The scheduling database
- The memory management database
- Nonpaged pool allocation

Dynamic spinlocks differ from system to system. For example, consider the spinlocks for UCBs or CRBs.

A vector of longword addresses (called the spinlock vector) contains the addresses of all static spinlocks in the system. This vector is created at compile time in module LDAT of the SYS facility. The base address of this vector is SMP$GL_SPNLKVEC.

Structures that are protected by dynamic spinlocks will be modified to contain the address of the appropriate spinlock entry. For example, UCBs are protected by a spinlock common to all UCBs on the same adapter or common to the entire system, depending upon the type of device. Each UCB will contain the address of the spinlock entry that is appropriate for the particular UCB. Multiple UCBs on the same controller will contain the same spinlock address.

Spinlocks for dynamic structures, such as UCBs, will be created when the corresponding CRB is created. The address of the spinlock will be saved in the CRB and later copied to the UCBs as the UCBs are created for each unit on the controller.

To create a dynamic spinlock, a call is issued to a new system routine (SMP$ALLOC_SPL) to allocate a new spinlock. Later, a call must be made to another new routine (SMP$INIT_SPL) to initialize the spinlock with information (such as the IPL needed to acquire the spinlock).

NOTE

Dynamic spinlocks are not present as indexed entries in the system spinlock vector. As such, they are not appropriate for use in forking. Forklocks and forking mechanisms are discussed later in this document.

The spinlock vector includes the following:

SMP$GL_SPNLKVEC::

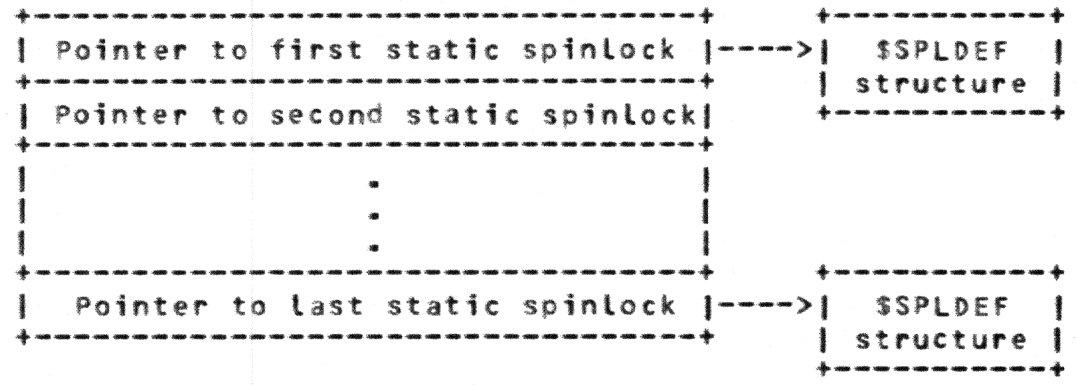


Figure 1: Spinlock Vector

3.2.6.1 Static Spinlock Database

The static spinlock database contains all of the information associated with each spinlock, including the following:

- Name of spinlock (ASCII text to be used by display programs) (TBI)
- IPL at which spinlock must be acquired/released
- State of spinlock (owned/unowned)
- CPU identifier of the owning CPU (if owned)
- Rank (order number for acquiring locks)
- Reference count that is incremented for each LOCK and decremented or set to 0 by the UNLOCK macro. This allows nested spinlock acquisitions on the same CPU.
- Statistic counters that include the following:
  - Count of waiting CPUs
  - Maximum number of waiting CPUs at any given time (TBI)
  - Cumulative count of spinloops executed by all waiting CPUs

- Last PC of a waiting CPU

- An eight longword vector that holds the last eight PCs at which the spinlock was acquired or released

- An index into the vector of the last eight PCs that points to the next PC entry to use

Each spinlock has an associated name, rank and IPL. To acquire a spinlock, the CPU must be executing at an IPL equal to or below the IPL associated with the spinlock. The CPU cannot lower its IPL to acquire a spinlock. Thus, a CPU acquiring a spinlock may or may not raise the IPL, depending upon whether the CPU was already executing at the IPL at which the particular spinlock must be acquired.

The spinlock database should contain the IPL for each spinlock. The VMS code will also indicate the IPL at which to acquire or release a spinlock. Including the IPL in the spinlock database allows the acquire and release spinlock routines to verify that each reference to a lock is supplying the correct IPL. Including the IPL as part of the caller's acquire/release spinlock request clarifies the code as it appears in the VMS listing, thus easing maintenance.

There are two possible states for the spinlock:

o Owned

o Unowned

If a spinlock is owned, then the owner field will contain an identifier that indicates which CPU in the SMP system owns the spinlock. This is especially useful when debugging hung systems or analyzing crash dumps.

NOTE

Only a CPU can own a spinlock, a process cannot own one.

There will be times when VMS code must acquire more than one spinlock. To prevent deadlocks, the spinlocks must be acquired in a particular order. Spinlocks may be released in any order, provided that no attempt to reacquire those spinlocks is made without such acquisition occurring in the correct order. The acquisition ordering and enforcement is defined by the RANK field in the spinlock database. Deadlocks are prevented in an SMP system by requiring that spinlocks be acquired in a pre-determined order. Each spinlock is assigned a unique rank.

When a CPU acquires a spinlock, the spinlock database is checked to determine if this CPU already owns other spinlocks. If it owns other spinlocks, then the rank of all owned spinlocks must be less than the rank of the spinlock being acquired. (This code may be turned off using a SYSGEN parameter, MULTIPROCESSING. However, it should be possible to activate the code at any customer site for debugging purposes.)

NOTE

Only one dynamic spinlock (at each IPL) may be acquired at one time. This is because all of the dynamic spinlocks will have the same rank (-1). Spinlocks may be acquired more than once by the CPU that first acquires them.

3.2.6.2 Spinlock Data Structure Format

The $SPLDEF structure includes the following:

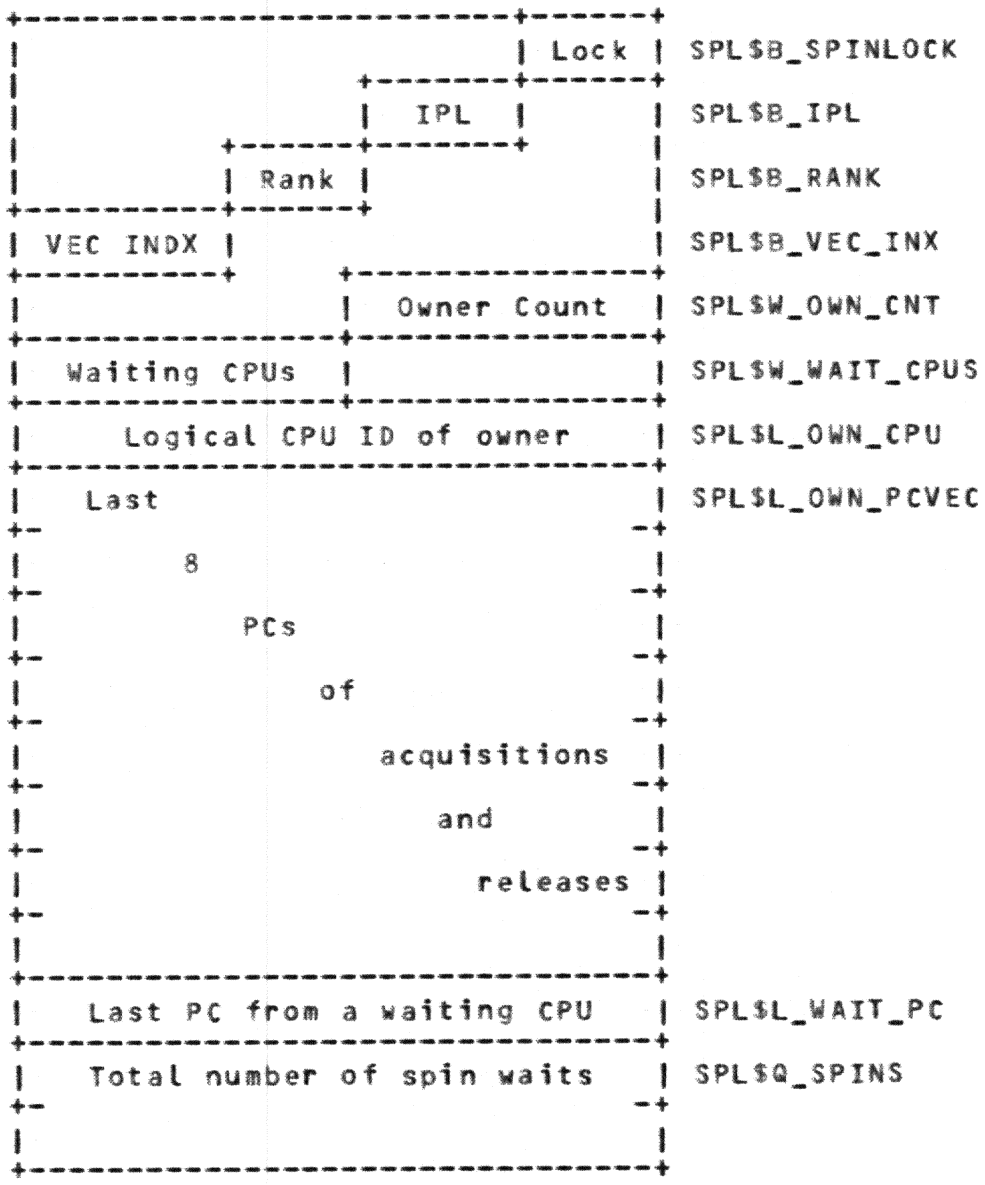


Figure 2: $SPLDEF Structure

The WAIT_CPUS field contains the count of CPUS that are waiting to acquire the spinlock.

The VEC_INDX field contains the index of the next entry to write in the SPL$L_OWN_PCVEC vector within the $SPLDEF structure.

3.2.6.3 System Spinlocks

Table 1 lists the static spinlocks, as currently defined. The rank for each spinlock is listed. A description of what is locked by each spinlock is included to prevent confusion regarding appropriate spinlock usage. Some IPLs do not have an associated spinlock. These IPLs (for example, IPL 3 and IPL 7) are used to order events, rather than to synchronize events. IPL 2 is used to synchronize events within a given process, rather than to synchronize events between processes.

Rank	Lockname	IPL	Description
0	QUEUEAST	6	Lock QUEUEAST queue (not used for locking, only for forking to IPL 6)
1	FILSYS	8	Lock on file system structures
2	MMG	8	Memory management, PFN database, swapper, modified page writer, per-CPU database vectors
3	IOLOCK8/SCS	8	Locks fork IPL 8 (SCS is an alias name); it also locks all SCS-related code (such as CNXMAN, LCKMGR, and DUDRIVER)
4	PR_LK8	8	PRIMARY lock for IPL 8
5	TIMER	8	Timer Queue Entries
6	SCHED	8	PCBs, scheduling database, acquisition/ release of mutexes
7	IOLOCK9	9	Lock on IPL 9
8	PR_LK9	9	PRIMARY lock on IPL 9
9	IOLOCK10	10	Lock on IPL 10
10	PR_LK10	10	PRIMARY lock on IPL 10
11	IOLOCK11	11	Lock on IPL 11
12	PR_LK11	11	PRIMARY lock on IPL 11
13	MAILBOX	11	Lock on sending messages to mailboxes (OPCOM)
14	POOL	11	Non-paged pool database
15	PERFMON	15	I/O performance monitoring lock
16	IPINT	21	Interprocessor interrupt wait IPL
17	HWCLK	22/24	Locks both system time cells (that is, the quadword time representing the first TQE, and EXE$GQ_SYSTIME)
??	EMB	??	Allocation and deallocation of error logging buffers. Note that non-SMP VMS is capable of these actions at IPL 31.

Table 1: System Spinlocks

These assigned ranks may not be so densely defined in a released version of VAX/VMS SMP.

The rank for each system spinlock is unique. The lowest rank is 1. A spinlock with a rank of 1 cannot be allocated after any other locks have been allocated by the same CPU. If more than one static spinlock must be held, then they must be acquired in order (that is, by order of increasing rank). However, spinlocks of intervening rank do not have to be acquired. For example, if the FILSYS and SCHED spinlocks are needed, they must be acquired in that order. However, no other spinlocks would have to be acquired (such as the MMG spinlock). The HWCLK spinlock is associated with the IPL at which the 10 millisecond interval timer interrupts. On certain VAX processors, this IPL is 22; on other processors, it is 24. Therefore, a new system cell, EXE$GB_HWCLKIPL, has been added to the spinlock database and is the address of the IPL field of the HWCLK spinlock data structure. EXE$GB_HWCLKIPL will contain the appropriate IPL for

the system (it will be set by CPU-specific code when VAX/VMS is booted).

All RSN$xxx resources need to be under the umbrella of a lock: either a mutex, a spinlock, or some other type of lock.

3.2.6.4 VMS Spinlock Usage

Any code in VMS that raises or lowers IPL must be changed, as well as interrupt service routines and machine check handlers. These locations are found by searching the sources for all references to SETIPL, ENBINT, DSBINT, REI, and MTPR x,PR$IPL. Such references should be changed to one of the following macros:

- LOCK - acquire spinlock
- UNLOCK - release spinlock
- SETIPL - set IPL; that is, do nothing to locks
- REIMAC - validate state of spinlocks and then REIs

These synchronization macros (and all others that are defined later in this document) preserve the contents of the general registers.

NOTE

The variants of the LOCK macro (FORKLOCK and DEVICELOCK) are also used to replace SETIPL and similar macro references.

The rules for changing SETIPL, DSBINT, and ENBINT macros are as follows:

Old Macro	Current IPL	Old Function	New Macro	New Function
SETIPL IPL >2	All	Raise IPL	LOCK name,ipl	Acq. spinlock
SETIPL IPL <3	>2	Lower IPL	UNLOCK name,ipl	Rel. spinlock
SETIPL IPL <3	<3	Lower IPL	SETIPL ipl	Lower IPL
DSBINT IPL	All	Save/raise IPL	LOCK name,ipl,sav	Save IPL/acq. spinlock
ENBINT	All	Lower IPL	UNLOCK name,ipl	Rel. spinlock

Table 2: Macro Changes

Notes:

- Not all SETIPL macros are converted; sometimes SETIPLs remain unchanged.

- The instances of "SETIPL 10$ ... 10$: .LONG IPL$xxx" references will be replaced by instances of the PMLREQ and PMLEND macros (PML is an acronym for Poor Man's Lockdown). Current instances are used to lock a maximum of three pages in memory and to raise IPL (to lock some resource at the same time). The new PMLREQ macro will call a memory management routine to lock as many pages as necessary into the system working set. However, all pages requested in a single PMLREQ macro call must be virtually contiguous - therefore, if current uses lock discontiguous memory then separate PMLREQ macro calls must be used.

  An LOCK macro call will also be added at these instances in order to lock the appropriate database. Note that all pagefaults must occur before the spinlock is acquired, since the IPL will raise and this will prohibit pagefaults.

  NOTE

  Refer to the Poor Man's Lockdown subsection for a complete description.

- The SETIPL macro should be enhanced to verify that the current PC is within 514 bytes of 10$ (of the previously mentioned instance). This is important because making a minor change (such as fixing broken branch displacements) can alter or break the locking assumptions provided by the code.

- The LOCK macro saves the current IPL in the DST parameter (if it is specified), raises IPL, and then locks the specified spinlock and associated database.

- The UNLOCK macro unlocks the specified spinlock and associated database, and lowers IPL if an IPL value is specified. If an IPL value is not specified, IPL is assumed to remain unchanged.

The rules for changing REI instructions are as follows:

- The REI instruction is replaced with an REIMAC macro, which calls a routine (SMP$REI_CHECK) to verify consistency of the REI instruction.

- The REIMAC macro must be preceded with a UNLOCK macro, for all databases intended to be released by the REI instruction. Thus, all spinlocks acquired above the IPL being lowered to must be released before the REIMAC macro is executed.

3.2.6.5 LOCK Macro

The LOCK macro is used to acquire a spinlock, using an interlocked instruction, to provide correct SMP synchronization. This macro takes four parameters:

- LOCKNAME - name of resource to lock
- LOCKIPL - IPL at which spinlock is acquired
- SAVIPL - optional, location to save current IPL
- CONDITION=(xx) - optional, special action
  where xx is:
  NOSETIPL - do not execute a SETIPL

The LOCKNAME parameter is used to generate a literal, which is the index into the system spinlock database. For example, if the lockname is SCHED, then the literal, SMP$C_SCHED, is generated. This literal is the index for the SCHED spinlock in the system spinlock vector and is resolved at link time. The list of system spinlock names is shown in Table 1.

We did consider defining dynamic spinlocks in the data structures that they locked. However, as an example, the addition or deletion of new spinlock statistic fields, would require the reassembling of all modules that define or reference a data structure that includes a spinlock. Instead, if spinlocks are included in the system spinlock database, only one module (LDAT) needs to be reassembled and then SYS relinked in order to incorporate changes to the spinlock structures.

All system spinlocks have defined indexable positions in the system spinlock vector (the base of which is SMP$GL_SPNLKVEC). No dynamic spinlocks are contained in the system spinlock vector. Therefore, the LOCK macro cannot be used to lock a dynamic spinlock.

The LOCKIPL parameter is specified for design consistency checking; that is, for ensuring that, if a spinlock is requested at a particular IPL, all requests are consistently given. (Although the LOCKIPL parameter is not absolutely necessary to the function of the macro, it will be useful to developers reading VMS code to have the IPL in the listing and guaranteed as correct.) The spinlock rank in the database is also checked during acquisition to make sure that multiple spinlocks are

acquired in a consistent order, to prevent deadlocks.

The SAVIPL parameter, if specified, will hold or save the current IPL, before it is raised to the IPL specified by the LOCKIPL parameter.

The CONDITION=(xx) parameter, if specified, is used to control different environments. For example, if an LOCK is done at a hardware device interrupt level, then no SETIPL is necessary. The possible keyword for this parameter is:

- NOSETIPL - specifies "don't do a SETIPL", to verify that the current IPL is correct for whatever spinlock is being acquired. This avoids overhead in situations where it is undesirable and unnecessary to do verify the IPL.

3.2.6.6 UNLOCK Macro

The UNLOCK macro is used to release a spinlock. This macro takes three parameters:

- LOCKNAME - name of resource to unlock
- NEWIPL - optional if it precedes REIMAC macro; IPL to lower to after spinlock is released
- CONDITION=(xx) - optional, special action
  where xx is:
  RESTORE - decrement the refcount and unlock only if the refcount is 0

The LOCKNAME parameter is used to generate a literal, which is the index into the system spinlock vector. For example, if the lockname is SCHED, then the literal SMP$C_SCHED is generated. This literal is the index for the SCHED spinlock into the system spinlock vector and is resolved at link time. The list of system spinlock names is shown in Table 1.

A spinlock is released using an interlocked instruction, to guarantee correct SMP synchronization. After the spinlock is released, the IPL is set to the value of the NEWIPL parameter. This may or may not change the current IPL of the processor.

The UNLOCK macro cannot be used to release dynamic spinlocks, which are not represented as index-accessible values in the system spinlock vector.

3.2.6.7 Spinlock Acquire and Release Subroutines

The following two subroutines are significantly modified versions of the actual spinlock Acquire and Release subroutines. They are provided here for readers who understand concepts better when they see code fragments. For clarity, code that does consistency checking and performance statistics gathering has been removed. This is the "bare bones" code, and probably much less than we would ever ship. The "logical CPU ID" mentioned in the comments below is explained in the section called "Per-CPU Database and Stacks".

```
SMP$ACQUIRE::                                   ;R0 = spinlock address.
        find_cpu_data   R1                      ; Get CPU database in R1.
        MOVZBL  CPU$B_LOG_CPUID(R1),R1          ; Get logical CPU ID.
30$:    ; The next instruction tries to acquire the spinlock,
        ; and falls through if successful, else it spin waits.
        BBSSI   #SPL$V_OWN,SPL$B_SPINLOCK(R0),60$
        ; The lock is now acquired.
        MOVL    R1,SPL$L_OWN_CPU(R0)    ; Remember which CPU acquired it.
40$:    RSB

60$:    ; We are about to busy wait
        ; First, we must verify if we are the lock owner.
        ; If so, then proceed, else we must wait.
        CMPL    R1,SPL$L_OWN_CPU(R0)    ; Do we own lock?
        BEQL    40$                     ; Yes, continue with lock held.
        ; Lock is owned by another processor - wait.
        ; The next sequence is the busy wait.
        ; We don't use the interlocked instructions to do the test.
        ; This avoids memory writes. However, it only checks that it
        ; is free and that we may acquire it.
        ; Then, go try the full, interlocked instruction.
80$:    BLBS    SPL$B_SPINLOCK(R0),80$  ; Loop if not available.
        ; Lock is now free. Try to acquire lock again.
        BRB     30$

INCONSISTENT:
        BUG_CHECK SPINLINCONS, FATAL            ; BUG OUT.

SMP$RELEASE::
        find_cpu_data   R1                      ; Get CPU database in R1.
        MOVZBL  CPU$B_LOG_CPUID(R1),R1          ; Get logical CPU ID.
                                                ; Check if correctly acquired.
30$:    CMPL    R1,SPL$L_OWN_CPU(R0)            ; Do we really own lock?
        BNEQ    INCONSISTENT                    ; Branch if no, then error.
        ; The next instruction releases the spinlock,
        ; and falls through if the lock was owned, else it bug checks.
        BBCCI   #SPL$V_OWN,SPL$B_SPINLOCK(R0),INCONSISTENT
        RSB
```

3.2.6.8 REIMAC Macro

The REIMAC macro will replace most REI instructions. It provides a mechanism for easily verifying that any owned spinlocks are released, before IPL is lowered. This mechanism is a design consistency check, intended to ensure that spinlocks are held and released correctly.

