Box 34
folder 10

Windows NT, 1993

Digital Equipment Corporation records
Engineers' papers: Mike Uhler papers: Alpha and PRISM devel
X2675.2004, *Box 34* 102749943

5/12/93 DECWEST

- BYTE, WORD support is a distraction. Attempt to quantify the impact - perf and development cost - to determine whether an architecture change and IN change is required.

- Superpage support for NT for Bost I & D.

- Need TB entries qualified VIVA ASN because of system calls begin context switches. 6-7 bits seems about right.

- Allow locking first based on ASN, VA, or both.

- Make up synch fast.

- NT has hard-coded 8K page size and will likely be that way for the foreseeable future

(Message ev6:41)
Return-Path: gmu
Received: by snaab.cdad.hlo.dec.com (5.57/ULTRIX-fma-041391);
        id AA19715; Wed, 5 May 93 12:22:41 -0400
To: AD::FIELDING
Cc: AD::MEYER, ad::lowney, gmu
Reply-To: uhler@ad.enet.dec.com
Subject: Agenda for the DECwest visit
Date: Wed, 05 May 93 12:22:41 -0400
From: "Mike Uhler, DTN 225-4735, HLO2-3/D11" <gmu>
X-Mts: smtp

Here are the list of topics that we'd like to discuss with
the DECwest folks, as reviewed and updated by Dirk and Geoff:

Here are the list of topics that I'd like to discuss with
the DECwest folks. Any modifications?

- Speculative execution and the potential impacts on NT. Geoff will sketch the alternatives to provoke discussion.

- NT management of the virtual and physical address space including such things as page coloring.

- NT support for the performance monitoring utilities (e.g., PC sampling, IPROBE, PERFMON/PMT, etc.) that now run on VMS and OSF/1.

- Methods of thread and MP synchronization.

- The NT calling standard, exception handling standard, and the dynamics of library and system calls to determine what the "weight" of each type of call is.

- NT impact of putting restrictions on virtual aliases (to evaluate the possibility of a large virtual Dcache).

- Other more generic topics like access to sources (with restrictions), for analysis purposes, etc.

/gmu

NAME	GROUP / INTEREST	NODE	DTN
DIRK MEYER	EV6 / ARCHITECTURE	AD::	225-6325
KEVIN FIELDING	EV6 / PROD MGT	AD::	225-6808
Tom Van Baak	NTSG /	MOLD:: TVB	548-8892
Steve Jenness	NTSG	NTHEAD:: JENNESS	548-8838
Dave orbits		DECWET:: ORBITS	
JEFF MCLEMAN	NTSG	NTHEAD:: MCLEMAN	548-887?
Rod Gamache	NTSG	DECWET:: GAMACHE	548-8855
Eric Rehm	NTSG / user mode Graphics, Client-Server OS Issues	rust::rehm	548-8904
Joe Notarangelo	NTSG	NTHEAD:: JOE	548-8908
John DeRosa	NTSG	NTHEAD:: DEROSA	548-8893
MIKE UHLER	SEG	AD::	225-4735
BENN SCHREIBER	NTSG	DECWET::	548-8703
[illegible]	[illegible]	VSSE::	

HARDWARE REQUIREMENTS FOR EISA AND PCI BASED ALPHA AXP SYSTEMS RUNNING THE WINDOWS NT OPERATING SYSTEM

David G. Conroy
dgc@dgcvax.cdad.hlo.dec.com
Last edit: 04-Feb-93

1.0 Introduction

This document is a specification for Alpha AXP systems, intended to run the Microsoft Windows NT operating system and its applications, which use the EISA and/or PCI busses as their principal I/O busses. There are a number of reasons why it is desirable to have such a specification.

1) Windows NT applications must be able to make functional assumptions about the systems upon which they will run when they design their user interfaces. This results in requirements surrounding display resolution, number of display colors, keyboard key labels, number of mouse buttons, and so on. Since these requirements must be the same on all systems upon which the Windows NT system runs (that is, the same set of requirements must apply to Intel, MIPS, and Alpha AXP systems) most are determined by the existing Intel base.

2) The Windows NT operating system itself makes unwritten, but nevertheless fundamental, assumptions about the hardware upon which it runs. Violating these assumptions can make porting the operating system impossible. This results in requirements surrounding low-level aspects of the system architecture. Most of these requirements are architecture independent; a small set of them are Alpha AXP specific.

3) The Windows NT system is an open system, and it is expected that the same industry which builds expansion hardware for Intel based systems will build expansion hardware for Alpha AXP based systems. This results in requirements to ensure that certain key pieces of system software (for example, device drivers) can be shrink-wrapped, and may, eventually, result in recomendations surrounding external connectors.

This document does not define a system which can be cloned mindlessly, and which will run some standard version of the Windows NT software. Rather, it defines the edge of the envelope within which a certain class of Alpha AXP systems should be designed, and offers some recommendations, which, if followed, may make building a version of the Windows NT software easier.

Cloning of hardware is vital in the world of Intel based PCs. MS-DOS provides such a threadbare set of services to its applications that many are forced to manipulate the hardware directly, and the only way to achieve application compatibility is to have

absolute hardware compatibility. Although the Windows NT operating system shields its applications (and, in fact, almost all of itself as well) from most of the low level details of the system upon which it runs, it is hard for the designer of an Intel-based system to exploit this fact as long as it is possible that the system might run MS-DOS.

Alpha AXP systems have no such history. Designers of Alpha AXP systems are free to exploit the fact that the Windows NT operating system has a very abstract view of hardware, and can build their systems in whatever manner they desire, as long as they are willing to accept the additional cost of building system specific software. Designers of systems which are far from the traditional PC design center (very high performance systems, multiprocessor systems, and so on) may be willing to accept this additional cost. It is, therefore, important to understand exactly where the edge of the envelope lies.

This is not to say that certain standardized Alpha AXP system configurations will not become popular. Building new versions of software, even if it is easy, takes time and costs money, and so there will be pressure (especially within large organizations, like Digital itself, where even doing nothing takes a long time and costs a lot of money) to make new systems copy most of the attributes of their predecessors. The availability of standardized integrated circuits with interesting characteristics (like the LCA microprocessor, and the HARLEY and APECS chipsets) will encourage this even more. However, it seems naive to believe that a single system configuration can suffice, or that Digital can determine all of the characteristics of such configuration(s).

A particular system can implement functions beyond those described in this document (that is, the absence of a function in this specification does not imply that a function is forbidden), and it can implement a particular function described in this document in a way which is not compatible with the recommended implementation. In both these cases the system developer must provide the additional software required to support the function.

This document does not discuss Alpha AXP systems which do not use the EISA and/or PCI busses as their principal I/O busses, nor does it discuss systems which access these busses indirectly. Much of the information in this document is relevant to the design of Alpha AXP systems which are bounded and/or are built around other system busses; determining what sections are relevant is (unfortunately) left as an exercise to the designers of such systems.

This document only describes details which are visible to high-level software (in this specification, the term high-level software means the operating system and above, including device drivers and system initialization; the term low-level software means PAL, HAL, and any console functions). It does not describe system specific registers which would only be manipulated by low-level code, nor does it describe internal busses within the processor or I/O systems which are not directly visible to software (note that the PCI bus will fit into this class in many systems).

2.0 Processor Section

The processor section is the collection of processors, caches, memories, and internal busses which can execute programs defined by the Alpha AXP architecture. Most of the requirements imposed upon the processor section are imposed by the Alpha AXP architecture, and are described in detail in the Alpha AXP System Reference Manual (SRM).

The SRM (section I, 4.1.1) allows floating point support to be subsetted. The Windows NT operating system requires full support for IEEE floating point (S and T format), and does not require any support for DEC floating point (F, D, and G format). Any gaps in the processor section's support of floating point must be filled in by PAL code; there is no floating point emulation in HAL or in any high-level software.

Like VMS and OSF/1, the Windows NT operating system has, in addition to the standard PAL functions required by the Alpha AXP architecture, its own operating system specific PAL functions. These functions are not described in this specification; eventually a description of the PAL functions used by the Windows NT operating system will be ECOed into the SRM.

2.1 Regions

The SRM (section I, 5.2) specifies that the physical address space is divided into four regions based on the two most significant implemented physical address bits. The actual address bits which function as region selection bits are known only by low-level software.

2.1.1 Memory Region

The system must implement at least 16M of memory, and this memory must appear to high-level software as a contiguous memory-like region (as defined in section I, 5.2.4 of the SRM) starting at offset 0 in region 00. Small areas of this contiguous block may be marked as unavailable to high-level software because low-level software has claimed them for it own use or has determined that they are defective.

It is TBD how this information is passed to high-level software, but something very similar or identical to that described in the bootstrapping sections of the SRM is likely.

Memory systems built from interchangeable array modules of differing sizes (for example, SIMMs) must include hardware which can select the array modules in a way which allows a contiguous view. Memory systems are not required to do this automatically (that is, completely in hardware) as long as the necessary low-level software runs before any high-level software.

The memory region is allowed to be, but not required to be, a cached region. The cache policy (write-through, write-back, and so on) is unspecified, but all caches must comply with the coherence rules detailed in the SRM (section I, 5.4) assuming that the EISA

DMA transfer controller and EISA bus direct masters are processors; software enforced cache coherence is forbidden.

Clearly it is possible to run Windows NT on a machine with software cache coherence, because if it wasn't possible, the JAZZ machine would not work. If we don't forbid software cache coherence we have to define a set of Alpha AXP specific software cache coherence HAL procedures, arrange that these procedures are called from all the appropriate places in the high-level software, and have stubs for these procedures on the machines which have hardware cache coherence, which is probably most of them. Given that building coherent caches isn't that hard, it seems easier to just kill software cache coherence here and now.

It is unspecified what happens if high-level software references a physical address in the memory region beyond the actual size of physical memory reported by low-level software. This eliminates any need for the memory system to detect such references, lets the memory system introduce physical addressing aliases to simplify the implementation of the contiguous view, and lets the system eliminate tag storage for physical address bits beyond those needed to address its largest possible physical memory.

It is unspecified if the memory system provides any error checking and/or error correction hardware. The format of the information logged on memory errors (and, in fact, on any machine check) is system specific, since all machine checks are processed in PAL or HAL. The logging interrupt generated when a correctable error is actually corrected is delivered at IRQL 7; the system must contain whatever hardware (if any) is necessary to make the delivery of the interrupt subject to the current IRQL.

Jeff and Joe: The Windows NT PAL code implements a non mapped kernel space between virtual addresses 80000000 and BFFFFFFF. Because the space is non mapped (that is, a virtual address in this space can be transformed into a physical address in the memory region using simple arithmetic) the actual location of the kernel's pages in the memory region is somewhat visible. Does this have any implications on console and/or diagnostic software; for example, would it be better for console and/or diagnostic software to steal any pages it needs from the high end of the memory region so that the system would always be loaded at location 0?

2.1.2 Non-Memory Regions

The system has almost unlimited flexibility in regions 01, 10, and 11. These regions will contain control and status registers for the memory system (manipulated by low-level software), bootstrap and diagnostic ROMs, control and status registers for system specific I/O devices (manipulated by system specific device drivers) and so on.

The memory and I/O spaces of the EISA and/or PCI bus(es) must be assigned blocks of addresses in one of the non-memory regions. The base address of these blocks can be system specific, since the base addresses are supplied by HAL, but the blocks must have the same behavior as viewed by high-level software (that is, it must be possible to use the

READ_PORT and WRITE_PORT macros in both memory and I/O space; the recommended way to accomplish this is to make their semantics exactly the same).

2.2 Locks in the Memory Region

The system must implement the standard semantics of load-locked (section I, 4.2.4) and store-conditional (section I, 4.2.5) in the memory region. Uniprocessor systems will, in most cases, implement an extremely degenerate implementation of the lock hardware, consisting only of the lock flag (that is, the lock flag is set on load-locked, cleared and tested on store-conditional, and cleared on DMA writes to any address in the memory region). Multiprocessor systems will, in most cases, implement the lock address register and the lock address comparator as well.

The Windows NT operating system uses its standard interlocks on uniprocessor systems. System performance will suffer if uniprocessor systems use low-performance implementations of load-locked and store-conditional under the (incorrect) assumption that they will only be extensively used on multiprocessors.

If you arrange that the Alpha AXP lock flag cannot be set when there is an outstanding EISA or PCI lock (that is, wire the EISA or PCI lock signal to the asynchronous reset of the lock flag) then you can write Alpha AXP sequences with the same semantics an Intel LOCK XCHG as long as the program does not depend on byte or word atomicity.

Recently I have been working on a feasability study for building an Alpha AXP processor chip which is functionally compatible (and perhaps even pin compatible) with the Intel Pentium processor. Since the Pentium specification promises much about what can happen when the Pentium LOCK# signal is asserted, it seems that the only way it can be implemented is in PAL. This means that a Pentium compatible Alpha AXP processor would implement both Alpha AXP locks (with a lock flag and a lock address register) and Intel locks (with some special hardware to assert the LOCK# signal at the right time, and the appropriate CALL_PAL instructions to request services). This seems to be an acceptible solution (and perhaps the only one, if it is necessary to implement locks with byte granularity) since the only clients of such locks would be the miniport drivers for some peripherals.

Joe: The SRM says that the lock flag is cleared by a CALL_PAL REI. There is no mention of any change to the state of the lock flag in the description of CALL_PAL RTI or CALL_PAL RETSYS, at least in the ancient version of the PAL specification which I have. Is this just an oversight?

2.3 Multiprocessor Considerations

The Windows NT operating system can support multiprocessor Alpha AXP systems, as long as the system meets certain requirements. Most of the requirements (read and write ordering, synchronization facilities, and so on) are part of the general Alpha AXP

architecture, and are covered by the SRM; however, the Windows NT operating system imposes additional requirements.

1) All processors in a multiprocessor system must be identical. It must be possible to suspend a thread on one processor and resume it on any other processor. In addition, the same suspend/resume code is used on all processors, so no reformatting of machine state is possible.

2) All processors in a multiprocessor system must have identical views of the memory system and the I/O system. Asymmetric arrangements in which the I/O system can only be seen by distinguished processors (and in which I/O can only initiated on those processors) are unacceptable. In addition, asymmetric arrangements in which the physical address of registers in the I/O system differ from processor to processor are unacceptable as well, since the same set of page tables are used on all processors.

3) Each processor in a multiprocessor system must have a block of per-processor memory associated with it, which is accessed in exactly the same way (that is, by exactly the same code) on each processor. This can be implemented by having a per-processor base register which is accessed the same way on each processor (for example, by a PAL call which returns the address of the per-processor data block for the current CPU), or by having distinct physical memory mapped to the same address in the virtual address space of each processor (by means of a PAL code hack and/or dedicated DTB entries, since the same set of page tables is used by all processors).

4) At system startup time a distinguished processor (the boot processor) must begin running, and all other processors must be stopped. The other processors must be able to be initialized and started by software running on the boot processor (that is, it must be possible to implement the HalStartNextProcessor procedure).

5) Each processor in a multiprocessor system must be able to send interprocessor interrupts to all other processors (including itself) in order to implement the HalRequestIpi procedure. Interprocessor interrupts must be delivered to a processor at an IRQL above all device interrupts and above the interval timer interrupt, and be masked by IRQL in the usual way.

6) It must be possible to deliver all interrupts to the boot processor. It is recommended, but not required, that interrupts be delivered to all processors in a way that distributes the interrupt load over all processors, taking care to avoid executing multiple copies of the interrupt handler for a particular source (that is, interrupts through a particular interrupt vector) on multiple processors at once; executing multiple interrupt handlers from distinct sources which just happen to be assigned the same IRQL is fine.

3.0 I/O Section

There were several requirements on the design of the EISA and/or PCI I/O section.

1) The EISA bus and the PCI bus are longword wide busses with four byte enables (that is, they allow byte and word width reads and writes), whereas the Alpha AXP architecture allows only longword and quadword width reads and writes. This makes it necessary to devise a scheme which uses Alpha AXP longword and quadword loads and stores to generate masked reads and writes. In addition, neither bus makes any attempt to conceal its Intel heritage, so it is also necessary to devise a schemes to generate I/O space cycles, interrupt acknowledge cycles, PCI configuration cycles, and PCI special cycles.

2) The Windows NT operating system assumes that a block of physically contiguous addresses in I/O or memory space corresponds to a block of physically contiguous addresses in the address space seen by the processor section (the HalTranslateBusAddress procedure returns a single address, not a vector of addresses). This prohibits implementations which use high order physical address bits to generate byte enables.

3) The Windows NT operating system assumes that all locations in memory space on all I/O busses can be directly accessed at all times. It expects to be able to compute the physical address corresponding to a particular memory space address on a particular I/O bus (by calling HalTranslateBusAddress), to be able to map that block of physical addresses into some virtual address space (by calling MmMapIoSpace), and then be able to context switch between that address space other address spaces (which may also contain mapped regions of EISA memory space) without special actions. This prohibits implementations which use bank switching registers to generate high order I/O bus memory space address bits or to select between multiple I/O busses. No such assumption is made in I/O bus I/O space or in I/O bus memory space which is accessed only by device drivers, since all references are made through the READ_PORT and WRITE_PORT macros.

3.1 Byte Enable Encoding

The EISA bus and the PCI bus require the ability to generate byte masked longword read and write cycles. Although many implementations are possible, encoding this information in the address makes synchronization easier, since masked cycles are generated by unprivileged code.

One possible implementation (which has been more-or-less proven in practice) uses bits a[04..03] of the physical address as size bits. The remaining physical address bits are used as I/O bus address bits, with the physical address bit a[05] taken to be byte address bit a[00]. The two size bits and the two least significant address bits determine which byte(s) participate in a read or write. In EISA systems the two least significant physical address bits are only used to generate the byte enables on some internal bus which is eventually transformed into the EISA bus. In PCI systems somewhat more complicated logic is needed because of the more complicated rules regarding what is driven on AD[01..00] during PCI address phase.

This implementation is only more-or-less proven in practice because JENSEN shifts everything two bits to the left; that is, it places the size bits in a[06..05]. This simplifies

the logic a little, but restricts the size of the external address space to 32M. The described implementation is being used on all new machines (actually, I heard through the grapvine recently that there is a PCI design for TurboLaser which uses the old Jensen scheme). The described implementation can always be built, since the SRM mandates quadword granularity of reads and writes in all regions of the address space.

The size encoding is described in the following table. It is unspecified what byte enables are generated for address and length combinations not in this table, but since regions of I/O bus memory space will be mapped into areas of the virtual address space which are accessible by non privileged software, they must not be security holes (so designs cannot, for example, hide configuration or interrupt acknowledge cycles in the unspecified encodings).

a[06..05]	a[04..03]	Enables
00	00	FFFT
01	00	FFTF
10	00	FTFF
11	00	TFFF
00	01	FFTT
01	01	FTTF
10	01	TTFF
00	10	FTTT
01	10	TTTF
00	11	TTTT

Reads and writes of I/O and memory spaces must always be performed using longword loads and stores; systems built using CPU chips with cache-line external interfaces are unlikely to be able to determine the size (longword vs quadword) of a read. The system performs no repositioning or masking of data on reads or writes; data is always read or written from the natural byte lane of the longword, and software must perform the necessary shifting and masking. The system performs no special sequencing of reads and writes of I/O and memory spaces; if software needs to ensure that a sequence of reads and writes happen in a particular order, then it must include the necessary MB instructions.

This implementation has no way to generate a 64 bit read or write on a 64 bit PCI bus. This is harder than it sounds, because on EV4 there is no way to determine the size (longword vs quadword) of the load instruction which caused a read miss.

This encoding is fairly low performance, because it discards much of the potential bandwidth of the processor chip's pins, and much of the data gathering abilities of the processor chip's write buffers. This is not much of an issue on pure EISA systems like Jensen, since the EISA bus is slow, the ISA option cards which are plugged into many of the EISA slots are worse, and the processor chip side of most EISA chipsets can only handle single longword transactions (and most cannot even run those cycles back-to-

back). Systems which allow PCI options may have higher expectations, and higher performance encodings may need to be developed.

The longword dense address space implemented by the HARLEY chipset is a step in this direction. Note that if it were quadword dense then standard Alpha byte manipulation sequences aimed at a block of memory would work; the performance might be very poor, but the code would function. The longword dense vs quadword dense arguments happen because the Alpha SRM requires only quadword granularity in non-memory regions, which encouraged EV4 to adopt an external interface in which it was not possible to tell the size of the load instruction which caused a read miss (writes in non-memory regions on EV4 have longword granularity, but it's an artifact), which forced systems to assume some size for I/O space reads. The EV4 external interface was changed between pass 1 and pass 2, at the request of the workstation group, to bring a[2] to the pins on read misses; this makes the HARLEY longword dense space possible. In retrospect, one wonders why the workstation group didn't ask for a size indication at the same time, and/or why nobody noticed that their request put the correctness of the quadword non-memory region access granularity in doubt.

3.2 Memory Space Accesses

Processor reads and writes in the region of the physical address space assigned to I/O bus memory space are translated into the appropriate masked memory space reads and writes on the I/O bus, using the encoding for low order address bits and byte enables defined for the system. Such references must be uncached.

Since several of the processor's low order address bits may be consumed by a size field and by any shift required to ensure that the all the necessary address bits are visible outside the CPU chip, there may not be enough CPU physical address bits left to generate full 32 bit I/O bus memory space addresses, or even I/O bus memory space addresses beyond the end of system memory. The inability to generate I/O bus memory space addresses beyond the end of system memory on systems which use direct-mapped DMA addressing (section 3.4) is serious, since I/O bus memory space devices must be configured so that their contribution to I/O bus memory space lies beyond the end of system memory to prevent their memory from casting a configuration-dependent shadow on system memory where DMA does not work.

Since the Windows NT operating system assumes that all locations in I/O bus memory space on all I/O buses can be directly accessed at all times, and since the I/O bus memory space will be wildly discontiguous in any system which has an Intel style address map (network cards and VGAs have memory space buffers between 640K and 896K, and large frame buffers need to be placed in memory space above system memory, which is hundreds of megabytes away), the missing high order address bits cannot be supplied from a single register. However, in systems which have an Intel style address map two registers are enough; one would always contain 0 (and be used to access the low 16M of memory space) and the other would contain a system and/or configuration specific value (and be used to access the large frame buffers, which are configued above system

memory in memory space). The registers would be loaded once, early in the life of the system, and never changed. Note that since one register always contains 0 it does not really need to me implemented as a register; a set of carefully placed AND gates will suffice.

The Jensen implementation has a single register, and uses it at all times. System software puts a 0 in this register, and configuration restrictions arrange that all EISA memory space devices are assigned addresses in the low 32M. With this implementation it is all but impossible use a device which needs a large block of EISA memory space. If it is necessary to address an EISA memory device above 1M (which would cast a shadow on main memory at a place where it might matter) then system software can avoid the problem by simply arranging to not use the memory upon which the shadow is cast. One way to do this would to be to mark it as defective.

3.3 I/O Space Accesses

Processor reads and writes in the region of the physical address space dedicated to I/O bus I/O space are translated into the appropriate masked I/O space reads and writes, using the encoding for low order address bits and byte enables defined for the system. Such references must be uncached.

EISA I/O space addresses are only 16 bits wide so there will always be enough physical address bits to allow direct access. PCI I/O space addresses are 32 bits wide, but since Intel processors cannot generate such addresses, and since Intel is very strongly recommending that peripheral chips not use PCI I/O space addresses above 64K, it seems pointless for systems to be able to generate, and unwise for systems to require the use of, PCI I/O space addresses above 64K. However, all 32 PCI address bits must be driven.

Note that an address in the low 64K of an address space is also in the low 16M of an address space, so high order PCI address bits can be generated the same way in memory space and in I/O space.

3.4 DMA Accesses

Memory space addresses generated by I/O bus direct masters (or, in EISA systems, by the DMA transfer controller) sometimes need to be treated as DMA references to the cache and memory system in the processor section. Two implementations are possible.

In the direct-mapped scheme memory space reads and writes between location 0 and the highest physical memory address implemented by the system (which is not necessarily installed in the system) are treated as DMA references, with the exception, in EISA systems, of any holes needed to allow correct operation of ISA options.

Holes are needed because ISA memory space options respond based on a 24 bit address decode and the MEMR/MEMW strobes (this is even true on old 8-bit cards, since a[23..20]=0000 is implied by the SMEMR/SMEMW strobes), resulting in address

conflicts during DMA between the option and the (direct-mapped) main memory. Most conflicts happen between 640K and 896K, where the BIOS ROMs and the display memory window of VGA compatible display cards lie; a single hole between 512K and 1M is recommended. Correct operation of DMA transfers targeted at ISA memory space options which must be addressed between 1M and 16M requires additional holes. Note that the Windows NT system has no mechanism for dealing with the fact that a DMA transfer cannot be targeted at the main memory whose addresses conflict with such an option; in a system using direct-mapped DMA little can be done with such main memory other than mark it bad to keep it away from the system.

In the scatter-gather mapped scheme memory space reads and writes in a system specific address space window are treated as DMA references. Address translation hardware, manipulated by the IoMapTransfer and IoFlushAdaptorBuffers procedures, determines the system memory address corresponding to each memory address in the window. The address translation hardware must have 8K (that is, Alpha AXP minimum pagesize) byte granularity.

The Windows NT operating system makes extensive use of DMA transfers to and from buffers which are virtually contiguous, but not necessarily physically contiguous. The inability to perform these transfers directly may have serious performance consequences, since it may force a large fraction of I/O transfers to be copied through a contiguous staging buffer.

Experience with DMA devices in other operating systems has shown that scatter-mapped DMA is a good thing, and there is no reason to believe that the Windows NT operating system will be different (especially considering that there is a wonderful comment in one of the workbooks which talks about "certain archaic systems" which require that DMA transfers be aimed and physically contiguous pages). The LCA and APECS PCI interfaces implement scatter-mapped DMA, and do so in a compatible fashion, which uses a table in main memory to hold the physical page numbers, and keeps a small translation buffer to avoid looking at the table in main memory on every DMA cycle. The designers of the HARLEY chipset have agreed to implment this scatter-mapping scheme in a future version of the HARLEY chipset should performance measurements on the MORGAN system justify the change.

The use of the direct mapped DMA addressing and peripheral controllers which implement their own scatter-gather can be, in many systems, a reasonable alternative to dedicated scatter-mapping hardware, as long as the consequences of having to deal with devices which do not have scatter-gather capability and/or ISA device which cast main memory shadows between 1M and 16M are understood.

The cache and memory system never responds to I/O space addresses generated by direct masters or by the EISA DMA transfer controller. The cache and memory system never responds to IACK, configuration, or special cycles generated by PCI masters.

3.5 DMA Transfer Controller

EISA systems must implement a DMA transfer controller which is functionally compatible to that described in the EISA specification (that is, each of the seven DMA channels must support type A/B/C/compatible timing, 8/16/32 bit devices, single/demand/block transfer modes, and extended addressing). Since the DMA transfer controller is manipulated only by HAL code, it need not be register compatible to with the controller described in the EISA specification.

Jeff: It looks to me like there are features on the standard EISA DMA controllers which are not accessible via the HAL interface, like ring buffer mode, and ISA compatible segment/offset addressing Can we eliminate these features from this specification, so that systems which are not using the Intel chips can eliminate them, or do device drivers sometimes stuff bits into the DMA controllers behind the back of HAL?

3.6 Interrupt Controller

The system must implement an interrupt controller which is prioritizes the various interrupt sources, and, if the priority of an interrupt source is above a priority level set by HAL, delivers an interrupt request and the identity of the interrupt source to the processor section.

The interrupt controller in EISA systems is not required to be register compatible to the interrupt controller described in the EISA specification, since it is only manipulated by HAL. The interrupt controller is required to implement programmable mode control (active high edge trigger vs active low level trigger) on any request line which connects to an EISA slot.

If a system's interrupt controller is implemented using Intel components then its processor section will need to implement a system specific register to generate the Intel-style interrupt acknowledge read cycle. Hardware need only generate a read cycle with the correct semantics; HAL can mimic the interrupt acknowledge behavior of an Intel processor by generating two interrupt acknowledge reads separated by an appropriate delay, and discarding the result of the first.

The following table is a list of EISA interrupt requests (listed in priority order, from highest priority to lowest priority) and the conventional use, if any, in Intel PCs.

IRQ0	ISP Timer (Windows NT profile clock)
IRQ1	Keyboard
IRQ8	(RTC on Intel, unused on Alpha AXP)
IRQ9	EISA bus
IRQ10	EISA bus, (COM3)
IRQ11	EISA bus, (COM4)
IRQ12	Mouse (see section 4.1)
IRQ13	(NDP on Intel, unused on Alpha AXP)
IRQ14	EISA bus, Hard disk

IRQ15	EISA bus
IRQ3	EISA bus, COM2
IRQ4	EISA bus, COM1
IRQ5	EISA bus, (LPT2)
IRQ6	EISA bus, Floppy disk
IRQ7	EISA bus, LPT1

Note that some interrupt requests , like IRQ3, might be shared by the EISA bus and by an I/O device, masquerading as an EISA device, which is, in fact, on the system board. It is unclear what is the best way to hook up such interrupts. One approach, which does work, is to just connect the local I/O device interrupt request to the EISA bus wire, on the grounds that any conflict which happens is no worse than the conflict which would have happened in an Intel based system. A better approach is to add the OR gates needed to cleanly share the interrupt request wires, and to implement a clean interrupt request sharing scheme. This software part of this is straightforward, since the interrupt handlers of Windows NT device drivers are required to return a BOOLEAN which indicates if they serviced an interrupt, and the interrupt dispatcher is required to keep walking the chain of device objects associated with a particular interrupt source as long as interrupt handlers keep returning FALSE. The hardware part (the OR gate) is a little tricky, since the polarity of EISA bus interrupt requests is programmable. This may not actually be a problem, since the EISA bus interrupt requests upon which these sharing situations occur have stong de-facto assertion levels (IRQ3 and IRQ4 are, for example, active high).

3.7 Interval Clock

The system must implement an interval clock which generates periodic timer interrupts. Although the Windows NT operating system prefers an interval timer rate of about 100 timer interrupts per second, and prefers that the interval clock generate an integral number of timer interrupts per second (that is, 100 ticks per second is better than 102.4 ticks per second), just about any interval timer rate can be used.

Systems whose interval clock cannot generate timer interrupts which meet the Alpha SRM requirements (section II, 6.4.3) may be unable to run alternative operating systems such as OpenVMS and OSF/1.

Interval clock interrupts must be delivered to software at a higher IRQL than any device interrupt; the system must take care when bringing interval clock interrupts through the interrupt controllers that interval clock interrupts are never masked by device interrupts. It must be possible to mask interrupt clock interrupts, since there are IRQLs at which interval clock interrupts are blocked.

In a multiprocessor system interval clock interrupts must be delivered to all processors. There is no need for these interrupts to be synchronized.

3.8 Profile Clock

The system must implement a profile clock which generates periodic interrupts between TBD Hz and TBD Hz. The delivery of profile clock interrupts need not be precise (it is, after all, only used by the profiler), so they can be treated just like device interrupts if desired. It must be possible to mask profile clock interrupts, since there are IRQLs at which profile clock interrupts are blocked.

In a multiprocessor system profile clock interrupts must be delivered to all processors. There is no need for these interrupts to be synchronized.

The EISA specified interval timer 1 of counter 0, which counts at 1.193MHz and delivers an interrupt through IRQ0 of the EISA interrupt controller, is used as the profile clock in all Intel based systems and in the Jensen system. Nobody seems to have any idea how the rate of the profile clock should be related to the cycle time of the processor.

3.9 Realtime Clock

The system must implement the BB_WATCH hardware as described in the SRM; that is, a realtime clock with at least 1 second resolution, with at least 50 ppm long term stability, with an update time of 1 second or less, and with battery backup with powerfail detection.

Any of the PC standard time-of-year clocks, such as the Motorola MC14681A or the Dallas DS1287A can be used to implement the realtime clock. In addition, these chips all provide a periodic interrupt source, one configuration of which generates an interrupt every 976.562 uS, which can be used as an SRM compliant interval clock.

3.10 Configuration Memory

The system must provide at least TBD bytes of non-volatile configuration memory. This configuration memory can be implemented as a CMOS RAM backed up by a battery, or it can be implemented as an electrically erasable ROM. There is no need for powerfail detection on a battery backed up configuration memory since all data in it is protected by software means.

John: The value of TBD will actually be a formula, since the size is a function of the number of I/O busses and the number of slots per bus. Do you have any idea for the actual values would be in the formula? Does anybody have any idea what the formula would be for PCI?

3.11 Error Registers

The system board I/O control function section of the EISA specification requires that system errors be reported to the CPU via a non-maskable interrupt controlled by a number of special purpose registers (NMI status, NMI enable, extended NMI, software NMI). The system is not required to implement these registers as described; instead, they should implement whatever error detection and reporting hardware is appropriate, and

control it with system specific registers. EISA systems should support IOCHK* detection, EISA bus timeouts, and programmable EISA bus reset. The software fail-safe timer and the software generated NMI features of EISA can be omitted, because they are useless.

The Windows NT operating system treats the non-maskable interrupt as a fatal condition which prevents even orderly system shutdown. It is not appropriate for the system to report recoverable I/O errors, power failures, or recoverable memory errors using a non-maskable interrupt.

3.11 Power Fail

The Windows NT operating system supports power fail recovery.

A system which implements power fail recovery must arrange that its memory system retains data during the power failure (and no bits can be scrambled as the system transitions from normal power to battery power), and must report the power failure to all processors using a distinctive power fail interrupt. It must be possible mask the power fail interrupt, since there are IRQLs at which power fail interrupts are blocked.

It must be possible for a system which implements power fail recovery to differentiate between a cold start (where the content of memory is unknown) and a warm start (where the content of memory is a Windows NT system whose state was saved by a power fail interrupt). In multiprocessor systems only the boot processor needs to be able to differentiate between these two situations.

3.12 Copy Acceleration

Digital's Windows NT development team has suggested to Microsoft that standard HAL procedures be defined for bulk transfers between main memory and I/O bus memory, much like the procedures which already exist for bulk transfers between main memory and FIFO-like devices in I/O space, and it seems likely that Microsoft will accept this recommendation. This means that if a system implements specialized hardware to assist in these bulk transfers (which should be common; this is the operation used to transfer a screen image created in main memory to a frame buffer on the I/O bus) that high level software can use it in a standardized way.

Systems are not discouraged from investigating special hardware for the purpose of accelerating these bulk transfers. However, since there is no hard data to defend or refute the value of such hardware, systems should not be designed so that the acceleration hardware reduces the performance of the standard access methods.

4.0 Required Peripherals

The system is required to implement a number of peripherals. It is usually preferable to implement a required function in a way which is completely compatible to the Intel based PC implementation, since such an implementation would be very cost effective, and the standard driver from the Windows NT system can be used.

The system has considerable flexibility in assigning physical addresses to the peripherals, since they are manipulated by high-level software via READ_PORT and WRITE_PORT HAL calls, and almost any addressing model can be accommodated by the appropriate collusion between HalTranslateBusAddress and procedures invoked by the macros. However, introducing complexity into these often used paths is not good, so it is recommended that peripherals be accessed via regions of the physical address space which have the same semantics as I/O bus cycles (section 3.1). In EISA systems peripherals which are Intel compatible can placed at their normal locations in EISA I/O space, since there is no danger of them casting a shadow; such peripherals will often resist being hooked up in any other way.

4.1 Keyboard and Pointing Device

The system must support a keyboard and a pointing device.

The Windows NT keyboard model is based on a virtual keyboard, which reports all events as key-up and key-down transitions, and uses a set of virtual keycodes which are keyboard hardware independent (the VK_ codes in "winuser.h"). The actual keycodes used by the keyboard hardware are mapped into virtual keycodes by the keyboard driver. It must be possible to generate virtual key-up and key-down events for all keys on the keyboard, including modifier keys like the shift keys and the control key(s).

The virtual keycode set was designed with the key labeling of the 102- key IBM-PC keyboard in mind, and so it is easy to write a portable program whose user interface is aware of that labeling. Such applications will be very hard to use on keyboards with unusual labeling, or which generate some of the virtual keycodes using multikey sequences.

The Windows NT pointing device model is based on a mouse. The pointing device driver uses the positioning data provided by the hardware to update the current X-Y position, and transforms the actual pushbutton codes used by the hardware into virtual left, middle, and right mouse button presses and releases (the three mouse buttons are assigned virtual keycodes just like the keyboard). Windows NT applications expect to have at least two mouse buttons.

The standard Windows NT keyboard driver handles a 102-key PS/2 compatible keyboard, using the standard pre-programmed microcontroller (or its gate-array equivalent), addressed at its standard place (ports 0060H and 0064H) in EISA I/O space. The standard Windows NT mouse driver handles a PS/2 compatible mouse, sharing a controller with the keyboard. The keyboard interrupt is active high edge triggered, and is

brought to IRQ1 on the EISA interrupt controller. The mouse interrupt is active high edge triggered, and is brought to IRQ12 on the EISA interrupt controller.

Systems whose mouse interface cannot handle a 3 button mouse may be difficult to use with alternative operating systems such as OpenVMS and OSF/1.

Jeff: I think that we should recommend that the keyboard and mouse use the same interrupt request (IRQ1), rather than two different interrupt requests (IRQ1, IRQ12) like an Intel PC. Using different interrupt requests for the keyboard and mouse creates the situation where there are two interrupt handlers at two different priority levels accessing the same registers in the keyboard and mouse interface, which means the low priority handler will need to raise IRQL to access the registers.

4.2 Display

The systems must support a display.

The display must have at least 1024x768 pixels (with a 1:1 aspect ratio). It is recommended that systems use color displays (even though the Windows NT GDI can correctly deal with bilevel and grayscale displays) since many Windows NT applications will have their roots in Windows, where color displays are the norm.

The standard format bitmaps manipulated by the Windows NT GDI have 1, 4, 8, 16, 24, or 32 bits per pixel. Although the Windows NT GDI can correctly deal with display devices which use device-specific bitmap formats (and, in fact, the VGA is an example of a device in this class), it is recommended that the display be able to be accessed as a simple frame buffer holding pixels in one of the standard formats, since the driver for such a device is extremely simple.

4.3 Disk

The system must support a disk (there are no diskless Windows NT systems). A disk 120 megabytes in size will hold the basic system software, although a disk TBD megabytes in size is needed to hold the complete system, along with a useful suite of utilities and a reasonable quantity of user data.

The system assumes that disk reads and writes which are aligned on sector boundaries on the disk and are for multiples of full sectors can be performed directly to and from client buffers (that is, it assumes the NO_BUFFERING open option does something). The system does not assume that unbuffered disk reads and writes can be performed to memory addresses which are not aligned on 512 byte boundaries (code in the system can determine the actual required alignment by calling NtQueryInformationFile, but always has to deal with being told the required alignment is 512 bytes).

The I/O system documents sometimes express the disk and/or memory address alignment requirements as "sector boundary" and sometimes as "512 byte boundary". The description of the FILE_FLAG_NO_BUFFERING option of CreateFile and OpenFile calls in the Win32 API demands sector alignment of both disk and memory addresses. The FILE_ALIGNMENT_INFORMATION subfunction of NtQueryInformationFile has no return code for alignments stricter than 512 bytes. I think what is really being said here is that the disk address must be on a sector boundary, that the transfer count must be for full sectors, that drivers cannot demand memory address alignment more strict than 512 bytes, and that programs coded to the Win32 API will overly-constrain their memory addresses (or, more likely, will use 512 byte alignment).

Is a more general statement being made here regarding devices which might ever be used in a NO_BUFFERING context, like the CD-ROM? Also, is there, in fact, a hidden assumption here that disks have 512 byte physical sectors? How much code in, say, the file system would keep working if it could not use 512 as a count on a file which was opened with the NO_BUFFERING option, because the file was aimed at a disk with a 1024 byte sector?

4.4 Audio

The system must support a small speaker driven by a tone generator which can generate tones between TBD Hz and TBD Hz, so that the HalMakeBeep procedure can be implemented. This can be a simple square wave generator; in EISA systems the standard tone generator, implemented by interval timer 1 of counter 2, and enabled by bits in the NMI status and control port, is acceptable.

The speaker, the tone generator, and the HalMakeBeep procedure seem to be historical dregs left over from Intel PCs, which use them to make irritating noises during system startup . They make little sense on an unattended server machine. Can we just delete the hardware and no-op the HAL procedure?

5.0 Recommended Peripherals

It is recommended that the system implement a number of additional peripherals, because there is an important standard function for that peripheral. Note that the fact that a peripheral would be generally useful is not enough for inclusion in the recommended peripherals list; for example, there is no mention of audio, although most systems will, in fact, be required by the force of the market to have audio capabilities.

5.1 Serial Line

It is recommended that the system support an RS232 compatible serial port interface, with modem control, for use by the Windows NT kernel debugger.

The standard Windows NT serial port driver handles standard PS/2 serial ports (including the deep silos, as implemented in the newer versions of the serial I/O chips) addressed at the standard locations (line 1 at 03F8H - 03FFH, line 2 at 02F8H - 02FFH) in EISA I/O space. Serial port interrupts are active high edge triggered. If they are connected directly to an EISA bus line, interrupts from line 1 go to IRQ4 on the EISA interrupt controller, and interrupts from line 2 go to IRQ3 on the EISA interrupt controller.

Systems which use standard PS/2 serial interface chips but which do not bring the serial port interrupts through the EISA interrupt controllers should be aware of the fact that much existing serial port code uses bit 3 of the MCR as a global interrupt enable, blissfully unaware that the word enable in this context refers to the tristate enable on the chip's interrupt output pin (quite a different meaning from the word enable as used in the description of the bits in the IER). The interrupt output from the serial chip is active high and has a pullup on it, so disabling the interrupt in the MCR actually unconditionally asserts the interrupt request. Because the EISA interrupt controllers are low-to-high edge triggered, as long as interrupts are disabled when there is an interrupt pending, it appears that the right thing happened. The driver is tristate (not open-drain) so gentle pulldowns may be more appropriate if the serial port interrupt request is brought to a level triggered interrupt request input.

5.2 Floppy Diskette

It is recommended that the system be able to read and write 720K, 1.44M, and 2.88M 3.5 inch floppy diskettes which follow the formatting conventions of Intel based systems. Such diskettes are used both as a distribution media for shrink-wrapped applications and for the exchange of small amounts of data between systems not interconnected by a network. Some systems will deliver part of the Windows NT distribution kit on floppy diskette as well (the platform independent parts of the system will be on the CD-ROM, and the platform specific parts will be on floppy diskette); in such systems support for the floppy diskette is, of course, required.

The standard Windows NT floppy diskette driver handles the standard PS/2 compatible controller addressed at its standard place (ports 03F0H - 03F7H) in EISA I/O space. Floppy diskette controller interrupts are active high edge triggered, and, if connected to an EISA bus line, go to IRQ6 on the EISA interrupt controller.

5.3 CD-ROM

It is recommended that the system support a CD-ROM drive. CD-ROMs are used as the primary distribution medium for the Windows NT system itself, as well as for large shrink-wrapped applications.

It is recommended that a SCSI CD-ROM drive be used, since this enables the system to share a controller between the disk and the CD-ROM, and, courtesy of the external SCSI connector, enables users to share a single CD-ROM drive among a group of Windows NT systems, which will reduce cost-of-ownership in environments where the CD-ROM

is only used as a distribution medium. It is also recommended that a CD-ROM drive with a audio jacks be used, since the incremental cost of having such a drive is low (often it is zero), and the Windows NT systems includes standard software to use the CD-ROM drive as an audio player.

I need to understand some low-level CD-ROM issues. What is the format used by the Microsoft distribution CD-ROMs. What is the extra functionality needed in the CD-ROM drive to allow Kodak PHOTO-CD disks to be read (I know that there is something, because the QuickTime kit says that some old drives cannot read the Kodak disks).

Windows NT for Alpha AXP Calling Standard

Revision: 1.6

15–March–1993

Issued by:

Ron Brender, Digital Equipment Corporation

digital™

Digital Equipment Corporation
Maynard, Massachusetts

CONTENTS

FIGURES

TABLES

Preface

This document defines the *Windows NT for Alpha AXP Calling Standard*. There are, at present, three closely related Alpha calling standards. This document describes the Windows NT for Alpha AXP flavor which is used for the Windows NT for Alpha AXP product.

Acknowledgements

It should be noted that this document has been derived from various other works related to the Alpha program. The following people contributed significant amounts of time and energy towards the creation of this and related source documents:

- Gary Barton
- Ron Brender
- Peter Craig
- Mark Davis
- Terry Grieb
- Rich Grove
- Steve Hobbs
- Ayub Khan
- Ray Lanza
- Ken Lesniak
- Bill Noyce
- Chip Nylander
- Mike Rickabaugh
- Tom Van Baak

and of course the numerous others that have provided "special" review.

In addition, some specifications are based in part on published specifications and related correspondence from Microsoft Corporation, whose assistance is hereby acknowledged.

Revision History

Date	Revision Number	Author	Summary of Changes
23-Mar-1992	0.0	T. Group	This document has a lot of history from the Alpha/VMS and Alpha/OSF-1 versions as well as other documents. As such the *authoring* part of the following notes is somewhat bogus and should more appropriately be called *edited by*. (A version)
23-Mar-1992	1.0	T. Grieb	Start with Alpha/OSF-1 document and "NT"-ize it. Change GP def and change register mix. (Spring version)
12-Jun-1992	1.1	R. Brender	Integrate Windows NT exception handling, collect procedure descriptor and call chain material in new Chapter 9, update with applicable recent ECOs to the Alpha VMS calling standard.
18-Jun-1992	1.2	R. Brender	Continue editorial cleanup, re-organize topics between Chapters 8 and 9, conditionalize to enable creation of either OSF/1 or Windows NT versions.
5-Jul-1992	1.3	R. Brender	Revise GP handling and procedure descriptors to match Windows NT MIPS, add actual target names where known.
26-Jul-1992	1.4	R. Brender	Updates from reviews.
15-Jan-1993	1.5	R. Brender, M. Davis	Resolve open issues, updates from reviews.
15-Mar-1993	1.6	R. Brender, M. Davis	Cleanup typos, minor clarifications

Notation Used in This Document

The specifications in this document are presented as follows:

- Editorial Comment

```
\\
All text enclosed in double backslashes, illustrated by this paragraph, is
editorial comment, is not formally a part of the specification, and will not
necessarily be in future revisions of this document.
\\
```

- Constants

  Constants are presented symbolically with their value given at the point of definition in this standard. Concrete language bindings for each constant are provided in system definition files external to this standard.

```
\\
Note that the symbols used in this document do NOT follow any particular
naming conventions nor do they adhere to POSIX or other conventions.
We need to decide what we want the Alpha naming conventions to be (e.g. are
future naming conventions affected by the presence of open, standard APIs on
multiple operating systems), and then this document will be brought into
conformance.
\\
```

- Functional Interfaces

  Functional interface syntax is presented in abstract form. Concrete language bindings for each functional interface are provided in system definition files external to this standard.

  The semantic capabilities of each functional interface are defined in American English.

- Algorithms

  Algorithms are presented precisely, as a series of steps, in American English.

- Conventions

  All conventions that are important to correct program execution are presented in a form appropriate to each convention.

- Methods

  Actual or recommended methods are presented informally, using examples, suggestions, or other appropriate form.

- Numbering

  All numbers are represented as decimal values unless otherwise indicated. Non decimal numbers are typically represented with the name of the base in parentheses following then number, E.G. 1B(Hex).

- Figures

  Figures that represent memory or register layouts follow the convention that increasing addresses run from the top to bottom and right to left of a page. Most significant bits are on the left and least significant bits are on the right.

- Code Examples

All code examples in this document are supplied strictly for purposes of explanation. They are presented in a form that expresses the relevant concept with clarity. They do *not* reflect optimized and properly scheduled code sequences that a compiler would generate.

Assembler syntax used follows the *Alpha System Reference Manual* (including Appendix A, section A.4.3 on Stylized Code Forms) and does not represent actual Alpha Assembler notation.

- Data Structures

  Data structures are defined in terms of the physical memory format of each structure. Concrete language bindings for each data structure are provided in system definition files external to this standard.

- Data Structure fields

  Record fields are referred to by using the name of the record or subrecord followed by a dot, followed by the field name; as in RECORD_NAME.SUB_RECORD.FIELD.

CHAPTER 1

INTRODUCTION

This standard defines the run time data structures, constants, algorithms, conventions, methods, and functional interfaces that enable a native user mode procedure to operate correctly in a multilanguage and multithreaded environment on Windows NT for Alpha AXP systems. These properties include the contents of key registers, the format and contents of certain data structures, and actions that procedures must perform under certain circumstances.

This standard also defines properties of the run time environment that must apply at various points during program execution. These properties vary in scope and applicability. Some properties apply at all points throughout the execution of user mode code, and must therefore be held constant at all times. Examples of such properties include those defined for the stack pointer and various properties of the call chain navigation mechanism. Other properties apply only at certain points, such as call conventions that apply only at the point of the transfer of control to another procedure.

Furthermore, some properties are optional depending on circumstances. For example, compilers are not obligated to follow the argument list conventions when a procedure and all of its callers are in the same module, have been analyzed by an interprocedural analyzer, or have private interfaces (such as language support routines).

NOTE

The specifications in this document are presented in an *as if* definition. This simply means that all conformant code must behave *as if* the specifications have been met. In particular, this calling standard is designed such that additional link-time information can be utilized to optimize or even remove instructions in critical code paths and as such achieve higher performance levels.

In addition, in many cases significant performance gains can be realized by selective use of non-standard calls when the safety of such calls is known. Compiler writers are encouraged to make optimal use of such optimizations as appropriate while always ensuring that procedures outside the compilation unit can proceed *as if* the letter of the standard were met.

The conventions specified in this standard are intended to fully exploit the architectural and performance advantages of the Alpha AXP hardware architecture and are designed to provide a leadership execution environment for applications and languages on Windows NT for Alpha AXP. Some of these conventions are visible to the high level language programmer, and therefore may require source level changes in high level language programs when moving them from other environments.

To achieve source level compatibility and portability between the Windows NT for Alpha AXP environment and various other environments users should not depend on the properties of this architecture except indirectly through high level language facilities that are portable across architectures.

By definition, many of the conventions described in this standard differ from other software implementation architectures. Therefore programs that depend on properties of this architecture may not be portable to other architectures.

1.1 Applicability

This standard defines the rules and conventions that govern the *native user mode run time environment* on Alpha AXP systems. It is applicable to all products executing in native user mode on the Windows NT for Alpha AXP operating system.

Specific examples of uses of this standard are:

- All externally callable interfaces in standard system software
- All intermodule calls to major software components
- All external procedure calls generated by language processors without the benefit of interprocedural analysis or permanent private conventions (such as those used for language support runtime library routines).

1.2 Architectural Level

This standard defines an *implementation level run time software architecture* for Alpha AXP operating systems.

The interfaces, methods, and conventions specified in this document are primarily intended for use by implementors of compilers, debuggers and other run time tools, run time libraries, and base operating systems. These specifications may be, but are not necessarily, appropriate for use by higher level system software and applications.

Compilers and run time libraries may provide additional support of these capabilities via interfaces that are more appropriate for compiler and application use. This specification neither prohibits nor requires such additional interfaces.

1.3 Related Documents

This calling standard is a component of the larger Alpha AXP Software Architecture, and depends on certain standards and conventions that are not described by this document.

Those standards, described by other documents, include:

- Object language (including link-time optimizations) and object file format
- Status values and message definition, formatting, and reporting
- Heap memory management and dynamic string management
- Multithread Architecture

- Names and naming conventions

The above topics as well as other related topics may be found in:

- Digital Equipment Corporation, Alpha AXP System Reference Manual
- Digital Equipment Corporation, OpenVMS Calling Standard, October 1992
- Digital Equipment Corporation, OSF/1 for Alpha AXP Calling Standard, Revision 1.6, November 1992
- POSIX 1003.1, IEEE Standard Portable Operating System Interface for Computer Environments - IEEE Std 1003.1-1988
- American National Standard for Information Systems Programming Language C - ANSI X3.159-1989 (or its ISO equivalent, ISO 9899)

1.4 Definitions

The following terms are used in this standard:

- *Address*: A 32-bit value used to denote a position in memory. In the case of Windows NT for Alpha AXP systems, an address is sometimes represented by a 64-bit value in which case the high-order 33 bits of that value must be the same (that is, the 64-bit value must equal the sign-extended 32-bit value).
- *Argument list*: A vector of quadword entries that represents a procedure parameter list and possibly a function value.
- *Asynchronous procedure call*: An asynchronous interruption of normal code flow caused by some software event. This interruption shares many of the properties of hardware exceptions that includes forcing some out-of-line code to execute.
- *Bound procedure*: A type of procedure that requires knowledge (at run-time) of a dynamically determined larger enclosing scope to execute correctly.
- *Call frame*: The body of information that a procedure must save to allow it to properly return to its caller. A call frame may exist on the stack or in registers. A *call frame* may optionally contain additional information required by the called procedure.
- *Descriptor*: A mechanism for passing parameters where the address of a descriptor is an entry in the argument list. The descriptor contains the address of the parameter, data type, size, and additional information needed to fully describe the data passed.
- *Dynamic-link library (DLL)*: A form of shared image on Windows NT for Alpha AXP systems which can be loaded into a process for execution.
- *Exception code*: A 32-bit value used to uniquely identify an exception condition. An exception code can be returned to a calling program as a function value (status) or raised using the exception handling mechanism.
- *Exception condition (or exception)*: Some exceptional condition in the current hardware and/or software state that should be noted or fixed. Its existence causes an interrupt in program flow and forces execution of out of line code. Such an event may be caused by

exceptional hardware state such as arithmetic overflows, memory access control violations, and so on or by actions performed by software, such as subscript range checking, assertion checking, or asynchronous notification of one thread by another.

While the normal control flow is interrupted by an exception, that condition is termed *active*.

- *Exception handler*: A procedure designed to handle exceptions (conditions) when they occur during the execution of a thread.
- *Function*: A procedure that returns a single value in accordance with the standard conventions for value returning. Additional values are returned by means of the argument list.
- *Hardware exception*: A particular category of exceptions that directly reflect an exceptional condition in the current hardware state that should be noted or fixed by the software. Hardware exceptions may occur synchronously or asynchronously with respect to the normal program flow.
- *Image*: A collection of compiled modules that are combined by a linker into a form that is ready to be loaded for execution.
- *Immediate value*: A mechanism for passing input parameters where the actual value is provided in the argument list entry by the calling program.
- *Language support procedure*: A procedure called implicitly to implement higher-level language constructs. Such procedures are not intended to be explicitly called from user programs.
- *Library procedure*: A procedure explicitly called using the equivalent of a call statement or function reference. Such procedures are usually language independent.
- *Natural alignment*: An attribute of certain data types that refers to the placement of the data, such that the lowest addressed byte of the data has an address that is a multiple of the size of the data in bytes. Natural alignment of an aggregate data type generally refers to an alignment, such that all members of the aggregate are naturally aligned.

  There are five natural alignments defined by this standard:

  — Byte—any byte address
  — Word—any byte address that is a multiple of two
  — Longword—any byte address that is a multiple of four
  — Quadword—any byte address that is a multiple of eight
  — Octaword—any byte address that is a multiple of 16
- *Procedure*: A closed sequence of instructions that is entered from and returns control to the calling program.
- *Procedure value*: An address value that represents a procedure – the address of the first instruction of the procedure to be executed.
- *Process*: An address space and at least one thread of execution. Selected security and quota checks are done on a per process basis.

This standard anticipates the possibility of the execution of multiple threads within a process. An operating system that only provides a single thread of execution per process is considered a *special case* of a multithreaded system where the maximum number of threads per process is one.

- *Reference*: A mechanism for passing parameters where the address of the parameter is provided in the argument list by the calling program.
- *Shareable image*: An image that can shared by multiple processes. On Windows NT for Alpha AXP, a single copy of the image can be simultaneously included at the same address in multiple using processes. (That is, a shareable image is generally not position independent.)
- *Signal*: A POSIX defined concept used to cause out-of-line execution of code.
- *Standard call*: Any transfer of control to a procedure by any means that presents the called procedure with the environment defined by this document and does not place additional restrictions, not defined by this document, on the called procedure.
- *Standard conforming procedure*: A procedure is said to be standard conforming if it adheres to all the relevant rules set forth in this document.
- *Thread of execution (or thread)*: An entity scheduled for execution on a processor. In language terms, a thread is a computational entity utilized by a program unit. Such a program unit might be a task, procedure, loop, or some other unit of computation.

  All threads executing within the same process share the same address space and other process context, but have unique per-thread hardware context that includes program counter, processor status, stack pointer, and other machine registers.

  This standard applies only to threads that execute within the context of a user mode process and are scheduled on one or more processors according to software priority. All subsequent uses of the term *thread* in this standard refer to such user mode process threads only.
- *Thread safe code*: A property of code compiled in such a way as to ensure it will execute properly when run in a threaded environment. This usually adds extra instructions to do certain run-time checks and requires that thread local storage be accessed in a particular fashion.
- *Undefined*: Operations or behavior for which there is no directing algorithm used across all implementations that support this standard. Such operations may or may not be well defined for a single implementation, but still remain undefined with reference to this standard. The actions of undefined operations may not be required by standard conforming procedures.
- *Unpredictable*: Any results of an operation that cannot be guaranteed across all implementations of this standard. These results may or may not be well defined for a single implementation, but remain unpredictable with reference to this standard. All results caused by operations defined in this standard but where the results are not explicitly specified in this standard are considered *unpredictable*. No standard conforming procedure may depend on *unpredictable* results.

CHAPTER 2

BACKGROUND

This section describes various background information that served as the basis for many of the decisions made during the process of generating this standard.

2.1 Goals

The design of the Windows NT for Alpha AXP calling standard was accomplished with the following goals in mind:

- The standard must support nearly perfect compatibility with the user visible characteristics of the NT operating system as implemented on Intel x86 and MIPS Rx000 systems.
- The standard must be applicable to all intermodule callable interfaces in the native software system. Specifically, the standard must consider the requirements of important compiled languages including Ada, BASIC, BLISS, C, C++, COBOL, FORTRAN, Pascal, LISP, PL/I, and calls to the operating system and library procedures. The needs of other languages that may be supported in the future must be met by the standard or by compatible revision to it.
- The standard should not include capabilities specifically for lower level components (such as assembler routines) that cannot be invoked from the higher level languages.
- The calling program and called procedure can be written in different languages. The standard attempts to reduce the need for use of language extensions for mixed language programs.
- The standard should contribute to the writing of error free, modular, and maintainable software. Effective sharing and re-use of software modules are specific goals.
- The standard should provide the programmer with control over fixing, reporting, and flow of control when various types of exceptional conditions occur.
- The standard should provide subsystem and application writers with the ability to override system messages to provide a more suitable application oriented interface.
- The standard should add no space or time overhead to procedure calls and returns that do not establish exception handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.
- The standard should be optimized for newer, more complex compilation techniques including interprocedural analysis, link-time code transformations, and other mechanisms that are conducive to optimal performance. While holding this goal in mind, the standard must not require any such mechanisms for correctness.
- Provide support for a multilanguage, multithreaded execution environment.