Since REI instructions are used to change from one access mode to another (such as from supervisor mode to user mode), not all REI instructions will be changing IPL. Therefore, the consistency checks should start by checking the current mode. If the current mode is not kernel mode, then no other checks are needed and the REI instruction can be executed immediately.

3.2.6.9 SETIPL Macro

The SETIPL macro will be used to change from one IPL to another. Through subsequent releases since VMS Version 1.0, it also locked out certain kernel mode activity to provide synchronization. However, on an SMP system, this would only lock out kernel mode activity on the currently executing CPU. All other CPUs could do whatever they want, potentially causing race conditions and unpredictable crashes. Therefore, it must be recognized that in the SMP system, the SETIPL macro is used to synchronize on only one CPU. Spinlocks must be used to extend the desired level of synchronization across all CPUs.

All code that references the IPL internal processor register directly ("MTPR xx, PR$_IPL"), will be changed to use the SETIPL macro.

The practice of locking pageable code into memory, using either a SETIPL or DSBINT macro, must also be modified to use spinlock acquisition. Note, however, that such practices are infrequent.

3.2.7 WHAMI Macro

A new macro, WHAMI (for WHO AM I), returns a unique physical CPU identifier which never varies in any mode of CPU operation. This CPU identifier is a longword number, used to differentiate one CPU from another in a multiprocessor system configuration. This number is only used to uniquely identify the local processor currently executing.

For example, on a BI-based system configuration, such as the 8300, the BI node ID of the requesting CPU is returned by this macro. On the VAX 8800, the Left/Right bit from the SID register is returned, thus providing physical ID values of '1' or '0',

respectively.

The CPU's unique physical ID is used, for example, when searching for the corresponding per-CPU database and interrupt stack during machine restarts. It can also be used in Error Logging to provide additional context for each error, if necessary.

WHAMI has one parameter, DST. The longword value for the physical CPU ID is placed in DST. The following is an example of how WHAMI is used.

```
        WHAMI   R0      ; Get Physical CPU ID

                        ; The WHAMI macro executes a case instruction on
                        ; the CPU type to find a unique, physical CPU ID
```

In this example, R0 will now contain a longword ID that uniquely defines a specific processor within a multiprocessor configuration.

Use of the WHAMI macro should be limited to specific areas of the system, where structures must be locked/unlocked by a given processor. Current uses include XDELTA's internal locking and unlocking of its read/write databases, and use of the WHAMI return as the destination of a interprocessor interrupt.

The WHAMI macro can only be executed while in kernel mode, because of the possible use of MFPR instructions. Furthermore, the WHAMI macro can only be used when running above IPL 2 to prevent rescheduling of the code thread to another CPU during execution of the macro. Care must be taken to prevent such rescheduling after issuing the macro for as long as the information returned by WHAMI is being used.

3.2.8 FIND_CPU_DATA Macro

The FIND_CPU_DATA macro takes a longword parameter and returns the virtual address of the per-CPU database for the CPU on which the code is currently executing. The FIND_CPU_DATA macro does this by doing the following:

1. Extract the current ISP value from IPR 4.

2. Perform a complemented AND operation (using a BICL instruction) upon the ISP value using the modulo mask, SMP$GL_BASE_MSK. The use of this mask rounds down the address extracted from IPR 4 to the base of the per-CPU context area, which is the per-CPU database itself.

This algorithm requires that each per-CPU database be aligned on the appropriate page boundary. For more information, see the section that describes the per-CPU context area.

The following is an example of how the FIND_CPU_DATA macro is used.

```
FIND_CPU_DATA  R0                    ; Get base of CPU database
MOVL    CPU$L_CURPCB(R0),R1          ; Get current PCB address
```

In this example, R0 now contains the base address of the current CPU's (that is, the CPU this code is executing on) database.

The FIND_CPU_DATA macro can only be used when memory management is enabled, and only by kernel mode code because the macro uses an "MFPR PR$_ISP,xx" instruction to find the ISP value. If kernel mode code is used only to locate the current PCB address and P1 space of the current process is available, then it is recommended that such code use the P1 location CTL$GL_PCB to find the current PCB address.

The FIND_CPU_DATA macro can only be used when running above IPL 2 to prevent rescheduling of the code thread to another CPU during execution of the macro. Care must be taken to prevent such rescheduling after issuing the macro for as long as the information returned by FIND_CPU_DATA is being used.

The FIND_CPU_DATA macro expansion for the above example is:

```
MFPR    #PR$_ISP,R0             ; Get current ISP value
BICL    G^SMP$GL_BASE_MSK,R0    ; Round down to per-CPU base
```

3.2.9 Forking and Stalling

The concept of forking has always been supported by VMS. Forking allows high IPL code to do the following:

- Continue executing a particular code thread at a lower IPL
- Synchronize with other code executing at the lower IPL

Usually, forking is done by drivers that service device interrupts at IPLs 20-23 (decimal), and then lower their IPL to 8-11 to complete the processing of the interrupt request that is not time critical and to synchronize with code threads initiating I/O. This permits other device interrupts to occur, while previous interrupt threads are completing.

On an SMP system, forking is still necessary. However, the VMS fork block will no longer contain the IPL at which the fork thread is executed. Instead, the fork block will contain a forklock number. The fork requesting code will use the forklock number to look up the IPL (at which to queue the fork thread) in the spinlock database. This approach serves two purposes:

1. It allows the requester of a fork thread to select the IPL at which the fork thread is to be executed.

2. It provides the fork dispatcher with a mechanism to acquire the correct spinlock just prior to executing the fork thread and no sooner.

The forklock number must be the same for all code threads that must synchronize with each other (that is, that fork to the same IPL). In its simplest form, a forklock simply synchronizes the right of a fork process to execute at a specified IPL. This is an example of the broad approach to extending VMS to operate as an SMP system, that is, take all existing IPL based synchronizations and simply extend them across the entire SMP system with a spinlock. However, multiple forklocks may be defined for any given fork IPL, thus providing a greater degree of parallelism at the fork processing level. Herein lies one difference between forklocks and other system spinlocks. A forklock is assigned at boot or run-time and the selected spinlock index is then used to lock the required database as needed. The spinlock index is saved in a byte field and is one of the arguments on the forklock macro, when attempting to acquire access to the particular database. A forklock is a general synchronization tool which can be used by any set of fork processes that must synchronize with each other for some unspecified resource, which may simply be the right of each fork process concerned to execute at a given IPL.

The second purpose listed above for implementing this kind of fork synchronization is very important in the concept of forking. Although fork threads can be used as mechanisms to lower IPL, they are also used to stall execution of a code path without lowering the IPL.

For example, consider two code paths of execution, threads A and B, executing at like IPLs. Thread A is a code path that (by nature of the subroutines that it calls) may stall (fork) to wait for the availability of a resource. (One such resource might be the allocation of UNIBUS map registers for a DMA device.) Thread B is the code path that releases some map registers and resumes the stalled thread A. It is possible that thread B did not use the same locks that code thread A used, just prior to being stalled. (Note that thread A must not stall before releasing any spinlocks it may have acquired, to prevent deadlocks.)

However, if instead we force threads A and B to use the same spinlock at the outset by identifying that spinlock as being required in order for one of these threads to execute at the desired IPL, deadlocks will not occur and all code paths will be suitably interlocked.

The example stated above is illustrated as follows:

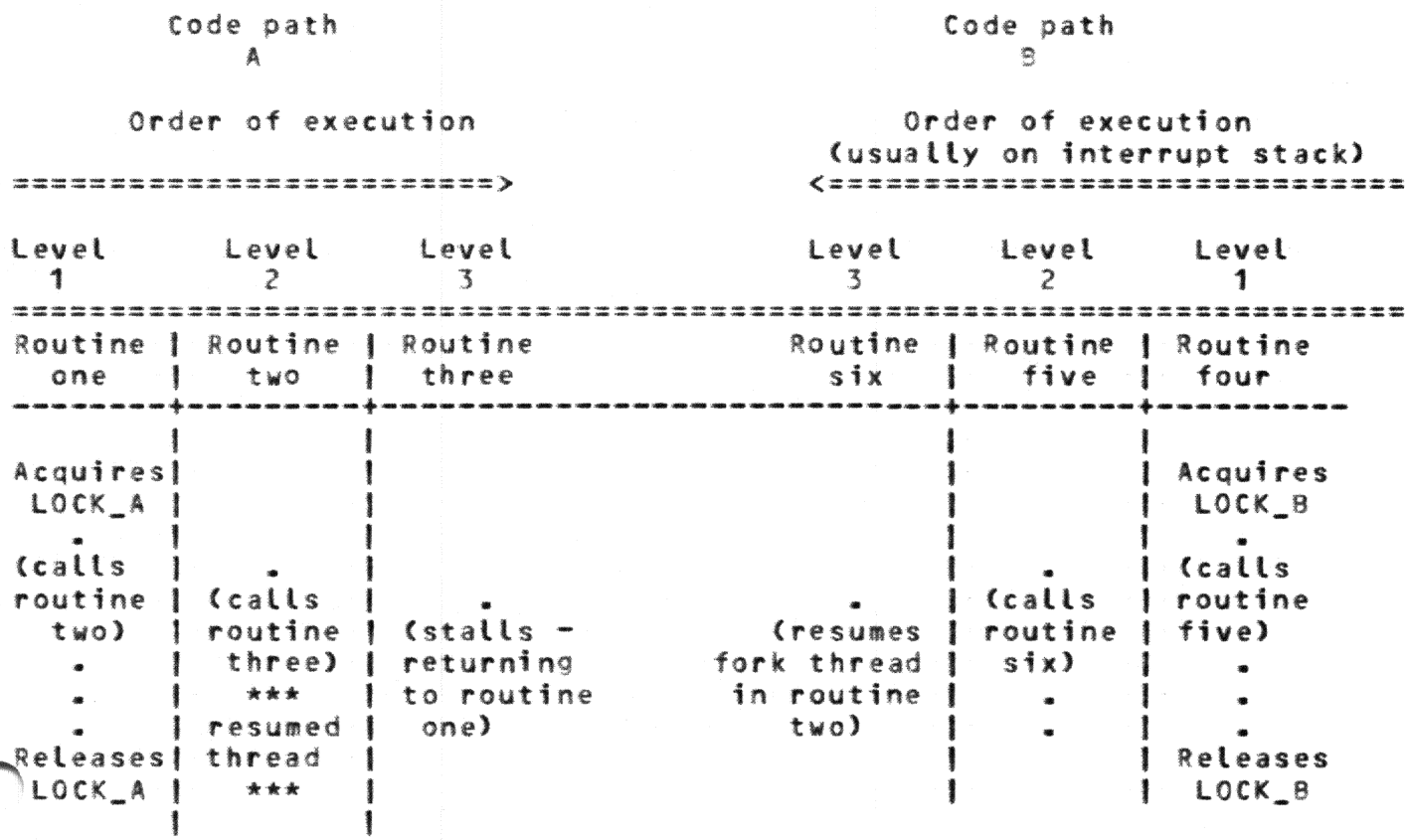


Figure 3: Fork Example

In the example above:

- The routines are executed in order: Level 1, Level 2, and Level 3.

- The Level 1 routines acquire and release the spinlocks (routines one and four in the example). Assume that thread A acquires spinlock LOCK_A and that thread B acquires spinlock LOCK_B.

- The Level 2 routine for code path A (routine two) calls a subroutine (routine three) which will either return immediately to its caller (routine two) or fork and return to its caller's caller (routine one). If the fork is not taken, the remainder of routine two will execute under LOCK_A. Then, routine two will return control to routine one and LOCK_A will be released.

  However, if the fork is taken, then routine two's context (at the time of the call to routine three) will be saved in a fork block. The remainder of the

execution of routine two is postponed. LOCK_A will be released when control is returned to Level 1, at routine one. Later, thread B will execute at routine four and LOCK_B will be acquired. Routine five will then call routine six to release a resource, which will cause the stalled fork thread, the remainder of routine two, to be executed. (This section of the example is highlighted by asterisks, "*"). However, routine two will now be executing under LOCK_B, which is not the same as if the resource had been readily available. Therefore, LOCK_A and LOCK_B must be the same lock for this to work.

Spinlocks should be taken out at the earliest (highest) level to allow for lower level routines to fork correctly. This is why we intend to have the fork dispatcher take out the predetermined spinlock, the forklock.

The fork dispatcher will only attempt to take out one spinlock. Therefore, routines that fork or stall must require only one spinlock, or they must take care to acquire additional locks that they may need after they are resumed from the fork or stall.

3.2.9.1 FORKLOCK Macro

The FORKLOCK macro is used to acquire a forklock spinlock using an interlocked instruction, to provide correct SMP synchronization. This macro is similar to the LOCK macro, except that it takes a spinlock number rather than a spinlock name. The FORKLOCK macro takes the following four parameters.

- LOCK - spinlock number of resource to lock
- LOCKIPL - IPL at which spinlock is acquired
- SAVIPL - optional, location to save current IPL
- CONDITION=(xx) - optional, special action
  where xx is:
  NOSETIPL - do not execute a SETIPL

The LOCK parameter selects the index of the particular spinlock in the system spinlock vector.

The LOCKIPL parameter is specified for design consistency checking (that is, for making sure that a spinlock is requested at a particular LOCKIPL). LOCKIPL ensures that spinlock acquisition requests are done consistently. (Although the LOCKIPL parameter is not absolutely necessary to the function of the FORKLOCK macro because the desired IPL is available from the spinlock database, it will be useful to developers reading VMS code to have the IPL in the listing and guaranteed as correct.)

The spinlock rank in the database is also checked to make sure that multiple spinlocks are acquired in a consistent order, to prevent deadlocks.

The SAVIPL parameter, if specified, will hold or save the current IPL before it is raised to the LOCKIPL.

The FORKLOCK macro cannot be used to lock a dynamic spinlock, nor can it be used to lock system spinlocks that are not intended to be used as forklocks.

3.2.9.2 FORKUNLOCK Macro

The FORKUNLOCK macro is used to release a forklock spinlock. This macro is similar to the UNLOCK macro. It takes the following three parameters:

- LOCK - spinlock number of resource to unlock
- NEWIPL - optional, IPL to lower to after spinlock is released
- CONDITION=(xx) - optional, special action
  where xx is:
  RESTORE - decrement the refcount and only unlock if the refcount is 0

The LOCK parameter selects the index of the particular spinlock in the system spinlock vector. The spinlock is released using an interlocked instruction to guarantee correct SMP synchronization.

After the spinlock is released, the IPL is set to the value of the NEWIPL parameter, if specified. This may or may not change the current IPL of the processor.

The FORKUNLOCK macro cannot be used to release dynamic spinlocks, nor can it be used to release the system spinlocks that are not intended to be used as forklocks.

3.2.9.3 System Routines Called During Device Initialization

There is a class of system subroutines that deserves special mention here. The class of subroutines includes:

- Pool allocation
- Memory management

o Process scheduling

These subroutines enforce synchronization of their function by forcing the IPL to a predetermined level, regardless of the caller's IPL. The theory is that no code threads will call these subroutines from an IPL that is higher than the subroutine's synchronization IPL, because by doing so, the caller would break synchronization of the subroutine's function.

For example, the system subroutine that allocates nonpaged pool has a synchronization IPL of 11. This subroutine sets (raises) IPL to 11 to synchronize all allocations of nonpaged pool. To call this subroutine from a higher IPL would be disastrous if that higher IPL code thread had pre-empted another code thread already in the process of allocating pool. The end result is likely to be nonpaged pool corruption.

These subroutines deserve special mention because there are cases in VMS today where this synchronization technique has been modified for special purposes. For example, device drivers are called at their unit and controller initialization routines by the SYSGEN utility and during VMS system initialization (module INIT). Some device drivers need to allocate nonpaged pool, but since these initialization routines are called at IPL 31, allocating pool would have the side effect of lowering IPL down to the pool allocation IPL. Lowering IPL in this manner cannot be allowed. Therefore, these device drivers take steps to prevent this from happening by 1) temporarily modifying the allocation IPL to be 31 for the duration of their particular pool allocation, and then 2) restoring the allocation IPL to 11, when their allocation is done.

In an SMP environment, taking such wanton liberties with the synchronization techniques will not always work. Changing the synchronization IPL is not enough to ensure synchronization across the entire multiprocessor system. The accepted way to achieve the desired function, in a synchronized manner, is for these system subroutine callers to lower IPL by forking to an IPL that is equal to or lower than the synchronization IPL of the subroutine that will be called. In calling the subroutine by first forking, that subroutine will then be able to successfully synchronize its function across the entire multiprocessor system.

A CRB fork block must be made available to allow these driver routines to fork as described. If the UNIT_INIT routine requires that the CONTRL_INIT routine must have completed before it runs, then it must wait by forking. Care must be taken here because the fork thread will execute at an IPL that allows device interrupts. If interrupts are allowed to occur before the fork thread can execute, then the driver must be ready to handle the interrupt right away. This may require, say, preallocation of pool at a higher IPL, or simply not enabling the device interrupts until after the fork thread has executed.

If the forking thread is not successful (for example, in pool allocation), it must fork and wait before attempting the allocation again. Use of the FORK_AND_WAIT mechanism that exists in VMS V4.0 is recommended.

3.2.10 I/O Subsystem

Changes to the I/O subsystem are necessary to fully synchronize an SMP system. These changes are described in the following subsections.

3.2.10.1 Fork IPL

Forking is a concept that is relevant to the functioning of the I/O subsystem, as well as to the functioning of other parts of the executive. The fork mechanisms, as described in earlier sections of this document, will be used by the I/O subsystem, instead of the current forking and fork IPL locking. Therefore, instead of only raising to fork IPL, a call must also be made by I/O subsystem routines to the forklock macro. This call will raise IPL and also acquire the appropriate spinlock. Thus, any I/O subsystem code that must synchronize with a driver's fork thread, must be at the same IPL and must synchronize its execution with that fork thread by acquiring the same forklock.

3.2.10.2 CRB Fork Block

The following considerations have made it necessary to add a fork block to each CRB:

- Controller initialization (during which there may or may not be any UCBs defined) may need to fork. For example, some executive routines may have to be called that require memory allocation and require execution at a lower IPL for correct synchronization.

- CRB timeout routines cannot lower IPL while holding a spinlock. Currently, in VMS V4.0, CRB timeout routines are called at IPL powerfail. In an SMP system, it is not legal to lower IPL below the IPL at which a spinlock being held had been acquired. It appears that the CRB timeout routines are called at the powerfail IPL, because there is no way to predetermine the correct IPL at which to call them. Powerfail is presumably chosen because this IPL is higher than all device synchronization IPLs. With the addition of a fork block in the CRB, it is possible to specify a forklock for that timeout code thread. Thus, when the CRB timeout

routine is called, it will be called at that fork IPL with the forklock already acquired. If the timeout code thread in the driver requires more synchronization, it must do so itself (for example, synchronize with the device's synchronization IPL, or with the powerfail IPL).

- If existing drivers have been modified to fork when necessary, rather than accomplishing the desired synchronization by bending some IPL rules, they will not be modified and should continue to work. A fork block will still be added to the CRB as a standard part of that data structure.

3.2.10.3 Device IPL Synchronization

There are three cases where a driver must run at device IPL:

- In its interrupt service routine
- Raising from its fork IPL, to synchronize with the interrupt service routine
- During a device timeout, to synchronize with the interrupt service routine

The second and third cases are synchronized with each other because both must have acquired the forklock before they attempt to raise IPL and acquire the device lock.

An interrupt service routine usually does not hold the forklock. However, it may have pre-empted a thread holding the forklock or a fork thread may be running in parallel on another CPU. Therefore, an interrupt service routine must not access or change any fields in the UCB that are protected by the forklock.

The devicelock is used primarily to synchronize access to the device or adapter registers. Also, a driver may have some storage in the UCB (or in another data structure) that must only be accessed under the devicelock.