- Provide an efficient mechanism for calling lightweight procedures that do not need to pay the overhead of setting up a stack call frame.
- Provide for the use of a common calling sequence to invoke lightweight procedures that maintain only a register call frame and heavyweight procedures that maintain a stack call frame. This should allow a compiler to determine whether or not to use a stack frame based on the complexity of the procedure being compiled. A recompilation of a called routine that causes a change in stack frame usage should not require a recompilation of its callers.
- Provide exception handling, traceback, and debugging for lightweight procedures that do not have a stack frame.
- Make efficient and effective use of the Alpha AXP hardware architecture.
- Minimize the cost of procedure calls
- Support a 32-bit address user mode environment
- Provide a building block for the next 20 years of computing

2.2 Constraints

This standard was developed under the following constraints:

- The standard must be implementable on all Alpha AXP platforms
- The standard must be implementable by third party compiler writers.
- There is a short development cycle for the first wave of products
- The standard must not require any complex compilation techniques (such as link-time code movement) for correctness.

CHAPTER 3

BASIC CONSIDERATIONS

This section describes some fundamental concepts of the Windows NT for Alpha AXP calling standard.

3.1 Address Representation

One feature of the Windows NT for Alpha AXP flavor of the calling standard is that in **all** cases bits <63:31> of all values that represent addresses are identical. In most cases this fact is used to save storage since only 32 bits are needed to represent an address. Structures that contain low-level machine state sometimes allocate a full 64 bits for addresses even though bits <63:31> are always the same.

3.2 Procedure Representation

One of the distinguishing characteristics of any calling standard is how procedures are represented. The term used to denote the value which uniquely identifies a procedure is a *procedure value*. If the value identifies a bound procedure then it is called a *bound procedure value*.

In the Windows NT for Alpha AXP calling standard a simple (not bound) procedure value is defined as the address of the first instruction of that procedure's entry code (see Section 4.2.6, Entry and Exit Code Sequences).

A bound procedure value is defined as the address of the first instruction of an instruction sequence that establishes the correct execution context for the bound procedure.

In addition, procedures in the Windows NT for Alpha AXP calling standard are associated with a data structure called a procedure descriptor. This structure describes various aspects of the procedure's code which are required for correct and robust exception handling. The exception processing described by this standard is based on the assumption that any given program counter value can be mapped to an associated procedure descriptor that describes the currently active procedure.

3.3 Register Usage Conventions

This section describes the usage of the Alpha AXP hardware general purpose (integer) and floating point registers.

3.3.1 Integer Registers

In a standard-conforming procedure the general purpose, integer registers are used as shown in Table 3–1.

Table 3–1: General Purpose, Integer Register Usage

R0	Function value register. In a standard call that returns a non-floating point function result in a register, the result must be returned in this register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R1..R8	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R9..R14	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
R15	FP, Stack Frame Base register. For procedures with a run time variable amount of stack, this register is used to point at the base of the stack frame (fixed part of the stack). For all other procedures this register has no special significance. If a standard-conforming procedure modifies this register, it must save and restore it.
R16..R21	Argument registers. In a standard call, up to six non-floating point items of the argument list are passed in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R22..R25	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
R26	RA, Return Address register. In a standard call, the return address must be passed and returned in this register.
R27	Conventional scratch register. In a standard call, this register may be modified by the called procedure without being saved and restored.
R28	Volatile scratch register. The contents of this register are always *unpredictable* after any external transfer of control either to or from a procedure. **This applies to both standard and nonstandard calls.** This register may be used by the operating system for external call fixup, autoloading and exit sequences.
R29	GP, Global Pointer or saved register. In a main (static) image, this register contains a pointer to a region of global storage (addresses and variables) constructed by the linker. In a Dynamic-Link Library (DLL), this register must not be used in any way.
R30	SP, the Stack Pointer. This register contains a pointer to the top of the current operating stack. Aspects of its usage and alignment are defined by the hardware architecture. Various software aspects of its usage and alignment are defined in Section 4.2.1, Call Conventions.
R31	RZ, ReadAsZero/Sink. Hardware defined: binary zero as a source operand, sink (no effect) as a result operand.

3.3.2 Floating Point Registers

In a standard-conforming procedure the floating point registers are used as shown in Table 3–2.

Table 3–2: Floating Point Register Usage

F0	Floating point function value register. In a standard call that returns a floating point result in a register, this register is used to return the real part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F1	Floating point function value register. In a standard call that returns a complex floating point result in registers, this register is used to return the imaginary part of the result. In a standard call, this register may be modified by the called procedure without being saved and restored.
F2..F9	Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it.
F10..F15	Conventional scratch registers. In a standard call, may be modified by the called procedure without being saved and restored.
F16..F21	Argument registers. In a standard call, up to six floating point arguments may be passed by value in these registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F22..F30	Conventional scratch registers. In a standard call, these registers may be modified by the called procedure without being saved and restored.
F31	ReadAsZero/Sink. Hardware defined: binary zero as a source operand, sink (no effect) as a result operand.

3.3.3 Register Names

The register names used in this standard follow the conventions used in the *Alpha System Reference Manual*. That is, the names of general registers begin with "R" and the names of floating point registers begin with "F". These same names are also used in the examples in this text.

This register naming convention is not necessarily followed by compiler or assembler tools that are used on Alpha AXP systems. However, in most cases a simple name substitution can convert from the notation used here to the appropriate convention. For example, the following might be used in conjunction with the standard C preprocessor as a prelude to the examples in this text.

```
/*   replacement     also known as */
#define R0     $0     /* v0          */
#define R1     $1     /* t0          */
...
#define R8     $8     /* t7          */
#define R9     $9     /* s0          */
...
#define R14    $14    /* s5          */
#define R15    $15    /* fp or s6    */
#define FP     $15
```

```
#define R16    $16    /* a0          */
...
#define R21    $21    /* a5          */
#define R22    $22    /* t8          */
...
#define R25    $25    /* t11         */
#define R26    $26    /* ra          */
#define RA     $26
#define R27    $27    /* t12         */
#define R28    $28    /* at          */
#define R29    $29    /* gp          */
#define GP     $29
#define R30    $30    /* sp          */
#define SP     $30
#define R31    $31    /* zero        */

#define F0     $f0
#define F1     $f1
#define F2     $f2
...
#define F9     $f9
#define F10    $f10
...
#define F15    $f15
#define F16    $f16
...
#define F21    $f21
#define F22    $f22
...
#define F30    $f30
#define F31    $f31
```

3.4 Program Image Layout

The Windows NT for Alpha AXP calling standard does not define many aspects of an executable image. However, there is one basic concept that is defined to permit optimal access to static data.

A hardware architecture that has the property that instructions cannot contain full virtual addresses is sometimes referred to as a base register architecture. The Alpha AXP architecture is such an architecture. In a base register architecture, normal memory references within a limited range from a given address are expressed by using displacements relative to the contents of some register which contains that address (usually referred to as a *base* register). Base registers for external program segments, either data or code, are usually loaded indirectly through a program segment of address constants.

To optimize this base register access method, this standard allows the main (static) image of an executable program to have a single global storage region that is addressable by the GP register. Together, the linker and the compilers arrange that various static data is collected together into a single such region that is shared across all procedures in the main image.

During the compilation process a compiler generates object language to designate data as belonging in the global segment. The linker pools these contributions to form the segments; in particular, address constants, literals and external storage from multiple compilations can often be combined. The total contributions of all modules of the main image must not exceed 64K bytes of contributions after pooling.

For dynamic-link libraries, no shared global segment is provided. A separate compilation mode is required that neither makes contributions to a global segment nor assumes the availability of a global pointer for addressing. (In this latter compilation mode, the GP must not be used in any way.)

CHAPTER 4

FLOW CONTROL

The following sections contain descriptions of various aspects of the calling standard that deal with the flow of control of a program (as opposed to data manipulation which comes later in Chapter 5).

4.1 Procedure Types

This standard defines three basic types of procedures. A compiler may chose which type to generate based on the requirements of the procedure in question.

The standard procedure types are:

- Stack frame procedure - A procedure that maintains its caller's context on the stack
- Register frame procedure - A procedure that maintains its caller's context in registers
- Null Frame procedure - A procedure that executes in the context of its caller

Some procedures maintain their call frame on the stack, others maintain their call frame entirely in registers (although they may use the stack). Very simple procedures do not necessarily maintain any call frame at all and simply execute in the context of their caller. The calling procedure need not distinguish these cases.

4.1.1 Procedure Descriptor Overview

Every procedure other than a null procedure (see Section 4.1.4) *must* have a structure associated with it that describes which type of procedure it is as well as various other characteristics of the procedure. This structure, called a *procedure descriptor*, is a longword-aligned data structure that provides basic information about a procedure. This data structure is used to interpret the call chain at any point in a thread's execution. It is normally built at compile time and is not normally accessed at run time except in support of exception processing or other rarely used code execution.

Table 4–1 contains a brief summary of the properties of a procedure that can be determined from its associated procedure descriptor. Some of these properties are explicitly represented in the procedure descriptor and others must be derived by examination of the code in a procedure's prologue. (Some fields only apply to certain kinds of procedures.) This summary is included here because many of these properties are mentioned in the descriptions that follow in this Chapter. For a complete description of procedure descriptors, see Section 9.1, Procedure Descriptor Representation.

Table 4–1: Procedure Properties Summary

Field Name	Description
REGISTER_FRAME [1]	Indicates a register (or null) frame procedure rather than a stack frame procedure
BASE_REG_IS_FP [1]	Indicates register R15 is used as a frame pointer rather than as just a preserved register
HANDLER_VALID [1]	Indicates there is an associated exception handler
EXCEPTION_MODE	Indicates the error reporting behavior expected of certain called mathematical library routines
ENTRY_RA [1]	Register that contains the return address at the time of a call. This is always R26
SAVE_RA [1]	Register in which the return address is saved (when not saved on the stack)
FRAME_SIZE [1]	Size in longwords of the (fixed part of the) stack frame
SP_SET [1]	Offset in instructions from the beginning of the procedure to the instruction that changes the stack pointer
ENTRY_LENGTH [1]	Number of instructions in the procedure prologue
BEGIN_ADDRESS	Address of the first instruction (and entry point) of the procedure
PROLOG_END_ADDRESS	Address of the first instruction following the prologue of the procedure
END_ADDRESS	Address of the first location following the last instruction of the procedure
HANDLER_ADDRESS	Address of an associated exception handling procedure
HANDLER_DATA	Supplementary per procedure data to be passed to an associated exception handler

[1] This field is not explicitly represented in the procedure descriptor (see Section 9.1).

4.1.2 Stack Frame Procedure

A *stack frame procedure* is one that allocates space for and saves its caller's context on the stack. This type of procedure is sometimes called a "heavyweight procedure" referring to the cost of storing this context in memory.

Such a procedure can save and restore registers and may make standard calls to other procedures.

The stack frame of this type of procedure consists of a fixed part (the size of which is known at compile time) and an optional variable part. Certain optimizations can be done if the optional variable part is not present. Compilers must be careful to recognize situations that can effectively cause a variable part of the stack to exist in non-intuitive ways such as:

- A called routine may use the stack as a means to return certain types of function values (see Section 5.1.7, Returning Data, for details).

If any such situation exists a compiler *must* choose to use a variable size stack frame procedure when compiling the caller so that an unwind operation can be done correctly.

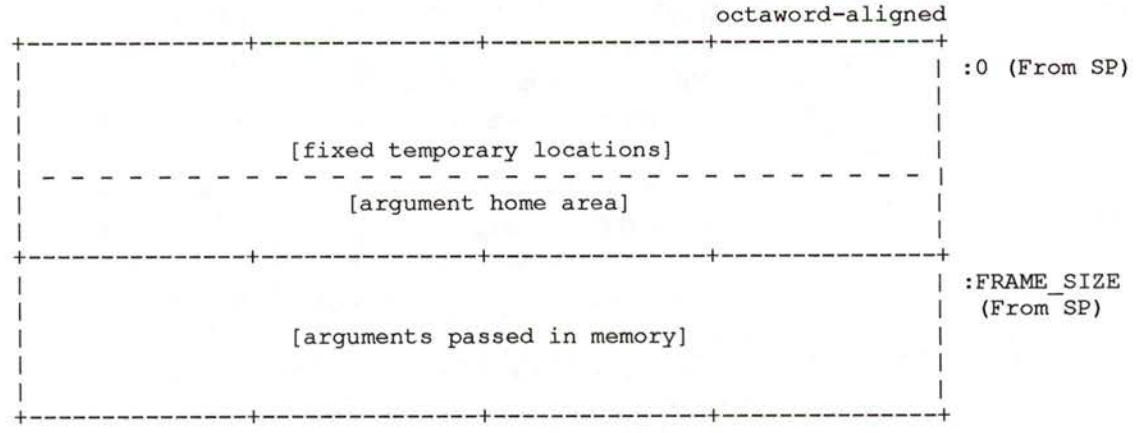
Stack Frame Format

Even though the exact contents of a stack frame are determined by the compiler there are certain properties common to all stack frames. The two basic flavors of stack frames are described below.

The following figure illustrates the format of the stack frame for a procedure with a fixed amount of stack which uses the SP as the stack base register (i.e. BASE_REG_IS_FP is 0). In this case, R15 is simply another saved register and otherwise has no special significance.

Some parts of the stack frame are optional and occur only as required by the particular procedure. Brackets surrounding a field's name indicate the field is optional.

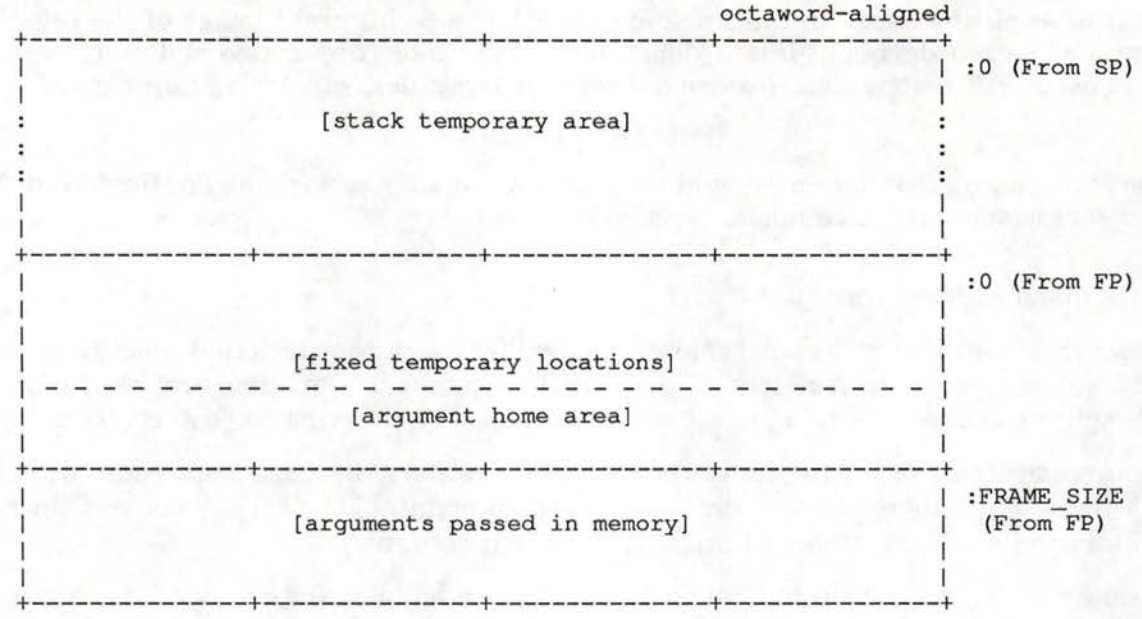
Figure 4–1: Fixed Size Stack Frame Format



The following figure illustrates the format of the stack frame for procedures with a varying amount of stack which use the FP as the stack base register (i.e. BASE_REG_IS_FP is 1).

Some parts of the stack frame are optional and occur only as required by the particular procedure. Brackets surrounding a field's name indicate the field is optional.

Figure 4–2: Variable Size Stack Frame Format



In either case the portion of the stack frame designated by FRAME_SIZE must be allocated and initialized by the entry code sequence of a called procedure with a stack frame.

Fixed temporary locations are optional sections of the stack frame that contain language-specific locations required by the procedure context of some high level languages. This may include, for example, register spill area, language-specific exception handling context (such as language dynamic exception handling information), fixed temporaries, etc.

If a compiler chooses, the fixed temporary locations adjacent to the area pointed to by the frame base register plus FRAME_SIZE can be used for a special purpose termed the *argument home area*. The argument home area is a region of memory used by the called procedure for the purpose of assembling in contiguous memory the arguments passed in registers, adjacent to the arguments passed in memory, so that all arguments can be addressed as a contiguous array. This area may also be used to store arguments that are passed in registers if an address for such an argument must be generated. Generally, either 6 or 12 contiguous quadwords of stack storage will be allocated by the called procedure for this purpose (see Section 5.1.3, Homed Memory Argument List Structure for details).

A compiler may use the **stack temporary area** for fixed local variables, such as constant-sized data items and program state, as well as for dynamically sized local variables. The stack temporary area may also be used for dynamically sized items with a limited lifetime, for example, a dynamically sized function result or string concatenation that can't be directly stored in a target variable. When a procedure uses this area, the compiler must keep track of its base and reset SP to the base to reclaim storage used by temporaries.

The high-address end of the stack frame is defined by the value stored in FRAME_SIZE plus the contents of SP or FP, as indicated by BASE_REG_IS_FP. The high-address end is used to determine the value of SP for the predecessor procedure in the calling chain.

NOTE

Floating registers saved in the stack are stored as a 64-bit exact image of the register, i.e. no reordering of bits is done on the way to or from memory. Compilers must use a STT instruction to store the register regardless of floating point type.

RATIONALE

The above note is so that an unwind routine can properly restore the floating point registers without more complete type information.

4.1.3 Register Frame Procedure

A *register frame procedure* does not maintain a call frame on the stack and must therefore save its caller's context in registers. This type of procedure is sometimes referred to as a "lightweight procedure" referring to the relatively fast way of saving the call context.

Such a procedure cannot save and restore nonscratch registers. Because a procedure without a stack frame must therefore use scratch registers to maintain the caller's context, such a procedure cannot make a standard call to any other procedure.

A procedure with a register frame can have an exception handler and can handle exceptions in the normal way. Such a procedure can also allocate local stack storage in the normal way, although it will not necessarily do so.

NOTE

Lightweight procedures have more freedom than might be apparent. By use of appropriate agreements with callers of the lightweight procedure, with procedures that the lightweight procedure calls, and by the use of unwind handlers, a lightweight procedure may modify nonscratch registers, and may call other procedures.

Such agreements may be by convention (as in the case of language support routines in the RTL) or by interprocedural analysis. Calls employing such agreements are, however, *not* standard calls, and might not be fully supported by a debugger since, for instance, it might not be able to find the contents of the preserved registers.

Since such agreements must be permanent (for upwards compatibility of object code), lightweight procedures should in general follow the normal restrictions.

4.1.4 Null Frame Procedure

A *null frame procedure* is a particularly simple, special case of a register frame procedure, in which:

- The entry return address register is R26 (ENTRY_RA = 26).
- The return address is not saved in any other register (SAVE_RA = ENTRY_RA)
- No stack space is allocated (SP_SET = 0 and FRAME_SIZE = 0)
- As a result of the above, the prologue requires no instructions (ENTRY_LENGTH = 0)
- There is no associated exception handler (HANDLER_ADDRESS = 0).

This special case of a register frame procedure is of interest because such a procedure need not have an associated procedure descriptor (see Section 9.1, Procedure Descriptor Representation).

4.2 Transfer of control

A standard procedure call can use any sequence of instructions that presents the called routine with the required environment (see Section 1.4, Definitions). Although this is true it is believed that the vast majority of standard conforming external calls will be done with a common sequence of instructions and conventions. This common set of call conventions is so pervasive that it is included for reference as part of this standard.

One important feature of the calling standard is that it is designed such that the same instruction sequence can be used to call each of the different types of procedure. That is, the caller does not have to know which type of procedure is being called.

4.2.1 Call Conventions

The call conventions describe the rules and methods used to communicate certain information between the caller and the called procedure during invocation and return. For a standard call these conventions include the following:

- **Return Address**

  The calling procedure must pass to the called procedure the address to which control must be returned during a normal return from the called procedure. In most cases the return address is the address of the instruction following the one which transferred control to the called procedure. For a standard call the return address is passed in the return address register (R26).

- **Argument List**

  The *argument list* is an ordered set of zero or more *argument items*, that together comprise a logically contiguous structure known as an *argument item sequence*. This logically contiguous sequence is in practice mapped to registers and memory in a fashion that produces a physically discontiguous argument list. In a standard call, the first six items are passed in registers R16..R21 and/or registers F16..F21. (See Section 5.1.2, Normal Argument List Structure, for details of argument-to-register correspondence). The remaining items are collected in a memory argument list that is a naturally-aligned array of quadwords. In a standard call, this list (if present) must be passed at 0(SP).

- **Function Result**

  If a standard-conforming procedure is a function, and the function result is returned in a register, then the result is returned in R0, F0, or F0..F1. Otherwise, the function result is returned via the first argument item or dynamically as defined in Section 5.1.7, Returning Data.

- **Stack usage**

  SP must at all times denote an address that has octaword alignment. (This has the side effect that the in memory portion of the argument list, if any, will start on an octaword boundary.) During a procedure invocation the SP may never be set to a value that is higher than the value of the SP at entry to that procedure invocation.

  The contents of the stack located above the portion of the argument list which is passed in memory (if any) belong to the calling procedure and should therefore not be read or written by the called procedure, except as specified by indirect arguments or language-controlled up level references.

The stack pointer (SP) is used by the hardware in raising exceptions and asynchronous interrupts. It must be assumed that the contents of the stack below the current SP value and within the stack for the current thread are continually and unpredictably modified, as specified in the *Alpha AXP System Reference Manual* and as a result of asynchronous software actions.

4.2.2 Linkage

In a main (static) image, the GP register is initialized to point to the shared global storage region at the beginning of program execution. The GP register must not be modified by any procedure that is included in either the main image or a dynamic-link library.

Because there is no GP or other base pointer available to code that is part of a dynamic-link library image, addressing data that is not part of the same compilation generally requires using a LDAH/LDA sequence to dynamically create a suitable base address. Such code thereby becomes non-position independent (non-PIC). That is, it requires relocation if it is to be moved and executed at a different virtual address. The number of such relocations in the code stream can be minimized by loading a *linkage pointer* (a sort of "procedure local GP" value) near the beginning of a procedure's execution with the address of a *linkage section*, if needed, where the linkage section contains variables, addresses and/or literal data that is needed by the code. (This decreases the density of non-PIC references in the code stream.)

4.2.3 Link-Time Optimization

The design of this calling standard assumes that the system linker will assist in implementing a transfer of control to a target location that is too far away to reach with a BSR instruction.

A typical standard call looks like:

```
        BSR     FOO'
```

where FOO' is the displacement from the location of the BSR instruction to the entry point of the target procedure FOO. If the displacement is too large to be represented in the 21-bit displacement field of the BSR instruction, or if the displacement is not known at link-time, then the linker must intervene to create an *exit transfer vector* such as the following:

```
FOO":   LDAH    R28, FOO-hi(R31)
        LDA     R28, FOO-lo(R28)
        JMP     (R28)
```

The value of FOO' is then the displacement from the location of the BSR instruction to the FOO" entry in the exit transfer vector. The exit transfer vector loads the full absolute target address for procedure FOO and completes the transfer of control. Note that control does not return to the transfer vector; rather, it returns to the location following the BSR instruction.

NOTE

This design requires that the Windows NT for Alpha AXP operating environment manage the offsets in the LDAH/LDA instructions of the exit transfer vector whenever a dynamic-link library image is loaded and/or relocated.

4.2.4 Calling Computed Addresses

Most calls are made to a fixed address whose value is completely determined by the time the program starts execution. There are, however, certain cases which cause the exact address not to be determined until the code is actually executed. In this case the procedure value representing the procedure to be called will be computed in a register.

Suppose R4 contains such a computed procedure value (simple or bound). An example of the code to call the procedure that it describes is:

```
JSR     R26, (R4)
```

4.2.5 Bound Procedure Values

There are two distinct classes of procedures:

- Simple procedure
- Bound procedure

A *simple procedure* is a procedure that does not need direct access to the stack of its execution environment. A *bound procedure* is a procedure that does need direct access to the stack of its execution environment, typically to reference an up-level variable or to perform a non-local goto. Both a simple procedure and a bound procedure will have an associated procedure descriptor as described in previous sections. Bound procedure values are designed for multilanguage use and allow callers of procedures to use common code to call both bound and simple procedures.

When a bound procedure is called, the caller must pass some kind of pointer to the called code that allows it to reference its up-level environment; typically such a pointer is the frame pointer for that environment but many variations are possible. When the caller is itself executing within that outer environment then it can usually make such a call directly to the code for the nested procedure without recourse to any additional mechanism. However, when a procedure value for the nested procedure must be passed outside of that environment to a call site that has no knowledge of the target procedure, a special bound procedure is created so that the nested procedure can be called just like a simple procedure.

The procedure value of a bound procedure is defined as the address of the first instruction of a sequence of instructions that establish the proper environment for the bound procedure and then transfer control to that procedure.

One direct scheme for constructing a jacket to a bound procedure that can be called like a simple procedure is to allocate a sequence of instructions on the stack and use the address of those instructions as the procedure value. Assume that a bound procedure named PROC expects its static link to be passed in R1. Then, a suitable sequence of instructions might look like the following:

```
LDAH    R1,frame-hi(R31)      ;Create up-level pointer in R1
LDA     R1,frame-lo(R1)       ;
LDAH    R28,PROC-hi(R31)      ;Materialize address of bound procedure
LDA     R28,PROC-lo(R28)      ;
JMP     (R28)                 ;Transfer to the bound procedure
```

Note that this sequence can only be created by code that is executing within the context of the containing procedure so that the appropriate frame pointer value is known and can be

easily incorporated in the sequence illustrated. The lifetime of this sequence is, of course, limited to the lifetime of the stack frame in which it is allocated.

After creating the above jacket instructions, it is necessary to execute an IMB instruction prior to execution of them to assure instruction cache coherence, as described in the *Alpha AXP System Reference Manual.*

4.2.6 Entry and Exit Code Sequences

This section describes the steps that must be executed in procedure entry and exit sequences. These conventions must be followed in order for the call chain to be well defined at every point during thread execution.

Entry Code Sequence

Since the value of the PC defines the currently executing procedure, it must be possible using the procedure descriptor associated with that PC, as well as the instructions of the procedure itself, to recover the identity (PC) and environment of the caller. This leads to the following restriction: A standard call cannot be made from the prologue.

NOTE

If an exception is raised or an exception occurs in the prologue of a procedure, that procedure's exception handler (if any) will not be invoked since the procedure is not *current* yet. This implies that if a procedure has an exception handler compilers may not move code into the procedure prologue that might cause an exception that would be handled by that handler.

When a procedure is called, the code at the entry address must synchronize (as needed) any pending exceptions caused by instructions issued by the caller, save the caller's context, and make the called procedure current (by executing the last instruction of the procedure prologue).

This is done by performing the following actions in the order given:

1. If stack space is allocated (FRAME_SIZE $\neq$ 0), then set register SP = SP – FRAME_SIZE.

   After any necessary calculations and stack limit checks, this step must be completed by exactly one instruction that modifies SP.

2. For a stack frame procedure (REGISTER_FRAME is 0): Store the return address and any registers that are being preserved in the stack frame. The stack frame reference must use SP.

   NOTE

   If the locations allocated for saving the registers must be more than 32768 bytes away from the top of stack (for example, because the procedure contains an actual parameter list whose size exceeds 32768 bytes) then it is not possible to allocate the entire stack in a single step. In this case, the variable stack size form of stack frame must be used. (Recall that memory reference instructions in the Alpha AXP architecture can address at most 32768 bytes relative to the contents of some general register.)

3. Execute TRAPB if required (see Section 6.1.12, Other Considerations, for details).

4. For a variable sized stack frame procedure (BASE_REG_IS_FP is 1), copy register SP to register FP.

   This step must be completed by exactly one instruction that modifies FP.

When the above steps have been completed, the executing procedure is said to become *current* for the purposes of exception handling. The handler for a procedure will not be called except when that procedure is current.

For each of the steps described above, there are specific instructions or sequences of instructions that must be used. These are the only instructions that are interpreted when unwinding a stack frame for purposes of exception dispatching or stack walking. (Such prologue interpretation is sometimes called *reverse execution*. See Section 8.3, Getting a Procedure Invocation Context with a Routine for related discussion.)

1. Allocate stack

   Let N be the number of bytes to allocate for the fixed part of the stack. The simpler case occurs when the size of the extension does not exceed the value of MAX_NOCHK_EXTEND (4096, see Implicit Stack Limit Checking) and can therefore be represented in the 16-bit offset field of an LDA instruction. In this case, use

```
LDA    SP, -N(SP)
```

   If the previous case does not apply, then use the sequence

```
load   Rx, N
...
SUBQ   SP, Rx, SP
```

   where Rx is a scratch register for the procedure and *load* stands for one of a number of alternatives for loading the constant N into register Rx. The load of Rx and update of the stack pointer are separated by instructions that check that the stack limit is not exceeded by the allocation as described in Section 7.1, Stack Limit Checking.

   The following instructions may be used to load a constant value N into a register Rx:

```
BIS    R31, N, Rx        ; 0 ≤ N ≤ 255
```

or

```
LDA    Rx, N(R31)        ; 0 ≤ N ≤ 32767
```

or

```
LDAH   Rx, Hi(R31)       ; 0 ≤ Hi ≤ 32767, N = Hi*65536
```

or

```
LDAH   Rx, Hi(R31)       ; 0 ≤ Hi ≤ 32767,
LDA    Rx, Lo(Rx)        ; -32768 ≤ Lo ≤ 32767,
                         ; N = Hi*65536 + Lo
```

In the last case, the LDAH and LDA instructions need not be contiguous. [2]

2. Save return address and preserved registers

   General purpose registers that are saved on the stack must be saved using

```
STQ    Rx, n(SP)
```

   where Rx is the register being saved (including the return address register) and *n* is the offset in the stack for saving that register. Similarly, floating point registers that are saved on the stack must be saved using

[2] As a short term expedient, the instruction ADDQ R31, N, Rx is also interpreted during reverse execution.

```
STT     Fx, n(SP)
```

where Fx is the register being saved and *n* is the offset in the stack for saving that register. (Note that floating point registers are never saved using STF, STD, STG or STS because the called routine has no a priori information about the type of the variable contained in the register. Similarly, LDT is always used to restore a floating register.)

Preservation of registers in this manner is not limited to just the standard preserved registers defined in Section 3.3.1, Integer Registers and Section 3.3.2, Floating Point Registers.

A register that is preserved may also be saved by moving it to a scratch register in the case of a register frame procedure.

There are no requirements concerning the order in which registers are saved or the position within the stack frame where they are saved.

The first use of a register that is preserved within a prologue must be to save it. The save operation may consist of a sequence of moves that start with a given register, copy the contents to other registers, and optionally store the register in the stack. (This may be necessary, for example, to temporarily preserve the return address if a routine is called in the prologue to perform stack limit checking (see Section 7.1, Stack Limit Checking). Note that such a called routine is necessarily non-standard.) The last location in this sequence (whether a register or a stack location) must thereafter not be modified during the execution of the procedure. The registers involved in such a sequence (other than the last if the contents is not stored in the stack) can be used for other purposes provided, either

- they do not occur in any of the instructions described in this section, or
- they occur following the last instruction of the sequence.

3. Force pending exceptions

   The TRAPB instruction is used to assure that any pending exceptions occur prior to the end of the prologue (prior to the routine becoming current).

4. Initialize frame pointer register

   The frame pointer must be set using

   ```
   MOV     SP, FP
   ```

5. Move a register to another register

   A general register Rx must be moved (MOV) to another general register Ry using [3]

   ```
   BIS     R31, Rx, Ry
   ```

   A floating point register Fx must be moved to another general register Fy using

   ```
   CPYS    Fx, Fx, Fy
   ```

NOTE

The SP register is normally "saved" and "restored" by adjusting its contents by a fixed value that is part of the instruction stream rather than by copying its contents to another register or to a location on the stack. However, a STx SP,n(SP) or MOV SP,Rx instruction *is* reverse executed as just described in bullets 2 and 5 above.

[3] As a short term expedient, the instructions BIS Rx, Rx, Ry and BIS Rx, R31, Ry are also interpreted during reverse execution.

It follows that these instructions can be included in a procedure prologue only if their reverse execution is in fact intended as part of restoring the SP contents. In particular, code scheduling must not allow such instructions, whose use of the SP contents is unrelated to SP maintenance, to be moved into the prologue of a procedure.

Prologue Length As a general rule, it is valid to include instructions in the prologue in addition to those required above in order to take advantage of available processor cycles that are not otherwise used. However, to avoid undesirable reverse execution overhead during exception dispatching or unwinding the length of the prologue should not be increased in ways that do not exploit such available cycles except possibly to include instructions that have an unusually long latency. In any case, it must not be possible for such additional instructions to cause an exception that would be handled by that procedure if that exception were raised after the procedure became current. (An exception may be considered to be not handled by a procedure if it is known a priori that the handler will always resignal that exception.)

It follows from these considerations that the length of the prologue of a procedure should typically be no larger than two instructions times the number of registers that are preserved in that procedure, in addition to those instructions that perform stack limit checking (if any). In Windows NT for Alpha AXP, there must not be more than 1024 instructions in a prologue. If a larger number is detected, then the containing program may be aborted.

Frame Pointer Conventions

After procedure prologue completion, the register indicated by BASE_REG_IS_FP must contain the *frame pointer* of the stack frame, which is the address of the lowest-addressed byte of the fixed portion of the stack frame allocated by the procedure prologue. The value of the frame pointer is the value of FRAME_SIZE subtracted from the value of the stack pointer upon procedure entry.

For fixed frame procedures, the frame pointer is the stack pointer, which is not modified by that procedure after the instruction in that procedure prologue specified by SP_SET.

Entry Code Example for a Stack Frame

This example assumes that this is a stack frame procedure, that registers R9..R11 and F2..F3 are saved and restored, that the procedure has a static exception handler that does not reraise arithmetic traps, and that the procedure uses a fixed amount of stack (BASE_REG_IS_FP is 0).

```
LDA     SP,-SIZE(SP)    ;Allocate space for new stack frame
STQ     R26,16(SP)      ;Save return address
STQ     R9,24(SP)       ;Save first integer register
STQ     R10,32(SP)      ;Save next integer register
STQ     R11,40(SP)      ;Save next integer register
STT     F2,48(SP)       ;Save first floating point register
STT     F3,56(SP)       ;Save last floating point register
TRAPB                   ;Force any pending hardware exceptions to be raised
;Called procedure is now the current procedure
```

Entry Code Example for a Register Frame

The following entry code example is for a called procedure that has no static exception handler, both SAVE_RA and ENTRY_RA specify R26 and the procedure utilizes a fixed amount of stack storage (BASE_REG_IS_FP is 0).

```
LDA     SP,-SIZE(SP)        ;Allocate space for new stack frame
;Called procedure is now the current procedure
```

Exit Code Sequence

The end of procedure entry code can be determined easily given a PC value together with the ENTRY_LENGTH value. Since there may be multiple return points from a procedure, detecting that a procedure exit sequence is being executed is not quite as straight forward. Unwind support routines *must* be able to detect if the stack pointer has been reset or not and if not, how to reset it. This is done by using a reserved instruction sequence.

Reserved Instruction Sequence for Procedure Exit

To allow the stack to be properly restored during an unwind, a reserved instruction or sequence of instructions must be used. None of these sequences may be used in any other way.

The following reserved instruction **must** appear at every exit point from any procedure that uses stack (FRAME_SIZE $\neq$ 0):

```
RET     R31,(Rn),0001       ;Return to caller with usage hint 0001
```

NOTE

***Usage hint* refers to the value of the branch prediction bits encoded in the RET instruction. The Alpha AXP System Reference Manual documents that these bits, <13:0> of the instruction longword, are reserved to software when the instruction is *RET* or *JSR_COROUTINE*. (See Section 4.3 of the *Alpha AXP System Reference Manual*.) This calling standard further requires that these bits contain the value 0001 (hex) for procedure returns and 0000 otherwise. It is the occurrence of the usage hint value 0001 that identifies a RET instruction as one that is reserved for use only as described in this section. RET instructions may be used for other purposes provided they contain a usage hint of 0000. Such RET instructions will not be recognized and treated in a special way for the purposes of exception handling or unwinding.**

Furthermore, for any such procedure that does not return a value on the stack, the above instruction must be *directly* preceded by one of the two reserved stack resetting instructions as in:

```
LDA     SP,*                ;Reset stack
RET     R31,(*),0001        ;Return to caller with usage hint
```

or

```
ADDQ    *,*,SP              ;Reset stack
RET     R31,(*),0001        ;Return to caller with usage hint
```

That is, any LDA instruction whose destination is the SP register or any ADDQ instruction whose destination is the SP register is interpreted as part of a procedure exit sequence when immediately followed by the reserved procedure return instruction. The LDA instruction must have the form *LDA SP,n(Rx)* if the amount by which the stack is being adjusted can be represented as the offset n. Otherwise, the latter form is used, in which case the ADDQ instruction must have the form *ADDQ Rx, Ry, SP* where, in particular, Ry is the name of a register rather than a literal operand.

A stack resetting instruction may not be present (in the case of a procedure that returns a result on the top of the stack). However, if present, it will occur immediately preceding the reserved RET instruction.

Furthermore, for any such procedure that has BASE_REG_IS_FP set to 1, the resulting sequence must be directly preceded by the FP reloading instruction as in:

```
LDQ     FP,*                ;Restore FP
LDA     SP,*                ;Or ADDQ *,*,SP to Reset stack
RET     R31,(*),0001        ;Return to caller with usage hint
```

That is, any LDQ instruction whose destination is the FP register (R15) is interpreted as part of a procedure exit sequence when it is immediately followed by the reserved procedure return instruction or by a stack resetting instruction as described above that is immediately followed by the reserved procedure return instruction.

Procedures that do not use the stack need not use these reserved instruction sequences.

The unwind support code uses the above sequences to make the following assumptions about an interrupted PC value:

- If the PC points within the prologue then the effect of the prologue performed so far is undone via *reverse execution* starting at PC – 4 (see Section 8.3, Getting a Procedure Invocation Context with a Routine , and the unwind can proceed.
- If the PC points at a RET R31,(*),0001 instruction then SP has already been reset, the registers have already been restored and the unwind can proceed.
- If the PC points to either a LDA SP,* or an ADDQ *,*,SP instruction that is immediately followed by the instruction described previously then the registers have already been restored but the SP must be incremented by FRAME_SIZE before the unwind can proceed.
- If the PC points to a LDQ FP,* instruction that is immediately followed by either of the instructions described previously and REGISTER_FRAME is 0, then all registers other than FP have been restored, FP still retains the frame base pointer which should be copied to SP, then FP must be restored, SP must be incremented by FRAME_SIZE and the unwind can proceed.
- Otherwise, the registers must be restored, SP reset, and the unwind can proceed.

When a procedure has executed the first instruction of one of the reserved sequences just described, the procedure becomes no longer *current* for the purposes of exception handling. The handler for a procedure will not be called in the midst of one of these reserved instruction sequences in that procedure.

Exit Code Sequence Steps

When a procedure returns, the exit code must restore the caller's context, synchronize any pending hardware exceptions, and make the calling procedure current by returning control to it.

This is done by performing the following actions:

Perform step 1, followed by steps 2 - 5 in any order, followed by steps 6 - 8 in that order.

1. For a variable size stack frame procedure that does not return a value on the top-of-stack (BASE_REG_IS_FP is 1), copy FP to SP.

2. For a stack frame procedure (REGISTER_FRAME is 0), reload any saved registers. (For a fixed size stack frame procedure (BASE_REG_IS_FP is 0), R15 is reloaded if it was saved on entry.)
3. Reload the register that held the return address on entry with the saved return address if necessary. For a stack frame procedure (REGISTER_FRAME is 0), load the register designated by ENTRY_RA (R26 in a standard call) with the return address.
4. Execute TRAPB if required (see Section 6.1.12, Other Considerations, for details).
5. For a variable size stack frame procedure (REGISTER_FRAME is 0 and BASE_REG_IS_FP is 1), reload R15 (FP) as for any other saved register.

   After any necessary calculations, this step must be completed by exactly one instruction as described above.
6. If a function value is not being returned on the stack, then restore SP to the value it had at procedure entry by adding the value that was stored in FRAME_SIZE to SP. (In some cases the returning procedure will leave SP pointing to a lower stack address than it had on entry to the procedure, as specified in Section 5.1.7, Returning Data).

   After any necessary calculations, this step must be completed by exactly one instruction as described above.
7. Execute the RET R31,(Rn),0001 instruction as described above to return control to the calling procedure.

Note that the called routine does not adjust the stack to remove any arguments passed in memory. This responsibility falls to the calling routine which may choose to defer their removal due to optimizations or other considerations.

Exit Code Example for a Stack Frame

The following is the return code sequence for the stack frame example above.

```
LDQ     R26,16(SP)      ;Get return address
LDQ     R9,24(SP)       ;Restore first integer register
LDQ     R10,32(SP)      ;Restore next integer register
LDQ     R11,40(SP)      ;Restore next integer register
LDT     F2,48(SP)       ;Restore first floating point register
LDT     F3,56(SP)       ;Restore last floating point register
TRAPB                   ;Force any pending hardware exceptions to be
                        ; raised  (see Section 6.1.12)
LDA     SP,SIZE(SP)     ;Restore SP
RET     R31,(R26),0001  ;Return to caller with usage hint
```

Exit Code Example for a Register Frame

The following is the return code sequence for the register frame example above.

```
LDA     SP,SIZE(SP)     ;Restore the SP
RET     R31,(R26),0001  ;Return to caller with usage hint
```

CHAPTER 5

DATA MANIPULATION

This section deals with the passing and storage of data.

5.1 Data Passing

The fundamental unit of data which gets passed between procedures has been abstracted for purposes of discussion to a concept called an *argument item*. An argument item represents one unit of data being passed.

5.1.1 Argument Passing Mechanisms

This calling standard distinguishes three classes of argument items according to the mechanism used to pass the argument:

- Immediate value
- Reference
- Descriptor

Argument items are not self-defining; interpretation of each argument item depends on agreement between the calling and called procedures.

This standard does not dictate which of the above mechanisms must be used by a given language compiler. Language semantics and/or interoperability considerations may require different mechanisms to be used in any given situation.

Immediate Value

An *immediate value* argument item contains the value of the data item. The argument item, or the value contained in it, is to be directly associated with the parameter.

Reference

A *reference* argument item contains the address of a data item such as a scalar, string, array, record, or procedure. That data item is to be associated with the parameter.

Descriptor

A *descriptor* argument item contains the address of a descriptor, which contains structural information about the argument's type (such as array bounds) and the address of a data item. That data item is to be associated with the parameter.

This standard does not define a standard set of descriptors. Consequently, descriptors can not be used as part of a standard call.

5.1.2 Normal Argument List Structure

The *argument list* in an Alpha AXP call is an ordered set of zero or more argument items, which together comprise a *logically* contiguous structure known as the *argument item sequence*. An argument item is represented in 64 bits.

An argument item may be used to pass immediate arguments, arguments by reference, and arguments by descriptor. The standard permits any combination of these mechanisms in an argument list.

Although the argument items form a logically contiguous sequence, they are in practice mapped to integer and floating point registers and to memory in a fashion that may produce a physically discontiguous argument list. Registers R16..R21 and F16..F21 are used to pass the first six items of the argument item sequence. Additional argument items must be passed in a memory argument list which must be located at 0(SP) at the time of the call.

RATIONALE

The caller needs to have enough scratch registers to compute arguments. In addition, if the caller must make a call to compute some of the arguments, any that are in scratch registers must be saved to memory (but only once - so this is no worse than passing them in memory).

The following table specifies the standard locations in which argument items can be passed.

Table 5–1: Argument Item Locations

Item	Integer Registers	Floating Point Registers	Stack
1	R16	F16	
2	R17	F17	
3	R18	F18	
4	R19	F19	
5	R20	F20	
6	R21	F21	
7...n			0(SP)...(n-7)*8(SP)

The general rules that determine the location of any specific argument can be summarized as follows:

1. All argument items are passed in the integer registers or on the stack, *except* argument items that are floating point data passed by immediate value.
2. Floating point data passed by immediate value is passed in the floating point registers or on the stack.
3. Only *one* location in any row in the above table may be used by any given argument list. So, for example, if argument item 3 is an integer passed by value, and argument item 4 is a single precision floating point number passed by value, then argument item 3 is assigned to R18 and argument item 4 is assigned to F19.

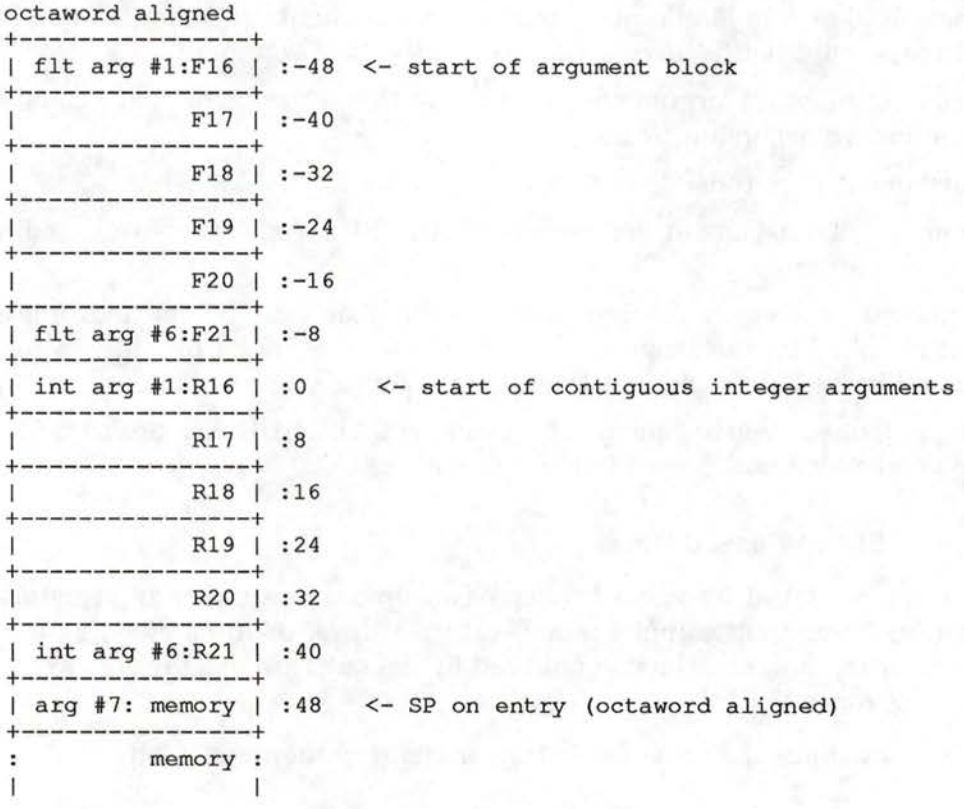
4. A single or double precision complex value passed by immediate value is treated as two arguments for the purpose of these rules, with the real part coming first. In particular, the real part of a complex value may happen to be passed as the sixth argument item in register F21 in which case the imaginary part will then be passed as the seventh argument item in memory.

The argument list, including both the in memory portion as well as the portion that is passed in registers, may be read and written by the called procedure. The calling procedure must therefore not make any assumptions about the validity of any part of the argument list after the completion of a call.

5.1.3 Homed Memory Argument List Structure

In certain cases (for example, for C procedures that use varargs), it is useful to form a contiguous in memory structure that includes the contents of all of the formal parameter values. In nearly all cases, a compiler can arrange to allocate and initialize this structure so that the parameter values that are passed in registers are placed adjacent to the parameters that are passed on the stack (without making a copy of the stack arguments). Storage for the parameters passed in registers is called the *argument home area* (see Figure 4–1 and Figure 4–2). The resulting in memory homed argument list structure is illustrated in Figure 5–1.

Figure 5–1: In Memory Homed Argument List Structure



Because it is generally not possible to tell statically whether a particular argument is an integer or floating point argument, it is necessary to store both integer and floating point register argument contents in this structure. However, it is sometimes possible to determine statically that there are no floating arguments anywhere at all (either in registers or on the stack), in which case the first six entries can be omitted. To facilitate this special case, the address used to reference this structure is always the address of the first integer argument position.

The C language type va_list is defined as follows:

```
typedef longlong quad;
typedef struct {
    quad     *base;
    int      offset;
    } va_list;
```

To load the next integer argument, read the quadword at location (*base+offset*), and add 8 to *offset*. To load the next floating argument, if *offset* $\leq$ 6*8, read the quadword at location (*base+offset*-6*8); otherwise, read the quadword at location (*base+offset*); in either case add 8 to *offset*.

5.1.4 Argument Lists and High Level Languages

High level language functional notations for procedure call arguments are mapped into argument item sequences according to the following rules:

1. Arguments are mapped from left to right to increasing offsets in the argument item sequence. R16 or F16 is allocated to the first argument, and the last quadword of the memory argument list (if any) is allocated to the last argument.
2. Each source language argument corresponds to one or more contiguous Alpha AXP calling standard argument items.
3. Each argument item consists of 64 bits.
4. A null or omitted argument, for example CALL SUB(A,,B), is represented by an argument item containing 0.

   No arguments passed by the immediate mechanism may be omitted unless a default value is supplied by the language. (This is to enable called procedures to distinguish an omitted immediate argument from an immediate value argument with the value 0.)

   Trailing null or omitted arguments, for example CALL SUB(A,,), are passed by the same rules as embedded null or omitted arguments.

5.1.5 Unused Bits in Passed Data

Whenever data is passed by value between two procedures either in registers, as is the case for the first six input arguments and return values, or in memory, as is the case for arguments after the first six, the bits not used by the data are usually sign extended or zero extended as appropriate.

The table below defines the rules for setting or clearing the unused bits.

Key:

- *Sign32* means sign extended to 32 bits - The state of bits <63:32> is unpredictable

- *Sign64* means sign extended to 64 bits
- *Zero32* means zero extended to 32 bits - The state of bits <63:32> is unpredictable
- *Zero64* means zero extended to 64 bits
- *Data32* means data is 32 bits - The state of bits <63:32> is unpredictable
- *2*Data32* means two single precision parts of the complex value are stored in memory as independent floating point values (each handled as *Data32*)
- *Data64* means data is 64 bits
- *2*Data64* means two double precision parts of the complex value are stored in memory as independent floating point values (each handled as *Data64*)
- *Hard* means passed in the layout defined by the *Alpha AXP System Reference Manual*
- *2*Hard* means two double precision parts of the complex value are stored in a pair of registers as independent floating point values (each handled as *Hard*)
- *Nostd* means that the state of all high order bits not occupied by the data is unpredictable across a call or return

Table 5–2: Unused Bits in Passed Data

Data Type	Type Designator	Data Size (bytes)	Register Extension Type	Memory Extension Type
byte logical	BU	1	Zero64	Zero64
word logical	WU	2	Zero64	Zero64
longword logical	LU	4	Sign64	Sign64
quadword logical	QU	8	Data64	Data64
byte integer	B	1	Sign64	Sign64
word integer	W	2	Sign64	Sign64
longword integer	L	4	Sign64	Sign64
quadword integer	Q	8	Data64	Data64
F floating	F	4	Hard	Data32
D floating	D	8	Hard	Data64
G floating	G	8	Hard	Data64
F floating complex	FC	2 * 4	2*Hard	2*Data32
D floating complex	DC	2 * 8	2*Hard	2*Data64
G floating complex	GC	2 * 8	2*Hard	2*Data64
IEEE floating single S	FS	4	Hard	Data32
IEEE floating double T	FT	8	Hard	Data64
IEEE floating single S complex	FSC	2 * 4	2*Hard	2*Data32
IEEE floating double T complex	FTC	2 * 8	2*Hard	2*Data64
Structures	N/A		Nostd	Nostd
Small arrays of 8 bytes or less	N/A	≤8	Nostd	Nostd
32-bit address	N/A	4	Sign64	Sign64
64-bit address	N/A	8	Data64	Data64

NOTE

***Sign64* applied to a longword logical duplicates bit 31 through bits <63:32>, which may cause the 64-bit integer value to appear negative. However, careful use of 32-bit arithmetic and 64-bit logical instructions (but no right shifts) will preserve the 32-bit unsigned nature of the argument.**

Because of the varied rules for sign extension of data when passed as arguments it is important that both calling and called routines must agree on the datatype of each argument. No implicit data type conversions can be assumed between the calling procedure and the called procedure.

5.1.6 Sending Data

Sending Mechanism

The following represents the rules which govern the allowable mechanisms for sending data.

By immediate value An argument may be passed by immediate value only if the argument is one of the following:

- One of the noncomplex scalar data types with a size known (at compile time) to be $\leq$ 64 bits
- A record with a known size (at compile time)
- A set, implemented as a bit vector, with a size known (at compile time) to be $\leq$ 64 bits

No form of string, array or complex data type may be passed by immediate value in a standard call.

A standard immediate argument item must fill 64 bits. This means that unused high-order bits of all data types (excluding records) must be zero-extended or sign-extended, as appropriate depending on the data type, to fill all unused bits. (See Table 5–2, Unused Bits in Passed Data, for details.)

Large Immediate Arguments Record values are passed by immediate value as follows:

- Allocate as many fully occupied argument item positions to the argument value as are needed to represent the argument.
- The value of the unoccupied bits is undefined in a final, partially occupied argument item position.
- An argument item that is passed in one of the registers is passed in the integer registers (never in a floating register).

Non-Standard Immediate Arguments Non-record argument values that are larger than 64 bits can be passed by immediate value using non-standard conventions, typically using a method similar to that used for records. Thus, for example, a 26-byte string could be passed by value in four integer registers.

By Reference—Non-parametric Non-parametric arguments (that is, arguments for which associated information such as string size and array bounds are not required) may be passed by reference in a standard call.

By Reference—Parametric Parametric arguments (that is, arguments for which associated information such as string size and array bounds must be passed to the caller) may be passed by reference followed by one or more immediate arguments for the parametric values. (The parametric values need not immediately follow the reference arguments to which they apply.)

NOTE

Interlanguage conventions for such calls are not defined by this standard.

By Descriptor Parametric arguments (that is, arguments for which associated information such as string size and array bounds must be passed to the caller) may be passed by a single descriptor.

NOTE

A standard set of descriptors for interlanguage use is not defined by this standard.

Order of Argument Evaluation

Since most higher level languages do not specify the order of evaluation (with respect to side effects) of arguments, those language processors can evaluate arguments in any convenient order. The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. Programs should not be written that depend on the order of evaluation of arguments.

5.1.7 Returning Data

A standard function must return its function value by one of the following mechanisms:

- immediate value
- reference
- descriptor

These mechanisms are the only standard means available for returning function values, and they support the important language independent data types. Functions that return values by any mechanism other than those specified here are non-standard, language-specific functions.

Function Value Return By Immediate Value

This section describes the two types of immediate value function return.

Non-Floating Function Value Return By Immediate Value

A function value is returned by immediate value in register R0 *only* if the type of function value is one of the following:

- Non-floating point scalar data type with size known (at compile time) to be $\leq$ 64 bits
- Record with size known (at compile time) to be $\leq$ 64 bits
- Set, implemented as a bit vector, with size known (at compile time) to be $\leq$ 64 bits

No form of string or array may be returned by immediate value.

Two separate 32-bit entities cannot be returned in R0.

A function value < 64 bits returned in R0 must have its unoccupied bits extended as appropriate, to a full quadword, depending on the data type. (See Table 5–2, Unused Bits in Passed Data for more details).

Floating Function Value Return By Immediate Value

A function value is returned by *immediate value* in register F0 if, and *only* if, it is a non-complex single or double precision floating point value (F,D,G,S,or T).

A function value is returned by *immediate value* in registers F0..F1 if, and *only* if, it is a complex single or double precision floating point value (complex F,D,G,S,or T). The real part is in F0 and the imaginary part is in F1.

Function Value Return By Reference

A function value is returned by reference *only* if the function value satisfies both of the following criteria:

- Its size is known to both the calling procedure and the called procedure, but the value cannot be returned by immediate value (because the function value requires more than 64 bits, the data type is a string or an array type, and so on).
- It can be returned in a contiguous region of storage.

The actual-argument list and the formal-argument list are shifted to the right by one argument item. The new, first argument item is reserved for the address of the function value.

The calling procedure must provide the required contiguous storage and pass the address of the storage as the first argument. This address *must* specify storage that is naturally aligned according to the data type of the function value.

The called function must write the function value to the storage described by the first argument.

Function Value Return By Descriptor

A function value is returned by descriptor *only* if the function value satisfies all of the following criteria:

- It cannot be returned by immediate value (because the function value requires more than 64 bits, the data type is a string or an array type, and so on).
- Its size is not known to either the calling procedure or the called procedure.
- It can be returned in a contiguous region of storage.

Function results may *not* be returned by descriptor in a standard call.

Typically, the called routine creates the return object on its stack and leaves it there on return. This is referred to as the *stack return* mechanism. The exit code of the called routine does not restore SP to its value before the call (otherwise the return value would be left unprotected in memory below SP). The calling routine must be prepared for SP to have a different value after the call.

5.2 Data Allocation

5.2.1 Alignment

On the Alpha AXP architecture, memory references to data that is not naturally aligned may result in alignment faults, which can *severely* degrade the performance of all procedures that reference the unnaturally aligned data.

For this reason, data values on Alpha AXP systems (including variables, aguments, function results and so on) should be naturally aligned. For example, 8-bit character strings should start on byte boundaries; 16-bit integers should start at addresses that are a multiple of at least 2 (word alignment); single precision real values should start at addresses that are a multiple of at least 4 (longword alignment); double precision real values should start at addresses that are a multiple of at least 8 (quadword alignment); and so forth. Single precision complex values should start at addresses that are a multiple of 4 (longword alignment) and double precision complex values should start at addresses that are a multiple of 8 (quadword alignment).

Data types larger than 64 bits should use quadword or greater alignment. Alignments larger than quadword are language-specific or application defined.