To provide a locking mechanism at device IPL, an additional structure will be created by INIT, SYSGEN, and INIADP when the CRB is created. This structure will be a dynamic spinlock for the controller. A new field, CRB$L_DLCK, will be defined in the CRB to contain the address of this dynamic spinlock allocated for the CRB. When the corresponding UCBs are allocated for each unit of the controller, the address of the spinlock will be copied from the CRB to a new field in the UCB, UCB$L_DLCK. This spinlock must be acquired in addition to raising to device IPL

(DIPL), and must be acquired in each device's interrupt service routine.

The creation of multiple device spinlocks, one per controller (CRB), should increase the granularity of the locking database and help to achieve a high degree of parallelism in the execution of code threads on multiple CPUs. This should be true even for I/O-intensive systems that request many I/O operations.

3.2.10.4 DEVICELOCK Macro

The DEVICELOCK macro is used to acquire a device spinlock using an interlocked instruction, to provide correct SMP synchronization. This macro is similar to the LOCK macro, except that it takes the address of a dynamic spinlock as a lock parameter, rather than a spinlock name. The DEVICELOCK macro takes three parameters:

- LOCKADDR - address of dynamic spinlock to lock
- SAVIPL - optional, location to save current IPL
- CONDITION=(xx) - optional, special action
  where xx is:

  NOSETIPL - do not execute a SETIPL

The LOCKADDR parameter is the address of a particular spinlock that locks access to the device, controller, or adapter. No LOCKIPL parameter is specified, because device drivers will specify the desired device IPL in the UCB field UCB$B_ODIPL (the old DIPL field). The spinlock acquisition code can ensure that IPL is raised to the required IPL.

The SAVIPL parameter, if specified, will hold or save the current IPL before it is raised to the LOCKIPL.

The NOSETIPL value for the CONDITION parameter can be used in interrupt service routines where the IPL has already been raised to the correct level by virtue of the interrupt having occurred.

The DEVICELOCK macro must be added to every interrupt service routine to ensure correct synchronization in an SMP system.

3.2.10.5 DEVICEUNLOCK Macro

The DEVICEUNLOCK macro is used to release a device spinlock. This macro is similar to the UNLOCK macro. It takes three parameters:

- LOCKADDR - address of dynamic spinlock to lock
- NEWIPL - optional, IPL to lower to after spinlock is released
- CONDITION=(xx) - optional, special action
  where xx is:
  RESTORE - decrement the refcount and only unlock if the refcount is 0

The LOCKADDR parameter is used as the address of a particular spinlock that unlocks access to the device, controller, or adapter. The spinlock is released using an interlocked instruction, to guarantee correct SMP synchronization.

After the spinlock is released, the IPL is set to the value of the NEWIPL parameter, if specified. This may or may not change the current IPL of the processor.

The DEVICEUNLOCK macro must be added to every interrupt service routine, to correspond with any needed DEVICELOCK macro that is added.

3.2.10.6 Device Synchronization with Power Failure

The rules associated with raising IPL to the powerfail IPL to block powerfail interrupts require modification. A typical sequence for such device access synchronization will be the following:

1. Acquire device lock, thus raising IPL to device IPL (DIPL) and synchronizing access to the device in general
2. Raise IPL to the powerfail IPL to block powerfail interrupts during the critical path of the desired access
3. Access the required bits or data related to the device
4. Lower IPL to DIPL when it is no longer necessary to block powerfail interrupts
5. Release device lock, restoring the previous IPL and allowing access by other CPUs

The Wait For Interrupt Keep Channel system routine (WFIKPCH macro) will need to be modified. It will have to be called with the devicelock held. It will conditionally release the devicelock, before returning to its caller's caller (which should release the devicelock, if it was previously acquired). Therefore, if the caller's caller had acquired the devicelock, then the devicelock will still be held on return.

3.2.11 Process Scheduling

The scheduler has been changed to no longer run a NULL process. Rather, it executes in an idle loop, looking for any change to SCH$GL_COMQS. The memory cell SCH$GL_COMQS reflects the availability of computable processes, one bit for each priority level that has a computable process. The idle loop is executed at IPL 3 on the interrupt stack, with no spinlocks held. Therefore, when a process becomes computable, the SCH$GL_COMQS cell will be updated under the SCHED spinlock. All idle processors note the change to SCH$GL_COMQS and then contend for the SCHED spinlock. Only one processor will be granted access to the SCHED database at a time; work will be dispatched to processors as they enter the SCHED database (until no more computable processes are available). In essence, any change to SMP$GL_COMQS, while running the idle loop, is regarded as a hint to the availability of work to be done (work may currently be available, but by the time the CPU searches the work queues, there may be no more work). While the CPU is executing in the idle loop, the CPU$L_CURPCB field will point to the NULL PCB; the CPU$B_CUR_PRI field (of the per-CPU database) will contain a 255 (for the highest priority process executing).

Allowing the scheduler to run in the idle loop enables the code in RSE (which determines whether to execute a SOFTINT macro to reschedule the current process) to not issue the RESCHED request when the CPU is in an idle state. The idle loop will pick up the process as soon as enough REI instructions are executed to get back to the idle loop. In fact, not setting the CPU$B_CUR_PRI field to 255, will cause RSE to execute the SOFTINT macro and, when the processor eventually resumes the idle loop and schedules the computable process, the CPU will immediately return to a RESCHED state to again schedule the same process. (This is because of the pending RESCHED interrupt requested by RSE.)

These enhancements to the scheduler mean that a CPU issues a RESCHED request only if it needs to pre-empt the current process. At all other times, the CPU picks up computable processes for free, while running in the idle loop.

Because the idle loop is executing on the interrupt stack, it must be accounted for when considering null time for displays such as MONITOR. Since I/O requests complete on the interrupt

stack, we must also ensure that proper system accountability be maintained. Therefore, interrupt stack time, while executing at IPL 3 (with the CPU$L_CURPCB field pointing to the NULL PCB) is only counted as null time. All other interrupt stack time is counted as interrupt stack time. This implies that all mode times kept in the CPU data area are correct and need not be adjusted to subtract null time from either kernel or interrupt time.

Another enhancement can be made to have the idle loop check a bitmask of pending software interrupts (equivalent to the OR of all CPU's SISRs). When each CPU inserts a work item into one of the fork queues, it should also set a corresponding interlocked bit in the global memory cell, SMP$GL_FORKQS. When all items have been removed from the fork queue, then the summary bit will be cleared. The testing of the bit mask could be added to the idle loop.

3.2.12 Mutexes

Mutexes must be acquired and released with the SCHED spinlock acquired. This ensures correct synchronization in an SMP system. A mutex is used for data structures or resources that will be accessed over a lengthy period of time or when executing at low IPLs, such as IPLs 2 to 5. Routines that require a mutex will not have to be changed, because the code to acquire/release the SCHED spinlock will be placed in the mutex lock/unlock routines (SCH$LOCKR, SCH$LOCKW, SCH$UNLOCK, and so on).

Mutexes are usually acquired by processes. However, mutexes must also be acquired by system-level software that does not have process context. Currently, such system-level code executes at IPL$_SYNCH (the synchronization IPL) and checks the ownership count field of the mutex to determine if any other software owns the mutex. Depending on whether the mutex is owned, one of the following will occur:

1. If the mutex is not owned, then the system-level code continues. Although this code does not explicitly own the mutex, it works correctly only because no other code owns the mutex and no other code can possibly acquire the mutex as long as this code remains at this high IPL.

2. If the mutex is owned, then the system-level code must delay and try again later.

However, on an SMP system, mutex acquisition will not work as described above, unless the mutex is explicitly acquired. Therefore, two new routines have been added in module MUTEX to explicitly acquire a mutex for READ or WRITE access. These

routines are SCH$LOCKREXEC and SCH$LOCKWEXEC, respectively. The system-level code described above which was able to avoid explicit acquisition of the mutex will be modified to call these routines to synchronize access to the mutex across the SMP system. Also, a new routine has been added to module MUTEX to release a mutex with only system context. It is called SCH$UNLOCKEXEC.

As a first pass, the design team assumes that most of the mutexes in VAX/VMS Version 4.0 will remain mutexes for SMP. However, some of these may become spinlocks or interlocked queues, as needed.

3.2.13 Per-CPU Database and Stacks

There is certain information that must be kept for each CPU in an SMP system, for example:

- The interrupt stack address
- The current PCB address
- The physical CPU ID
- The logical CPU ID

Each CPU will be assigned its own copy of this data in a CPU-specific database, referred to in this document as the per-CPU database. The layout of the per-CPU database is defined by the $CPUDEF macro and illustrated in Figure 5.

A new system cell, SMP$GL_CPU_DATA, is the address of a vector of pointers that locates each per-CPU database, as noted in the following figure. This vector can be used by the System Dump Analyzer Utility (SDA) to locate and display each CPU's per-CPU database.

SMP$GL_CPU_DATA:

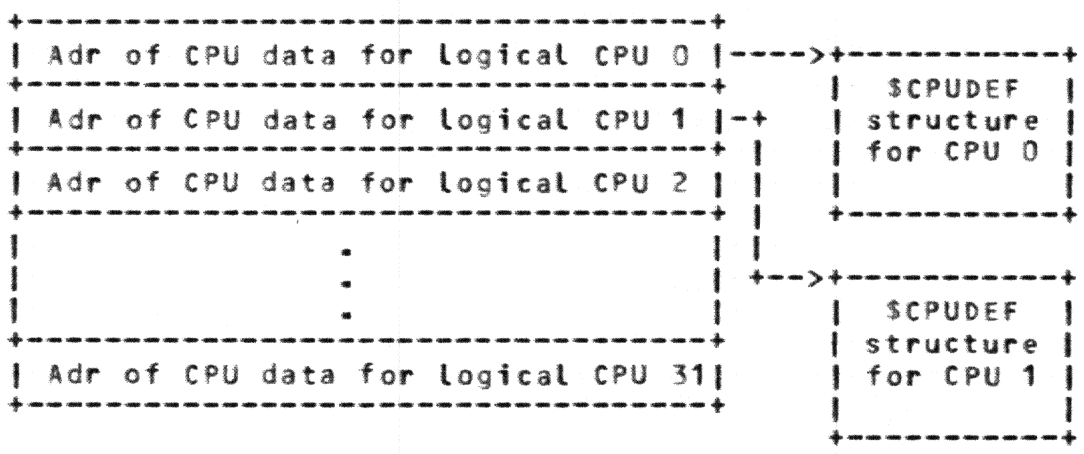


Figure 4: Per-CPU Vector Pointers

The CPU numbers in Figure 4 are logical CPU ID numbers. Logical CPU ID zero always indicates the boot CPU.

Also specific to each CPU is a pair of stacks for use by that particular CPU alone. There are two stacks defined for each CPU, the boot stack and the interrupt stack.

3.2.13.1 Per-CPU Boot Stack

The boot stack is virtually contiguous to the per-CPU database. It serves the purpose formerly served by the end of the Restart Parameter Block (RPB). It is a small stack that is used by the CPU whenever memory management is disabled, and is used virtually in certain exception scenarios where the normal interrupt stack for that CPU cannot be relied upon. The creation of separate boot stacks virtually eliminates the problem of having only one RPB and, thus, having only one boot stack (at the end of the RPB) for multiple CPUs.

It should be noted that the RPB will still be used as the boot stack when the boot CPU is first booted. As a part of the VMS system initialization sequence that the boot CPU will carry out, a separate boot stack will be created for the boot CPU and also for all other CPUs as they are brought online, thus obviating the need for the RPB to be used as a boot stack for any CPU from that moment on.

3.2.13.2 Per-CPU Interrupt Stack

Also defined for each CPU is an interrupt stack to be used in all normal VMS operations. This stack is only used when memory management is enabled. It is bracketed by two guard pages (marked as no-access) to prevent the stack pointer from exceeding the address bounds of the stack without detection. This stack, together with its guard pages, is virtually contiguous to the per-CPU database and boot stack that are described above. Thus, the entire per-CPU context is defined as a single virtually contiguous block of system address space. See Figure 5 for an illustration of the entire per-CPU context area.

3.2.13.3 Per-CPU Database Alignment

The virtual organization of each per-CPU context area provides an elegant mechanism to be used whenever a CPU needs to quickly and efficiently locate its own per-CPU database. This mechanism works by using modulo arithmetic, as follows.

First, the size of the per-CPU context area is calculated by adding the following items:

- The page count of the interrupt stack
- The two guard pages that bracket the interrupt stack
- The page count of the per-CPU database and boot stack

The resulting page count is rounded up to the closest power of 2. This modulo 2 value is then used to specify the system address space alignment at which the per-CPU context area will be allocated. Thus, the base of the per-CPU database will be aligned at a virtual page boundary that is consistent with the calculated power of 2.

This modulo 2 page count is converted to a byte count (less 1) and is saved at the global location SMP$GL_BASE_MSK. This bit pattern can then be used to round down any virtual address from within the per-CPU context area to the base of the per-CPU database (at the base of that context area). Such a virtual address is always guaranteed to be available in normal system operation by examining the interrupt stack pointer register. Because the interrupt stack is a part of the per-CPU context area, one can take the current interrupt stack pointer and apply the modulo 2 bit pattern stored in SMP$GL_BASE_MSK to quickly calculate the address of that CPU's per-CPU database.

The mechanism to find the base of the per-CPU database has been encoded into the FIND_CPU_DATA macro, the use of which is the recommended way to access the CPU's per-CPU database.

However, since this macro must execute more than one instruction, scheduling must be disabled while contents of the per-CPU data area are being retrieved.

3.2.13.4 Boot CPU's per-CPU Database

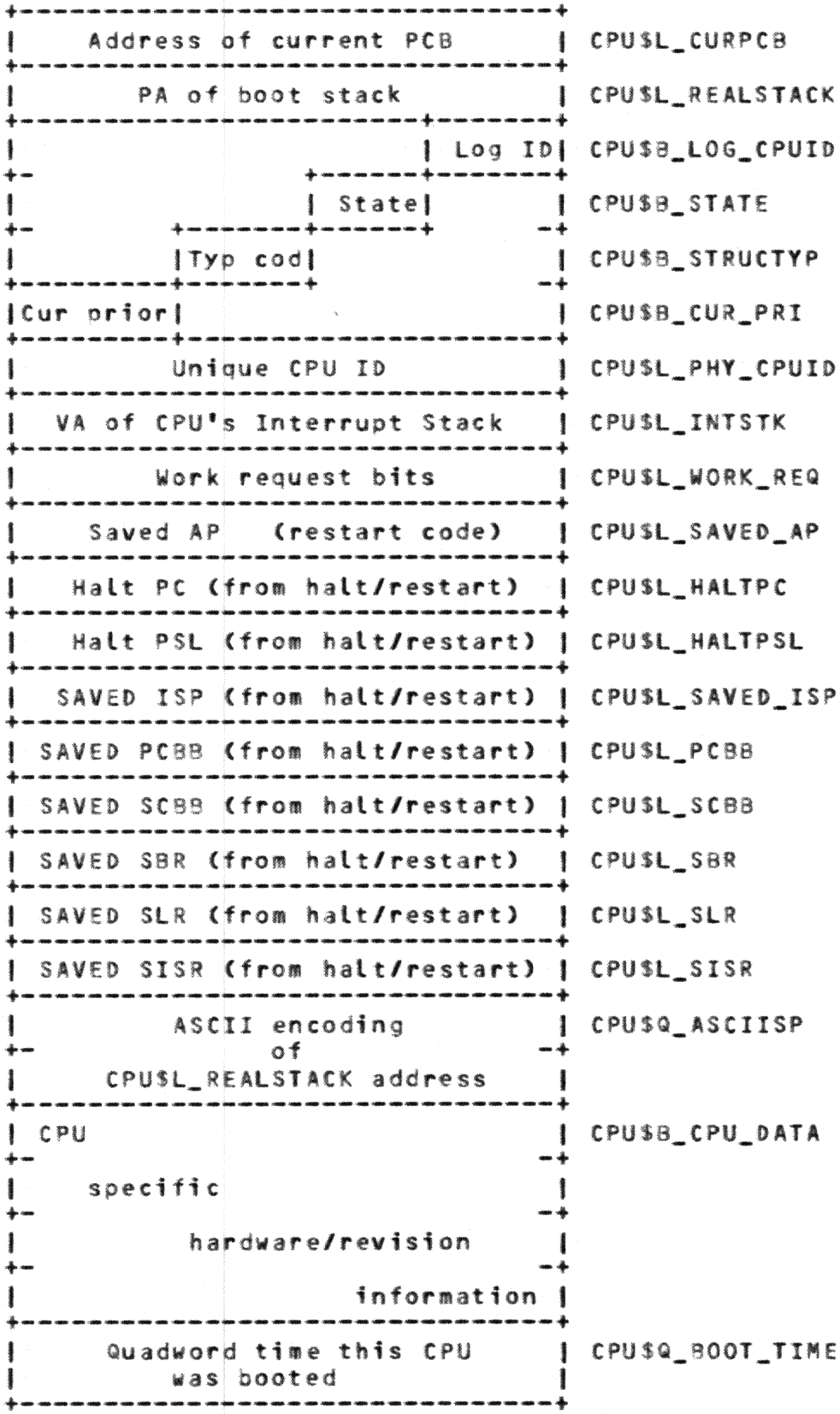
For the boot CPU, the per-CPU database is allocated by SYSBOOT (the secondary bootstrap program) and initialized by INIT (the system initialization routine). To allow for other potential CPUs on multiprocessor configurations, SYSBOOT will also reserve enough SPTEs in the system page table for the interrupt stacks and per-CPU databases (for the maximum number of supported CPUs). Code in SYSLOA (the CPU-dependent loadable image) will allocate SPTEs and PFNs for the interrupt stacks and per-CPU databases for all other CPUs as they are brought online.

3.2.13.5 $CPUDEF Structure

Figure 5 shows the contents of the per-CPU database that is defined by the $CPUDEF macro. Certain VMS V4.0 system cells, such as SCH$GL_CURPCB, have been relocated within the $CPUDEF structure for each CPU. This was done because these fields must be replicated once for each CPU in the system. Therefore, all references to cells (such as SCH$GL_CURPCB) must be removed. The following global cells have been relocated to the $CPUDEF database:

- SCH$GL_CURPCB, now structure offset CPU$L_CURPCB
- SCH$GB_PRI, now structure offset CPU$B_CUR_PRI
- EXE$GL_INTSTK, now structure offset CPU$L_INTSTK

The $CPUDEF data structure includes:



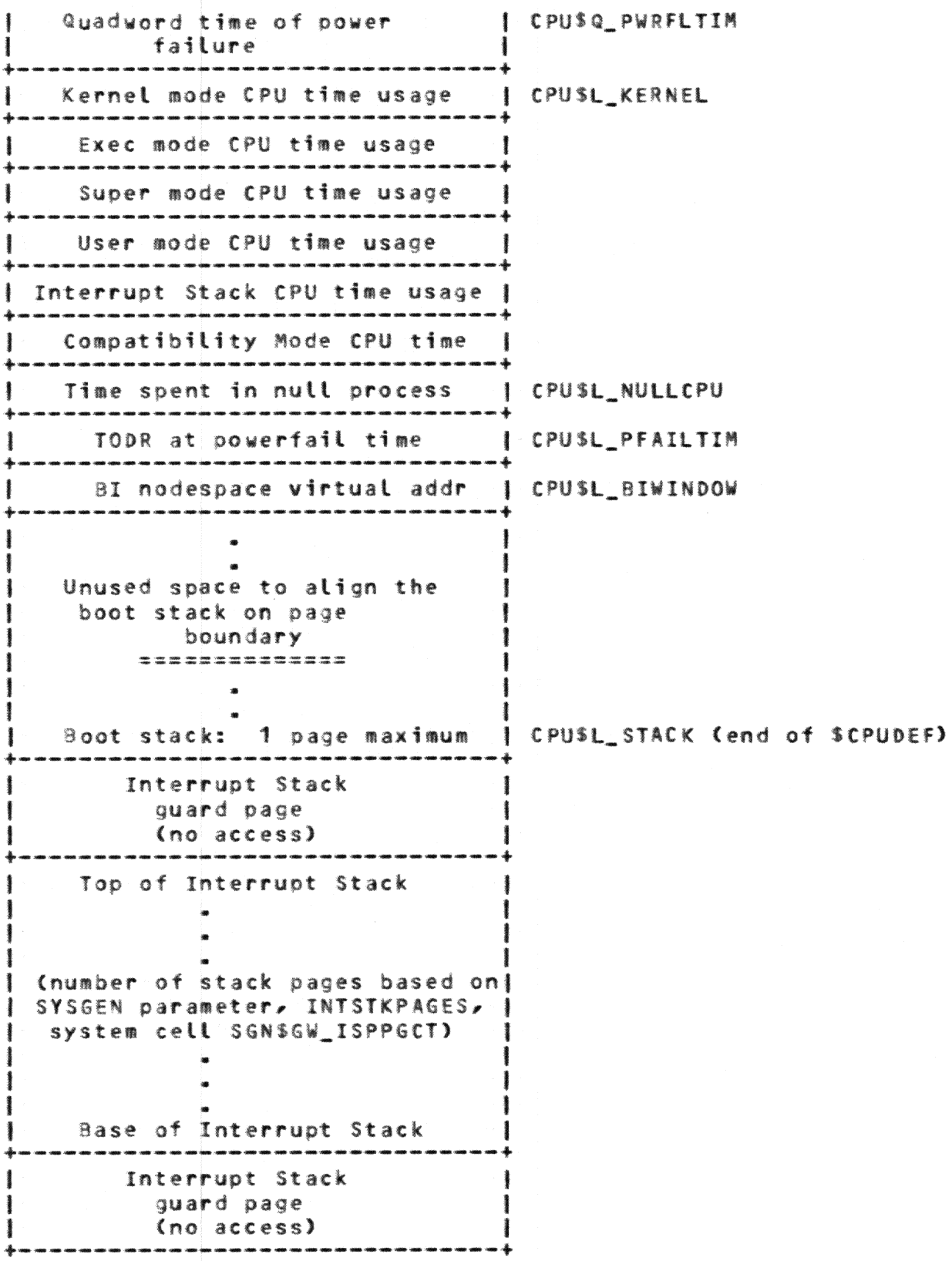


Figure 5: Per-CPU Database and Stacks

The following are CPU states (in CPU$B_STATE):

- CPU$C_INIT - CPU is initializing.
- CPU$C_RUN - CPU is running.
- CPU$C_STOPPING - CPU is stopping; that is, the CPU has seen the STOP request bit.
- CPU$C_STOPPED - CPU is stopped.