For aggregates such as strings, arrays, and records, the data type to be considered for purposes of alignment is *not* the aggregate itself, but the elements of which the aggregate is composed. The alignment requirement of an aggregate is that all elements of the aggregate should be naturally aligned. Varying 8-bit character strings must, for example, start at addresses that are a multiple of at least 2 (word alignment) because of the 16-bit count at the beginning of the string; 32-bit integer arrays start at a longword boundary, irrespective of the extent of the array.

5.2.2 Granularity

On the Alpha AXP architecture, memory is byte addressable but the smallest unit in which memory may be accessed (the *granularity*) is a longword. Moreover, even for longword sized data it is often expedient for execution efficiency to access memory in quadword units. In the presence of multiple threads of execution (whether on multiple processors or just one processor), allocation of more than one data element within a single quadword can lead to more complicated access sequences (for example, using LDx_L/STx_C) and/or latent and hard to diagnose bugs because of non-obvious and implicit data sharing. As a result, it is generally recommended that independent variables (that is, variables not combined in a larger aggregate) should be allocated on quadword boundaries.

5.2.3 Record Layout Conventions

The Alpha AXP calling standard record layout rules are designed to provide good run time performance on all implementations of the Alpha AXP architecture.

Only the standard record layouts may be used across standard interfaces or between languages. Languages may support other language-specific record layout conventions, but such other record layouts are not standard.

Aligned Record Layout

The aligned record layout conventions ensure that:

- All components of a record or subrecord are naturally aligned.
- The layout and alignment of record elements and subrecords is independent of any record or subrecord in which they may be embedded.
- The layout and alignment of a subrecord is the same as if it were a top level record.
- Declaration in high level languages of standard records for interlanguage use is reasonably straightforward and obvious, and meets the requirements for source level compatibility between Alpha AXP environments and other environments.

The aligned record layout is defined by the following conventions:

- The components of a record must be laid out in memory corresponding to the lexical order of their appearance in the high level language declaration of the record.
- The first bit of a record or subrecord must be directly addressable; i.e. it must be byte aligned.
- Records and subrecords must be aligned according to the largest natural alignment requirements of the contained elements and subrecords.
- Bitfields (packed subranges of integers) are characterized by an underlying integer type which is a byte, word, longword or quadword in size together with an allocation size in bits. A bitfield is allocated at the next available bit boundary provided that the resulting allocation does not cross an alignment boundary of the underlying type. Otherwise, the field is allocated at the next byte boundary that is aligned as required for the underlying type. (In the latter case, the space skipped over is left permanently not allocated.) In addition, the alignment of the record as a whole is increased to that of the underlying integer type (if needed).
- Unaligned bit strings, unaligned bit arrays, and elements of unaligned bit arrays must start at the next available bit in the record; no fill is ever supplied preceding an unaligned bit string, unaligned bit array, or unaligned bit array element.
- All other components of a record must start at the next available naturally aligned address for the data type.
- Strings and arrays must be aligned according to the natural alignment requirements of the data type of which the string or array is composed.
- The length of an array element is a multiple of its alignment, even if this leaves unused space at its end. The length of the whole array is the sum of the lengths of its elements.

CHAPTER 6

EVENT PROCESSING

This chapter discusses specifications related to events that are outside the normal program flow.

6.1 Exception Handling

This section on exception handling discusses the considerations involved in the notification and handling of exceptional events during the course of normal program execution.

6.1.1 Exception Handling Requirements

The exception handling capabilities specified in this standard are for support of:

- Reliable programmer and program control over response to exceptions and reporting of such exceptions, and over the flow of control when exceptions occur.
- Orderly termination of layered applications.
- Correct and predictable exception handling in a multilanguage environment
- The construction of modular, maintainable multilanguage applications.
- Support for parallel multithreaded application execution, including
  - Per-thread exception handling.
  - Handling of asynchronous exceptions.
  - Safe thread exit in a multithreaded environment.
- Capability for subsystems and applications to override system messages to provide a more suitable application oriented interface, specifically including modular, multinational message and error reporting.

6.1.2 Exception Handling Overview

When an exception occurs (is *raised*), the normal flow of control in the current thread is interrupted, the context is saved, and control is transferred to the exception handling support code. This support code marshals the exception information and then enters a section of the support code called the *exception dispatcher*. The exception dispatcher searches for exception handlers and invokes them in the proper sequence.

When a handler is invoked, it is called as a procedure with arguments that describe the nature of the exception, the environment within which the exception was raised, and the environment within which the handler was established. When the handler is called the exception is said to be *delivered* to the handler.

The handler may respond to the exception in several ways, including various combinations of the following:

- Perform some action that affects the context of the thread (possibly correcting the circumstances that led to the exception being raised).
- Modify or augment the description of the exception.
- Raise a *nested exception*, causing another exception to occur in the context of the exception handler or in a procedure called directly or indirectly by the handler.

When an exception handler has finished processing an exception, it must indicate this in one of the following ways:

- Indicate that the exception handling support code should *Reraise* the exception and resume the search for another handler.
- Indicate that the exception handling support code should *Continue* execution of the interrupted thread at the location indicated by the saved exception program counter.
- *Unwind*, which causes the exception handling support code to resume execution of the thread at a point different than the point at which it was interrupted, or terminate the execution of the thread.

All exceptions are handled with the same interfaces, data structures, and algorithms. That is, there is unified exception handling for all kinds of exceptions, regardless of their origination.

Each exception has an *exception value*, which identifies the exception (such as subscript range violation, or memory access control violation). Exceptions may also have associated with them one or more *exception qualifiers* (such as the name of an array and the subscript which was out of range, or an address associated with a memory access control violation).

6.1.3 Kinds of Exceptions

Exceptions may be divided into these kinds:

- Those caused by general software or hardware notification mechanisms (called *general exceptions*).
- Those caused by an unwind operation (called *unwind exceptions*).

General Exceptions

General exceptions may be further categorized as one of:

Software Caused

A software caused general exception is raised as the result of the invocation of an exception raising procedure and is always delivered to the thread which made the call.

Such an exception may be raised at any point during thread execution. Applications and language run time libraries may raise general exceptions to notify a thread of some exceptional (noteworthy) state in the current thread context. For example, subscript range checking failures and assertion checking failures may be raised as general exceptions.

Hardware Caused

A *hardware exception* occurs when a thread performs some action which causes an exceptional state to exist in the hardware. Such a state will cause the currently active thread to be interrupted. A hardware caused general exception is always delivered to the thread that executed the instruction which caused the exception.

Exactly which hardware events can result in an exception, the state of the machine when a hardware exception occurs, the interpretation of the exception-related information which is delivered to a user mode thread, and circumstances under which execution can be continued are specific to individual hardware exceptions. Hardware exceptions are fully defined by the *Alpha AXP System Reference Manual* which should be consulted for additional information.

In the Windows NT for Alpha AXP environment, hardware exceptions (that are not handled by the operating system itself) are reported to user level in the form of a general exception.

Unwind Exceptions

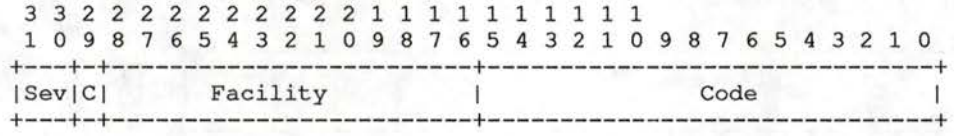
An *unwind exception* results from the invocation of the unwind support code by a thread, and is always delivered to the thread which invoked the unwind.

Unwind exceptions are delivered as part of the notification process that an unwind is in progress (see Section 6.2, Unwinding, for details).

6.1.4 Status Values and Exception Codes

A status value can be used as a return value from a procedure call to indicate success, failure and/or other information about the requested operation. A status value can also be used as an exception code to indicate the reason that an exception is being raised.
A status value is represented as shown in Figure 6–1.

Figure 6–1: Status Value Representation



The components of this representation are as follows:

Sev is a severity code, which can hold the following values.

Bit Encoding	Meaning
00	Success
01	Informational
10	Warning
11	Error

C is a flag that indicates that this status value is customer defined.

Facility is a facility code that indicates the software component that defines this status value.

Code is an identifier value for a particular status condition.

6.1.5 Exception Records

The fundamental data structure for describing exceptions is the *exception record.*

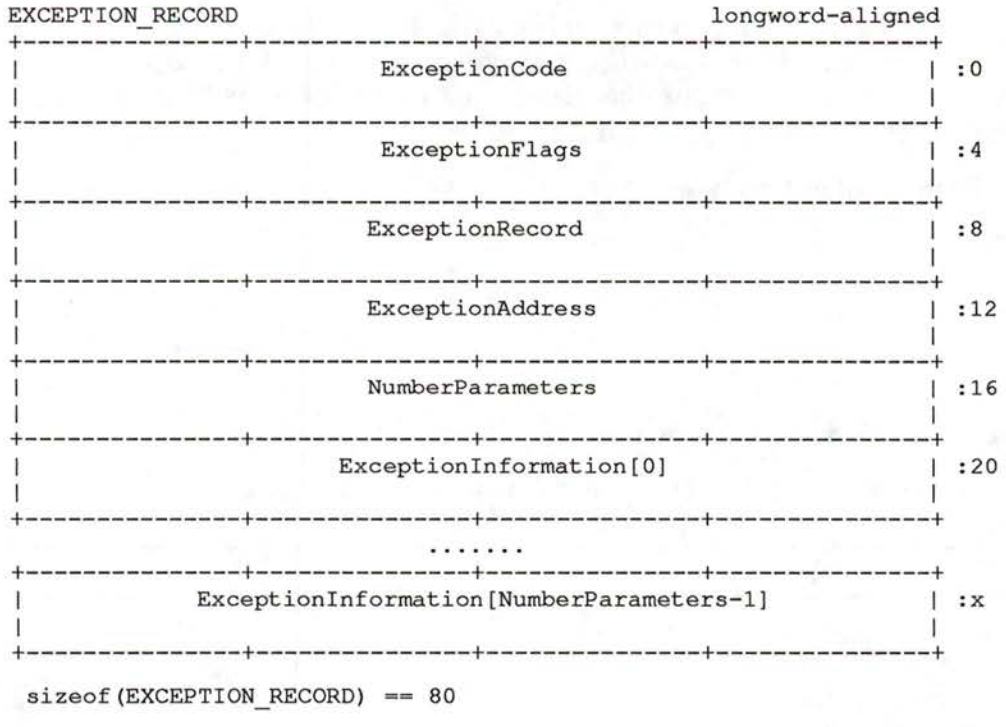
Exception records can form a linked list. Each record in a list describes one exception.

The first exception record in the list describes the *primary exception*. Additional *secondary exceptions* may be specified by additional exception records in the list. Secondary exceptions qualify or elaborate the primary exception; they may be raised at the same time as the primary exception, or a handler may add new secondary exceptions to the list before handling or reraising the exception.

Storage for exception records may be allocated in read-only memory. The exception record that is passed to a handler is a separate copy constructed from information in the original exception record augmented with additional information.

Exception records are defined as shown in Figure 6–2, Exception Record Format.

Figure 6–2: Exception Record Format



ExceptionCode is a status value (see Section 6.1.4, Status Values and Exception Codes).

ExceptionFlags is a bitfield of flags which further qualify the exception. These flag bits are significant only in the primary exception record; their state is *unpredictable* in secondary exception records. ExceptionFlags bits are logically divided into two groups. The first group (called detail flags) provide additional information about the exception. The second group (called environment flags) provide additional information about the environment in which the exception is being delivered.

Valid ExceptionFlags bits that give additional detail are:

- EXCEPTION_NONCONTINUABLE

  If EXCEPTION_NONCONTINUABLE is 1, an exception handler must not return ExceptionContinueExecution.

- EXCEPTION_EXIT_UNWIND

  If EXCEPTION_EXIT_UNWIND is 1, the exception handler is being invoked because of an unwind operation that will terminate execution of the thread.

- EXCEPTION_UNWINDING

  If EXCEPTION_UNWINDING is 1, the exception handler is being invoked because of a general unwind operation (with the semantics of *longjmp()*).

Valid ExceptionFlags bits that give additional information about the environment at the time of exception delivery are:

- EXCEPTION_NESTED_CALL

  If EXCEPTION_NESTED_CALL is 1, an exception or unwind is in progress at the time this exception is delivered.

- EXCEPTION_STACK_INVALID

  If EXCEPTION_STACK_INVALID is 1, the stack is invalid.

  NOTE

  This flag is for use by system software, and will never be 1 in an exception record delivered to a normal handler.

- EXCEPTION_TARGET_UNWIND

  If EXCEPTION_TARGET_UNWIND is 1, this is the target frame of an unwind operation. (This may be useful to allow a language specific handler to perform proper handling for the last step of an unwind.)

- EXCEPTION_COLLIDED_UNWIND

  If EXCEPTION_COLLIDED_UNWIND is 1, an unwind collision has occurred (see Section 6.2.5, Multiply Active Unwind Operations).

All FLAGS bits other than those defined above must be zero.

ExceptionRecord is either zero or is the address of the next exception record in the list.

ExceptionAddress is the address of the instruction causing the exception.

For a hardware exception or asynchronous software exception, this is the address of the instruction at which the hardware exception or asynchronous exception, interrupted execution of the thread.

For synchronous software exceptions and unwind exceptions, this is the address of the call instruction that invoked the library routine for raising the exception or starting the unwind, respectively.

This field is significant only in the primary exception record; its contents are *unpredictable* in secondary exception records.

NumberParameters is the number of exception-specific qualifiers in the exception record.

Each **ExceptionInformation[n]** value is a single longword which provides additional information specific to the exception, and may contain information intended for display in messages.

Exception Records for General Exceptions

Software Caused Exceptions The information in exception records for general and unwind exceptions may vary widely from a simple single exception value to a long chain of exceptions and exception qualifiers. This specification defines the conventions for constructing these exception records. A complete enumeration of all possible combinations is beyond the scope of this document.

Hardware Caused Exceptions All Alpha AXP hardware exceptions have exception information associated with them. This information may be as little as the exception type and exception PC or as much as three additional registers worth of information. The specific information that is supplied with each exception type is defined by the *Alpha AXP System Reference Manual.*

When a hardware exception occurs, the Windows NT for Alpha AXP operating system may handle the exception internally or it may cause the exception to be raised in user mode. When a hardware exception is raised in user mode, the exception record passed to exception handlers includes the exception information supplied by the operating system. This information is marshaled into the exception record to produce an exception record, typically as follows:

ExceptionCode is an operating system defined code unique to the exception condition.

ExceptionFlags is as appropriate (see above).

ExceptionRecord is 0 (null).

ExceptionAddress is the PC of the instruction that caused the hardware exception.

NumberParameters and each **ExceptionInformation[n]** contain additional information specific to the exception condition.

Details on the treatment of particular hardware exceptions are system defined.

Exception Records for Unwind Exceptions

Unwind exceptions are characterized by having at least one of the EXCEPTION_UNWINDING or EXCEPTION_EXIT_UNWIND or EXCEPTION_TARGET_UNWIND flags set to 1.

The reason code for the unwind as well as any supplied qualifiers is represented in the ExceptionCode in the same way as for general exceptions.

6.1.6 Exception Handlers

A frame-based handler is established when a procedure whose descriptor specifies an exception handler becomes current. Thus, frame-based handlers are usually associated with a procedure at compile time, and are located at run time via the procedure descriptor. These exception handlers are normally used to implement a particular language's exception handling semantics.

The frame-based handlers which may be invoked are those established by active procedures, from the most current procedure to the oldest predecessor.

Handling Exceptions

An exception handler that conforms to this standard generally should not handle any exception that its establisher did not cause unless there is a prior agreement between the writers of the exception handler and the writers of the code that raised the exception.

Access to Memory

Exceptions can be raised and unwind operations (which cause exception handlers to be called) can occur when the current value of one or more variables is in registers rather than in memory. Because of this, a handler, and any descendant procedure called directly or indirectly by a handler, must not access any variables except those explicitly passed to the procedure as arguments or those that exist in the normal scope of the procedure.

This rule can be violated for specific memory locations only by agreement between the handler and all procedures which might access those memory locations. The effects of such agreements is not specified by this standard.

6.1.7 Establishing Handlers

The list of established frame-based handlers for a thread is defined by the thread's procedure invocation chain (see Chapter 8, Procedure Invocations and Call Chains).

A procedure descriptor for which HANDLER_VALID is 1 must specify in HANDLER_ADDRESS the procedure value of an exception handler. The exception handler specified by a procedure descriptor is established when that descriptor is added to the invocation chain (that is, when the procedure designated by the descriptor becomes current), remains established as long as that procedure invocation is part of the invocation chain, and is revoked when that descriptor is removed from the invocation chain (that is, when the procedure invocation designated by the descriptor terminates, either by returning or being unwound).

Thus, the set of frame-based handlers which is established at any moment is defined by the current procedure call chain.

Dynamic activation and deactivation of exception handlers is not defined by this calling standard (and in fact not permitted within the semantics of many language standards). If this capability is required it must be defined on a language by language basis. Compilers which choose to support this functionality may set up language-specific static exception handlers that provide the dynamic exception handling semantics of that language. These static handlers would be established by means of the procedure descriptor of the establishing procedure. If a language compiler decides to support dynamic activation of exception handlers it must be prepared to recognize code that intends to use this feature. This requirement stems from the need to add appropriate TRAPB instructions and other compile time considerations needed to make dynamic exception handling function correctly.

NOTE

There may be additional protocols and conventions for dynamic exception handling. These may be needed, for example, to enable a debugger to do a good job within the language exception handling environment. These conventions are driven by the requirements of the languages and the language support utilities, and are not addressed by this calling standard.

6.1.8 Raising Exceptions

Raising General Exceptions

A thread may raise a general exception in its own context by calling a system library routine defined as follows:

RtlRaiseException (ExceptionRecord)

Arguments:

ExceptionRecord	The address of a primary exception record.

Function Value:

None.

RtlRaiseException() sets ExceptionAddress to the address of the invoking call instruction.

If *RtlRaiseException()* detects that the exception record passed via the first argument is not a valid exception record, it raises the exception STATUS_INVALID_EXCEPTION.

Raising General Exceptions Using GENTRAP

The *Alpha AXP System Reference Manual* defines a GENTRAP PALcall which provides a means for software to raise hardware-like exceptions at minimum cost.

This mechanism is suitable for use in low levels of the operating system or during bootstrapping when only a limited execution environment may be available. In a constrained environment, the GENTRAP can be handled directly via the SCB Vector by which the trap is reported. In a more complete environment, the GENTRAP parameter is transformed into a corresponding exception code and reported as a normal hardware exception. Because of this, low level software can use this mechanism to report exceptions in a way that is independent of the execution environment. Compiled code may also use this means to raise common generic exceptions more cheaply than making a full procedure call to *RtlRaiseException*.

The PALcall is defined as follows:

GENTRAP (EXPT_CODE)

Argument:

EXPT_CODE	Code for the exception to be raised.

If the EXPT_CODE value is one of the small negative values shown in the following table, then that value is mapped to a corresponding Windows NT for Alpha AXP exception code as shown. Otherwise, STATUS_ALPHA_GENTRAP is raised with the unmapped value included in the exception record as the (first and only) qualifier value. Note that there is no means to associate any parameters with an exception raised using GENTRAP.

GENTRAP sets ExceptionAddress to the address of the PALcall instruction.

Code	Symbol	Status Code
-1	GENTRAP_INTEGER_OVERFLOW	STATUS_INTEGER_OVERFLOW
-2	GENTRAP_INTEGER_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO
-3	GENTRAP_FLOATING_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO
-4	GENTRAP_FLOATING_OVERFLOW	STATUS_FLOAT_OVERFLOW
-5	GENTRAP_FLOATING_UNDERFLOW	STATUS_FLOAT_UNDERFLOW
-6	GENTRAP_FLOATING_INVALID_OPERAND	STATUS_FLOAT_INVALID_OPERATION
-7	GENTRAP_FLOATING_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT
-14		STATUS_STACK_OVERFLOW
-18		STATUS_ARRAY_BOUNDS_EXCEEDED

Raising Unwind Exceptions

The mechanism used to raise an unwind exception is described in Section 6.2, Unwinding.

6.1.9 Invocation of Exception Handlers

The search for an exception handler begins with the program counter value that indicates the address at which the exception was raised. As a general rule, a program counter value is associated with a procedure descriptor (Section 9.1, Procedure Descriptor Representation) which provides information needed to identify the procedure that contains the code and to interpret those parts of the stack frame that are needed to walk the procedure call chain. (See also Chapter 8, Procedure Invocations and Call Chains.)

If a null frame procedure or other fragment of code does not have an associated procedure descriptor, then it is assumed that an appropriate initial program counter value is located in the normal return address register (R26). If there is a procedure descriptor associated with this address, then the search for an exception handler begins using that address. Otherwise, a fatal exception is raised (and the executing thread is terminated).

Order of Invocation

When an exception is raised, established exception handlers are invoked in a specific order.

Any frame-based handlers are invoked in order from that established by the most current procedure invocation to the oldest predecessor in the invocation chain.

If no frame-based handlers have been established, or if all reraise the exception, then the system last chance handler is invoked.

Nested Exceptions A nested exception occurs if an exception is raised while an exception handler is active.

When a nested exception occurs, the structure of the procedure invocation chain, from the most recent procedure invocation to the oldest predecessor, is as follows:

1. The procedure invocation within which the nested exception was raised.
2. Zero or more procedures invoked indirectly or directly by the most recently invoked (most current) handler.
3. The most current handler.

   This is the same invocation as item 1 (that in which the nested exception was raised) if there are zero invocations in item 2. If this case, items 1 and 3 count as just one invocation.
4. The procedure invocation within which the active exception that immediately preceded the nested exception was raised; that is, the invocation in which the exception was raised for which the most current handler was invoked.
5. Zero or more procedure invocations, all established handlers which were invoked for the exception that immediately preceded the nested exception, and all of which reraised.
6. The establisher of the most current handler.

   This is the same as item 4 (the invocation in which the exception that immediately preceded the nested exception was raised) if there are zero invocations in item 5.
7. Zero or more procedure invocations for which no established handlers have yet been invoked.

Established handlers are invoked in reverse order with respect to that in which their establishers were invoked; that is, the search of stack frames for procedure invocations which have established handlers is in order from 1 to 7.

If further nested exceptions occur, this procedure invocation chain structure is repeated for those further nested exceptions, and frame-based handlers are invoked according to the above rules, in order from those established by the most current procedure to those established by the oldest predecessor.

Steps for Locating and Invoking Handlers for Exceptions When an exception is raised, the steps that implement the above explanation are detailed in the following. (Note that these steps cover only the search of stack frames for a handler proper and do not address the mapping of a POSIX signal to an exception.)

- Let *current_invocation* be the procedure invocation in which the exception was raised.
- [loop]: If *current_invocation* does not establish a handler, go to step [check-begin] below.
- Invoke the handler established by *current_invocation*.
- If the handler returns ExceptionContinueExecution or initiates an unwind, exit these steps.
- [check-begin]: If *current_invocation* is the beginning of the procedure invocation chain, go to step [last-chance] below.

- If *current_invocation* is an active handler, let *current_invocation* be the invocation in which the exception was raised that invoked this active handler, and go to step [loop] above.
- Let *current_invocation* be the procedure invocation which invoked *current_invocation*.
- Go to step [loop] above.
- [last-chance]: Invoke the system last chance handler.

Invalid Thread Stack If, during the search for and invocation of frame-based handlers, the exception dispatcher detects that the thread's stack is corrupt, then the following steps take place:

1. The EXCEPTION_STACK_INVALID flag is set to 1.
2. The search for handlers immediately proceeds to the system last-chance handler.

Handler Invocation and Arguments

Every exception handler is invoked as a function which returns a status value. The function call is defined as follows:

(*ExceptionHandler)
(ExceptionRecord, EstablisherFrame, ContextRecord, DispatcherContext)

Arguments:

ExceptionRecord	The address of a primary exception record.
EstablisherFrame	Virtual frame pointer of the establisher (see Section 8.1, Referring to a Procedure Invocation).
ContextRecord	Address of an invocation context block containing the saved original context at the point where the exception occurred. During an unwind, this is the address of the invocation context block for the establisher.
DispatcherContext	Address of a control record for the exception dispatcher (see below).

Function Value:

STATUS	A value indicating the action to be taken upon handler return. The valid values are ExceptionContinueExecution and ExceptionContinueSearch. Note: the exception dispatcher allows additional return values from its own exception handlers.

The control record pointed to by DispatcherContext provides communication between the handler and the exception dispatcher (the system routine that actually invokes the handler). This record provides information about the establisher. Of the fields listed below, all but ControlPC are read-only to exception handlers (except for handlers for the exception dispatcher itself).

ControlPC contains the PC where control left the establisher of the exception handler, i.e., the PC of the call instruction or the instruction that caused the exception. This field may be updated by a handler. If a nested exception occurs during unwinding while the handler is still active, then the value of the PC used for the establisher will be the updated value of ControlPC. This mechanism can be employed to retire nested exception handling scopes local to a procedure to assure that each is executed at most once (even in the presence of

a nested exception within such a handler). The ControlPC value must, however, always be an address within the procedure whose handler is executing.

FunctionEntry contains a pointer to the procedure descriptor for the establisher.

6.1.10 Modification of Exception Records and Context by Handlers

The exception records, exception qualifiers, invocation context blocks, and control records that are passed to an exception handler are always allocated in writable memory. Handlers may write to any location in these data structures. The exception records and exception qualifiers that are passed to a handler are copies of the original ones. Modifications to them are seen by other subsequently called handlers (within the limits defined below) but do not affect the original data structures.

The effect of a handler modifying passed exception information is as follows:

1. If the EXCEPTION_NONCONTINUABLE flag in the primary exception record is changed from 0 to 1, then the exception handler which made the modification *must not* return ExceptionContinueExecution, nor may any handler subsequently invoked for the exception return ExceptionContinueExecution.

   If ExceptionContinueExecution is returned after the EXCEPTION_NONCONTINUABLE flag has been changed from 0 to 1, then a nested exception is raised with ExceptionCode = STATUS_NONCONTINUABLE_EXCEPTION, indicating that an attempt was made to continue from a noncontinuable exception. This second exception is also noncontinuable.

2. If any flags in ExceptionFlags in the primary exception record are modified except as specified above, there is no effect after the exception handler completes; all handlers subsequently invoked for the exception receive a primary exception record with the flags unmodified.

   In particular, if an exception handler changes the EXCEPTION_NONCONTINUABLE flag from 1 to 0, that handler must not return ExceptionContinueExecution, and any and all handlers subsequently invoked for the exception will be invoked with the EXCEPTION_NONCONTINUABLE flag set to 1.

3. If the contents of the record specified by ContextRecord or DispatcherContext are modified by a handler (except for ControlPC), the results are *unpredictable*, and such a handler does not conform to this standard.

4. Except as specified above, all changes made to the exception information will be visible to handlers subsequently invoked for the exception. Any other effects of modifying the exception information are not defined by this standard.

6.1.11 Handler Completion and Return Value

When an exception handler has finished all its processing, it completes by performing one of the following actions:

- *reraising* the exception
- *continuing* execution of the thread
- initiating procedure invocation *unwinding*

Completion by Reraise

If an exception handler determines that additional handlers should be invoked for the exception (because it could not completely handle the exception), it can reraise the exception by returning ExceptionContinueSearch.

Reraise causes the next exception handler to be invoked (see Section 6.1.9, Invocation of Exception Handlers).

If all exception handlers established by the thread reraise the exception, the system last chance handler is invoked, with system dependent results.

Completion by Continue

By returning ExceptionContinueExecution, an exception handler can continue execution of the thread at the address specified by the continuation PC in the ContextRecord, with the context of the interrupted procedure restored.

If ExceptionContinueExecution is returned and the EXCEPTION_NONCONTINUABLE flag is 1, then a nested exception is raised with ExceptionCode = STATUS_NONCONTINUABLE_EXCEPTION. This second exception is also noncontinuable.

Continuation from Unwind

When an unwind is in progress, the status returned by handlers must be ExceptionContinueSearch; otherwise STATUS_INVALID_DISPOSITION is raised. That is, handlers may not continue during an unwind operation.

Continuation from Signal Exceptions

The legality and effects of continuation from a signal exception are governed by the underlying signal, as specified by the implementation of the POSIX environment.

Completion by Unwinding

The unwind type of completion is more complex than simply returning a value. See Section 6.2, Unwinding, for details and considerations on unwinding.

6.1.12 Other Considerations

The following details certain aspects of the Alpha AXP architecture that have significant implications for exception handling. The rules presented are designed to assure correct operation across all implementations of that architecture. As with all aspects of this calling standard, optimization information may assure correct behavior *as if* these rules were followed without appearing to explicitly do so.

Alternative approaches that exploit implementation specific characteristics are also possible, but are outside the scope of this standard.

Exception Synchronization

The Alpha AXP hardware architecture allows instructions to complete in a different order than that in which they were issued, and for exceptions caused by an instruction to be raised after subsequently issued instructions have been completed. Because of this, the state of the machine when a hardware exception occurs cannot be assumed with precision unless it has been guaranteed by bounding the exception range with the appropriate insertion of TRAPB instructions.

The rules for bounding the exception range are as follows:

- If a procedure has an exception handler that does not simply reraise all arithmetic traps caused by code not contained directly within that procedure then it must issue a TRAPB instruction before it establishes itself as the *current procedure*.

  RATIONALE

  The above is required because a standard procedure is not allowed to handle traps that it might not have caused.

- If a procedure has an exception handler that does not simply reraise all arithmetic traps caused by code not contained directly within that procedure or by any procedure that might have been called while that procedure was *current* then it must issue a TRAPB instruction in the procedure epilogue while it is still the *current procedure*.

  RATIONALE

  The above is required because handlers established by previous invocations in the call chain might not be able to handle exceptions from a procedure invocation that is no longer active.

- If a procedure has an exception handler that is sensitive to the invocation depth then it must issue a TRAPB instruction immediately before and after any call. Furthermore, the handler must be able to recognize exception PC values that represent TRAPB instructions immediately after a call and adjust the depth appropriately.

These rules ensure that exceptions are detected in the context within which exception handlers have been set up to handle them.

These rules do *not* ensure that all exceptions are detected while the procedure within which the exception-causing instruction was issued is current. For example, if a procedure without an exception handler is called by a procedure that has an exception handler which is not sensitive to invocation depth, an exception detected while that called procedure is current may have been caused by an instruction issued while the caller was the current procedure. This means that the frame, designated by the exception handling information, is the frame that was current when the exception was detected, *not* necessarily the frame that was current when the exception-causing instruction was issued.

Continuation from Exceptions

The Alpha AXP architecture neither guarantees that instructions are completed in the same order in which they were fetched from memory nor that instruction execution is strictly sequential. Continuation after some exceptions is possible, but there are restrictions as reflected in the following discussions.

Software raised general exceptions are, by definition, synchronous with the instruction stream and can have a well defined continuation point. Thus, a handler may have the option of requesting continuation from a software raised exception. However, since compiler-generated code typically relies on error free execution of previously executed code, continuing from a software raised exception may produce unpredictable results and unreliable behavior unless the handler has explicitly fixed the cause of the exception in such a way as to be transparent to subsequent code.

Hardware faults on Alpha AXP systems follow rules that, loosely paraphrased, state that if the offending exception is fixed, re-execution of the instruction (as determined from the supplied PC) will yield correct results. This does *not* imply that no instructions following the faulting instruction have been executed (see the *Alpha AXP System Reference Manual* for more details). Hardware faults can therefore be viewed as similar to software raised exceptions and can have well defined continuation points.

Arithmetic traps cannot be restarted since all the information required for a restart is not available. The most straightforward and reliable way in which software may guarantee the ability to continue from this type of exception is by placing appropriate TRAPB instructions in the code stream. Although this does allow continuation, this technique must be used with extreme caution due to the negative side effect on application performance. A more sophisticated technique that requires typically one TRAPB per basic block is described in the *Alpha AXP System Reference Manual*, Section 4.7.5.1, Imprecise/Software Completion Trap Modes.

6.2 Unwinding

The unwinding capabilities specified in this section are for support of:

- Correct and predictable nonlocal GOTO support in a multilanguage environment
- Support for the construction of modular, maintainable multilanguage applications

6.2.1 Unwind Basic Considerations

Unwinding refers to the action of returning from a procedure or a chain of procedures by a mechanism other than the normal return path. Performing an *Unwind* operation in a thread causes a transfer of control from the location at which the unwind operation is initiated to a *target location* in a *target invocation*. This transfer of control also results in the termination of all procedure invocations, including the invocation in which the unwind request was initiated, up to the target procedure invocation. Thread execution then continues at the target location.

Before control is transferred to the unwind target location, the unwind support code invokes all frame-based handlers which were established by procedure invocations that are being terminated, plus the handler for the target invocation. These handlers are invoked with an indication that an unwind is in progress. The exception record passed to the target invocation's handler also has EXCEPTION_TARGET_UNWIND set to 1. This gives each procedure invocation the chance to perform clean-up processing before its context is lost.

Once all the relevant frame-based handlers have been called and the appropriate frames have been removed from existence, the target invocation's saved context is restored and execution is resumed at the specified location.

The results of attempting an unwind operation to any invocation previous to the top level procedure of a thread is *undefined* by this standard.

Unwinding does not require an exception handler to be active; it may be used by languages to implement non-local GOTO.

6.2.2 Types of Unwind

There are two types of unwind requests:

General unwind

A *general unwind* transfers control to a specified location in a specified procedure invocation.

The target procedure invocation is specified by a frame pointer (see Section 8.1, Referring to a Procedure Invocation).

The target location is specified with an absolute PC value.

When a general unwind is completed, the registers are updated from the invocation context for the target frame. Register R0 obtains its value from the *ReturnValue* argument to unwind, allowing a status to be returned to the target of the unwind.

Exit Unwind

It is valuable for a thread which is terminating execution to be able to clean up its use of shared resources. In a single-threaded process, these might be global resources shared among processes, such as files, locks, or shared memory. For multithreaded processes, global resources plus process-wide resources like a heap might need to be restored to a known state.

Because of this, user mode thread exit may be accomplished *only* by unwinding. A special case form of unwind, termed *exit unwind*, invokes all established frame-based handlers with an exception record specifying that an exit unwind is in progress, terminates all procedure invocations up to the beginning of the call chain, and terminates execution of the thread. Threads that use any other mechanism are not considered to be standard and their behavior is *undefined*.

6.2.3 Unwind Invocation Types

There are two cases under which an unwind may be invoked. Those initiated while an exception is active and those initiated while no exception is active.

Unwind with No Active Exception

An unwind which is initiated when no exception is active is usually done to perform a non-local GOTO, that is, to transfer control directly to some code location which is not part of the currently executing procedure or is not statically known. Even this type of operation must provide a mechanism to allow cleanup operations of terminated invocations (including restoring a consistent set of register values) to be performed. The unwind mechanism is used to support this type of operation.

Unwind during an Active Exception

The handler, or any descendant procedure called directly or indirectly by the handler, can continue execution of the thread at a different location than that at which the exception was raised by initiating an unwind operation.

An unwind operation specifies a target invocation in the procedure invocation chain and a location in that procedure. The operation terminates all invocations up to the target invocation, and continues thread execution at the specified location in that procedure.

Before control is transferred to the target location, the unwind operation invokes each frame-based handler which was established by any procedure invocations being terminated, plus the handler for the target invocation.

6.2.4 Unwind Initiation

Initiating a General Unwind

A thread may initiate a general unwind operation by calling one of two system library routines. The routines differ only in how their first argument specifies the target frame: as a *virtual frame pointer* or a *real frame pointer* (see Section 8.1, Referring to a Procedure Invocation). These routines are defined as follows:

RtlUnwind (VirtualTargetFrame, TargetPC, ExceptionRecord, ReturnValue)

RtlUnwindRfp (RealTargetFrame, TargetPC, ExceptionRecord, ReturnValue)

Arguments:

VirtualTargetFrame	If non-zero, the virtual frame pointer of the target procedure invocation to which the unwind should be done. If zero, an exit unwind is initiated. In addition, the EXCEPTION_EXIT_UNWIND flag is set to 1 in the exception record.
RealTargetFrame	If non-zero, the real frame pointer of the target procedure invocation to which the unwind should be done. If zero, an exit unwind is initiated. In addition, the EXCEPTION_EXIT_UNWIND flag is set to 1 in the exception record.
TargetPC	The address within the target invocation at which to continue execution. If TargetFrame is zero, then TargetPC is ignored.
ExceptionRecord	If non-zero, the address of a primary exception record. If zero, specifies that a default exception record should be supplied (see below).
ReturnValue	The value to use as the return value (contents of R0) at the completion of the unwind.

Function Value:

None.

If the ExceptionRecord argument is zero, then *RtlUnwind()* or *RtlUnwindRfp()* supplies a default exception record which specifies exactly one exception record in which ExceptionCode is STATUS_UNWIND if a non-zero TargetFrame is specified, and STATUS_EXIT_UNWIND otherwise. For either an explicit or a default exception record, the EXCEPTION_UNWINDING flag is set to 1, and the EXCEPTION_EXIT_UNWIND flag is set to 1 if a null TargetFrame is specified. ExceptionAddress is set to the address of the call to *RtlUnwind()* or *RtlUnwindRfp()*.

If the ExceptionRecord argument is specified when the unwind is initiated, then all other properties of the exception record are determined by ExceptionRecord. If *RtlUnwind()* or *RtlUnwindRfp()* detects that a specified exception record is not a valid unwind record, it will raise the exception of STATUS_INVALID_EXCEPTION. If the TargetFrame cannot be found, then the last chance handler will be called since all procedures have been terminated.

Once an unwind is initiated, control never returns from the call.

6.2.5 Multiply Active Unwind Operations

During an unwind operation, another unwind operation may be initiated. This may occur, for example, if a handler which is invoked for the original unwind initiates another unwind, or if an exception is raised in the context of such a handler and a handler invoked for that exception initiates another unwind operation.

An unwind which is initiated while a previous unwind is active is either a *nested unwind* or an *colliding unwind.*

Nested Unwind

A nested unwind is an unwind which is initiated while a previous unwind is active, and whose target invocation in the procedure invocation chain is not a predecessor of the most current active unwind handler. That is, a nested unwind is one which does not terminate any procedure invocation which would have been terminated by the previously active unwind.

When a nested unwind is initiated, no special rules apply. The nested unwind operation proceeds as a normal unwind operation, and when execution resumes at the target location of the nested unwind, the nested unwind is complete and the previous unwind is once again the most current unwind operation.

Colliding Unwind

An colliding unwind is an unwind which is initiated while a previous unwind is active, and whose target invocation in the procedure invocation chain is a predecessor of the most current active unwind handler. That is, an colliding unwind is one which terminates one or more procedure invocations that would have been terminated by the previously active unwind.

An colliding unwind is detected when the most current active unwind handler is terminated. This detection of an colliding unwind is termed a *collision*.

When a collision occurs, the second (more recent) takes precedence and the prior unwind is abandoned.

The next action is to reinvoke the most current active unwind handler, since its establisher has not been unwound. The EXCEPTION_COLLIDED_UNWIND flag will be set in the exception record to indicate this situation to the handler.

6.2.6 Unwind Completion

When an unwind completes the following conditions are true:

- The target procedure invocation is the most current invocation in the procedure invocation chain.

- The environment of the target invocation is restored to the state when that invocation was last current, except for the contents of scratch registers.
- Register GP contains a pointer to the shared global storage region for a target procedure that is contained in the main image.
- Register R0 contains the return value which was passed by the routine that invoked the unwind.
- Execution continues at the target location.

6.2.7 Unwinding Coexistence with setjmp/longjmp

The procedure invocation unwinding facility defined by this standard can coexist and interoperate with *setjmp / longjmp* facilities. It is sufficient for the *jmp_buf* array to consist of just the frame pointer and program counter values that are needed as arguments to RtlUnwind or RtlUnwindRfp. A null pointer can be provided for the ExceptionRecord argument and the value of the longjmp expression can be provided for the ReturnValue argument.

Any environment which conforms to this standard must implement non-local gotos by using RtlUnwind or RtlUnwindRfp (or an equivalent means) to allow all procedures being terminated to cleanup any local or global state as appropriate.

6.2.8 Exceptions Raised During Unwinding

During an unwind operation, an exception may be raised, causing the invocation of handlers. This may occur, for example, if a handler which is invoked for the original unwind causes an exception to be raised in this context.

Unwinding may be implemented in several fashions, including removing each stack frame after its procedure is processed, and removing all of the procedures' frames only when the target invocation is reached. For this reason, if an exception is raised during unwinding, it is undefined whether handlers for unwound procedures are invoked for this exception.

CHAPTER 7

MULTITHREADED ENVIRONMENT CONVENTIONS

This Chapter defines the conventions to support the execution of multiple threads in the multilanguage Windows NT for Alpha AXP environment. Specifically, it defines how compiled code must perform stack limit checking. While this calling standard is compatible with a multithreaded execution environment, the detailed mechanisms, data structures and procedures that support this capability are not specified here.

For a multithreaded environment, the following characteristics are assumed:

- There can be one or more threads executing within a single process.
- The state of a thread is represented in a *thread environment block* (TEB).
- The TEB of a thread contains information that determines a stack limit, below which the stack pointer must not be decremented by the executing code (except for code that implements the multithread mechanism itself).
- Exception handling is fully reentrant and multithreaded.
- The correct way to terminate a thread is by returning from the initial procedure in which the thread began execution, or by a call to RtlUnwind specifying a null target environment or some other procedure that includes this effect. That is, correct thread termination involves unwind processing for all of the active frames of that thread.

7.1 Stack Limit Checking

A program that is otherwise correct can fail because of stack overflow. Stack overflow occurs when extension of the stack (by decrementing SP) allocates addresses not currently reserved for the current thread's stack.

Detection of a stack overflow situation is important. Without it, a thread, writing into what it considered to be stack storage, could modify data allocated in that memory for some other purpose. This would most likely produce unpredictable and undesirable results and/or unreproducible application failures.

The requirements for procedures that can execute in a multithread environment include checking for stack overflow. This section defines the conventions for stack limit checking in a multithreaded environment.

In the following sections, the term *new stack region* refers to the region of the stack from the old value of SP – 1 to the new value of SP.

7.1.1 Stack Guard Region

In a multithreaded environment, the memory beyond the limit of each thread's stack is protected by contiguous *guard pages*, which form the stack's *guard region*.

7.1.2 Stack Reserve Region

In some cases it is desirable to maintain a stack *reserve region*, which is a minimum sized region that is immediately above a thread's guard region. A reserve region may be desirable to assure that exceptions or asynchronous signals have stack space to execute on a thread's stack or to assure that the exception dispatcher and any exception handler that it might call have stack space to execute *after* an invalid attempt to extend the stack has been detected.

This standard does not require a reserve region.

7.1.3 Methods for Stack Limit Checking

Since there may be accessible memory at addresses lower than those occupied by the guard region, compilers must generate code such that the stack is never extended past the guard pages into accessible memory not allocated to the thread's stack.

The general strategy is to access each page of memory down to and possibly including the page corresponding to the intended new value for the stack pointer SP. If the stack is to be extended by an amount that is larger than the size of a memory page, then a series of accesses is required that works from higher to lower addressed pages. If any access results in a memory access violation, then the code has made an invalid attempt to extend the stack of the current thread.

NOTE

An access can be performed using either a load or a store operation. However, care must be taken to use an instruction that is guaranteed to make an access to memory. For example, do not use a *LDQ R31, instruction because the Alpha AXP architecture allows it to result in no memory access at all rather than a memory read access whose result is discarded as a result of the R31 destination.**

There are two methods for stack limit checking: implicit and explicit.

Implicit Stack Limit Checking

Two mutually exclusive strategies for implicit stack limit checking are of interest:

1. If the lowest addressed byte of the new stack region is guaranteed to be accessed prior to any further stack extension, then the stack can be extended by an increment that is equal in size to the guard region (without any further accesses).
2. If some byte (not necessarily the lowest) of the new stack region is guaranteed to be accessed prior to any further stack extension, then the stack can be extended by an increment that is equal in size to one-half the guard region (without any further accesses).

The stack frame layout (see Section 4.1.2, Stack Frame Procedure) and entry code rules (see Section 4.2.6) generally do not make it feasible to guarantee access to the lowest address of a new stack region without introducing an extra access solely for that purpose. Consequently, this calling standard uses the second strategy. While the amount of implicit stack extension that can be achieved is smaller, the check is achieved at no additional cost.

This calling standard requires that the minimum guard region size is 8192 bytes, the size of the smallest memory protection granularity allowed by the Alpha AXP architecture.

These considerations lead to the following rule:

> If the stack is being extended by an amount less than or equal to 4096 and no reserve region is required, then no explicit stack limit checking is required.

However, because asynchronous interrupts and calls to other procedures may also cause stack extension without explicit stack limit checking, stack extension with implicit limit checking must follow a strict set of conventions:

1. Explicit stack limit checking must be performed unless the amount by which the SP is decremented is known to be less than or equal to 4096 and no reserve region is required.
2. Some byte in the new stack region must be accessed before SP can be decremented for a subsequent stack extension.

   This access can be performed either before or after the SP is decremented for this stack extension, but it must be done before it can be decremented again.
3. No standard procedure call can be made before some byte in the new stack region is accessed.
4. The system exception dispatcher ensures that the lowest addressed byte in the new stack region is accessed if any kind of asynchronous interrupt occurs after SP is decremented but before the access in the new stack region occurs.

These conventions ensure that the stack pointer will not be decremented such that it points to accessible storage beyond the stack limit without this error being detected (by either the guard region being accessed by the thread or an explicit stack limit check failure).

As a matter of practice, the system can provide multiple guard pages in the guard region. When a stack overflow is detected as a result of access to the guard region, one or more guard pages can be unprotected for use by the exception handling facility, and one or more guard pages can remain protected to provide implicit stack limit checking during exception processing. However, the size of the guard region and the number of guard pages is *system defined*, and is not defined by this standard.

Explicit Stack Limit Checking

If the stack is being extended by an amount of unknown size or known size greater than the maximum implicit check size (4096) then a code sequence which follows the rules for implicit stack limit checking can be executed in a loop to access the new stack region incrementally in segments smaller than or equal to the minimum page size (8192 bytes). At least one access must occur in each such segment. The first access must occur between SP and SP – 4096 because, in the absence of more specific information, the previous guaranteed access relative to the current stack pointer may be as much as 4096 bytes greater than the current stack pointer address. The last access must be within 4096 bytes of the intended new value of the stack pointer. These accesses must occur in order, starting with the highest addressed segment and working toward the lowest addressed segment.

NOTE

A simple algorithm that is consistent with this requirement (but makes up to twice the minimum number of accesses) is to perform a sequence of accesses in a loop starting with the previous value of SP, decrementing by the minimum no check

extension size (4096), to but not including the first value that is less than the new value for the stack pointer.

The stack must **not** be extended incrementally in procedure prologues. A procedure prologue that needs to extend the stack by an amount of unknown size or known size greater than the minimum implicit check size must test new stack segments as just described in a loop that does not modify SP, and then update the stack with one instruction that copies the new stack pointer value into SP.

NOTE

An explicit stack limit check may be performed either by *inline* code that is part of a prologue or by a run-time support routine that is specially tailored to be called from a procedure prologue.

Stack Reserve Region Checking

The size of the reserve region, if any, must be included in the increment size used for stack limit checks, after which it is not included in the amount by which the stack is actually extended. (Depending on the size of the reserve region, this may partially or even completely eliminate the ability to use implicit stack limit checking.)

7.1.4 Stack Overflow Handling

If a stack overflow is detected, the result will be one of the following:

- Exception STATUS_STACK_OVERFLOW may be raised.
- The system may transparently extend the thread's stack, reset the TEB stack limit value appropriately, and continue execution of the thread.

Note that, if a transparent stack extension is performed, a stack overflow that occurs in a called procedure might cause the stack to be extended. Because of this, the TEB stack limit value must be considered volatile and potentially modified by external procedure calls and by handling of exceptions.

CHAPTER 8

PROCEDURE INVOCATIONS AND CALL CHAINS

Mechanisms for each of the following three functions are needed to support procedure call tracing:

- Mechanism to refer to a given procedure invocation
- Mechanism to provide the context of a procedure invocation
- Mechanism to traverse (walk) the procedure call chain

This chapter describes the data structures and procedures that support these functions.

8.1 Referring to a Procedure Invocation

When referring to a specific procedure invocation at run-time, either the *virtual frame pointer* or the *real frame pointer* for that invocation can be used. The *virtual frame pointer* of a procedure invocation is the contents of the stack pointer at entry to the procedure. The *real frame pointer* of a procedure is the contents of the stack pointer after the size of the fixed part of the stack frame has been subtracted from the virtual frame pointer. Note that the virtual frame pointer of an invocation is not the value used for addressing by the procedure itself. The contents of the SP register is modified in the procedure prologue and the resulting real frame pointer value then possibly copied into a FP register (in the case of a variable size stack frame). The resulting real frame pointer is then used for addressing local storage throughout the remainder of the procedure.

The real frame pointer is not, of itself, sufficient to unambiguously identify all possible procedure invocations. For example, a null frame procedure has the same real frame pointer as its caller because the null frame procedure allocates no stack storage. This ambiguity is of no consequence for the purposes of this calling standard because the real frame pointer value is always used in combination with a program counter value that identifies an instruction within a particular procedure.

The static link used in calling nested procedures in languages such as Pascal and Ada is usually either the virtual frame pointer or the real frame pointer value. The actual choice is implementation dependent and can vary from language to language and release to release.

8.2 Invocation Context Block

The full context of a specific procedure invocation is provided through the use of a data structure called an *invocation context block*. An invocation context block is identical in structure to the system defined *CONTEXT* structure.

8.3 Getting a Procedure Invocation Context with a Routine

A thread can obtain its own context by calling a system library function defined as follows:

RtlCaptureContext (ContextRecord)

Arguments:

ContextRecord	Address of an invocation context block into which the procedure context of the caller is written.

In fact, the context actually corresponds to that of the called *RtlCaptureContext()* procedure. However, that context is the same as that the caller except for the contents of R16, R26 and R28. The contents of R26 in the caller prior to the call can be obtained using *RtlVirtualUnwind()* (see below).

A thread can obtain the invocation context of the procedure preceding another procedure context by calling a system library function defined as:

RtlVirtualUnwind (ControlPC, FunctionEntry, ContextRecord, InFunction, EstablisherFrame, ContextPointers)

Arguments:

ControlPC	Address where control left the function.
FunctionEntry	Address of the function table entry for the function.
ContextRecord	Address of an invocation context block. The given context block is updated to represent the context of the previous (calling) frame.
InFunction	Address where the value 0 or 1 is written. Zero indicates that the ControlPC value is in either the prologue or the epilogue code of the function. One indicates that the ControlPC is in the body of the function.
EstablisherFrame	Pointer to a structure where both the real frame pointer value and the virtual frame pointer value of the context is written. Th real frame pointer value is defined iff InFunction is 1.
ContextPointers	Address of a context pointers record.

Function Value:

PreviousControlPC	Address where control left the previous frame.

This procedure takes an invocation context block together with its associated procedure descriptor and updates the context to reflect the state of the caller at the point where it made the call.

For exception and interrupt frames, the ControlPC is obtained from the trap frame continuation address as follows. For faults, ControlPC is both the last instruction that executed in the previous frame and the next instruction to execute if control were to resume in that frame. For synchronous or asynchronous traps, the ControlPC is the continuation address. For normal call frames, the ControlPC "where control left the function" is the address of the call instruction and not the return address for that call.

If the given context corresponds to a leaf procedure, then the ControlPC can be obtained from the saved PC value found in the context record. Otherwise, the ControlPC should be the result of a previous call of this procedure for the previous context.

If a context pointers record is specified (non-null), then the address where each register is restored from is recorded in the appropriate field of the record. This record can then be used to modify the state of some currently active context when control returns to that context. See the Windows NT for Alpha AXP documentation for full details.

The operation of *RtlVirtualUnwind()*, which is also a building block for exception dispatching and unwinding, is based both on the procedure descriptor and the code of the procedure associated with the ControlPC.

If the ControlPC points to a reserved RET instruction (see Section 4.2.6) then the context of the caller is the same as the given context and the ControlPC of the caller is taken as the contents of the register holding the return address (which need not be R26) minus 4.

If the ControlPC points to stack adjustment instruction that is part of a reserved return sequence, then the context of the caller is formed by adjusting the value of the SP and the ControlPC of the caller is taken from the contents of the register holding the return address minus 4.

Otherwise, the context of the caller is recreated by restoring saved registers and by incrementing the stack pointer if the function decremented it. This is done by *reverse execution* of instructions in the prologue. Instructions are processed in the reverse of the normal order and their normal effect is reversed: informally speaking, a store becomes a load, a subtract becomes an add, a move-to becomes a move-from, and so on.

If the ControlPC is within the prologue, then reverse execution begins at the last completely executed instruction: if the instruction at ControlPC caused a fault, then reverse execution begins at ControlPC - 4; otherwise, at ControlPC. If the ControlPC is not within the prologue, then reverse execution begins with the last instruction of the prologue.

The instructions in the prologue are processed backwards sequentially. When reverse execution begins, and before the value of SP is used, it is assumed that either SP has a valid value, or that FP has a valid value and a "MOV SP, FP" instruction will be encountered before SP is used.

Basically, the operations performed are:

- If a register was saved on the stack, its value is restored from that location on the stack.
- If a register was copied to another register, its value is restored from that same register.
- If the stack pointer was decremented by a value, it is incremented by that same value.
- Any other instructions contained in the prologue are allowed, but ignored—they are assumed to have no effect on the context.
- The ControlPC of the caller is the value of register R26 at the conclusion of the reverse execution minus 4.

See Section 4.2.6, Entry and Exit Code Sequences for a detail description of the instructions that take part in reverse execution.

8.4 Walking the Call Chain

During the course of program execution it is sometimes necessary to navigate the call chain. Frame based exception handling is one case where this is done. Call chain navigation is only possible in the reverse direction (latest to earliest or top to bottom procedure).

The steps to perform for call chain navigation are:

1. Build an invocation context block when given a program state (which contains a register set).

   For the current routine, an initial invocation context block can be obtained by calling *RtlCaptureContext()*.

2. Repeatedly call *RtlVirtualUnwind()* until the end of the chain has been reached.

Compilers are allowed to optimize high-level language procedure calls in such a way that they do not appear in the invocation chain. For example, in-line procedures never appear in the invocation chain.

No assumptions should be made about the relative positions of any memory used for procedure frame information. There is no guarantee that successive stack frames will always appear at higher addresses.

8.5 Updating an Invocation Context

A given procedure's invocation context fields can be updated with new register contents using the ContextPointers optional parameter of *RtlVirtualUnwind()* (see Section 8.3).

CHAPTER 9

PROCEDURE DESCRIPTORS

Procedure descriptors serve two functions:

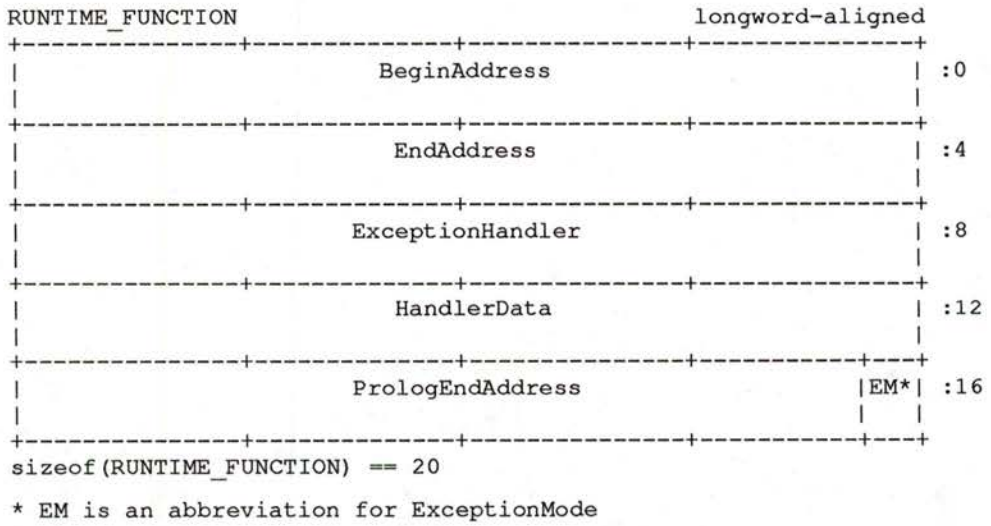
1. They provide the means to map from an arbitrary program counter value to the descriptive information associated with the code at that address.
2. They provide information about a procedure (what registers are saved and where, how long the prologue is, and so on) that is needed for call chain walking in general and exception handling in particular.

Every procedure must have an associated procedure descriptor except for null frame procedures (see Section 4.1.4, Null Frame Procedure).

9.1 Procedure Descriptor Representation

Procedure descriptors on Windows NT for Alpha AXP consist of a structure as shown in Figure 9–1.

Figure 9–1: Procedure Descriptor



The fields of this structure are defined in the following.

BeginAddress is the address of the first instruction and entry point of a procedure.

EndAddress is the address of the first instruction following the code for the procedure.

ExceptionHandler is the address of the exception handler for the procedure. If there is no associated handler, then this field contains null (0).

HandlerData is the address of the associated data for use by the handler for this procedure. If ExceptionHandler is null then this field must also be null.

PrologEndAddress is the address of the first instruction following the prologue of the procedure. (In a procedure that has no prologue, this will be the same value as the BeginAddress.)

If the procedure descriptor indicates a segment of the procedure that does *not* include the prologue, i.e., BeginAddress is not the beginning of the procedure, then PrologEndAddress contains the beginning address of the procedure. (Note that this is the only case in which PrologEndAddress will not be between BeginAddress and EndAddress.) This mechanism permits different segments of a procedure to be represented by different procedure descriptors, e.g., the segments are not contiguous, or the segments need different handlers or handler data.

Note that PrologEndAddress is a longword address. The low two bits of the longword that contains this field are used for the ExceptionMode field.

ExceptionMode encodes the caller's desired exception reporting behavior when calling certain mathematically oriented library routines. The possible values for this field are as follows:

Name	Meaning
EXCEPTION_MODE_SILENT	Raise no exceptions, create only finite values (no infinities, denormals nor NaNs). In this mode, either the function result or the C language *errno* variable must be examined for any error indication. This is the default mode.
EXCEPTION_MODE_SIGNAL	Raise exceptions for all error conditions except for underflow, which results in a zero result.
EXCEPTION_MODE_SIGNAL_ALL	Raise exceptions for all error conditions (including underflow).
EXCEPTION_MODE_FULL_IEEE	Raise no exceptions except as controlled by separate IEEE exception enable bits, create infinite, denormal and NaN values according to the IEEE floating point standard.

For a more complete description of these exception modes and how they are used, see the Windows NT for Alpha AXP system documentation.

The set of procedure descriptors of an image are all collected together in a single contiguous array and sorted according to the values of the BeginAddress field.