The following are work request longword bits (in CPU$L_WORK_REQ):

- CPU$V_INV_TBS - invalidate TB for address in SMP$GL_INVALID.
- CPU$V_INV_TBA - invalidate TB for all addresses.
- CPU$V_TBACK - acknowledge that the requesting CPU has invalidated its translation buffer and that this CPU can now proceed.
- CPU$V_BUGCHK - request a bugcheck.
- CPU$V_BUGCHKACK - acknowledge that CPU has saved process context and per-CPU data so crash CPU can continue doing bugcheck.
- CPU$V_STOP - request a CPU to stop.
- CPU$V_UPDASTLVL - request to update AST level.
- CPU$V_TIMKEEPER - indicates this CPU is keeping track of time for the entire SMP system and that it owns the console terminal. This CPU was also the boot CPU if the system has not switched the console terminal to another CPU, for any reason.
- CPU$V_UPDTODR - request to update the TODR IPR.

The NULL PCB may be referenced by CPU$L_CURPCB, in more than one CPU, at any point in time. Therefore, the NULL PCB time is cumulative for all CPUs. Since the longword that accumulates clock ticks for CPU time utilization in the NULL process's PCB is not interlocked, this longword does not necessarily contain accurate data. Thus, the idle time for each CPU is accumulated in a separate cell, CPU$L_NULLCPU, in the per-CPU data structure.

Usually, only one virtual address in the translation buffer (TB) is invalidated at a time. This is because the CPU that requests such action (for example, due to a change in the system

working set) must be holding the MMG spinlock. Occasionally it is also necessary to invalidate the entire TB. A global memory location, SMP$GW_INV_CNT, contains the count of CPUs requested by the CPU holding the MMG lock to invalidate the virtual address. The field is decremented by these CPUs with ADAWIs and continually tested by the requester until it reaches zero, indicating a successful system-wide invalidation. Work that one CPU needs to have done by another CPU could be implemented by either of the following:

- Inclusion of a work bit mask and fields that will hold the other necessary information
- Definition of a work queue

The description above includes a bit mask and the necessary fields, rather than a work queue.

The physical CPU ID is a unique identifier, returned as a longword (by the WHAMI macro), that identifies a particular CPU within a multiprocessor configuration. The logical CPU ID is simply the longword index of the CPU's per-CPU database in the SMP$GL_CPU_DATA vector. When the term CPU ID is used, the logical CPU ID is most often what is being referenced. Use of the physical CPU ID, except in such places as the interprocessor interrupt requesting, CPU initialization, and XDELTA, is not recommended.

3.2.14 New System Cells

A new system cell, SMP$GL_INVALID, holds the system virtual address that must be invalidated for all CPUs in the translation buffer. This location is initialized by the CPU that is changing the system working set; it is read by all other CPUs.

The system cell, SMP$GW_INV_CNT, is used in conjunction with SMP$GL_INVALID. When an invalidate request on a single page must be performed on a system space address, the following events must be performed:

1. The virtual address of the page is stored in the SMP$GL_INVALID cell.
2. The number of CPUs that must be notified for invalidation is loaded into SMP$GW_INV_CNT.
3. Every other CPU is notified of the invalidate request.
4. When the other CPUs invalidate the TB address, they must also decrement SMP$GW_INV_CNT (with an ADAWI) to indicate that they have done the invalidation.

5. The other CPUs must then wait for the INVALIDATE ACK.

6. The requesting CPU then waits for SMP$GW_INV_CNT to reach zero (or timeout). When the requesting CPU finds that SMP$GW_INV_CNT has gone to zero, indicating that the invalidate has been performed on all other CPUs, then the requesting CPU can perform its own INVALIDATE. This is followed by an INVALIDATE ACK to signal all the other CPUs to let them resume their interrupted threads.

The new system cell, SMP$GL_SMPFLAGS, tracks dynamic system flags. For example, one bit (SMP$V_CRASHCPU) will indicate that a CPU is entitled to be the crash CPU. Only one CPU can successfully set that bit first, thus becoming the crash CPU. The CPU that becomes the crash CPU must store its CPU ID in the cell, SMP$GL_CRASH_CPU.

A new system cell, SMP$GW_BUGCHKCD, holds the bugcheck code. It is written by the CPU requesting the bugcheck. The CPU taking the bugcheck will also write its CPU number into a new cell, SMP$GL_BUGCHKCP, to assist the System Dump Analyzer.

The base mask, SMP$GL_BASE_MSK, used to find the CPU database (depending on the current interrupt stack) has already been mentioned in the description of the FIND_CPU_DATA macro.

The system cell SGN$GL_SMP_CPUS is discussed in the following subsection.

3.2.15 Multiprocessor Configuration

There are two aspects of booting additional CPUs into an SMP instance of VAX/VMS. First, in a given multiprocessor system, there will be a mechanism that provides control over which CPUs are to be automatically configured by VAX/VMS. Secondly, determination of the set of processors that exist and are available for booting must be possible.

3.2.15.1 SMP_CPUS SYSGEN Parameter

A new SYSGEN parameter, SMP_CPUS (system cell, SGN$GL_SMP_CPUS), contains one bit for each possible CPU that is allowed by the system manager to be booted automatically as part of the SMP system. Because this field is a quadword, automatic booting of CPUs into the SMP system is limited to 32 CPUs. If this parameter is set to -1, then all CPUs that are 'available' will be configured in at boot time. If this parameter is set to 0, then a single CPU system is booted. (Later, a system manager can start an SMP system by issuing the DCL command START/CPU, to

start the other CPUs.)

The bits in SMP_CPUS are assigned to the CPUs based on the physical CPU number (that is, bit 0 represents CPU 0, bit 1 represents CPU 1, and so on). If the corresponding bit is set, then the respective CPU is started automatically in the SMP system (provided that the CPU is 'available'). If the corresponding bit is clear, then that CPU is to be excluded (not automatically booted) during system initialization.

NOTE

These bits represent physical CPU IDs. Therefore, only VAXes with physical CPU IDs in the range 0 to 31 can be configured automatically. All other CPUs must be explicitly started using the DCL command START/CPU.

The exact definition of 'CPU availability' is system dependent. If a CPU is available, then it has passed self-test, or some processor specific set of tests such that the console subsystem believes that the CPU is capable of executing VAX instructions.

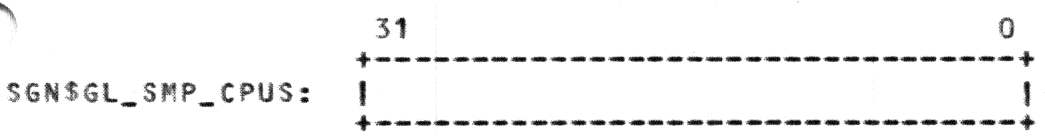


Figure 6: SMP_CPUS SYSGEN Parameter

3.2.15.2 Physical CPU Configuration

The SMP_CPUS SYSGEN parameter simply specifies which CPUs that actually exist in a given multiprocessor system are to be booted automatically into the SMP system during system initialization. This SYSGEN parameter has meaning only when it is known which processors do, in fact, exist.

It must be possible for VMS to determine the CPU configuration of the system during the VMS initialization sequence. Specifically, VMS must determine exactly how many CPUs are 'available' for booting into SMP operation, and what their physical IDs are. Once this set of CPUs is identified, then the SMP_CPUS SYSGEN parameter can be specified by the system manager to choose a subset to be booted automatically, provided that said physical IDs are in the range of 0-31, as described above.

3.2.16 Multiprocessor Initialization

A VMS SMP system will initially boot as a uniprocessor, just as is done for non-SMP instances of VMS. The processor that is selected by the hardware/firmware to be booted is, by definition, the boot CPU. It is expected that other processors are in a halted state before cold booting the boot CPU. This is true regardless of the reason for the cold boot sequence, whether it is originated by an operator request from the console terminal or by a request by the operating system software itself.

It is expected that the selection of a boot CPU will occur in a timely fashion. It does not matter to VMS how a CPU is selected by the console subsystem to be the boot CPU, as long as the selected CPU is, in fact, capable of booting VMS on that particular hardware configuration.

The boot CPU has the responsibility of bringing VMS up to a production level of operation. In the process of doing so, other CPUs may be brought online to begin execution.

3.2.16.1 VMB Execution

When the VAX first enters program mode, VMB has been loaded into memory and the stack pointer register has been initialized to point to the end of a physical page of memory which will be formatted by VMB into a Restart Parameter Block (RPB). This 'boot stack' can be used as the machine's stack until memory management is enabled (at which point the interrupt stack in the boot CPU's per-CPU context area will be used instead). Because no other CPUs will be running while the boot CPU is undergoing a cold-start sequence, there is no danger of corruption of the boot stack at the end of the RPB (by having multiple consumers of this stack). By the time other CPUs begin execution, the end of the RPB will not serve as a boot stack.

3.2.16.2 SYSBOOT Execution

VMB will load the secondary bootstrap program (SYSBOOT) into memory and transfer control to it. SYSBOOT carries out all traditional functions and also some additional functions that are specific to supporting symmetric multiprocessing hardware, to wit:

- Enough SPTEs are reserved in the system page table so that per-CPU context areas (including the per-CPU database and CPU-specific stacks) can be allocated in the future for the maximum number of CPUs (for the particular type of processor being booted).

- SPTEs are actually allocated for the boot CPU's per-CPU database, boot stack, interrupt stack and interrupt stack guard pages. The allocated SPTEs must be aligned so that the mechanism implemented by the FIND_CPU_DATA macro will correctly locate the base of the boot CPU's per-CPU database.

- Physical memory pages (PFNs) are allocated for the boot CPU's per-CPU database, boot stack and interrupt stack.

- The newly allocated SPTEs are mapped to the newly allocated PFNs, with appropriate page ownership and protection fields initialized. The pages are marked VALID. Note that no PFNs are allocated for the interrupt stack guard pages. The SPTEs for the guard pages are set up with a protection value of 'no-access'.

3.2.16.3 Execution in INIT (SYS.EXE) Module

SYSBOOT finishes its part of system initialization and transfers control to the INIT module of the VMS executive. In addition to all of its traditional duties, INIT will take the following steps related to SMP operation:

- Initialize the per-CPU database for the boot CPU. This is needed whether or not any other CPUs are ever brought online. For example, the address of the boot stack is initialized so that there is no further need to use the RPB as a boot stack.

- Store the virtual address of the boot CPU's per-CPU database in the first longword of the per-CPU database vector (SMP$GL_CPU_DATA). This position within the vector reflects the fact that the boot CPU has a logical ID of zero.

- Calculate and store the modulo base mask (SMP$GL_BASE_MSK), which is used by the FIND_CPU_DATA macro to locate the base of an issuing CPU's per-CPU database. This mask is directly related to the size of the per-CPU context area. It requires that the context area be aligned on an appropriate page boundary. From this point on, any CPU for which the interrupt stack pointer register has been loaded and memory management enabled can use this macro to locate its associated per-CPU database.

- Initialize SMP$GLPRIMID with the boot CPU's physical ID.

- Call the SMP$SETUP_SMP entry point in the system specific SYSLOA code to finish initializing the overall SMP environment and to bring additional CPUs online.

In current (non-SMP) versions of VMS, INIT modifies the synchronization IPL for nonpaged pool allocation to be 31. INIT then can use the standard pool allocation routines without incurring the undesirable side-effect of having IPL lowered too soon. As explained in the section describing "System Routines Called During Device Initialization", this is not guaranteed to work in a symmetric multiprocessing version of VMS and is considered to be bad form in the general case. Recognizing that INIT is a special case to begin with, INIT will accomplish the desired goal of nonpaged pool allocation without premature lowering of IPL by setting the synchronization IPL for pool allocation to 31 in the spinlock structures themselves. In fact, INIT will set the IPL of all spinlocks that it may have a need for to 31 (at least for the MMG and POOL spinlocks).

In summary, most spinlocks will be initialized by INIT to run at IPL 31, for the following reasons:

- To not restrict INIT's access to routines and their functions that are normally synchronized at an IPL lower than 31. This does not violate their synchronization requirements, because the system has never run below 31 at this point. Thus, there is no danger of having pre-empted a code thread running at a lower IPL.

- As insurance against other unforeseen problems similar to this one.

We will require that the forklock spinlocks not be set to IPL 31 to allow the system device to initialize itself in a fork routine.

The last code to execute in INIT will reset the IPL for all spinlocks to their correct values. Also, to have INIT correctly lower its IPL when it is ready to begin normal VMS operation, it will not simply RSB to the scheduler as it has done in the past. Rather, it will explicitly lower its IPL, using an REI with the SCHEDULING IPL and the address of SCH$SCHED on the stack.

3.2.16.4 CPU-Dependent Initialization - SYSLOA

For processor types that support symmetric multiprocessing, the SYSLOA code has the responsibility of completing the SMP environment and taking steps to bring other processors online. There are two major steps within SYSLOA that finish establishing the SMP environment.

It is expected at this point that the CPU configuration of the system is known. The mechanics of determining this configuration is processor-type specific. For NAUTILUS processors, this is done during SYSBOOT as a part of extracting general identification and revision information for the boot CPU. For SCORPIO systems, this is done by code that has been called by INIT to determine the configuration of all nodes on the VAXBI in general.

The first step in SYSLOA that is related to SMP initialization, at label SMP$SETUP_SMP, is called only once in the lifetime of the system from module INIT. The code in this section does the following bits of SMP initialization that are global in nature to the entire SMP environment:

- The scope of interprocessor and device interrupt visibility is expanded to include all processors and devices in the multiprocessor system. That is, adapters and other CPUs will be able to interrupt the entire set of CPUs that are running VMS. This strategy may or may not be modified to provide some kind of interrupt load-leveling across the participating CPUs.

- The address of the interprocessor interrupt service routine is loaded into the System Control Block (SCB). Prior to this step, this entry point in the SCB points to an unexpected interrupt handling routine.

- A physical page of memory is allocated and loaded with the first code that the new processor will be directed to execute. The code that is stored in that page is described below. This page is called the 'boot page'.

- Also stored in the boot page are various bits of information that will be needed by the new processor before memory management on that processor is enabled. For instance, the physical address of the RPB is stored in that page because the RPB contains information necessary for that CPU to find its own per-CPU database and to enable memory management.

- Another per-CPU database vector is established, analogous to the vector at label SMP$GL_CPU_DATA, but containing the physical addresses of the known per-CPU databases instead of the virtual addresses. This vector is ordered by logical CPU ID just as the virtual per-CPU database vector is ordered, and the boot CPU's per-CPU database is entered in the first cell in this vector. Again, this reflects the fact that the boot CPU has the logical ID of zero. This vector will be accessed by new CPUs as they begin execution in order to find their particular per-CPU context areas.

- The physical address of the vector of physical per-CPU database addresses is stored in the RPB, where it can be located by the new processor when it is booted.

- The SMP$SETUP_SMP routine finishes off by determining the set of additional processors that are to be brought online initially and calling SMP$SETUP_CPU for each of these processors. This set is determined by the value of the SMP_CPUS SYSGEN parameter and the actual CPU configuration that exists in the system.

The second step of SYSLOA's SMP initialization sequence is at label SMP$SETUP_CPU. This routine is called for every CPU (other than the boot CPU) that is to be brought online to operate in the SMP environment. The steps that must be taken for each CPU are:

- Allocate enough SPTEs to map the new CPU's per-CPU context area. Enough SPTEs must be allocated to ensure that the base of the new per-CPU database can be appropriately aligned for proper operation of the FIND_CPU_DATA macro.

- Allocate physical memory (PFNs) for the new CPU's per-CPU database, boot stack and interrupt stack. Map the newly allocated SPTEs to these PFNs.

- Initialize as much of the new per-CPU database as possible from the point of view of the boot CPU. When the new CPU starts executing, it will be in a position where it is able to finish initialization of its own per-CPU database.

- Execute the processor type specific procedure that is required to actually boot the target processor into the multiprocessing environment.

  For NAUTILUS processors, this procedure is to simply issue a console command that directs the console to boot the secondary processor. The address to be loaded into the secondary's PC register is that of a JMP instruction in the RPB, which in turn will cause the secondary to execute the code in the boot page. The end of the RPB is used as the secondary's boot stack, in the absence of some other means of specifying a per-CPU boot stack for the NAUTILUS secondary CPU.

  For SCORPIO multiprocessors, the boot procedure utilizes the VAXBI logical console interface to force-feed the target CPU with console commands. The boot CPU will, therefore, provide the PC of the boot page directly, as well as a per-CPU boot stack address.

5.2.16.5 New Processor Execution

When SMP$BOOT_CPU has finished executing, the new processor will enter program mode and begin executing code. The PC for the new processor has been set up so that the page of initialization code set aside just for this purpose by the SMP$SETUP_SMP routine in SYSLOA is where the CPU begins processing. This is the boot page. Also, the stack pointer in the newly booted processor is pointing to its own boot stack, or to the end of the RPB, for SCORPIOs and NAUTILI, respectively. It is hoped that future multiprocessors will provide a means of specifying a per-CPU boot stack for a processor before it begins executing VAX instructions, as is done for the SCORPIO. See the section called "Machine Restarts" for more discussion of the RPB's boot stack.

The code that the new processor must execute consists of the following steps:

- Locate its own per-CPU context area. Note that this cannot be done using the mechanism in the FIND_CPU_DATA macro because that mechanism relies upon contiguous virtual address space for the entire per-CPU context area. The contiguity of these pages is by no means assured when memory management is disabled. Therefore, the processor searches the vector of physical per-CPU databases previously set up for it until it finds a per-CPU database that contains its physical CPU identifier. Note that the physical CPU identifier field (and all fields that are required before memory management is enabled) must lie in the first page of the per-CPU database. The address of this per-CPU database vector is stored in the RPB which is accessible to the code in the boot page.

- Set up any immediate processor context that is required before memory management is enabled.

- Enable memory management.

- Load the interrupt stack pointer so that subsequent invocations of the FIND_CPU_DATA macro can work as expected.

- Finish initializing the per-CPU database. Perform some sanity checks, such as verifying that the CPU is at a revision level that is compatible with that of the boot CPU.

- Lower IPL and begin processing within VMS as a full partner in the SMP environment.

3.2.17 Interprocessor Interrupt Requirements

The operation of the SMP system depends, of course, upon the ability to generate interprocessor interrupts. This mechanism is used for a number of reasons in SMP, as described in the next section of this document. Described here are some general statements about the required functionality of interprocessor interrupts in an SMP instance of VAX/VMS.

In short, VMS requires the ability to generate an interprocessor interrupt from one CPU to a subset of other CPUs. This subset may consist of a single CPU, it may consist of all CPUs that exist in the SMP environment, or it may consist of a smaller number of CPUs that lies between these two extremes.

Note that there is no requirement that the interprocessor interrupt mechanism (in any given VAX implementation) be able to generate an interrupt from a CPU to itself. There is no perceived need in VAX/VMS for one CPU to interrupt itself via the interprocessor interrupt mechanism.

There is no requirement for a CPU that has taken an interprocessor interrupt to be able to discern the originator(s) of that interrupt. This level of knowledge will be embedded in software mechanisms that will be required anyway to sort out the reason(s) for the interprocessor interrupt.

SCORPIO systems provide the required functionality as described above by allowing a CPU to specify a bitmask of BI node numbers that specify which subset of CPUs is to receive an interprocessor interrupt. A CPU may specify a single bit in the mask that corresponds to a specific CPU to be interrupted, or multiple bits in the mask may be specified to interrupt multiple CPUs.

NAUTILUS systems provide the required functionality by virtue of there being only one other CPU to be interrupted. That is, the subset of CPUs that can be interrupted always has one member only, the opposite CPU.

3.2.18 SMP Use of Interprocessor Interrupts

Interprocessor interrupts are used for the following:

1. Request other processors to carry out some work request action

2. Coordinate some action that requires the cooperation of more than one processor

An interprocessor interrupt is requested by calling a CPU specific routine in SYSLOA. The physical CPU ID is passed in R0. Prior to calling the routine, some data is usually written to a request specific location (for example, SMP$GL_INVALID) and a work request bit is set in the CPU$L_WORK_REQ field in the per-CPU database of the processor being interrupted.

The work requests as currently defined are:

- Bugcheck request.

  Another processor is bugchecking and is requesting the current processor to save some state and get out of the way. The current processor 1) raises IPL to 31, 2) if it is running a process it does a SVPCTX, 3) saves all volatile machine state, including GPRs, 4) it acknowledges the bugcheck request and, 5) finally, it loops.

- Translation Buffer (TB) Invalidate request.

  There are two cases here: 1) A specific system virtual address is to be invalidated because the requesting processor wants to turn off the valid bit in the PTE (typically because it is executing FREEWSL or PAGEFAULT). 2) The entire TB is to be invalidated because the requesting processor is doing a TBIA (Translation Buffer Invalidate All) (typically because a balance set slot is being reused and stale system space addresses, containing process page tables, must be invalidated).

  Both cases are handled similarly.

  1. The requesting processor writes the address to be invalidated or zero (if it is an invalidate all request) into SMP$GL_INVALID and the number of processors it is requesting into SMP$GW_INV_CNT. It then requests all other processors to invalidate their TBs by setting the appropriate work request bit and sending an interprocessor interrupt.

  2. The requesting processor then waits for acknowledgment that all TBs are invalidated by testing for SMP$GW_INV_CNT going to zero.

  3. The other processors invalidate their TB and decrement SMP$GW_INV_CNT with an ADAWI instruction.

  4. The other processors then wait, without ever referencing the virtual address just invalidated, until the requesting processor tells them to proceed. They do this by waiting for a bit (SMP$V_TBACK) in their work request field to be set.

5. The original requesting processor is now free to copy the M bit from the PTE and write it and the V bit back to the PTE clear, without the fear that some other source will access the PTE uninterlocked.

6. The requesting processor then invalidates its TB and tells the other processors to proceed by setting the bit (SMP$V_TBACK) in each processor's work request field.

This is all very inelegant, but it is necessary because DEC STD 032 does not require that accesses to PTEs be interlocked. Thus, one processor could turn off the V bit while another is trying to set the M bit, and neither CPU would get the correct result. However, invalidating system virtual addresses is a relatively infrequent occurrence.

NOTE

This can only happen on one CPU at a time and is controlled by the MMG spinlock.

- Update AST Level Register request.

A processor has queued an AST to a process that is currently executing on another processor. The interrupted processor sets the ASTLVL register to the value of the mode that the AST is to be delivered in, for example:

- Kernel mode = 0
- Exec mode = 1
- Super mode = 2
- User mode = 3

It gets the value from the process header of the process that it is currently executing.

- This Processor Receives a Stop request.

A STOP/CPU command has been executed. This processor must return the process it is currently executing to the scheduler queues and stop. The processor sets its state to stopping and requests a reschedule interrupt. The rest is handled by the scheduler. If the STOP/CPU command is executed on the processor to be stopped it merely changes the processor

state to stopping and requests a reschedule interrupt.

- Update Time of Day Register.

  A $SETIME system service has been executed. This processor must update its time-of-day register.

3.2.19 Machine Restarts

The machine context that is required for handling VAX/VMS restarts is stored in the Restart Parameter Block (RPB) that was built when VMS initialized itself. Much of this context is specific to the CPU, however, which will cause problems when there is more than one CPU that can potentially restart at any given moment. To give more dimension to the function of the RPB, that context which is CPU-specific was moved into each CPU's per-CPU database.

This enables each CPU to independently undergo any kind of halt/restart condition, whether it be a powerfail recovery restart or a microcode detected error halt condition, without the danger of corrupting another CPU's context or itself being stepped on by another CPU undergoing a restart sequence.

3.2.19.1 Powerfail Interrupt Service Routine

When executing the power-down sequence, the local CPU will save the ISP, PCBB, SCBB, SBR, SLR and SISR registers in the per-CPU database. This data was formerly stored in the RPB, but since the values in these registers may be different for each CPU in an SMP environment, they must be stored in a CPU-specific location, the per-CPU database. Other than these registers, all other context saving will occur as is currently done for non-SMP instances of VMS. Some volatile context will be stored in the RPB and the remainder will be stored on the CPU's interrupt stack in memory.

The interrupt service routine finishes by executing an REI instruction to flush any write buffers to memory, thus ensuring that all of the machine's volatile state is completely saved. The routine then loops forever. It loops rather than halts to avoid confusing the console subsystem with an unexpected error halt situation before the actual loss of power.

NOTE

The use of the REI instruction in this instance does not require that the next instruction execute after the entire contents of the write

buffer has been flushed. The use of the REI instruction merely requires that the write-back operation be initiated and that the local CPU operations remain coherent. Therefore, if a VAX system implemented an asynchronous write-back of the write buffers (that is, a dump-and-run write-back), then the design would still work.

3.2.19.2 Error HALT Conditions

There are a number of very serious conditions that are detected by microcode and which result in the processor being halted by force. For instance, an invalid interrupt stack pointer, a CPU double error fault, and the issuance of the HALT instruction itself are examples of conditions that result in the machine coming to a halt.

VMS does not get immediate control of these situations, by definition, but must be prepared to take appropriate action when the machine is restarted by the console subsystem.

3.2.19.3 Recovery

It is expected that the VAX is set up to automatically attempt a system restart when either a power failure occurs or following the occurrence of an error halt condition. At the moment that the console subsystem actually restarts the machine, there is little distinction about the kind of restart that is occurring. As specified in DEC STD 032, the reason for restart is stored in the AP register prior to restarting the processor. DEC STD 032 is less clear about the contents of R10 and R11, but the PC/PSL of the error halt may be loaded into these registers as well.

When the restart sequence is executed, the RPB (of offset 4) will point to a common restart routine to be executed by all CPUs. The CPU will begin execution at this routine with memory management disabled and with its IPL set to 31. The stack pointer register may point to the end of the RPB, the default boot stack. It is very important that this stack not be used as a stack at this point, so the restart routine will immediately locate its own boot stack, if necessary, to be used instead of the RPB boot stack, without incurring stack operations in the process of doing so.

The address of the vector that contains the physical addresses of the known per-CPU databases, which in turn contains the addresses of the CPU specific boot stacks, is found in the RPB. The RPB is available to the restart code.

NOTE

Herein lies a somewhat dangerous situation. If a CPU suffers a machine check at the point where the RPB is being used as a boot stack, the machine check frame will be pushed onto that stack by the hardware/firmware. If more than one CPU gets a machine check during the phase of locating its real boot stack, then multiple, interspersed machine check frames will be pushed into the same stack space.

A way to avoid this is to locate and use the per-CPU boot stack before the VAX enters program mode.

The correct per-CPU database is found by searching through the vector of per-CPU databases and comparing the physical CPU ID in each database with the searching CPU's own physical ID that is returned to it by the WHAMI macro. When the per-CPU database for the CPU is found, the address of the CPU's boot stack is found at offset CPU$L_REALSTACK in the per-CPU database. This address is loaded into the stack pointer, at which point the RPB's boot stack will cease to be of concern.

The VMS restart routine will then enable memory management and examine the contents of the AP register to determine the reason for the restart sequence. If the reason that the machine is being restarted is anything other than a powerfail recovery sequence, then this represents a situation where VMS has no choice but to bugcheck. It does so immediately to preserve as much context around the error halt as possible to facilitate analysis of the problem later.

If the restart routine is entered because of the restoration of power, then each CPU will independently restore its own specific machine context that was saved during the powerfail interrupt processing. However, there is a portion of powerfail recovery that is system-wide in nature, such as device driver reinitialization, and properly should done by only one CPU in the SMP environment. The other CPUs must be made to wait until this system-wide powerfail recovery is completed.

We believe that two control bits can be defined to enforce synchronization of the system-wide powerfail recovery sequence. These control bits are cleared during powerfail interrupt processing. They will be used to control the power-up sequence as follows:

- Each CPU tries to set the first bit after it has completed its CPU-specific portion of powerfail recovery. The first CPU to set this bit will have the

responsibility of executing the system-wide powerfail recovery sequence on behalf of all the other CPUs. This implies that the recovery CPU has access to all I/O devices.

- When that CPU has finished doing the system-wide powerfail recovery sequence, it sets the second bit as a signal to the other CPUs that the sequence has been completed. This CPU then resumes normal VMS operations.

- The other CPUs, having failed to set the first bit first, will begin examining the second bit which will indicate when the system-wide powerfail recovery sequence has been completed. When the second bit is set, that is the signal that these CPUs may resume normal VMS operations.

3.2.19.4 Restartability Versus Non-restartability

The above discussion assumes that the member CPUs of the multiprocessor system are all independently restartable for all machine restart situations. There is the possibility that only one CPU within a multiprocessor configuration may be restartable, while the remaining CPUs will simply remain halted after incurring one of the various types of halt condition. This, of course, is not very symmetric, but the existing multiprocessor VAXes (SCORPIO, NAUTILUS) are known to behave this way to varying extents.

In systems such as these, there are additional responsibilities placed on the one restartable CPU and VMS, as follows:

- The restartable CPU must be able to detect, in some fashion, the need to restart another CPU that is itself not restartable. Furthermore, it must be able to detect the reason why that CPU had incurred the unrestartable halt condition.

- The mechanism for detecting a halted CPU and the reason for the halt must be made available to VMS running on the restartable processor.

- A mechanism must be provided for VMS to take action and do whatever is required to restart the halted CPU. When a halted CPU resumes program mode operation, it should have not have to care whether the restart action was carried out by VMS running on another (restartable) CPU, or by the halted CPU's console subsystem, as would be the case if that CPU was independently restartable.

These are fundamental requirements for proper recovery of unexpected halt situations in an SMP system. If all CPUs in an SMP system are independently restartable for any halt condition, then there is no need to rely upon VMS to execute restarts of other CPUs, by definition. For any halt condition that is not restartable by the console subsystem for a subset of CPUs, the requirements above enable VMS on the restartable CPU to lend the console a hand to extend the property of restartability to the halted CPU(s). Either approach produces a wholly restartable SMP system.

3.2.20 New Interlocked Queues

Any data structure that needs to be accessed by multiple processors simultaneously, without holding a spinlock, must be accessed using interlocked instructions. This can be done either for efficiency reasons or because the data structure must be accessed from multiple IPLs. Spinlocks must always be acquired and released at the same IPL. Consequently, a number of data structures have been changed to be self-relative interlocked queues. The following queues have already been changed:

- The PQB look-aside list
- The IOPOST queue
- The fork queues
- The nonpaged pool look-aside lists
- The DELWCB queue

As more such queues are identified, they will be similarly changed.

3.2.21 XDELTA Enhancements

XDELTA has been enhanced to support multiple processors. XDELTA continues to maintain a single database of breakpoints and to have the same command interface. Thus, when a breakpoint is set, it will be visible to all processors. A breakpoint set from a console attached to CPU A will result in CPU C entering XDELTA upon execution by CPU C of the code path that contains the breakpoint. XDELTA allows only one processor to be actually in XDELTA at a given time. Any other processor trying to enter XDELTA will wait until the first CPU has left XDELTA. Indivisible single stepping of instructions is supported on a per-processor basis. This means that if CPU A is in XDELTA and it is single-stepped, then the single instruction will be

executed without any other CPU entering XDELTA.

Specific enhancements to XDELTA have been to add a busy flag that is tested and set with interlocked instructions. This controls access to XDELTA. If the least significant bit is set, XDELTA is busy.

XDELTA now maintains a private SCB. Upon entering XDELTA, the processor's SCBB register is changed to point to the private single page SCB that XDELTA maintains. All vectors in this SCB point to an error handler in XDELTA. Thus, any error encountered in XDELTA is vectored without affecting any of the other processors which are sharing the standard SCB. Note that XDELTA's SCB is only one page long. This is satisfactory as XDELTA only runs at IPL 31 during the time that this SCB is being used.

Knowledge of which CPU is currently in XDELTA for use during single step operations is determined by use of using the WHAMI macro.

3.2.22 Error Logging

There are currently two designs for enhancements to the error logging procedure for SMP operation: 1) a short-term solution, and 2) a long-term solution.

The short-term solution is to increase the number of separate control buffers from 2 to 8, and to provide a spinlock to access the next available/last written buffer. This design does not provide for separate buffers for each CPU in an SMP system and is prone to loss of error log entries if one CPU is continuously logging errors.

Therefore, there is also a long term solution for enhancing the error logger that includes providing separate control buffers for each CPU (two buffers per CPU). The buffers are queued to a central error log processing queue, using an interlocked queue. A bit in the control buffer is set to signal that the buffer is in use and that it is being processed by the error log process. When the error log process copies the contents of the error log buffer, the buffer is removed from the queue and the interlocked bit is cleared, signaling the CPU that the buffer is again available when the next error has to be logged.

In either case, definitive identification of which CPU logged the error is required. This may involve machine serial numbers and/or physical CPU ID values.

3.2.23 Shutting Down a CPU

TBD (MH)

3.2.24 Taking a Bugcheck

VMS SHUTDOWN OR TAKES A BUGCHECK

- Initially as a uniprocessor. One CPU either starts the shutdown sequence or takes a bugcheck.
- Optionally a "good" CPU becomes the "boot" CPU and takes control of the CONSOLE TERMINAL. This is controlled by VMS interacting with the console subsystem.
- The "boot" CPU is requested to complete a shutdown or bugcheck.
- The "boot" CPU then controls the shutdown or bugcheck.
- All of the other CPUs are asked by the "boot" CPU (by setting the appropriate bit in each one's CPU specific WORK REQUEST field) to fold up their current context and stop.
- They inform the "boot" CPU (by setting the appropriate bit in their CPU specific WORK REQUEST field) when they are finished.
- They then wait in in a spinloop at IPL 31.
- The "boot" CPU continues as a uniprocessor shutting down or by taking the bugcheck.
- Finally the "boot" CPU either loops or requests its CONSOLE PROGRAM to initiate an AUTOREBOOT by writing "F02" to its TXDB register.
- It resumes at START rebooting itself. Note it owns the CONSOLE TERMINAL and is the "boot" CPU.

All GPRs and IPRs will be saved for all CPUs that are running VMS.

3.2.25 Device Affinity

The design of device affinity in the SMP system is based on the following rule:

- All devices must be accessible to the primary CPU

3.2.25.1 Goals

The following list represents the primary goals for the design of device affinity on SMP systems:

- Affinity is absolutely required for certain CONSOLE SUBSYSTEMS
- A generalized form of device affinity that can work for devices other than those connected to the CONSOLE SUBSYSTEM.

The following list represents the secondary goals for the design of device affinity on SMP systems:

- Load leveling capability
- Improve real-time device responsiveness through device segregation
- Allow support of asymmetric hardware configurations such as the VAX-11/782

3.2.25.2 Device Affinity Design

First we must define some terms that will be used in the description of device affinity. The term physical connectivity represents how a device is connected to the CPU(s) in an SMP configuration. For example, a device can be connected to all CPUs in an SMP system via a common bus, such as the Backplane Interconnect (BI). A device can be connected directly to a CPU via a special bus (with an accompanying set of processor registers), such as a console terminal on a VAX 8300. In the former case no physical connectivity to a specific CPU (that is, the primary CPU) is required. In the latter case, physical connectivity is required. And since our rule states that the primary CPU must have access to all devices, when physical connectivity is required then it is required to be to the primary CPU.

Extending the physical connectivity to represent a set CPUs (that could possibly exclude the primary CPU) is a violation of the rule that the primary CPU has access to all devices. Also, since all known and planned CPU designs allow connectivity either to the primary CPU or all CPUs, we can take advantage of this

knowlege and use a single bit to represent physical connectivity. It should be noted that the design can be extended to represent a subset of the CPUs as long as such a subset includes the primary CPU.

The term device affinity represents the notion of which CPUs can initiate an I/O operation. As such, device affinity is a subset of physical connectivity that can change within the bounds of the physical connectivity requirements. For example, in the case of a BI device attached to a 4 CPU VAX 8300 system, the affinity for the device can be any subset of the set of CPUs {1,2,3,4}, assuming that this is the set of CPU IDs in the system. In the case of the console terminal on the VAX 8300 system, device affinity is bound to the primary CPU and can never be changed.

The use of device affinity can be used to implement a crude form of software load leveling by restricting certain portions of I/O processing to be performed on certain CPUs. Note that unlike the physical connectivity indicator, device affinity can be set to a CPU that is not the primary.

Lastly, there is the notion of which CPUs can service a device interrupt. Obviously, the set of CPUs that can service a device interrupt must be governed by the physical connectivity indicator. For example, if the physical connectivity bit is on, then device interrupts cannot be directed to a CPU that is not the primary CPU. Setting device interrupts to a subset of CPUs is a form of hardware load leveling. On certain VAX systems, the set of CPUs destined to receive interrupts from a particular device may be limited to a single CPU. This would be the preferred method on systems (such as 8800s and 8300s) that incur passive releases when more than one CPU is destined to receive the device interrupt. On other systems (such as CALYPSO) the device interrupts should be set to all CPUs and the hardware can load level the device interrupts, without passive releases.

Also note, that the device interrupt mask is a subset of the physical connectivity, yet it may be a different subset from the device affinity mask. Therefore, the device interrupt mask and the affinity mask may be treated as totally independent entities.

Given the preceeding definitions, the SMP design of device affinity has three levels of control: a physical connectivity indicator, a device affinity mask, and a device interrupt mask. The physical connectivity indicator is a bit, and the other two are 32 bit masks that represent 1 bit for each CPU in the SMP system.

The physical connectivity indicator is a binary bit, that if set indicates that physical connectivity of the device is restricted to the primary CPU (that is, affinity is set to the primary CPU). If the bit is not set, then no physical connectivity is required of the device. This connectivity is

selected on a per unit (UCB) basis. The physical connectivity indicator is selected by setting bit TBD in the UCB$L_DEVCHAR field of the specific UCB.

The physical connectivity indicator bit is checked whenever there is an attempt to switch affinity for a device from one CPU to another. If the physical connectivity bit is set and an attempt is made to switch connectivity away from the primary CPU, then an error is returned on the switch request.

The device affinity mask represents the set of CPUs that can initiate I/Os to a particular device unit (UCB). The affinity mask will be represented in the longword UCB$L_AFFINITY in the UCB structure. The device affinity mask will be checked each time the system makes an internal call to the driver. Therefore the list of entry points at which the device affinity mask will be checked includes (but is not limited to):

- STARTIO
- ALTSTART
- TIMEOUT
- CANCEL
- Controller initialization
- Unit initialization
- Mount verification
- Diagnostic buffer filling
- FORK-N-WAIT

If the device affinity mask does not allow the driver's entry point to be called from the currently executing CPU, then a fork thread must be started on a CPU that can initiate the I/O request. This involves folding up the the current context (such as IRP address and UCB address) into a fork packet, queueing the fork packet to a CPU that has affinity to the device, and then generating an interprocessor interrupt to get the fork thread going. This is all very complicated, but must be implemented at least for the CONSOLE SUBSYSTEM devices on certain processors.

The CPU-specific work exchange queue will be the CPU$Q_WORK_IFQ interlocked queue in the per-CPU database. When an entry is put on the CPU work queue of another processor, the CPU$V_WORK_FQP bit in the CPU$L_WORK_REQ field must also be set. Next an interprocessor interrupt must be generated to the solicited CPU. When the solicited CPU receives the interrupt, the work bit CPU$V_WORK_FQP indicates that a work packet is

resent on the work exchange queue.

To enhance the capability of device affinity, all fork queues and the I/O posting queue will be CPU-specific queues. This has two benefits: first, it reduces contention on the fork queues; and second, it allows fork threads to finish execution on the CPU that serviced the device interrupt. This saves having to check each fork thread, as they are started, to see if it can be run on the currently executing CPU.

For the STARTIO and ALTSTART entry points, the switching of context from one CPU to another based on the device affinity mask can be handled by saving the processing context in the CDRP portion (offset IRP$C_CDRP) of the IRP. The CDRP portion of the IRP is then queued to a CPU that has the correct affinity. The decision to run on the current processor must be done under the protection of the device's forklock. This will protect the device affinity mask from changing as the decision to initiate the I/O is proceeding.

This also requires that if a driver allows the device affinity mask to be changed, then the driver must change the affinity mask itself. The driver must first be sure that all work exchange packets have been processed and that the driver is running under the protection of the forklock.

The controller and unit initialization entry points can easily conform to the requirements of device affinity. This is easy to accomplish because these routines are normally only called during booting (from INIT), during device configuring (from SYSGEN), and during POWERFAIL recovery (from POWERFAIL??). Each of these routines can ensure that device affinity is honored before calling either of the driver's initialization routines.