Properties of Procedures

For purposes of call chain tracing (see Chapter 8, Procedure Invocations and Call Chains and unwinding (see Section 6.2, Unwinding), the properties of a procedure are determined in part by the values in the procedure descriptor as defined above and in part by *reverse execution* of the procedure prologue.

These properties are determined as follows.

REGISTER_FRAME is 1 iff there is no instruction in the prologue that stores into the stack.

BASE_REG_IS_FP is 1 iff the last instruction in the prologue is an instruction that copies the contents of the stack pointer (SP) to the frame pointer (FP).

HANDLER_VALID is 1 iff the HandlerAddress field of the procedure descriptor is non-null.

EXCEPTION_MODE is specified in the ExceptionMode field of the procedure descriptor.

SP_SET is the unsigned offset in instructions (longwords) from the entry address of the procedure given in the BeginAddress field of the descriptor to the one *and only one* instruction in the procedure prologue which modifies the stack pointer. (This offset is assumed to be zero when there is no such instruction because the procedure has a FRAME_SIZE of 0.)

ENTRY_LENGTH is the unsigned offset in instructions (longwords) formed by taking the PrologEndAddress less the BeginAddress and dividing by four.

FRAME_SIZE is the unsigned size in quadwords of the fixed portion of the stack frame for this procedure. This value is found in the displacement field (divided by eight) of the instruction that modifies the stack pointer, or is zero if there is no such instruction.

NOTE

If a procedure requires a frame size that is too large to be represented using the displacement field of an instruction, then the size must be loaded from memory or otherwise materialized into a register and then subtracted from the stack pointer using a SUBQ *,*,SP instruction.

BEGIN_ADDRESS is specified in the BeginAddress field of the procedure descriptor.

END_ADDRESS is specified in the EndAddress field of the procedure descriptor.

PROLOG_END_ADDRESS is specified in the PrologEndAddress field of the procedure descriptor.

HANDLER_ADDRESS is specified in the ExceptionHandler field of the procedure descriptor.

HANDLER_DATA is specified in the HandlerData field of the procedure descriptor.

ENTRY_RA is always register R26.

9.2 Procedure Descriptor Access Routines

A thread can obtain information from the descriptor of any procedure in the thread's virtual address space by calling system library functions.

9.2.1 Current Procedure

In the course of running and debugging a program there are times when there must be a way to identify which procedure is currently executing. During normal thread execution the current procedure must be determined any time an exception arises so that the proper handlers will be invoked. Also a debugger must know which procedure invocation is currently executing to enable it to find information about the current state of the execution environment.

In order to completely determine the current execution context not only must the currently executing procedure be determined but also which instance of that procedure. This context of the current procedure together with a specific instance of that procedure invocation is called the *current procedure invocation* (which is often shortened to *current procedure*). At any

point in the execution of a thread there is always exactly one procedure that is considered to be the *current procedure*.

In the Windows NT for Alpha AXP Calling Standard the value in the PC is used to indicate the *current procedure* by means of a lookup table (see Section 9.1, Procedure Descriptor Representation).

The following system supplied routine may be used to obtain the address of the procedure descriptor that corresponds with any given PC value within the current address space.

RtlLookupFunctionEntry (ControlPC)

Arguments:

ControlPC	A PC value in the current address space for which the procedure value is to be returned. If zero, indicates the value should be returned for the routine making the call.

Function Value:

PROC_DESC	Address of the procedure descriptor for the procedure containing the requested PC. If the return value is null, then the PC is not currently mapped.

AXP NT Status (8/21/93)

*** This report is COMPANY CONFIDENTIAL
*** There are approximately 100 applications still missing from this listing

✓ = Work completed
❑ = Work in process, but not completed

			Priority	Targeted	Committed	Demo	Shipping
- 5/5 Logiciel - 5/5 Base	Vertical	Europe	M	✓	✓		
- 5/5 Logiciel - 5/5 OCR	Vertical	Europe	M	✓	✓		
- A/Soft Development, Inc. - nu/TPU	Vertical	USA	M	✓	✓		
- ABB Syst. Cont. - MEMSYS	Vertical	USA	M	✓	❑		
- Abortext - Publisher	Horizontal	Publishing	M	✓	❑		
- Abraxas Software - CodeCheck	Enabler	CASE Tool	H	✓			
- Abraxas Software - PCYACC	Enabler	CASE Tool	H	✓			
- Accelerat8 - VMS to NT Tools	Enabler	Conversion Tool	H	✓			
- Accessware (TM) - AccessPoint	Vertical	USA	M	✓	✓		
- AccuCOBOL 85	Enabler	Compilers	M	✓	✓		
- Accumate - EIS	Horizontal	Analysis/Access	M	✓			
- ACT Financial - City Desk Trading	Vertical	USA	M	✓	❑		
- Acuity Management Systems Ltd. - Acuity/ES	Vertical	Europe	M	✓	✓		
- Adamation - Adamation	Horizontal	Technical Horiz.	M	✓	✓		
- ADEV - PRO Series	Vertical	USA	M	✓	✓		
- Adobe Illustrator	Horizontal	Graphics/Imaging	M	✓			
- Adobe Photoshop	Horizontal	Graphics/Imaging	H	✓			
- Adobe Premier - Video Editor	Horizontal	Graphics/Imaging	H	✓			
- Adobe Printshop	Horizontal	Graphics/Imaging	H	✓			
- ADP - FS Trade/FS Partner	Vertical	USA	H	✓	✓		
- ADRA - CADRA-III	Horizontal	Technical Horiz.	H	✓	✓	✓	❑
- Advance Geo - ProMAX	Vertical	USA	M	✓	✓		
- Advanced Mathematical Software - Lamps	Vertical	Europe	M	✓	✓		
- Advanced Micro Technology, Inc. - EasyAuthor	Vertical	USA	M	✓	✓		
- Advanced Multiphase Tech. - AMT	Vertical	USA	M	✓	✓		
- AGA Computing -GUI for SQL Access	Enabler	Other Enablers	M	✓	❑		
- AGA - CAiCE - DOT engineering software	Vertical	USA	H	✓	✓	✓	
- AGE Logic - Xoftware NT	Enabler		M	✓	✓		
- Alcatel-Alsthom - PABX Operation & Control	Vertical	USA	M	✓	❑		
- Aldus - Freehand	Horizontal	Graphics/Imaging	H	✓	❑		

- Aldus - Pagemaker	Horizontal	Publishing	H		✓	☐		
- Aldus - Persuasion	Horizontal	Graphics/Imaging	H		✓	☐		
- Aldus - Photostyler	Horizontal	Graphics/Imaging	H		✓	☐		
- Aldus - Trapwise	Horizontal	Graphics/Imaging	H		✓	☐		
- Algorithmics - RISKWATCH	Vertical	USA	M		✓	✓		
- Altamira - Macromedia	Enabler	Other Enablers	M		✓	☐		
- Alysys - ADA	Enabler	Compilers	M		✓			
- AMC - ICE	Vertical	USA	M		✓	☐		
- Answer - P&C	Vertical	USA	M		✓	☐		
- Applied Auto. Tech., Inc. - AutoQuality	Vertical	USA	M		✓	✓		
- Applied Auto. Tech., Inc. - AutoShip	Vertical	USA	M		✓	✓		
- Applied Auto. Tech., Inc. - AutoTime	Vertical	USA	M		✓	✓		
- Applied Auto. Tech., Inc. - AutoTrack	Vertical	USA	M		✓	✓		
- Applied Auto. Tech., Inc. - AutoWIP	Vertical	USA	M		✓	✓		
- Applied Information Sciences - UniAccess	Vertical	USA	M		✓	✓		
- Applied Terravision - Datavision	Horizontal	GIS Systems	M		✓	☐		
- Applied Terravision - Eagle	Horizontal	GIS Systems	M		✓	☐		
- Applied Terravision - Terraview	Horizontal	GIS Systems	M		✓	☐		
- Applix, Inc. - Aster*X	Horizontal	Groupware/Mail	M		✓	✓		
- Approach Software - Approach For Windows	Horizontal	Analysis/Access	M		✓			
- Arbor Software - EssBase	Enabler	Database	M		✓	☐		
- Arcadd, Inc. - Lottacadd	Vertical	USA	M		✓	✓		
- Archibus - ARCHIBUS/fm	Vertical	USA	M		✓	✓		
- ASA Hindsight	Enabler	CASE Tool	M		✓			
- ASK - MANMANX	Vertical	USA	M		✓			
- Aspen Technology, Inc. - Aspen Plus	Vertical	USA	M		✓	✓		
- Asymetrix - Multimedia Toolbook	Enabler	Other Enablers	H		✓	✓		
- Atria ClearCASE	Enabler	CASE Tool	H		✓	☐		
- ATT - Tuxedo	Enabler	Other Enablers	M		✓			
- AUDIOtechs - Hyperkit	Enabler	Other Enablers	H		✓	✓	✓	✓
- Auto-trol - Technical Illustrator	Horizontal	Publishing	M		✓	☐		
- AutoDesk - 3-D Studio	Horizontal	Technical Horiz.	H		✓	☐		
- AutoDesk - AutoCAD	Horizontal	Technical Horiz.	H		✓	☐		
- AutoDesk - Hyperchem	Horizontal	Technical Horiz.	H		✓	☐		
- Autologic - SoftPIP (high speed postscript print)	Horizontal	Publishing	M		✓	✓	✓	✓
- Automated Office Systems, Inc. - SEARCHmate	Vertical	USA	M		✓	✓		
- AVP Systems - Sales Tax System	Vertical	USA	M		✓	✓		

- Bacg Limited - Data Maintenance	Vertical	Europe	M		✓	✓		
- Bacg Limited - Item Group Maintenance	Vertical	Europe	M		✓	✓		
- Bacg Limited - Item Maintenance	Vertical	Europe	M		✓	✓		
- Bacg Limited - Organisational Structures	Vertical	Europe	M		✓	✓		
- Bacg Limited - Purchasing Management	Vertical	Europe	M		✓	✓		
- Bacg Limited - Warehouse Management	Vertical	Europe	M		✓	✓		
- Backstage Software - LMF	Vertical	USA	M		✓	✓		
- Bailey Engineers - Baileys Engn. Imag. Appl.	Vertical	USA	M		✓	✓		
- Banyon - Vines	Enabler	Communications	M		✓	❑		
- BASIS	Enabler	Database	H		✓			
- BBN - RS/1	Horizontal	Analysis/Access	H		✓			
- BCR Computing Corporation - OSCAR	Vertical	USA	M		✓	✓		
- Beacon Expert Systems, Inc. Negotiator Pro	Vertical	USA	M		✓	✓		
- Beckman - Lab Manager LIMS	Horizontal	Technical Horiz.	M		✓	✓		
- Beckman - PeakPro	Horizontal	Technical Horiz.	M		✓	❑		
- Beilfuss & Associates -IGrds	Vertical	USA	M		✓	✓		
- Bellcore - ISP - Integrated Services Periph.	Vertical	USA	M		✓	✓		
- Bentley Systems - Microstation	Horizontal	Technical Horiz.	M		✓	❑		
- Biles & Assosciates - AIMStation	Vertical	USA	M		✓	✓		
- BIO N.V. - BIS (R)	Vertical	Europe	M		✓	✓		
- Biorad/Sadther Division - VAX Search	Vertical	USQ	M		✓	✓		
- Biosym Tech - Discover, Etc.	Vertical	USA	M		✓	✓		
- Blue Sky Magic Fields	Enabler	CASE Tool	H		✓	✓		
- Blue Sky RoboHelp	Enabler	CASE Tool	H		✓	✓		
- Blue Sky Visual SQL	Enabler	CASE Tool	H		✓	✓		
- Blue Sky WindowMaker	Enabler	CASE Tool	H		✓	✓	✓	✓
- BMDP	Horizontal	Analysis/Access	M		✓			
- Borland - dBase IV	Enabler	Database	M		✓			
- Borland - Interbase	Enabler	Database	H		✓	✓		
- Borland - Paradox	Enabler	Database	H		✓			
- Borland C/C++	Enabler	Compilers	H		✓	❑		
- Borland QuattroPro	Horizontal	Spreadsheets	H		✓			
- Borland Turbo Pascal	Enabler	Compilers	H		✓			
- Bradly Associates Ltd. - GINO-F	Vertical	Europe	M		✓	✓		
- Bradly Associates Ltd. - Ginograf	Vertical	Europe	M		✓	✓		
- Bradly Associates Ltd. - Ginosurf	Vertical	Europe	M		✓	✓		
- BSO/Tasking - BSO/Assember	Vertical	USA	M		✓	✓		

- BSO/Tasking - BSO/C cross compilers	Vertical	USA	M		✓	✓		
- BSO/Tasking - Crossview	Vertical	USA	M		✓	✓		
- BT (City Business Products) Ltd. - OTS	Vertical	Europe	M		✓	✓		
- Byte S.H. S.P.A. - Sipert 2000	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Asset Ledger	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Bacs Cat	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Bill of Materials	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Busi Cat	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - C Comm	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - C State	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - CC 420	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Databse Manager	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Employee Analysis	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Fax Cat	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Job Costing	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Nominal Ledger	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Payroll/Personnel	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Picking Lists	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Pro Cat	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Purchase Ledger	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Purchase Order Processing	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Routings	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Sales Ledger	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Sales Order Processing	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Stock Control	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Synergy DBC	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Synergy DTK	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Synergy ICS	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Tabi Cat	Vertical	Europe	M		✓	✓		
- C Cat Ltd. - Travel Cat	Vertical	Europe	M		✓	✓		
- C-Pak Corporation - C-Pak's Financials App.	Vertical	USA	M		✓	✓		
- C-Pak Corporation - C-Pak's Long Term Health	Vertical	USA	M		✓	✓		
- C-Pak Corporation - C-Pak's Manufacturing Sys	Vertical	USA	M		✓	✓		
- Caci Limited - FMMS	Vertical	Europe	M		✓	✓		
- Caci Limited - Prophecy	Vertical	Europe	M		✓	✓		
- CAD - EPVS	Vertical	USA	M		✓	✓		
- Cadance - Allegro	Horizontal	Technical Horiz.	H		✓	❑		

- Cadance - Concept	Horizontal	Technical Horiz.	H		✓	☐		
- Cadance - Dracula	Horizontal	Technical Horiz.	H		✓	☐		
- Cadance - FrameWork	Horizontal	Technical Horiz.	H		✓	☐		
- Cadance - Verilog	Horizontal	Technical Horiz.	H		✓	☐		
- Cadance - Veritime	Horizontal	Technical Horiz.	H		✓	☐		
- Cadcentre Ltd. - GNC NC	Vertical	Europe	M		✓	✓		
- Cadre Teamwork	Enabler	CASE Tool	M		✓			
- CAL - CEAPAC Estate Agency	Vertical	USA	M		✓	☐		
- California Software - BABY/AS	Enabler	Conversion Tool	M		✓			
- California Software - BABY/4XX	Enabler	Conversion Tool	M		✓			
- California Software - BABY/36	Enabler	Conversion Tool	M		✓			
- Cambridge Scientific Computing - ChemDraw	Horizontal	Technical Horiz.	M		✓	✓		
- Cambridge Scientific Computing - ChemDraw Plus	Horizontal	Technical Horiz.	M		✓	✓		
- Cambridge Scientific Computing - ChemOffice	Horizontal	Technical Horiz.	M		✓	✓		
- Cambridge Technology Group	Enabler	4 GL	H		✓			
- CASEWORKS - CASE W VIP	Enabler	CASE Tool	H		✓			
- Cegelec - System Controller w/GE FAUC	Vertical	USA	M		✓	☐		
- Cegelec/Alcatel - System Controller SCADA	Vertical	USA	M		✓	☐		
- Centerline Codecenter	Enabler	CASE Tool	M		✓			
- Cerner - CareNET	Vertical	USA	M		✓	✓		
- Chesapeake Decision Sciences - MIMI	Horizontal	Technical Horiz.	M		✓	☐		
- CIMAGE - Document Manager (TM)	Vertical	USA	M		✓	✓		
- CIMAGE - Work Flow Mgr.	Vertical	USA	M		✓	✓		
- CIMLINC - Linkage	VErtical	USA	M		✓	✓		
- Citicorp FAME	Horizontal	Analysis/Access	M		✓			
- Claris - File Maker Pro	Enabler	Other Enablers	H		✓	✓	✓	
- CLM/SYSTEMS, INC. - CEAL	Vertical	US.*.	M		✓	✓		
- CO-CAM - Brock Phone Sales Admin/Mkgt.	Vertical	USA	M		✓	☐		
- Cognos	Enabler	4 GL	H		✓	✓		
- Cold Springs Harbor Lab - Genetic Mapp. Wkbch.	Horizontal	Technical Horiz.	M		✓	✓		
- Command Systems - Claims	Vertical	USA	M		✓	☐		
- Compushare Inc. - EIS	Horizontal	Financial Mgmt.	M		✓	✓		
- Computational Mech. Beasy Ltd. - Beasy Acou.	Vertical	Europe	M		✓	✓		
- Computational Mech. Beasy Ltd. - Beasy Cp	Vertical	Europe	M		✓	✓		
- Computational Mech. Beasy Ltd. - Beasy Linear	Vertical	Europe	M		✓	✓		
- Computer Aided Dev. Corporation - Wincad (R)	Vertical	Europe	M		✓	✓		
- Computer International Ltd. - TARC	Vertical	Europe	M		✓	✓		

- Computer Vision - Adv. Tech. Ctr.	Horizontal	Technical Horiz.	H		✓	✓	✓	✓
- Computer Vision - CV Design, Designview etc.	Horizontal	Technical Horiz.	H		✓	✓		
- Computron Tech. Europe, Ltd. - Optical Fiche	Vertical	Europe	M		✓	✓		
- Computron Tech. Europe, Ltd. - Records Mgt.	Vertical	Europe	M		✓	✓		
- Computron Tech. Europe, Ltd. - Workflow Mgt.	Vertical	Europe	M		✓	✓		
- Computron Technologies - Accounting	Vertical	Europe	M		✓	✓		
- Contact Software International - ACT!	Horizontal	Groupware/Mail	M		✓			
- Control Systems - VXL	Vertical	USA	H		✓	✓		
- CorelDRAW	Horizontal	Graphics/Imaging	M		✓			
- CORTEX CORPORATION - CorVision	Vertical	USA	M		✓	✓		
- CPI - ImageIn	Horizontal	Graphics/Imaging	H		✓	✓	✓	✓
- Cross-Z Intl. - Navigator	Horizontal	Analysis/Access	H		✓	✓		
- Cross-Z Intl. - Private Eye	Horizontal	Analysis/Access	H		✓	✓		
- CSA - Retran	Vertical	USA	M		✓	❑		
- Cybermedix - CLINIPLEX	Vertical	USA	M		✓	✓		
- Cyberscience Corporation Ltd. - COCS	Vertical	Europe	M		✓	✓		
- Cyberscience Corporation Ltd. - Visual Cyber.	Vertical	Europe	M		✓	✓		
- Cygnus - GNU Toolchain Modif.	Enabler	Other Enablers	M		✓	✓		
- Cyport (Sol'n Series) - Payroll/HR	Horizontal	Financial Mgmt.	M		✓	✓		
- Dash Associates - Xpress-Mp	Vertical	Europe	M		✓	✓		
- Data Research - DRA System	Vertical	USA	M		✓	✓		
- Database Mgmt. Tech., Inc. - DBAnalyzer	Vertical	USA	M		✓	✓		
- Datatab Computer Services Ltd. - Dist.& Mgr.	Vertical	Europe	M		✓	✓		
- Datatab Computer Services Ltd. - Dress	Vertical	Europe	M		✓	✓		
- Datatab Computer Services Ltd. - Financials	Vertical	Europe	M		✓	✓		
- Datatab Computer Services Ltd. - Retail Mgmt.	Vertical	Europe	M		✓	✓		
- DEC - Plexis	Horizontal	Publishing	M		❑			
- DEC ACA Services	Enabler	Other Enablers	H		✓	✓		
- DEC ADA	Enabler	Compilers	H		✓			
- DEC AUNT Assembler for Windows NT	Enabler	Compilers	H		✓	✓	✓	✓
- DEC C++	Enabler	Compilers	M		✓	✓	✓	
- DEC COBOL	Enabler	Compilers	M		✓			
- DEC DCE NT Client	Enabler	Other Enablers	H		✓	✓		
- DEC DCE NT Server	Enabler	Other Enablers	H		✓	✓		
- DEC DECadmire	Enabler	CASE Tool	H		✓	❑		
- DEC DECforms	Enabler	CASE Tool	H		✓	❑		
- DEC DECmessageQ	Enabler	Other Enablers	H		✓	✓		

- DEC DECps - client	Enabler	Other Enablers	H		✓				
- DEC DECscheduler Client for NT	Enabler	Sys. Management	M		✓	✓			
- DEC DECtalk for Windows NT	Enabler	Other Enablers	M		✓	✓	✓		
- DEC DECtrace	Enabler	Sys. Management	M		✓				
- DEC Desktop ACMS	Enabler	Other Enablers	H		✓	✓	✓	❑	
- DEC Digital Standard Mumps	Enabler	Compilers	H		✓	✓			
- DEC EDI	Enabler	Other Enablers	H		✓				
- DEC eXcursion	Enabler	Communications	H		✓	✓	✓	✓	
- DEC FORTRAN	Enabler	Compilers	H		✓	✓	✓		
- DEC Objectworks	Horizontal	Groupware/Mail	H		✓	❑			
- DEC Pathworks for Windows NT	Enabler	Communications	H		✓	✓	✓		
- DEC QFD Expert for Windows NT	Enabler	Other Enablers	H		✓	✓	✓		
- DEC RDB	Enabler	Database	H		✓	✓			
- DEC RDB - DBA tools incl. RDBexpert	Enabler	Sys. Management	M		✓				
- DEC RFM	Horizontal	Groupware/Mail	H		✓	❑			
- DEC SLS	Enabler	Sys. Management	M		✓				
- DEC Team Route	Horizontal	Groupware/Mail	M		✓				
- DEC Teamlinks	Horizontal	Groupware/Mail	M		✓				
- DEC X.400 and X.500 products	Horizontal	Groupware/Mail	H		✓				
- DEC XLIB	Enabler	Other Enablers	H		✓	✓			
- Decathlon Data Systems - GOLDMEDAL (R)	Vertical	USA	M		✓	✓			
- Decathlon Data Systems - TEAMWARE (TM)	Vertical	USA	M		✓	✓			
- Design CAD - Design CAD 2D, 3D	Horizontal	Technical Horiz.	M		✓	❑			
- Desktop Data - Data Feed Toolkit	Enabler	Other Enablers	H		✓	✓	✓		
- Desktop Data - NewsEDGE	Horizontal	Groupware/Mail	M		✓	✓	✓	✓	
- Devon Systems International - Devon Securities	Vertical	Europe	M		✓	✓			
- Dialogic Corporation - Voice/Call Processing	Enabler	Other Enablers	M		✓	✓	✓	✓	
- Digital BCFI AB - Dec Bank FBS	Vertical	Europe	M		✓	✓			
- Digital Matrix Services	Horizontal	GIS Systems	M		✓	❑			
- Digital Tools Autoplan	Enabler	CASE Tool	M		✓				
- Digitalk Smalltalk	Enabler	CASE Tool	H		✓				
- DNA Star - LaserGene	Vertical	USA	M		✓	✓			
- Dodge Group - Accounting	Horizontal	Financial Mgmt.	M		✓	✓			
- Dragon - Report Writer	Horizontal	Analysis/Access	M		✓	✓			
- DRD Corp. - ARIS-D	Vertical	USA	M		✓	✓			
- Dun and Bradstreet - GL	Horizontal	Financial Mgmt.	H		✓	✓			
- Dynix - Dynix	Vertical	USA	M		✓	✓			

- Easel - Workbench for Windows NT	Enabler	CASE Tool	H		✓			
- EBT - Dyna-Tex	Vertical	USA	M		✓	❑		
- Edinburg Petro Svcs. Pansystem	Vertical	Europe	M		✓	✓		
- EDS - Unigraphics	Horizontal	Technical Horiz.	M		✓	✓		
- EDS Scicon - Setcim	Vertical	Europe	M		✓	✓		
- EDS/GDS - MicroGDS	Horizontal	GIS Systems	M		✓	✓		
- Effective Mgmt. Systems, Inc. - TCM-EMS	Vertical	USA	M		✓	✓		
- Effective Mgmt. Systems, Inc. - TCM-SFIS	Vertical	USA	M		✓	✓		
- EMATEK GmbH - GSS*GKS Kernel System	Enabler	Other Enablers	H		✓	✓	✓	✓
- EMATEK GmbH - GSS*GGDT CGI Metafile Drivers	Enabler	Other Enablers	M		✓	✓		
- Entropic Research Laboratory - ESPS	Vertical	USA	M		✓	✓		
- Entropic Research Laboratory - HTK	Vertical	USA	M		✓	✓		
- ESCA Corp. - Process Csontrol SCADA App.	Vertical	USA	M		✓	✓		
- ESRI - ARC/Info.	Vertical	USA	M		✓	❑		
- Evets Computers Intl. Ltd. - Ada	Vertical	Europe	M		✓	✓		
- Evets Computers Intl. Ltd. - Asset Scan	Vertical	Europe	M		✓	✓		
- Evets Computers Intl. Ltd. - Data Collector	Vertical	Europe	M		✓	✓		
- Evets Computers Intl. Ltd. - Sales Support	Vertical	Europe	M		✓	✓		
- Evets Computers Intl. Ltd. - Time Collector	Vertical	Europe	M		✓	✓		
- Excalibur Technologies - PixTex/EFS	Horizontal	Graphics/Imaging	M		✓	✓	✓	
- Executive Software - File Alert	Enabler	Other Enablers	H		✓	✓	✓	✓
- Exsys, Inc. - EXSYS Prof. Expert System Dev.	Vertical	USA	M		✓	✓		
- FCMC - Software Workflow	Horizontal	Groupware/Mail	M		✓	❑		
- Filenet	Horizontal	Graphics/Imaging	H		✓			
- Fisher/Rosemount - PROVOX	Vertical	USA	M		✓			
- Fisions/VG - VG/CGROM/LIM	Vertical	USA	M		✓	❑		
- Fluor Daniel, Inc. - CMMS Plus	Vertical	USA	M		✓	✓		
- FMC - Pacer	Vertical	USA	M		✓	✓		
- FocusSoft - ZEMAX Optical Design	Horizontal	Technical Horiz.	M		✓	✓	✓	✓
- FORTE	Enabler	CASE Tool	H		✓	❑		
- Fractel Design Corp. - Fractel Design Painter	Horizontal	Technical Horiz.	H		✓			
- Frame - FrameMaker	Horizontal	Publishing	H		✓	❑		
- Franklin Quest - Ascend	Horizontal	Groupware/Mail	M		✓			
- Franz Lisp (Allegro CL)	Enabler	Compilers	M		✓			
- FSL - IBEX LIM Broking	Vertical	USA	M		✓	❑		
- FTP Software - NFS	Vertical	USA	M		✓	✓		
- Gaylord - Gaylord Galaxy	Vertical	USA	M		✓	✓		

- GCS Comp. Serv. - GENASYS	Vertical	USA	M		✓	✓		
- Gemsoft - Sapphire	Vertical	USA	M		✓	✓		
- Genisis Software - Cabinet Manager	Enabler	Sys. Management	M		✓	✓	✓	✓
- Gensym - G2	Vertical	USA	M		✓	❑		
- GeoGraphix - GES	Vertical	USA	M		✓	✓		
- Gerber Alley - Report Writer	Vertical	USA	M		✓	✓		
- Gimpel Software - FlexeLint	Vertical	USA	M		✓	✓		
- Glassco Park - Microtool Math Library	Horizontal	Analysis/Access	M		✓			
- GMA, Geophysical Icro Computer Ap. - UNISYS	Vertical	USA	M		✓	✓		
- GNU Tools	Enabler	CASE Tool	M		✓			
- GP2 - GPIC	Vertical	Europe	M		✓	✓		
- GP2 - Ordobox	Vertical	Europe	M		✓	✓		
- Graphic M*I*S, Inc. - Fingraph III	Vertical	USA	M		✓	✓		
- GSI Business Management - Totas	Vertical	Europe	M		✓	✓		
- GSI Transcom - TOLARS	Horizontal	GIS Systems	M		✓	✓		
- Gupta SQL Windows App. Dev. Syst.	Enabler	CASE Tool	M		✓			
- H G Usher & Co. Ltd. - U-Contact	Vertical	Europe	M		✓	✓		
- Hamilton "C" Shell	Enabler	CASE Tool	H		✓	✓	✓	✓
- Hare Research - Felix & D-Space	Horizontal	Technical Horiz.	M		✓	✓		
- Harlequin - KnowledgeWorks	Vertical	Europe	M		✓	✓		
- Harlequin - LispWorks	Vertical	Europe	M		✓	✓		
- Harlequin - ScriptWorks	Vertical	Europe	M		✓	✓	✓	
- Harley Systems - Matrix Lims	Vertical	USA	M		✓	✓		
- Harley Systems - Peakmaster	Vertical	USA	M		✓	❑		
- Harley Systems - Quality Auditor	Vertical	USA	M		✓	❑		
- Harley Systems - Skylight	Vertical	USA	M		✓	❑		
- Health Systems Int. - HSSI	Vertical	USA	M		✓	✓		
- Helix Systems, Inc. - ResearchStation	Vertical	USA	M		✓	✓		
- Henco Software - Business Intelligent	Vertical	USA	M		✓	✓		
- Henco Software - Synchrony	Vertical	USA	M		✓	✓		
- Hilco Tech - Monitrol	Vertical	USA	M		✓	✓		
- Holistic Systems - EIS	Vertical	Analysis/Access	M		✓	✓		
- Honeywell - CM50 Garteway	Vertical	USA	M		✓	✓		
- Hoskyns Group PLC - Pinnacle	Vertical	Europe	M		✓	✓		
- Hypersoft - Browser	Vertical		M		✓	✓		
- IBI Focus	Enabler	Database	H		✓			
- IBM CICS (client)	Enabler	Other Enablers	H		✓			