Each of the remaining entry points will be investigated to try and find a consistent method for ensuring that device affinity is obeyed.

3.2.25.3 Device Affinity Scenarios

- Consoles

  In this case, the physical connectivity indicator would be set to indicate that the primary CPU only has full access to the console devices. The device affinity and device interrupt masks will have only the primary CPU's bit set, meaning that I/O can be initiated and serviced on the primary CPU only.

- Real Time Device Segregation

  A system manager may wish to segregate a real time device to a specific CPU. By doing so, a driver can eliminate the code to acquire devicelocks from the interrupt service routine as well as from the fork level. This streamlines the interrupt level processing in order to realize some benefit in real time response.

  These devices may be connected to the primary CPU only, or to all CPUs. The first level of our device affinity control is rendered irrelevant by the next statement. The device affinity mask would specify a single CPU (not necessarily the primary CPU) and the device interrupt mask would also specify that same CPU. This restricts all device level interaction to a single CPU, that is, no other CPU will ever access the device and so there is no need for the specified CPU to incur the cost of system-wide synchronization for this device.

  Note that the console case described above is a specific instance of device/CPU segregation, where the segregation is restricted to the primary CPU only by the physical connectivity bit. Although streamlining of the device level synchronization mechanisms was not a goal as it was for realtime devices, such streamlining is nonetheless available to the console drivers as a side-effect of being segregated to a single CPU.

- VAX-11/782 Support

  The techniques used to accomplish console and realtime device segregation as described above can be used to segregate any device to a specific CPU. This technique can therefore be used to segregate all devices on the VAX-11/782 to the primary CPU by setting the physical connectivity bit for each device to indicate primary only access. This paves the way for supporting SMP operations on this processor.

- Load Leveling

  The device affinity design allows us to implement device load leveling. By mixing and matching various combinations of CPUs in the device affinity and device interrupt masks, we can attempt to balance the software and hardware-interrupt processing load for a device.

  Note that such load leveling, by definition, cannot occur for devices which have primary only connectivity.

3.2.26 Poor Man's Lockdown

The "poor man's lockdown" as described earlier in this document would not work correctly in an SMP system, unless some extra measures are taken to ensure its operation for pages within the system working set.

NOTE

The "poor man's lockdown" of pages in process space still works the same as it did in previous releases of VMS. However, these code segments must make sure that a proper synchronization lock, such as LOCK, FORKLOCK or DEVICELOCK is also used. This design is only concerned with pages that are in the system working set.

3.2.26.1 Alternatives

The problem is that if a page in the system working set is locked down on one CPU, there is nothing to prevent another CPU from removing that page from the system working set and turning off the valid bit in the PTE. This problem could be solved in several ways: turning off the SYSPAGING SYSGEN parameter, moving all such code segments into the non-paged portion of the executive, each CPU could record the list of pages locked down and IP interrupt processing code would verify which pages can removed from the system working set, or each CPU can lock its own pages into the system working set.

The last choice was the method that was decided upon. However, it would be helpful to go over the reasons why the other three methods were rejected. First, turning off the SYSPAGING SYSGEN parameter, would have been overkill in that not only would we have locked the desired code segments into the working set, but all code and data segments would also have been locked down. This method would have dramatically increased the size of physical memory needed by a VMS system. Therefore, this method would have placed a tremendous strain on small memory configurations, and possibly have eliminated certain systems from running VMS.

The second method, that of moving just the locked down code segments to a non-paged psect is a reasonable choice. However, the poor man's lockdown is also used to lock pages in the process working set while executing in system space. Also, data pages are locked as well as code pages. Therefore moving the code segments only helps solve part of the problem. It is also worth mentioning that there were approximately 30 cases of "poor man's lockdown" observed within executive of the V4.2 release of VMS. By moving these code segments into a non-paged psect, it was

estimated that there would be an extra 20 pages needed to run the VMS executive. This is in addition to the pages needed by an SMP enhanced VMS executive.

The third method, that of using IP interrupts, looked promising. Initially the SMP team believed that such a scheme would leverage off the existing code for TB invalidation of system space. The reason is that each time the system working set was changed a system PTE was also changed and the corresponding TB entry for that PTE in each CPU would have to be invalidated. Since this work is required anyway, the poor man's lockdown code could be layered on top of this code.

The problem with this scheme is that modification of the PTE happens before the TB invalidate request is made. Therefore, if this design was to work, then a second IP interrupt would have to be made when the decision about which working set entry and PTE was a valid candidate for removal from the system working set. Such a scheme was considered far too costly to implement two sets of IP interrupts - one IP interrupt for the working set entry selection and another IP interrupt for the actual TB invalidation.

This finally left the last method, that of each CPU temporarily locking its own pages into the system working set, as the only viable alternative. Therefore, each CPU will invoke the PMLREQ macro when it needs to perform the "poor man's lockdown" of pages in the system working set. If the code also needs to lock pages in the process working set, then it must use the current method - for example, SETIPL (R6), where R6 is the address of longword aligned longword in process space. Lastly, the code segment must add a call to one of the synchronization macros, LOCK, FORLOCK, or DEVICELOCK.

The new PMLREQ macro will call a memory management routine to lock as many pages as necessary into the system working set. The routine will be passed a virtual address and a paged count for the pages to be locked into the system working set. However, all pages requested in a single PMLREQ macro call must be virtually contiguous. Therefore, if current uses lock discontiguous memory then separate PMLREQ macro calls must be used. The PMLREQ macro call must be done at IPL 2 or lower, this is to allow pagefaulting to occur.

When the synchronized code segment is finished, it must invoke the PMLEND macro to release all previously locked pages. There must be exacly one PMLEMD macro call per PMLREQ macro call. Unpredictable, but assuredly disastrous, events will result if the PMLEND macros do not match the PMLREQ macros. The code segment must also invoke one of the UNLOCK, FORKUNLOCK or DEVICEUNLOCK macros to release any locks and finally it must lower IPL (either explicitly or through the use of one of the unlock macros).

3.2.26.2 PMLREQ Macro

The PMLREQ macro is used to lock down a paged code segment into memory. The macro causes the pages to be faulted into memory and locked into the system working set. The PMLREQ macro should only be used on system virtual addresses. This macro takes three parameters:

- START - the starting virtual address of the 1st page to be locked (this parameter is optional, and defaults to the current PC)
- END - the ending virtual address of the pages to be locked
- IPL - the IPL the locked code segment is to execute at

The START parameter is used to indicate the starting virtual address of the pages to be locked. If no START address is given, then the address of the next instruction is assumed to be the start of the pages to be locked.

The END parameter is the ending virtual address of the pages to be locked. This parameter is required. The IPL parameter is optional, and if supplied the IPL is set to the specified IPL. The code sequence should then synchronize by using one of the appropriate syncrhonization macros (LOCK, and so forth).

The PMLREQ macro may be used several times in succession to lockdown a series of discontiguous virtual addresses. However, each PMLREQ macro must be accompanied by a corresponding PMLEND macro. The reason is that PMLREQ macro generates a co-routine call, which must be paired with a PMLEND macro to undo the co-routine call.

3.2.26.3 PMLEND Macro

The PMLEND macro is used to terminate a lockdown code sequence. The PMLEND does a co-routine call-back to the lock down routine to unlock the pages. This macro takes one parameter:

- IPL - the IPL to continue executing at

3.2.26.4 PMLREQ Macro example

The PMLREQ and PMLEND macros can be used in the following manner:

```
                TSTB    (R0)                            ; Fault in page
30$:            PMLREQ  END=100$                        ; Lockdown pages
                LOCK    LOCKNAME=MMG,-                  ; Synch with MMG
                        LOCKIPL=#IPL$_SYNCH,-           ; Raise IPL
                        SAVIPL=-(SP)                    ; Save current IPL
                MOVL    W^MMG$GL_SYSPHD,R3              ; Get system PHD
                EXTZV   #VA$V_VPN,#VA$S_VPN,R0,R1       ; Get VPN
                MOVAL   @W^MMG$GL_SPTBASE[R1],R1        ; Get PTE address
                EXTZV   #PTE$V_PFN,#PTE$S_PFN,(R1),R1 ; Get PFN
                        PFN_REFERENCE -
                INCW    <@W^PFN$Ax_SHRCNT[R1]>          ; Increment SHRCNT
                        PFN_REFERENCE -
                MOVZWL  <@W^PFN$AW_WSLX[R1],R1>         ; Get WSLX
                BISL    #WSL$M_WSLOCK,(R3)[R1]          ; Lock page in working set
                UNLOCK  LOCKNAME=MMG,-                  ; Unlock MMG
                        NEWIPL=(SP)+                    ; Restore IPL
                PMLEND                                  ; Unlock pages
100$:
```

In the preceeding example the PMLREQ macro locks all pages between the 30$ and the 100$ labels into the system working set. The PMLEND macro does the co-routine return to unlock those pages locked by the PMLREQ macro call.

3.2.27 System Dump Analyzer Enhancements

TBD (SF)

Specific Work Items

This section lists the work that needs to be done to build an SMP product. The work items listed here are not all necessary to build the different phases of the SMP breadboard; however, they must be completed for a finished product. Thus, they are listed here, lest they be overlooked.

4.1 Contents of SMP Baselevels

Briefly, the baselevels are expected to include the following:

1. The first baselevel: it will only support a uniprocessor, but will have all of the spinlock code and spinlock acquisition checking enabled. Therefore, I/O support enhancements will be deferred, power failure, PTE and TB invalidation, and the ERROR LOGGING synchronization.

2. The second baselevel: it should include multiprocessor support for the I/O subsystem.

3. The third baselevel: it should include XDELTA support and SDA enhancements for SMP.

4. The fourth baselevel: it should include ERROR LOGGING synchronization, and PTE and TB invalidation and BUGCHECK (that is, mainly the interprocessor communications code).

5. The fifth baselevel: it should be the first time we try to run on a real SMP system! This baselevel will include code to initialize the SMP stuff (per-CPU data) and to start the other processors.

6. The sixth baselevel: it should contain the idle loop that looks for work and process affinity.

7. The seventh baselevel: it should include POWERFAIL support, and support for the single console device.

APPENDIX A

REVISION HISTORY

Revision History:

Rev.#	Description	Author	Revised Date
1.0	First Draft	Kathleen D. Morse	July 16, 1985
1.1	Added: more definitions, RAS goals, timer interrupt, IPL usage, spinlock description.	Kathleen D. Morse	July 23, 1985
1.2	Added: final design sections, enhanced macro descriptions, design consistency goal, SETIPL descriptions, timer queue manipulation, mutex section. Changed: spinlock macro names and parameters. Added WHAMI description by Rod Gamache.	Kathleen D. Morse	July 26, 1985
1.3	Added: Definition of lock to glossary, Interrupt latency to performance section and removed "real-time devices", Acquisition/release order to spinlock design description, Reference count, cumulative spinloop count, and vector of last 8 PCs to spinlock database, Dynamic spinlock description,	Kathleen D. Morse	July 31, 1985

Ranks for spinlocks,
ERRLOG spinlock,
ADAWI protection of spinlock
database,
Cell needed to hold 10ms timer IPL,
WHAMI example.
Changed:
"Lock" to spinlock in
alternative design section,
NEXTTQE spinlock name to TIME,
Description of SETIPL 10$ usage.
Removed:
Device affinity.

1.4 Changed: Kathleen D. Morse Sept 10, 1985
Description of spinlock
pointer vector.
Added:
figures for various
data structures (SPL, CPU, and so on),
work items,
definition of boot CPU, other
CPU, and crash CPU,
SGN$GL_SMP_CPUS SYSGEN parameter.

1.5 Added: Kathleen D. Morse Sept 18, 1985
detailed source changes by Rod Gamache
and Bill Laing in Appendix A,
multiprocessing console description by
Bill Laing as Appendix D.

1.6 Added: Rod Gamache Oct 15, 1985
forklocks, and new forklock and
FORKUNLOCK macros. Also added performance
goals relating to running N-highest priority
processes and problems caused by large caches.

1.7 Added: Bill Laing Oct 15, 1985
Notes on routines called at IPL 31, (for examples UNIT_INIT
and CONTRL_INIT), use of executive routines that may need
to acquire locks.

1.8 Changes and clarifications. Bill Laing Oct 17, 1985
Fixed a number of typos.
Clarified forklocks, especially definition.
Added parallel applications under competitive analysis.

1.9 Comments from 1.8 Bill Laing 22-Oct-1985
Clarified Product Overview definition of SMP.
Clarified definition of SPINLOCK to note owned by a processor.
Noted ENQ/DEQ preferred application way of synchronizing.
Noted that crash CPU may, under some circumstances, be the
CPU requesting the bugcheck.
Added that documentation should be available for user-written

system services.
Clarified scheduling of highest priority processes.
Noted in non-goals types of configuration not supported.
Rewrote description of timer support and IPL 7 SOFTINT.
Clarified example on forking.
Corrected numerous typos and added other clarifications.

1.10 Added: Bill Laing 23-Oct-1985
Added Device Locking and initial thoughts on powerfail.

1.11 Added: Rod Gamache Feb 12, 1986
Changed per-CPU database: data contained in area, where it is located, how it is found.
Added description of how SCHEDULING/NULL PROCESS works.
Clarified DEVICELOCK description.

1.12 Michael Harvey Feb 24, 1986
Clarified a number of passages. Cleaned up text and added some descriptions of real console constraints.

1.13 William A. Laing Feb 26, 1986
Added descriptions of XDELTA changes, interlocked queues, and interprocessor interrupts.

1.14 Cathy Foley Feb 27, 1986
Added substantial edits (as approved by design team).

1.15 Michael Harvey Feb 27, 1986
Added a description of the booting sequences for the boot CPU and all other CPUs. Described how the CPUs join in to the SMP environment. Rewrote the section describing the per-CPU database, interrupt stack and the rationale behind the operation of the FIND_CPU_DATA macro. Added some insights into the concept of boot stacks and what we did to extend the old notion of boot stack support into the SMP environment. Various cleanup of text. Move alternative designs into an appendix.

1.16 Rod Gamache 27-Feb-1986
Minor corrections.

1.17 Bill Laing 27-Feb-1986
Corrections, typos, and clarifications.
Added DEC STD 032 as a reference.

1.18 Michael Harvey 28-Feb-1986
Improved the description of error halt actions and powerfail recovery. Pointer to physical per-CPU database vector moved to RPB.

1.19 Cathy Foley 28-Feb-1986
Inserted edits and approved revisions to various sections.

1.20 Cathy Foley 01-Mar-1986
Completed a very preliminary index for body of document.

2.1 Michael Harvey 25-Mar-1986

Clarified the text in a great many places. Incorporated review comments. Updated data structure figures. Refined the discussion of boot stacks. Included more NAUTILUS and SCORPIO references to clarify some points. Added changes to IPL 5 and IPL 15 regarding XDELTA.

2.2 Michael Harvey 1-Apr-1986

Clarified our requirements for interprocessor interrupts. Moved "SOURCE MODULE CHANGES" appendix out of this specification and into a separate document. Deleted the empty "SPINLOCK REFERENCES THROUGHOUT VMS" appendix. Deleted redundant (and obsolete) sections of the "MULTIPROCESSOR CONSOLES" appendix that have been obviated by clearer descriptions of various topics in the main body of the functional specification. Expanded other sections of that appendix with information that properly belongs there, and added new information. Incorporated some parting comments on consoles by Bill Laing.

Michael Harvey 7-Apr-1986

Replaced most of the "MULTIPROCESSOR CONSOLES" appendix to be more comprehensive and accurate in describing what VMS requires in a multiprocessor console. Clarified more terms and definitions.

2.3 Cathy Foley 5-May-1986

Completed index of entire document, corrected miscellaneous typos.

2.4 Rod Gamache 27-May-1986

Added design for device affinity.

APPENDIX B

ALTERNATE DESIGN APPROACHES

A number of different design approaches for symmetric multiprocessing had been proposed, evaluated and eventually rejected in favor of the design approach that was eventually chosen. These differing designs are included in this appendix to provide insight into some of the design decisions that were made during the formative stages of this project.

These various designs are summarized as follows:

1. Spinlock on each IPL, allowing simultaneous execution of different IPLs on different CPUs.

2. Spinlock on each IPL, requiring a CPU to own all spinlocks below its current IPL.

    a. A CPU blocks if another CPU owns a spinlock it needs

    b. A CPU pre-empts other CPUs that own lower IPL spinlocks (causes other CPUs to block)

3. Spinlocks on different classifications of IPLs: 0-2, 3-15, 16-24 and 30-31. These classifications would be modified to system spinlocks of different granularities in future phases of the SMP design.

Alternative 1: The design team was not convinced that this design would work correctly. Until now, the VMS locking mechanism has been to raise IPL to the desired level, thus locking out all activity that can occur at lower IPLs. If separate execution of threads at different IPLs is allowed, the locking out of all activity at lower IPLs cannot be ensured and system synchronization may be broken. There is no easy method for determining what area of VMS would be broken under this design. It is believed that while the actual locking code would be simple to write, the debugging of such a system would continue indefinitely. Also, this alternative did not offer an evolutionary improvement to locking granularity. The primary

ppeal of this approach was its apparent ease of implementation.

Alternative 2: The design team believes that approaches a and b could be designed to function correctly. However, this design would not provide for simultaneous execution of device interrupts on separate CPUs, nor would it provide an evolutionary method for migrating to the use of system resource spinlocks. System resource spinlocks will be needed to enable simultaneous execution of kernel mode code (for such system events as pagefaults, process creation, image activation, and so on).

Alternative 3: In this design, the different IPL classifications are divided into four categories. They are described as follows:

1. 31-30: Spinlock that prevents all other events from occurring in the system.

2. 24-16: Spinlock on device interrupts. This is expected to prevent one device interrupt thread from interrupting another. If a spinlock were placed on each UCB, then different CPUs would be able to handle interrupts from different devices simultaneously.

3. 15-3: Spinlock on software interrupts. This spinlock folds all the current VMS kernel mode threads into one spinlock. This spinlock would be separated into multiple spinlocks in future phases of the design implementation, to lock such data structures as PCBs, PFN database, GSDs, and so on.

4. 2-0: Spinlock on kernel mode execution. Currently, this spinlock is not clearly understood. Investigation is needed into the use of IPL 2. However, it is known that IPL 2 is used in VMS to prevent a process from being deleted while the process owns a mutex, has a pointer to pool, and so on. Another mechanism will be investigated for locking a single process against deletion, other than using IPL 2. IPL 0 is used for applications (such as MONITOR) that need to be in kernel mode to scan system data structures.

APPENDIX C

MISCELANEOUS WORK ITEMS

C.1 Console Support and Work Queue

The console will be local to the Time Keeper CPU. All other CPUs wanting to send to the OPA0 device should queue a work packet to the Time Keeper CPU.

A work queue is needed for the messages output from the non-boot CPU's to be output to the console terminal (which is owned by the boot CPU).

A console routine, CONS$xxxx, is needed that will write to another CPU's console to cause it to boot, halt, and so on.

The work items above will be included in the seventh baselevel.

C.2 Spinlocks

Spinlocks do not spin on an interlocked instruction, thus avoiding memory writes. They should also keep performance/debug data.

9/5/85: This work is completed in a new module in the SYS facility SPINLOCK

C.3 AUTOGEN

AUTOGEN needs to be taught about new SYSGEN parameters, such as SMP_CPUS, and how to set them.

C.4 Device Drivers

- Local to individual CPUs? (device affinity) Fork Process affinity

- Global to all CPUs?
- Powerfail recovery
- Important/initial drivers
  - CI port
  - NI port
  - BDA/BUA

9/5/85: This work will be deferred to the second baselevel.

C.5 QIO

Use interlocked queue instructions for the I/O post-processing queue.

9/5/85: This work has been completed for everything in the SYS facility. Still remaining to be changed are: XQP (first baselevel), DUTUSUBS (first baselevel), REMACP, NETDRIVER, F11ACP, MTAACP and CONUTIL (in SYSLOA).

10/16/85: Completed work on XQP, DUTUSUBS, REMACP and NETDRIVER. Work still remains on F11ACP, MTAACP and CONUTIL.

C.6 Fork Queues

Use interlocked queue instructions.

9/5/85: All done.

C.7 Pool Allocation

Use interlocked queue instructions for look-aside lists.

9/5/85: All known references have been changed except for: SHOMEMORY (in CLIUTL), MONITOR and SDA.

10/16/85: SHOMEMORY has been modified. Work remains for MONITOR and SDA.

For the variable pool list, we have created a POOL spinlock for synchronization.

9/5/85: Created a POOL spinlock for the variable pool.

C.8 Check System Services Reentrant

Check that there are no local variables in any system services.

9/5/85: Not believed to be a major problem because OWN storage wouldn't work if the process gets re-scheduled on a uniprocessor. And if the process is running above ASTDEL it should be holding a spinlock.

C.9 AST Delivery

Need to add a byte cell to the PCB or PHD to indicate which CPU the process is executing on if its state is CUR. This will aid the interprocessor mechanism that needs to set the ASTLVL register avoiding a search of each CPU's CURPCB. Have to send an interprocessor interrupt to the CPU, where the process is current so it can set its ASTLVL register.

The PCB$W_ASTCNT field of the PCB must be incremented and decremented with an ADAWI.

9/5/85: All references to the PCB$W_ASTCNT field have been modified to use interlocked instructions, except for: F11X (first baselevel), MTAACP, and SDA (third baselevel).

10/16/85: F11XQP has been modified.

C.10 Synchronization of Mutexes

Mutexes are synchronized by the SCHED spinlock.

9/5/85: Done.

C.11 Per-CPU Database

9/5/85: This work was started with changes to the SYSCOMMON module.

10/16/85: References to global symbols in the per-CPU database must be found and accessed indirectly using the FIND_CPU_DATA macro.

C.12 Interprocessor Interrupts

This work will be included in the fourth baselevel.

- BUGCHECK/SHUTDOWN
- TB invalidate
- PTE invalidate
- Setting ASTLVL register for CUR processes
- Queuing output to the OPA0 device
- Updating the TODR (this should be set initially as CPUs are started).

C.13 Scheduler

This work will be included in the sixth baselevel.

- Process affinity
- Idle loop must look for work (such as processing the FORK QUEUES)

C.14 Timer Queues and HWCLK Interrupts

9/5/85: This work is already completed.

10/16/85: All references to EXE$GQ_SYSTEM must be found and modified. Done for TIMESCHDL and EXSUBROUT.

C.15 Memory Management

This work will be included in the fourth baselevel.

- TB synchronization
- PTE synchronization

C.16 Error Logging

This work will be included in the fourth baselevel.

- Needs synchronization

C.17 Machine Restarts

This work will be included in the seventh baselevel.

C.18 XDELTA

This work will be included in the third baselevel.

C.19 Miscellaneous

Copying/comparing quadwords is not atomic, therefore need to change everywhere that the system time is compared/copied.

This work should be done for the first baselevel. (Need to search for all references to EXE$GQ_SYSTIME.)

APPENDIX D

SYSTEM DATA STRUCTURE PROTECTION

This appendix lists all the system data structures and how they are synchronized in the SMP system. This includes data structures for memory management, scheduling, I/O subsystem, and so on.

- o POOL look-aside lists: interlocked queues
- o The variable pool list is synchronized by the POOL spinlock.
- o SCHED database general rules:
  - The AST queue in the PCB is synchronized by the SCHED spinlock.
  - Access to mutexes is synchronized by the SCHED spinlock.
  - The PCB$W_ASTCNT in the PCB must be accessed with an ADAWI.
  - JIB quotas should require a new JIB MUTEX as part of the JIB.
- o HWCLK database consists of the following data:
  - EXE$GQ_SYSTIME - The SYSTEM time in 100 ns intervals since November 17, 1858. (The base time for the Smithsonian Institution astronomical calendar).

    This cell is updated by the HWCLKINT routine in TIMESCHDL.MAR. This cell should be read using the READ_SYSTIME macro.
  - EXE$GQ_1ST_TIME - The expiration time for the first TQE in the TQE list.

This cell is updated whenever an entry is moved to the front of the timer queue list. This cell is read by the HWCLKINT routine in TIMESCHDL.MAR.

o HWCLK database routines: TIMESCHDL.MAR:

- EXE$HWCLKINT - hardware clock (IPL 22/24) interrupt handler. This routine acquires the HWCLK spinlock when needed and then releases it.

o TIMER database consists of the following routines: TIMESCHDL.MAR:

- EXE$SWTIMINT - IPL 7 interrupt handler

The TIMER spinlock cannot be acquired before call, the TIMER lock is acquired when needed and then released.

- EXE$TIMEOUT - 1 second SYSTEM timer (ENTRY AT IPL$_TIMER)

EXSUBROUT.MAR:

- EXE$INSTIMQ -

R0,R1 = Expiration time

R5 = Address of TQE

TIMER spinlock doesn't have to be acquired, it is acquired when needed, the TIMER spinlock if acquired on entry is still acquired on exit.

HWCLK spinlock is acquired when updating EXE$GQ_1ST_TIME.

- EXE$RMVTIMQ -

R2 = Access mode

R3 = Request ID (zero implies all)

R4 = Type of entry (single and repeat)

R5 = IPID of process to remove entries for

TIMER spinlock doesn't have to be acquired, it is acquired when needed, the TIMER spinlock if acquired on entry is still acquired on exit.

HWCLK spinlock is acquired when updating EXE$GQ_1ST_TIME.

Calls: EXE$DEANONPAGED

- What synchronizes IACGL_IMAGE_LIST, IACGL_WORK_LIST and IAC$GL_ICBFL? Are they in P1 space? They are accessed at ASTDEL with no spinlock.

APPENDIX E

MULTIPROCESSOR CONSOLES

E.1 Introduction

There are a great many ways that VAX designers can build consoles. There are advantages and disadvantages to each. What this appendix tries to do is to define the attributes of a console that VMS must interact with, with particular emphasis on what attributes of a console are required for an SMP environment. The information in this appendix is derived from experience in the uniprocessor space, with ASMP operation, and from ideas taken from the SMP design team, other members of the VMS Development Group, and from new hardware development engineering where possible.

E.2 Console-Related Definitions

CONSOLE PROGRAM - The macro or micro code that runs in the CPU or dedicated console CPU that implements the console commands as documented in Chapter 11 of DEC SID 032 VAX Architecture Standard (that is, the program that prompts >>>). All output generated by the CONSOLE PROGRAM goes to the CONSOLE TERMINAL. The console program is sometimes referred to more generally as the 'console subsystem'.

CONSOLE TERMINAL (PHYSICAL) - The physical output device connected to a console port in a CPU. It is accessed by host software via the CPU's Internal Processor Registers (RXCS, RXDB, TXCS, TXDB), which act as the port to that terminal device, or via a port that may be implemented using some other CPU type specific method, for example, the method used for the VAXstation I and II. Access to the console terminal device is available to any CPU in a fully symmetric hardware configuration, that is, each CPU has a port that can access the console terminal device. The console terminal device and any port to that terminal is generally specified as "OPA0:", a name that has other meanings in other contexts.

CONSOLE TERMINAL (VMS) - The user terminal device as seen by the VMS terminal driver. The console terminal in this context is represented as a terminal device in the VMS I/O database and users are able to log into that terminal just as for any other terminal device supported by VMS. The console terminal in this sense is generally specified as "OPA0:" and is considered by VMS to be bound to a specific set of Internal Processor Registers on a single CPU. Thus, although there is a port (the IPRs) to the physical console terminal itself available to every CPU in a symmetric hardware configuration, the VMS operating system binds the user terminal representation of OPA0: to a specific CPU's console port, that of the PRIMARY CPU.

CONSOLE BLOCK STORAGE DEVICE - A non volatile media, for example, floppy disk, tape, or ROM, that holds some or all of the following, the CPU microcode, the boot device microcode, boot command files, restart command files, VMB. If the media is a disk or tape its online disk structure is important.

CONSOLE BOOT COMMAND FILES - Files that contain CONSOLE PROGRAM commands that cause and control booting from different devices and hardware configurations, for example, UDA, CI, BDA, or that will interleaving memory. Other important command files are RESTAR for powerfail recovery and the default boot command file DEFBOO used during an auto-reboot.

OPERATOR TERMINAL - The VMS terminal(s) that are designated by the system manager to receive operator messages generated by the operating system. The default OPERATOR TERMINAL is the VMS CONSOLE TERMINAL. It is possible to direct all operator messages in a VAXcluster to one or many operator terminals.

PRIMARY CPU - The concept of a 'PRIMARY CPU' in a symmetric multiprocessing system may seem incongruous, but it is in the area of console support where asymmetry does, in fact, exist. The PRIMARY CPU as defined from the VMS point of view has the following attributes:

- The PRIMARY CPU is a CPU in the configuration that is guaranteed to have full access to all of the console devices that VMS will be required to support. This does not mean to imply that other CPUs don't also have such access, just that VMS will depend upon only the PRIMARY CPU to have such access. This means that the console UCBs defined for VMS are bound (using device affinity) such that all interrupt-driven I/O to the various console devices will be executed by the PRIMARY CPU only.

  Implicit in this statement is the assumption that the PRIMARY CPU is the CPU to which the console terminal device remains physically 'bound' to. For instance, full VMS access to the console terminal device on a NAUTILUS requires that the target of the last SET CPU

'target' console command is, in fact, the NAUTILUS PRIMARY CPU.

- Initially, at least, the PRIMARY CPU is the boot CPU.

- Because the TOY clock is sometimes implemented in the console (true in NAUTILUS) and is therefore another console device, then by the first statement above, the PRIMARY CPU will be the CPU that VMS depends upon to be the TIMEKEEPER CPU in an SMP environment.

  Unless there is a compelling reason to do so, the identity of the TIME KEEPER and the PRIMARY CPUs will remain the same even in SMP systems where the TOY clock is not implemented in the console, for the sake of VMS programming consistency.

- The PRIMARY CPU must be capable of using all functions provided by the console. For instance, the PRIMARY CPU must be capable of booting and communicating with other CPUs in the configuration, using the tools provided by the console subsystem. This rule implies that both VMS and the console subsystem must agree on which CPU is the PRIMARY CPU.

  For instance, it is possible on NAUTILUS to boot VMS on the NAUTILUS SECONDARY CPU. In this case, the boot CPU (now the NAUTILUS SECONDARY) is the VMS PRIMARY CPU. However, the console 'knows' that it is the other CPU that is truly primary, thus requests by VMS to boot the VMS SECONDARY CPU will fail because VMS is already running on the NAUTILUS SECONDARY. In this case, VMS and the console disagree on which CPU is 'primary', with the end result of crippling the boot procedure for the other CPU. In this situation, VMS is not running on a viable PRIMARY CPU with respect to multiprocessing.

VMS PORT-LEVEL I/O - The class of I/O for which VMS cannot rely on the OPCOM facility (which drives the OPERATOR TERMINAL) to handle. Port-level I/O includes I/O that is related to XDELTA, BUGCHECK, etc. where non-interrupt driven I/O to the console port can occur without regard to which CPU is primary. Such I/O occurs directly to the CONSOLE TERMINAL DEVICE via the console terminal port. Note that this output occurs underneath the VMS CONSOLE TERMINAL interface which is interrupt-driven and remains bound to the PRIMARY CPU's console port.

E.3 Booting Requirements for Multiprocessor Consoles

It is up to the console subsystem to take the first steps in booting VMS, whether as a uniprocessor or, ultimately, as a symmetric multiprocessing system. Stated here are what VMS expects of a console that supports SMP (and uniprocessor) execution.

E.3.1 Per-CPU Context

One aspect of booting a multiprocessing system that VMS believes needs some attention is in the area of CPU-specific context. The problems come into being as multiple processors are being booted, as explained below.

VMS is initially booted as a uniprocessor system. The console subsystem selects a CPU to be the 'boot CPU', and boots VMS on that CPU. In the process of booting VMS, enough context is provided such that VMS is able to initialize itself and reach a steady state of operation. One very important part of this context is the 'boot stack'.

Initially, the boot stack for the boot CPU is defined to be the end of the Restart Parameter Block (RPB). The console subsystem finds a page of memory that will be passed to VMS as the place where the RPB is to be built. The stack pointer register (SP) is set to point to the end of the RPB, hence the machine has a stack provided for it. This is useful, for instance, if a machine check occurs, since there is now a place for the machine check frame to be pushed.

The problem with using the end of the RPB as a boot stack becomes apparent as multiple processors are booted into the SMP environment. It is very desirable to VMS that these additional processors be brought online in an asynchronous fashion, that is, that they are all booted in parallel and that VMS does not have to watch over them until AFTER they have 'come alive' and taken it upon themselves to join in SMP operations. Until that point, VMS doesn't have to worry about whether or not the CPU successfully boots, for if the CPU fails to boot, the remaining CPUs are unaffected since the failed CPU hasn't formally joined into the SMP environment.

This doesn't mean that VMS will never notice the failure of a CPU to boot. It will eventually notice and take appropriate action. What is desirable is that VMS not have to poll the status of the CPUs being booted; that they can be booted in parallel and asynchronously to allow the boot CPU to continue with system initialization without first having to wait for each CPU to respond in some fashion.

Given the desire by VMS to be able to boot additional processors in parallel, it becomes important that each processor have its own boot stack. Otherwise, each CPU that is booted in parallel by VMS will share the RPB as a boot stack (using the current approach of console subsystems for providing a boot stack) for a short period of time. During this time, a machine check taken by any of them will have its machine check frame pushed into the same stack space that is being used by other CPUs also undergoing the boot sequence (and potentially also getting machine checks). The end result is corruption of stack data.

It is felt by VMS that every additional CPU to be booted (or restarted) by VMS in parallel must have a unique boot stack address loaded into the SP register prior to allowing each CPU to execute instructions. Note that VMS calculates a unique boot stack address for each CPU prior to booting that CPU, a boot stack which is not in the RPB.

This is not a problem in current multiprocessors that are targeted for SMP operation, for the following reasons. In SCORPIO systems, the PRIMARY CPU force feeds console commands to the other CPUs in the process of booting them. One of these console commands loads the target CPU's SP register with a unique boot stack address. On NAUTILUS systems, there is no contention for the RPB as a boot stack, since there is only one other processor to be booted, and that processor is booted long after the PRIMARY CPU has stopped using the RPB as its stack.

As described in the next section, VMS creates a unique boot stack for each additional processor that is to be booted into the SMP environment. New SMP hardware implementations could provide a means whereby VMS could transfer that unique boot stack address to the console subsystem as part of the boot mechanism for the new processor, or it can solve the boot stack problem itself in some fashion. However the implementation eventually gets done, the important thing is that the boot stack problem gets solved.

Of course, one solution to the boot stack problem is to eliminate the option of booting additional CPUs in parallel and in an asynchronous fashion. If VMS is forced to serialize the booting of the additional processors, then the end of the RPB can still serve as a boot stack. However, this also implies that the additional CPUs cannot be independently restartable after error halts and powerfailures, since if they were, they would all, once again, be using the boot stack in the RPB. See the discussion on 'restartability' later in this appendix.

E.3.2 Initial Booting Mechanisms

A VMS SMP system will initially boot as a uniprocessor, just as is done for non-SMP instances of VMS. The processor that is selected by the console subsystem to be booted is, by definition,

the boot CPU. The boot CPU will be assumed by VMS to be the PRIMARY CPU. It does not matter to VMS how a CPU gets selected by the console subsystem to be the boot CPU, as long as such selection occurs in a timely fashion and that the selected CPU is, in fact, capable of booting VMS from the desired boot device.

It is expected that other processors are in a halted state before cold booting the boot CPU. This is true regardless of the reason for the cold boot sequence, whether the boot sequence originates from an operator request at the console terminal or by a request by the operating system software itself.

The boot CPU (PRIMARY) has the responsibility of bringing VMS up to a production level of operation. In the process of doing so, other CPUs will be brought online to begin execution. There are two requirements to be met here. First, VMS running on the PRIMARY CPU (or any subsequently booted CPU) must be capable of determining the CPU configuration of the system. Specifically, VMS must be provided with a mechanism for determining the set of CPUs that are 'available', that is, those CPUs which have passed any self-test criteria and are considered by the console subsystem to be bootable. This mechanism must provide the currently available CPU configuration that exists at the time that it is used. It is important that this mechanism not provide stale data.

If the physical CPU IDs take the form of a number from 0 to 31, this information could be made available to VMS in the form of a bitmask. Perhaps the bitmask could be returned to VMS via the MFPR instruction.

In SCORPIO systems, the configuration is determined by looking at the node type of each node on the VAXBI. On NAUTILUS, the CPU configuration is returned by the console in response to the miscellaneous console command, GET_CPU_INFO.

Secondly, a mechanism must be provided to VMS to boot the other CPUs. It is desirable that this mechanism be asynchronous in nature to allow for parallel booting of the additional processors and so that VMS does not have to watch over each CPU to ensure that it does or does not succeed in booting. In fact, VMS specifically assumes that once the boot request for a processor is issued, that VMS can safely ignore that processor until after it has successfully booted and taken it upon itself to join the current set of processors that are running VMS.

The important point here is that VMS can continue with system initialization while the other CPUs are being booted asynchronously. VMS would certainly notice, eventually, that the CPUs may have failed to boot and take appropriate action, but in the meantime, it doesn't have to wait around to find that out.

Note that if there is more than one additional processor to be booted, that it becomes risky to use the RPB as a boot stack for all of them, as described above. It is felt by VMS development that a unique boot stack address should be provided for each processor before it begins executing instructions. Such a boot stack address is calculated for that processor by VMS before it is booted, so a mechanism to transfer that address into the other processor's stack pointer register when that processor is being booted is one way to solve this problem.

If no mechanism is formulated to provide a unique boot stack address for additional CPUs being booted, then VMS will be forced to serialize the boot sequence for additional processors. This also implies that independent restartability of these CPUs is impossible without incurring the same risks of having only one boot stack shared amongst them.

E.4 "Primariness" - Access to Console

This section will attempt to extend the definition of "PRIMARY CPU" given at the beginning of this appendix by explaining what it means in terms of console device access at different levels and by specifying what it means to switch 'primariness' to a different CPU.

E.4.1 User-Level Console Access

The level of access to the console devices most commonly seen in a running VMS system, whether running as a uniprocessor or as a multiprocessor, can best be described as 'user level' access. There are various attributes of this level of console device access, as follows:

- This level of access is defined by the existence of UCBs and other data structures which provide the VMS I/O subsystem with operational context for each console device accessed by users. With these data structures and associated device drivers, users are able to issue I/O to the console devices using VMS utility programs or by using their own programs. This level of access is characteristically interrupt-driven.

- Access to the console devices is not replicated for each CPU. There appears to VMS to be only one set of console devices available to users, regardless of how many CPUs there may be in the system. For example, there is only one console terminal that VMS recognizes as OPA0: that a user may login on, rather than there being one for each CPU that may be running.

- It is recognized that only one CPU may be guaranteed of having full, independent access to the console devices via its set of console IPRs. Therefore, the console devices are 'bound' by VMS to that particular CPU, thus guaranteeing access to the console devices by any user. This binding is implemented by some form of device/CPU affinity at the data structure level and ensures that all I/O and interrupts for the console device in question is always serviced by the CPU that is known to have full access to that console device. The CPU to which these data structures are thus bound is considered by VMS to be the PRIMARY CPU and is initially the boot CPU.

  On SCORPIO, for instance, where the non-PRIMARY CPUs' consoles are logical interfaces across the VAXBI and must be linked by code that is running on the PRIMARY CPU to the real console, access to the console necessarily involves another CPU, thus placing a restriction on VMS access to the console via that non-PRIMARY CPU. On NAUTILUS systems, user level access to the console terminal device depends upon a console SET CPU command to match up the console terminal to a CPU that one might be interested in logging into. Since it is expected that the NAUTILUS user will leave the console terminal 'SET' to the NAUTILUS PRIMARY CPU, it behooves VMS to similarly bind its console terminal device to the same CPU.

- In the interest of simplicity, all console devices will always be bound to the same CPU, the PRIMARY. Interrupts for all console devices are only enabled on the PRIMARY CPU.

- There is a way for a user process to disable console device interrupts and communicate directly to the console devices via the console port on any CPU, provided that the CPU does, in fact, have full access to the console devices. For the reasons stated above for normal (interrupt-driven) I/O to the console devices, and to avoid collision with such I/O that might be going on in parallel on the PRIMARY CPU, this mechanism will be forced to be synchronized with normal interrupt-driven I/O activity by being similarly bound to the PRIMARY CPU.

  Note that there are not many consumers of this functionality in VMS, nor is this method of communicating with the console devices a recommended practice. However, the hooks for this communication do exist and are being treated here in this section for completeness.

E.4.2 Port Level Console Access

There are a number of cases in VMS where direct port level access to the console is required, that is, access that does not occur under control of device drivers operating in an interrupt-driven mode and utilizing VMS data structures to provide associated context. These cases all assume that each CPU has a port to the console devices that does, in fact, provide full access.

Examples of these cases are:

- XDELTA, the VMS system debugging tool. XDELTA may be invoked randomly on any one CPU and requires direct I/O to the console terminal device. Due to its nature as a system debugger, it cannot rely upon a device driver and associated data structures to carry out such I/O. Interrupts are disabled anyway while XDELTA is in control, so direct I/O is its only choice. Note that XDELTA requires both input and output from the console terminal device, regardless of which CPU it's running on.

  This presents an interesting situation on SCORPIO systems where the assumption of full and independent access to the console terminal is not true for secondary CPUs. XDELTA invoked on the secondary CPU depends upon the primary CPU to act as the interface between the actual console terminal and the secondary CPU. This situation was dealt with by VMS development by modifying our SCORPIO system to have multiple real consoles during system debugging.

- BUGCHECK, the VMS code that handles a system crash. The BUGCHECK code requires the ability to output to the console terminal device, and to be able to read in microcode if necessary from the console block storage device. Unlike with XDELTA, the BUGCHECK code can be trained to be bound to the PRIMARY CPU, if necessary, since BUGCHECK's function is not limited to a single processor in an SMP system as is the case with XDELTA.

A related aspect of console device access does not involve VMS at all. The console microcode in a CPU is fully capable of generating output to the console terminal device, regardless of what the CPU may be executing at the time. The constraints on writing to the console terminal are no different than the constraints described here for port level console access by VMS. If the CPU provides a full and independent path of access to the console terminal, then the console program is guaranteed to be successful when writing to the console terminal. Note the SCORPIO case described above where output from the SECONDARY CPU's console microcode is lost if the primary CPU is unable to act as the interface to the console terminal.

E.4.3 Switching "Primariness"

It may be desirable to change which CPU is to be the PRIMARY CPU. This may or may not require a VMS reboot to take effect.

E.4.3.1 NEXT_PRIMARY

One level at which primariness may be switched is to specify which CPU is to be chosen as the next PRIMARY CPU, effective as of the next time that VMS is booted on the system. What this means is that once VMS is booted on the system, the CPU that is currently the PRIMARY CPU will continue as primary until the next reboot, that is, primariness is not being switched dynamically while VMS is currently running.

On SCORPIO systems, the primariness cannot be switched except by swapping CPU boards in the VAXBI card cage. On NAUTILUS systems, NEXT_PRIMARY can be specified either by a console command typed by an operator, or VMS may issue a command to the console specifying that the next primary is to be the opposite of the current primary. In either case, primariness is not switched until the next time a console INITIALIZE command is issued during the cold boot sequence.

E.4.3.2 Switching Primariness Dynamically

Although there are currently no multiprocessor VAXes sold by DIGITAL that support the notion of dynamically switching primariness without having to reboot VMS, the idea is evaluated here as a first attempt to identify what might be required in order to accomplish this feat.

The important reason why this functionality might be useful involves the occurrence of a loss of the PRIMARY CPU itself. This can happen in a variety of ways. The PRIMARY CPU might not recover from a powerfailure, while other CPUs would. Or, due to some other kind of hardware failure, the CPU simply disappears. In all of these cases, VMS would eventually reach a point where it would wait for a response from that CPU and time it out. Once the decision is made by VMS that the CPU has disappeared, then it could take appropriate actions, such as switching primariness to another CPU if that option is available on that particular system.

Another case where this is useful involves the DCL STOP/CPU command, the command a system manager would use to shut down a specified CPU. On systems where dynamic switching of primariness is supported, the command would be able to shut down any CPU, including the PRIMARY CPU. Otherwise, it would be limited to only the non-PRIMARY CPUs.

The loss of any CPU, not just the PRIMARY CPU, must be handled properly by VMS. The actions that VMS must take to dynamically switch primariness are simply the consequences of the lost CPU having been the PRIMARY CPU. What VMS must do to switch primariness is described in more detail in the next section, but is briefly outlined here for convenience.

- All I/O that is currently active on the PRIMARY CPU's console port must be timed out.
- The binding of the console devices at the data structure level must be reassigned to a different CPU, the new PRIMARY CPU.

  Note that this implies a deterministic method for choosing a valid new PRIMARY CPU.
- A console command must be available that changes the binding of the console terminal itself to the new PRIMARY CPU. That is, reasons were listed earlier in this appendix why it is important that the console subsystem and VMS both agree on which CPU is the PRIMARY CPU. Thus, there must be some method by which VMS can inform the console that a new PRIMARY CPU has been chosen and that the console terminal itself should be correspondingly aligned. In NAUTILUS lingo, this amounts to a 'SET CPU' console command, which can be specified from code running on the VAX.
- The console I/O that had been previously timed out may now be retried.

E.5 Powerfail, Error Halts, and CPU Timeouts

There are requirements on a multiprocessor console for handling restartability, regardless of the reason for a required restart, and for proper handling of a CPU that has simply been lost for some reason. This section will describe what is required of the console subsystem at each step in a restart scenario, followed by a general discussion of restartability implementations.

E.5.1 Powerfailure and Error Halting

When a processor is about to lose power, it will take a powerfail interrupt and begin executing a routine which saves state. The powerfail interrupt service routine in VMS will save state in such a way that the possibility of having another CPU pick up that state and correctly resume operations will be

maximized. This will be done under the assumption that it is possible that a given CPU may not survive the loss of power, and that it would be up to another CPU in the multiprocessor system to 'cover for' the lost CPU if at all possible. Note that if the CPU that doesn't recover from a powerfailure happened to own some system locks at the time of the power loss, it is unlikely that the system will avoid a crash after the power is restored.

When a CPU incurs an error halt of some kind, the processor halts, thus giving the console subsystem control at this point.

E.5.2 Restarts

It is up to the console subsystem to restart a processor in some fashion following the restoration of power to the processor, or in order to recover from the error halt condition that caused the processor to be halted by force. Note that the implementation of restartability is treated later in this appendix.

Suffice it to say that the processor, once restarted, doesn't care how it was restarted. The restart code that the processor is executing simply has to provided with the halt PC, halt PSL and halt code in R10, R11 and AP, respectively. If the reason for the restart is anything other than for a powerfail recovery, then the system crashes immediately. Otherwise, the processors proceed with restoring state and resuming VMS operations.

Note that it doesn't matter to VMS which CPU is the first CPU to be restarted following a powerfailure. On some systems, only one CPU will be restarted, the PRIMARY CPU. The VMS powerfail recovery code will be able to restore as much state as it can before it reaches the point of evaluating whether recovery is successful, which requires detection of any CPUs that may not have survived the powerfailure and what effect that their loss may have on the remaining CPUs that did survive.

One important requirement on the console subsystem is that only CPUs that had been running prior to the powerfailure be restarted. That is, if a CPU had failed self-test before the system was booted, but miraculously passes self-test after a powerfail recovery, that that CPU not be restarted since it was not previously a participating CPU in the SMP environment.

One idea for implementing this is to define an enable/disable bitmask in the RPB. Software could set/clear bits in that mask to designate which CPUs are allowed to be booted or restarted. The console subsystem could examine this bitmask to determine if any given CPU has been disabled from execution by software.

The steps that VMS will take as the last part of powerfail recovery are:

- Wait for all CPUs that had been running VMS to recover from the powerfailure. This means that VMS must have some idea of which CPUs were participating in the SMP environment prior to the powerfailure. It also means that VMS once again will be requesting CPU configuration information from the console subsystem to discover if any of these CPUs has been lost.

- If any CPU has been lost from the SMP environment, time it out and take appropriate action.

E.5.3 When a CPU Times Out

There are a number of situations where a CPU may be timed out of the SMP environment. When VMS has decided that a CPU must be timed out, that is, it has been lost as an SMP participant, there are a number of actions that are required in an effort to minimize any damage.

First, the timeout code paths must ensure that the timed out processor remains inactive. There must be some mechanism provided by the console subsystem to render that processor inactive such that it can't come back later and cause damage due to VMS' belief that the CPU has gone away but really hasn't. On SCORPIO systems, the PRIMARY CPU is able to halt the timed out CPU.

Second, VMS must evaluate the state of the processor that was lost. If that processor possesses any system locks, then there is little choice but to cause a system crash. If no locks are outstanding, then it may be possible to continue system operation after first logging the fact that a process may have been lost on the timed out processor. Note that if the processor had timed out following a powerfailure, then enough process state should have been saved to prevent even the loss of a process.

Third, if the CPU that was timed out is the PRIMARY CPU, and the system supports dynamic switching of primariness, do the following:

- Time out I/O requests that are active on the console devices, if any.

- Select a new PRIMARY CPU (by some currently unspecified mechanism).

- Alter the binding of the console device data structures such that future I/O requests to the console devices will be issued via the new PRIMARY CPU.

- Retry any I/O requests to the console devices that were previously timed out.

- If there are any processes that have their CPU affinity set to the PRIMARY CPU, their affinity must follow the change in primariness.

- A special case that needs to be considered involves those processes that are attempting to access the console terminal directly, that is, without benefit of device drivers and associated data structures. They do this by raising their IPL to block console device interrupts, disabling console interrupts, and then doing their own I/O directly to/from the console IPRs.

  It is assumed that such processes will always have their affinity set to the PRIMARY CPU. This is because console interrupts are enabled only on the PRIMARY CPU. Because these processes will be running at a high IPL and, therefore, are not reschedulable, the loss of the old PRIMARY CPU will result in losing the process as well.

E.6 Restartability Considerations

It was assumed by the discussion above that the CPUs in an SMP environment are all independently restartable. This means that on a powerfailure or on an error halt restart, that the console subsystem is able to restart any one CPU without having to depend upon VMS running on another CPU for assistance. Of course, for this to work without risk of boot stack corruption, unique boot stacks must be specified for each CPU being restarted in parallel.

There is the possibility that only one CPU within a multiprocessor configuration may be restartable, while the remaining CPUs will simply remain halted after incurring one of the various types of halt conditions. This, of course, is not very symmetric, but the existing multiprocessor VAXes (SCORPIO, NAUTILUS) are known to behave this way to varying extents. For instance, only the PRIMARY CPU in a SCORPIO system is restartable for any reason. On NAUTILUS systems, both CPUs are independently restartable for error halt conditions, but only the PRIMARY CPU is restarted following a powerfailure.

For these systems, there are additional responsibilities placed on the one restartable CPU and VMS, as follows:

- The restartable CPU must be able to detect, in some fashion, the need to restart another CPU that is itself not restartable. Furthermore, it must be able to detect the reason why that CPU had incurred the unrestartable halt condition.

- The mechanism for detecting a halted CPU and the reason for the halt must be made available to VMS running on the restartable processor.

- A mechanism must be provided for VMS to take action and do whatever is required to restart the halted CPU. This includes providing the target CPU with halt code information and with a unique boot stack (as is currently done for SCORPIO systems).

  When a halted CPU resumes program mode operation, it should have not have to care whether the restart action was carried out by VMS running on another (restartable) CPU, or by the halted CPU's console subsystem as would be the case if that CPU was independently restartable.

These are fundamental requirements for proper recovery of unexpected halt situations in an SMP system. If all CPUs in an SMP system are independently restartable for any halt condition, then there is no need to rely upon VMS to execute restarts of other CPUs, by definition. For any halt condition that is not restartable by the console subsystem for a subset of CPUs, the requirements above enable VMS on the restartable CPU to lend the console a hand to extend the property of restartability to the halted CPU(s). Either approach produces a wholly restartable SMP system.

E.7 Console Logging

1. To log or not to log?

2. Log all CPUs equally well?

3. Replace OPA4: with log/nolog mechanism for OPA0:

4. SMP preference? How does this relate to primariness switching?

5. Display/control options or ideas?

6. On combined log device, port level routines must synchronize between CPUs to minimize the chances of interweaving bursts. When a port grabs the lock, output port ID string before putchars are done.

E.8 Remote Port

1. Do we need a separate console port? (OPA5, CSSE)

2. Nautilus and Venus already have it

3. If OPA5 is kept, do we need the SET TERM OPA5 command?

4. Port-level traffic problem

E.9 Miscellaneous Considerations

This section contains some miscellaneous console considerations.

E.9.1 Restart Flags

Do we need (or want) multiple sets of restart control flags, one for each CPU?

E.9.2 Physical CPU Identification

1. Physical ID specified in NEXT_PRIMARY

2. Physical ID specified in SET CPU command

3. Physical ID specified in DISABLE CPU command

E.10 Console Terminal Output and Input

The following lists are all the outputs that are written to the console terminal and all the inputs that are entered on the

onsole terminal.

- o CONSOLE PROGRAM output.
  - HALT executed
    PC = 800050D3.
  - CPU double error
  - Etc.
- o CONSOLE PROGRAM input.
  - CTRL/P
  - START <address>
  - EXAMINE <address>
  - BOOT <device>
  - And so forth...
- o SERIOUS VMS ERRORS.

  These messages are normally output directly to the console terminal because no other I/O may be possible.
  - BUGCHECK output
  - Mount Verification
  - Fatal Adapter Errors (for example, CI port re-initializations)
  - Nonpaged Pool Expansion Failures
  - PAGEFILE/SWAPFILE Full
  - CONNECTION MANAGER Messages in a VAXcluster
  - Possibly others.
- o INTERACTIVE BOOT COMMANDS
  - Changing SYSGEN parameters during SYSBOOT
- o DEBUGGING THE KERNEL

- XDELTA commands and responses

E.11 Operator Terminal Output and Input

The following lists are all the outputs that are written to the operator terminal and all the inputs that are entered on the operator terminal. Remember that this may not be the same terminal as the console terminal. The system manager can designate any user terminal as an operator terminal with the DCL command, REPLY.

- Security audit/breakin messages.
- DECnet messages
- Tape and disk mount requests
- Shutdown messages
- SYSGEN/AUTHORIZATION file changes
- And so forth...

5.12 Some Possible Multiprocessor Console Configurations

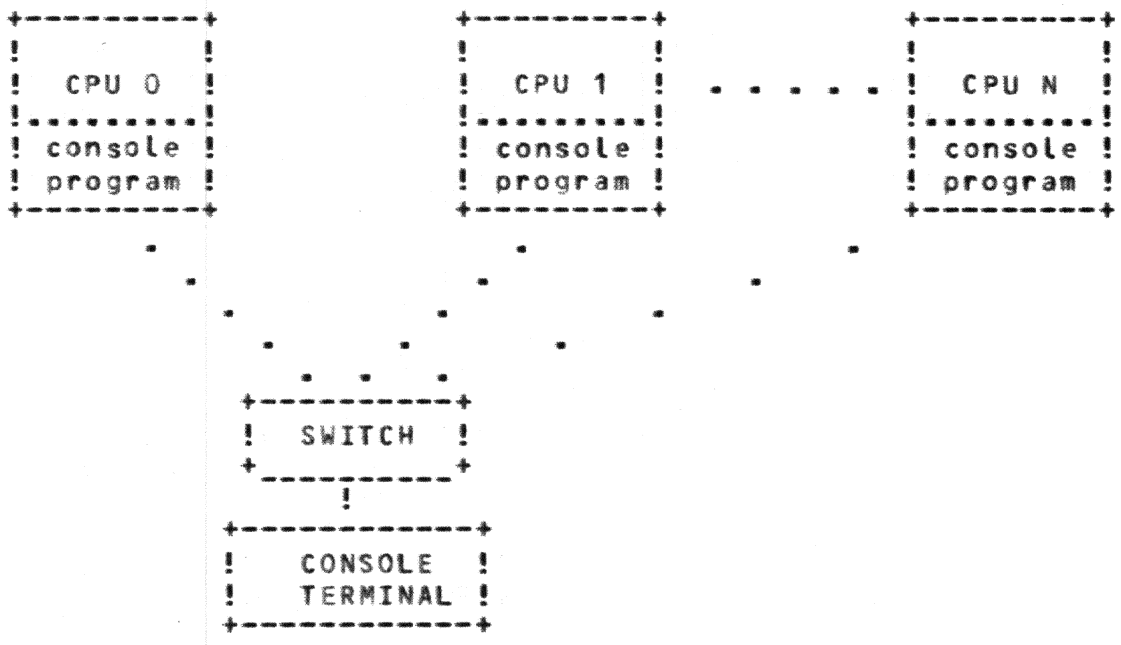


FIGURE 6: The Low End

In this configuration, each CPU is identical and has its own console program, either in microcode or macrocode. Each CPU has a connection to the console terminal. Some form of switch should be available to select which CPU should receive commands typed at the console terminal.

A mechanism should also be available for any CPU to send console commands to any other CPU. The logical console implementation using the VAXBI RXCD register is an example of this.

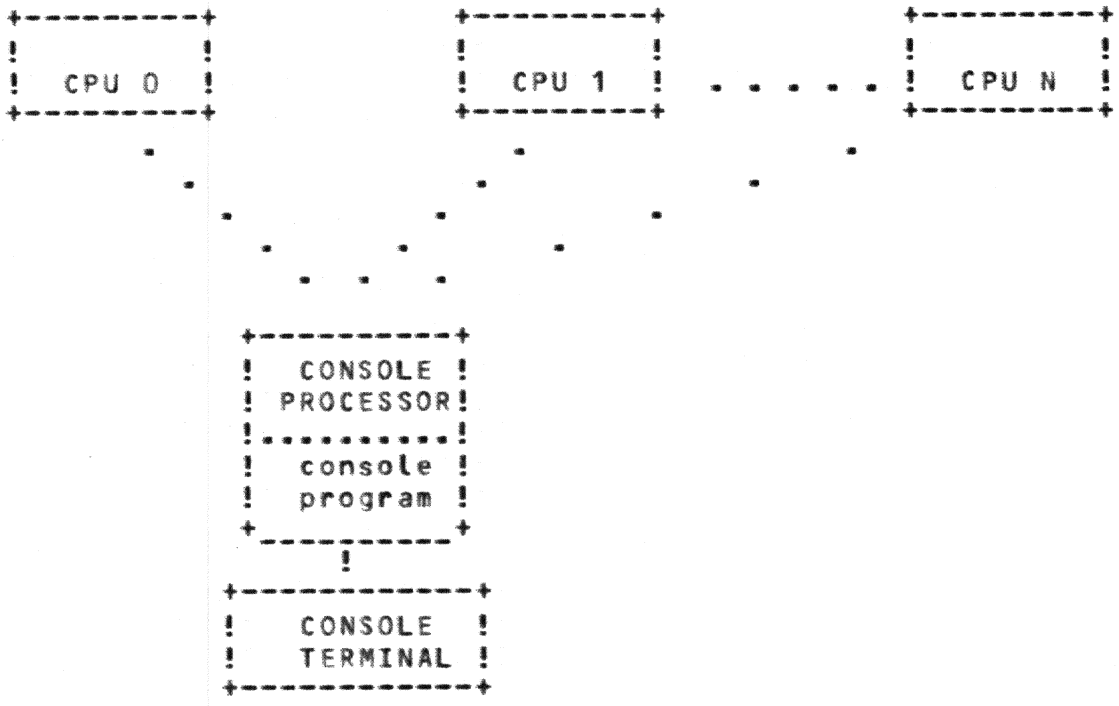


FIGURE 7: The Mid range

In this configuration, each CPU is identical and there is a separate console processor that implements the console program. A command to the console processor should be available to select which CPU to send console commands to.

A mechanism should also be available for any CPU to request the console processor to direct console commands to another CPU.

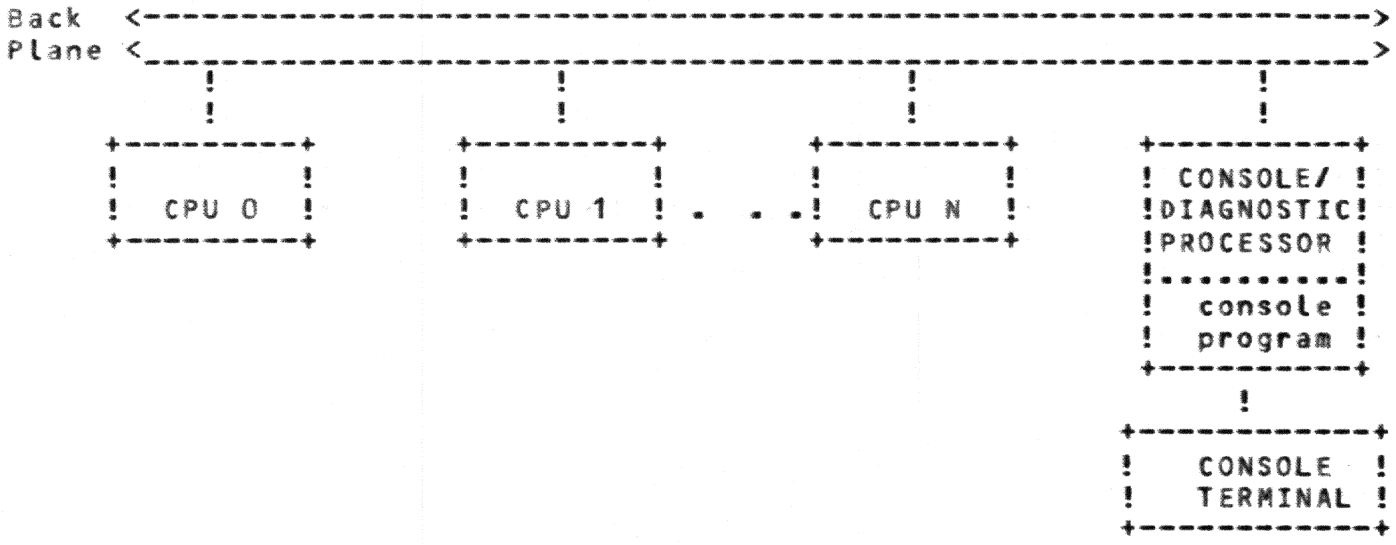


FIGURE 8: The High End.

In this configuration, the CPUs are identical and the console processor is connected directly to the backplane bus.