- IBSY Finance S.A. - Ibsy	Vertical	Europe	M		✓	✓		
- ICS - BX	Enabler	CASE Tool	M		✓			
- Idaho National Eng. - Relap 5	Vertical	USA	M		✓	❑		
- IDE - Software-thru-Pictures	Enabler	CASE Tool	M		✓	❑		
- IDX - Clinical Workstation	Vertical	USA	M		✓			
- Imbsen & Associates - AASHTO BDS (Bridge Dsgn)	Vertical	USA	M		✓	✓		
- Imbsen & Associates - AASHTO IGrds	Vertical	USA	M		✓	✓		
- Impact Software - Icon Manager	Enabler	Other Enablers	M		✓			
- IMSL - Statistical Library for Windows	Horizontal	Analysis/Access	M		✓			
- IMSR - Hyperion	Horizontal	Financial Mgmt.	M		✓	✓	✓	✓
- Indigo Software - Report Smith	Horizontal	Analysis/Access	M		✓			
- Infinity Ints. - Montage	Vertical	USA	M		✓	✓		
- Informix - HyperScript Tools	Enabler	Database	H		✓	❑		
- Informix - Viewpoint	Enabler	Database	H		✓	❑		
- Ingres	Enabler	Database	H		✓	✓		
- Insight Access Group - Kubota 64-bit Demo	Enabler	Database	M		✓	✓		
- Int'l Biotechnologies - IBI Sequence analysis	Horizontal	Technical Horiz.	M		✓	✓		
- Int'l Biotechnologies - MAC Vector	Horizontal	Technical Horiz.	M		✓	✓		
- InterCAP - Illustrator	Horizontal	Graphics/Imaging	M		✓	❑		
- Intercom Data Systems, Ltd. - Helpdesk	Vertical	Europe	M		✓	✓		
- Intergraph - I/EMS	Horizontal	Technical Horiz.	M		✓	✓		
- Intergraph - Microstation	Horizontal	Technical Horiz.	M		✓	✓		
- Interleaf - Interleaf V6	Horizontal	Publishing	H		✓	❑		
- Intermetrics - ADA	Enabler	Compilers	M		✓			
- Iowa State University - GAMMES	Horizontal	Technical Horiz.	M		✓	✓		
- IPG - IPG-Car	Vertical	Europe	M		✓	✓		
- IPG - IPG-Driver	Vertical	Europe	M		✓	✓		
- IPG - IPG-Graph	Vertical	Europe	M		✓	✓		
- IPG - IPG-Movie	Vertical	Europe	M		✓	✓		
- IPG - IPG-Test	Vertical	Europe	M		✓	✓		
- IPG - IPG-Tire	Vertical	Europe	M		✓	✓		
- IPG - Mesa Verde	Vertical	Europe	M		✓	✓		
- Ippolis Informatique - Avtis	Vertical	Europe	M		✓	✓		
- Ippolis Informatique - Contis	Vertical	Europe	M		✓	✓		
- Ippolis Informatique - Multis	Vertical	Europe	M		✓	✓		
- Ippolis Informatique - Statis	Vertical	Europe	M		✓	✓		
- Ippolis Informatique - Biblis	Vertical	Europe	M		✓	✓		

- IRI Express	Horizontal	Analysis/Access	H		✓	❑		
- Island Graphics - Island/stuff	Horizontal	Graphics/Imaging	M		✓	❑		
- Ithica Software - HOOPS A.I.R.	Enabler	CASE Tool	H		✓	✓	✓	✓
- Ithica Software - HOOPS Graphics Dev. System	Enabler	CASE Tool	H		✓	✓	✓	✓
- Ithica Software - HOOPS I.M.	Enabler	CASE Tool	H		✓	✓	✓	✓
- ITS - PMIS	Vertical	USA	M		✓	✓		
- Jackson Lab - Genetic Mapping Workbench	Horizontal	Technical Horiz.	M		✓	✓		
- JJ Wild - Meditech Optical Mgmt. Application	Vertical	USA	M		✓	✓		
- Jostens Learning Corp - Josten's Learning Systems	Horizontal	Technical Horiz.	M		✓	✓		
- JYACC JAM	Enabler	CASE Tool	H		✓			
- K & T Electrical - General Mail Facility	Vertical	USA	M		✓	✓		
- Kapiti Ltd. - Fist	Vertical	Europe	M		✓	✓		
- Kapiti Ltd. - Fist Market Window	Vertical	Europe	M		✓	✓		
- KEAterm - terminal emulator	Enabler	Communications	M		✓	✓		
- Kenan Technologies - Acumate (R)	Vertical	USA	M		✓	✓		
- Kevin G. Barkes Consulting Service - Translate	Vertical	USA	M		✓	✓		
- Kiefer & Veittinger GMBH - Comm. Mgr.	Vertical	Europe	M		✓	✓		
- Kiefer & Veittinger GMBH - Sales Manager	Vertical	Europe	M		✓	✓		
- Kiefer & Viettinger GMBH - Service Manager	Vertical	Europe	M		✓	✓		
- Kodak - Raster Image Processing	Enabler	Other Enablers	M		✓	✓		
- Kuck - KAP	Enabler	Compilers	M		✓	✓		
- Kurzwell AI -VoiceMed	Enabler	Other Enablers	M		✓	✓		
- Large Scale Biology - Kepler Software	Vertical	USA	M		✓	✓		
- Lateiner Dataspace - VOX-L Navigator	Horizontal	Analysis/Access	M		✓	✓	✓	✓
- LBMS Process Engineer	Enabler	CASE Tool	H		✓	✓	✓	✓
- LBMS Systems Engineer	Enabler	CASE Tool	H		✓	✓		
- Leiden University - Forcheck	Vertical	Europe	M		✓	✓		
- Liant - C/C++	Enabler	Compilers	M		✓			
- Liant - PL/1	Enabler	Compilers	M		✓			
- Liant - RGP II and III	Enabler	Compilers	M		✓			
- Liant Fortran 85	Enabler	Compilers	M		✓			
- Liant/Ryan McFarlan COBOL	Enabler	Compilers	H		✓			
- Lightworks Designs Ltd. - Lightworks ADS	Vertical	Europe	M		✓	✓		
- Logitech - FotoMan	Horizontal	Graphics/Imaging	M		✓			
- Logitech - FotoTouch	Horizontal	Graphics/Imaging	M		✓			
- Logitech - ScanMan	Horizontal	Graphics/Imaging	M		✓			
- Lorentzian - Gaussian 92	Horizontal	Technical Horiz.	M		✓	✓		

- Lotus - Ami Pro	Horizontal	Publishing	M		✓			
- Lotus - CC:Mail	Horizontal	Groupware/Mail	H		✓			
- Lotus - Freelance	Horizontal	Graphics/Imaging	M		✓			
- Lotus - Improv	Horizontal	Groupware/Mail	M		✓			
- Lotus - NOTES - Client	Horizontal	Groupware/Mail	M		✓	❑		
- Lotus - NOTES - Server	Horizontal	Groupware/Mail	H		✓	❑		
- Lotus 1-2-3	Horizontal	Spreadsheets	H		✓			
- Lotus One Source - C/D Investment	Vertical	USA	M		✓	❑		
- LUCID C Developement Environment	Enabler	CASE Tool	M		✓			
- Lynx Realtime	Enabler	Compilers	M		✓			
- M.S.E. Ltd. - ANVIL-5000	Vertical	Europe	M		✓	✓		
- Mango Systems - Network Driver	Enabler	Communications	M		✓	✓	✓	✓
- Mark V Systems - ADA Gen	Enabler	CASE Tool	M		✓	✓	✓	✓
- Mark V Systems - C Gen	Enabler	CASE Tool	M		✓	✓	✓	✓
- Mark V Systems - COBOL Gen	Enabler	CASE Tool	M		✓	✓	✓	✓
- Mark V Systems - ObjectMaker	Enabler	CASE Tool	M		✓	✓	✓	✓
- Mark V Systems - ProcessMaker	Enabler	CASE Tool	M		✓	✓	✓	✓
- Marketing Profiles, Inc.	Horizontal	Financial Magmt.	L		✓	✓		
- Martin & Associates - CBIS - Telephone Billing	Vertical	USA	L		✓	✓		
- MASSTECK - MaxEDS PC Board Layout	Vertical	USA	M		✓	❑		
- Matra - Datavision	Horizontal	Technical Horiz.	H		✓	❑		
- MC-Tel San Monaco Telematique - Videomail	Vertical	Europe	M		✓	✓		
- MC-Tel San Monaco Telematique - Videomask	Vertical	Europe	M		✓	✓		
- MC-Tel San Monaco Telematique - Videonet	Vertical	Europe	M		✓	✓		
- MC-Tel San Monaco Telematique - Videonet-Dev	Vertical	Europe	M		✓	✓		
- MC-Tel San Monaco Telematique - Videotelefax	Vertical	Europe	M		✓	✓		
- MC-Tel San Monaco Telematique - Videotelex	Vertical	Europe	M		✓	✓		
- McCabe ACT/Battlemap	Enabler	CASE Tool	M		✓			
- McIntyre Cnslt. Inc. - Medimanager	Vertical	USA	M		✓	✓		
- McIntyre Cnslt. Inc. - MShell/Medit	Vertical	USA	M		✓	✓		
- MCS - Anvil V2, V3	Vertical	USA	M		✓	✓		
- MCS - Sector	Vertical	USA	M		✓	❑		
- MCSS, Inc. - OpenUpTime	Vertical	USA	M		✓	✓		
- MCSS, Inc. - OpenUpTime Support Desk	Vertical	USA	M		✓	✓		
- MDTV - Prelude	Vertical	USA	M		✓	❑		
- MEC - Mass-11	Horizontal	Publishing	M		✓	❑		
- Mechanical Dynamics - ADAMS	Vertical	USA	M		✓	✓		

- Mechanical Dynamics - ADAMSView	Vertical	USA	M		✓	✓		
- Meditech, Inc. - Magic His	Vertical	USA	M		✓	✓		
- Megabyte Ltd. - Consumer Analysis Sys.	Vertical	Europe	M		✓	✓		
- Megabyte Ltd. - Midas	Vertical	Europe	M		✓	✓		
- Mentor - Design Archtect	Horizontal	Technical Horiz.	M		✓	✓		
- Mentor Graphics - ECAD Suite	Horizontal	Technical Horiz.	H		✓	❑		
- Metaware C/C++ multiplatform tool	Enabler	Compilers	H		✓			
- Mfg. and Cnslt. Services, Inc. - ANVIL	Vertical	USA	M		✓	✓		
- MicroEdge - SlickEdit	Enabler	CASE Tool	H		✓	✓	✓	✓
- Microfocus COBOL	Enabler	Compilers	H		✓	✓		
- Microfocus MicroCICS	Enabler	Conversion Tool	H		✓	✓		
- Micrognosis, Inc. - Leading Market Tech.	Vertical	USA	M		✓	✓		
- Micrognosis, Inc. - Lotus Realtime	Vertical	USA	M		✓	✓		
- Microimages, Inc. - Map and Image Process.	Vertical	USA	M		✓	✓		
- Microimages, Inc. - TNT-MIPS (TM)	Vertical	USA	M		✓	✓		
- Microsoft Advanced Server	Enabler	Sys. Management	H		✓	✓	❑	
- Microsoft Word for Windows	Horizontal	Publishing	H		✓	✓		
- Microsoft "C" V7	Enabler	Compilers	H		✓	✓	✓	✓
- Microsoft Excel	Horizontal	Spreadsheets	H		✓	✓		
- Microsoft Flight Simulator	Vertical	USA	M		✓	❑		
- Microsoft Foxbase	Enabler	Database	H		✓			
- Microsoft Mail	Horizontal	Groupware/Mail	M		✓	✓		
- Microsoft Powerpoint	Horizontal	Graphics/Imaging	H		✓	✓		
- Microsoft Project	Horizontal	Groupware/Mail	M		✓	❑		
- Microsoft SNA Communications	Enabler	Communications	H		✓	❑		
- Microsoft SQL Server	Enabler	Database	H		✓	✓		
- Microsoft Test	Enabler	CASE Tool	H		✓	❑		
- Microsoft Videoworks	Enabler	Other Enablers	H		✓	❑		
- Microsoft Visual Basic	Enabler	Compilers	H		✓	❑		
- Microsoft Visual C/C++	Enabler	Compilers	H		✓	✓		
- Midwest Stock Exchange - Exchange System	Vertical	USA	M		✓	✓		
- Mincom - MIMS/Lattice/Geolog/Minescape	Vertical	USA	M		✓	❑		
- Minesoft - Techbase	Vertical	USA	M		✓	❑		
- Mintec - Medsystem	Vertical	USA	M		✓	✓		
- MIT - CECI - AthenaMuse2	Vertical		M		✓	✓		
- Mitech - M/Power-NT	Vertical	USA	M		✓	✓		
- Mitech - Navigator-CGI	Vertical	USA	M		✓	✓		

- Molecular Applications Group - Macimdad	Vertical	USA	M		✓	✓		
- Montage Group Ltd. - Montage Video Editor	Vertical	USA	M		✓	✓		
- MSC - Nastran	Vertical	USA	M		✓	❑		
- Munro Engineering - Argus	Vertical	USA	M		✓	❑		
- National Center For Biotechnology Inf. - Entrez	Vertical	USA	M		✓	✓		
- National Center For Biotechnology Inf. - Blast	Vertical	USA	M		✓	✓		
- National Center For Biotechnology Inf. - Medline	Vertical	USA	M		✓	✓		
- Natron Software Maintenance Ltd. - RTL/2	Vertical	Europe	M		✓	✓		
- NCS - Bondmaster	Vertical	USA	M		✓	✓		
- NCS - Series 7/11	Vertical	USA	M		✓	✓		
- NCS - U/Trust	Vertical	USA	M		✓	✓		
- NetManage - Chameleon 32A - TDP/IP Utilities	Enabler	Communications	M		✓	✓	✓	✓
- Netmaster - Netmanage	Enabler	Sys. Management	M		✓	✓	✓	
- Neuron Data - Nexpert	Enabler	CASE Tool	H		✓	❑		
- Neuron Data - Open Interface	Enabler	CASE Tool	H		✓	✓		
- Neuron Data - Windowbuilder	Enabler	CASE Tool	H		✓	❑		
- Nexcom Ltd. - TArget Hotline	Vertical	Europe	M		✓	✓		
- Novell - Netware Client	Enabler	Communications	H		✓	✓		
- Numeral Apps. - Gothic	Horizontal	Technical Horiz.	M		✓	✓		
- Numetrix - Linx	Horizontal	Technical Horiz.	M		✓	✓		
- Oasys - 680x0 Cross Compiler Tools	Enabler	CASE Tool	M		✓	✓	✓	✓
- Object Design - Object Store	Enabler	CASE Tool	M		✓	✓		
- Objective Interface Systems, Inc. - Screen Machine	Vertical	USA	M		✓	✓		
- Objectivity OODB	Enabler	Database	M		✓			
- Odesta - ODMS	Horizontal	Publishing	M		✓	✓		
- ODI - OODB	Enabler	Database	H		✓			
- Oil Systems, Inc. - PI-Application Prog.	Vertical	USA	M		✓	✓		
- Oil Systems, Inc. - Plant Information System	Vertical	USA	M		✓	✓		
- OMR Systems - Trading Assis.	Vertical	USA	M		✓	✓		
- Oracle	Enabler	Database	H		✓	✓	✓	✓
- Orcad - OrcadSC	Horizontal	Technical Horiz.	M		✓	✓		
- Orion Scientific - LEADS - Law Enforcement Sys.	Vertical	USA	M		✓	✓		
- P-Stat, Inc. - P-Stat	Vertical	USA	M		✓	✓		
- P.S.I. - UltraPlanner	Horizontal	Groupware/Mail	H		✓	✓		
- Pacific Northwest Laboratory - Argus	Horizontal	Technical Horiz.	M		✓	✓		
- PADS - Productivity Analysis	Horizontal	Analysis/Access	M		✓			
- PageAhead Software - InfoPublisher	Horizontal	Publishing	M		✓			

- Palette Systems - EBRS	Horizontal	Technical Horiz.	M		✓	✓		
- Paragon Imaging - Paragon ELT/2	Horizontal	Publishing	M		✓	✓		
- Paragon Imaging - Paragon ELT/2Dev. Toolkit	Horizontal	Graphics/Imaging	M		✓	✓		
- Paragon Imaging - Paragon Image Proc. System	Horizontal	Graphics/Imaging	M		✓	✓		
- Paragon Imaging - Paragon Imaging Library	Horizontal	Graphics/Imaging	M		✓	✓		
- Paragon Imaging - Visualization Workbench	Horizontal	Graphics/Imaging	M		✓	✓		
- Parametric Technology - ProEngineer	Horizontal	Technical Horiz.	H		✓	✓	✓	✓
- ParcPlace Objectworks Smalltalk	Enabler	Compilers	M		✓			
- Parkside Organization - S/26 Migration Tools	Enabler	Conversion Tool	H		✓	✓		
- PDV - Systeme GBMH - Apollo	Vertical	Europe	M		✓	✓		
- Pennington Systems Incorporated - XTRAN	Vertical	USA	M		✓	✓		
- Pentamation - Leadership, Open	Vertical	USA	M		✓	✓		
- People Soft - Payroll/HR	Horizontal	Financial Mgmt.	M		✓	✓		
- Performance Computing - Video Decompression	Enabler	Other Enablers	M		✓	✓	✓	✓
- Petrotechnics - Completion Mgr.	Vertical	USA	M		✓	✓		
- Pilot Software Ltd. - Command Center	Horizontal	Analysis/Access	M		✓	✓		
- Pilot Software Ltd. - LightShip	Horizontal	Analysis/Access	M		✓	✓		
- Pilot Software Ltd. - TimeServer	Horizontal	Analysis/Access	M		✓	✓		
- Platinum - SeQueL GL	Horizontal	Financial Mgmt.	H		✓	✓		
- Post Software International - Total STORE	Vertical	USA	M		✓	✓		
- PowerCore - Network Scheduler 3	Horizontal	Groupware/Mail	H		✓			
- Powersoft - Powerbuilder	Enabler	CASE Tool	H		✓	✓		
- Practice Technology - Docuphase Image Server	Horizontal	Graphics/Imaging	M		✓	✓		
- Precision Nesting Systems, Inc. - PINS	Vertical	USA	M		✓	✓		
- Premier - Global Plus	Vertical	USA	M		✓	✓		
- Pro Systems, Inc. - MTO	Vertical	USA	M		✓	✓		
- Process Software - TCPware	Vertical	USA	M		✓	✓		
- Professional American Security - SCE	Vertical	USA	M		✓	✓		
- Progress DB	Enabler	Database	H		✓	✓		
- Promark - Rhobot/Win	Vertical	USA	M		✓	✓		
- Promis - PROMIS	Vertical	USA	M		✓	✓		
- Prosig USA Inc. - DATSplus	Vertical	USA	M		✓	✓		
- Public Systems Associates - LEGIScribe for Wind.	Vertical	USA	M		✓	✓		
- Publishers Software Sys. - Subscription Mgmt.	Vertical	USA	M		✓	✓		
- QSM - QSM PADS	Vertical	USA	M		✓	✓		
- QSM - SLIM	Vertical	USA	M		✓	✓		

- Qstar Technologies, Inc. - ES/Backup	Vertical	USA	M		✓	✓		
- Raima	Enabler	Database	H		✓	✓		
- Raindrop Software - Software Engineer	Enabler	CASE Tool	M		✓			
- Rapid-Gen Systems Ltd. - The Genius Sol.	Vertical	Europe	M		✓	✓		
- Rasna - Mechanica Applied Motion	Horizontal	Technical Horiz.	M		✓	✓		
- Rasna - Mechanica Applied Structure	Horizontal	Technical Horiz.	M		✓	✓		
- Rasna - Mechanica Applied Thermal	Horizontal	Technical Horiz.	M		✓	✓		
- Ready Systems (Realtime)	Enabler	Compilers	M		✓			
- Real Time - General U/writing (LIM Motor)	Horizontal	Financial Mgmt.	M		✓	✓		
- Realtime Performance - RPCluster	Vertical	USA	M		✓	✓		
- Realtime Performance - RPCore	Vertical	USA	M		✓	✓		
- Realtime Performance - RPGem	Vertical	USA	M		✓	✓		
- Recital Corporation - RECITAL	Vertical	USA	M		✓	✓		
- Red Computer S.A. - Info-Red	Vertical	Europe	M		✓	✓		
- Research & Plng., Inc. - Long Sales Cyc. Mgr.	Vertical	USA	M		✓	✓		
- Research & Plng., Inc. - Profit. Ana. & Rpt. Sys.	Vertical	USA	M		✓	✓		
- Ross Systems - Accounting	Horizontal	Financial Mgmt.	M		✓	✓		
- SAP - SAP GL	Horizontal	Financial Mgmt.	H		✓	✓		
- Sarena - PCMODEL	Horizontal	Technical Horiz.	M		✓	✓		
- SAS Institute - SAS family of products	Horizontal	Analysis/Access	H		✓	☐		
- SASI - Ansys	Horizontal	Technical Horiz.	M		✓	☐		
- Schlumberger - Bravo	Horizontal	Technical Horiz.	H		✓	☐		
- Scientific Computing - Linda	Horizontal	Technical Horiz.	M		✓	✓		
- Scott & Scott - DPA/G	Vertical	USA	M		✓	☐		
- SDRC - Ideas	Horizontal	Technical Horiz.	M		✓	✓		
- Semichem (Univ. Missouri KC) AMOAC4.0	Vertical	USA	M		✓	✓		
- SFGL - East	Vertical	Europe	M		✓	✓		
- SGI CASEvision	Enabler	CASE Tool	M		✓	☐		
- Siemens Analytic - Shelxtl	Vertical	USA	M		✓	✓		
- Siemens Automation - Industrial PCs	Vertical	USA	M		✓	✓		
- Sierra - Stratlog	Vertical	USA	M		✓	✓		
- Sigma Design - ARRIS	Vertical	USA	M		✓	✓		
- Silicon Valley Software - Fortran 77	Enabler	Compilers	M		✓			
- Smallworld Systems Ltd. - GIS	Horizontal	GIS	M		✓	✓		
- Smart Tools - OCR	Enabler	Other Enablers	M		✓			
- Smartstar	Enabler	4 GL	H		✓	✓		
- Smartsystems (UK) Ltd. - SmartStar	Vertical	Europe	M		✓	✓		

- Softdesk - SOFTDESK/XXX	Horizontal	Technical Horiz.	M		✓	✓		
- Softool - CCC	Enabler	CASE Tool	M		✓			
- Software AG	Enabler	Database	H		✓	✓		
- Software Mntce. & Deve. Sys. Inc. - Aide-De-cp.	Vertical	Europe	M		✓	✓		
- Software Mntce. & Deve. Sys. Inc. - Lakota Ap.	Vertical	Europe	M		✓	✓		
- Software Mntce. & Deve. Sys. Inc. -ADC/Ada	Vertical	Europe	M		✓	✓		
- Software Mntce. & Deve. Sys. Inc. -Lakota	Vertical	Europe	M		✓	✓		
- Software One Ltd. - Software One Exchange	Vertical	Europe	M		✓	✓		
- Software Publishing Corp - Harvard Graphics	Horizontal	Graphics/Imaging	M		✓	❑		
- Software Works, The - The Software Works	Vertical	USA	M		✓	❑		
- Softwr Maintenance & Dev. - ADA Scan	Enabler	CASE Tool	L		✓	✓	✓	✓
- Softwr Maintenance & Dev. - ADA Scan	Enabler	CASE Tool	L		✓	✓	✓	✓
- Softwr Maintenance & Dev. - Lakota App. Mgmt	Enabler	CASE Tool	L		✓	✓	✓	✓
- Softwr Maintenance & Dev. Aide-de-Camp	Enabler	CASE Tool	L		✓	✓	✓	✓
- SOS Software & Services Ltd. - Absencemaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Fleetmaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - HeRMeS	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Paymaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Recruitmaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Safetymaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Staffmaster	Vertical	Europe	M		✓	✓		
- SOS Software & Services Ltd. - Trainingmaster	Vertical	Europe	M		✓	✓		
- Spatial Tech. - ACES	Horizontal	Technical Horiz.	M		✓	❑		
- Speakeasy	Horizontal	Analysis/Access	M		✓	✓	✓	
- Spectrum Computer Systems PLC m- Cash Of.	Vertical	Europe	M		✓	✓		
- Spectrum Graphics S.P.A. - ICARO	Vertical	Europe	M		✓	✓		
- SPSS for Windows	Horizontal	Analysis/Access	M		✓			
- SQL*Builder Sosftware Company - SQL*Builder	Vertical	USA	M		✓	✓		
- Square D Co./CRISP Auto. Systems. - CRISP	Vertical	USA	M		✓	✓		
- SSI - Workbench	Horizontal	Technical Horiz.	M		✓	✓		
- St. Vincent's - Cliniplex	Vertical	USA	M		✓	✓		
- Stereographics - Crystal Eyes Virtuality Reality	Enablers	Other Enablers	M		✓	✓		
- Stirling EDI	Enabler	Other Enablers	H		✓			
- Stone & Webster ASDS, Inc. - Prod. Sch. Adv.	Vertical	USA	M		✓	✓		
- Strategic Systems International - TROPOS	Vertical	Europe	M		✓	✓		
- Structural Research & Analysis - CM Designer	Horizontal	Technical Horiz.	M		✓	✓		
- Structural Research & Analysis - CM Engineer	Horizontal	Technical Horiz.	M		✓	✓		

- Structural Research & Analysis - COSMOS\M	Horizontal	Technical Horiz.	M		✓	✓		
- Sun Gard - Global Securities Mgr.	Vertical	USA	M		✓	✓		
- SUN PC NFS	Enabler	Communications	H		✓	❑		
- Sybase - DEFT CASE Tools	Enabler	Database	H		✓	❑		
- Sybase - Gain Technology Multi-Media	Enabler	Database	H		✓	❑		
- Sybase - Interoperability Products	Enabler	Database	H		✓	❑		
- Sybase - SQL Advantage	Enabler	Database	H		✓	❑		
- Sybase - SQL Companion	Enabler	Database	H		✓	❑		
- Sybase - SQL Debug	Enabler	Database	H		✓	❑		
- Sybase - SQL Toolset	Enabler	Database	H		✓	❑		
- Symantec - Zortech C/C++	Enabler	Compilers	M		✓			
- Symbolics - Concordia	Enabler	CASE Tool	M		✓	❑		
- Symbolics - Genera	Enabler	CASE Tool	M		✓	❑		
- Symbolics - Joshua	Enabler	CASE Tool	M		✓	❑		
- Symicron Computer Com. Ltd. - Symdrive-nt	Vertical	Europe	M		✓	✓		
- Sysdrill Ltd. - IDEAS	Vertical	USA	M		✓	✓		
- System Software Assoc. - MRP II	Vertical	USA	L		✓	❑		
- System Software Associates - BPCS\WS Planners	Enabler	Other Enablers	M		✓	❑		
- Systemetrics - PageMate (paging software)	Enabler	Other Enablers	M		✓	✓	✓	✓
- Systems Union Ltd. - SunAccount Corp.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunAccount Fixed Asset	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunAccount Ledger	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunAccount Multi-Curr.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunBusiness Inventory	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunBusiness Pch. Inv.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunBusiness Pch. Ord.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunBusiness Sales Inv.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunBusiness Sales Odr.	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunQuery Report Write	Vertical	Europe	M		✓	✓		
- Systems Union Ltd. - SunSystems Sunlink	Vertical	Europe	M		✓	✓		
- T/One - Merlin	Vertical	USA	M		✓			
- Tactics International - Tactician	Horizontal	Analysis/Access	M		✓	✓	✓	✓
- TASC - TASC/MLSPLUS	Vertical	USA	M		✓	✓		
- Teknekron - PORTOLA	Horizontal	Technical Horiz.	M		✓	✓		
- Teknekron - TIBLINK	Horizontal	Technical Horiz.	M		✓	✓		
- Tema Studio Di Informatica - sT-Engine	Vertical	Europe	M		✓	✓		
- Tema Studio Di Informatica - sT-Mail	Vertical	Europe	M		✓	✓		

- Template Graphics - FIGT - PHIGS/PEX	Enabler	CASE Tool	M		✓				
- Template Graphics - PHIGS Development Kit	Enabler	CASE Tool	M		✓				
- Template Graphics - SNAP	Enabler	CASE Tool	M		✓				
- Texas Instruments - D3/S3	Horizontal	Technical Horiz.	M		✓	❑			
- Texas Instruments - IEF	Enabler	CASE Tool	H		✓	✓			
- The Dodge Group - OpenSeries Financials	Horizontal	Financial Mgmt.	M		✓	✓			
- The Sombers Group, Inc. - NetWeave	Vertical	USA	M		✓	✓			
- Tower Mountain S/W - TEMPO	Vertical	USA	M		✓	✓			
- Tradeware Technology S.A. -Toptrader	Vertical	Europe	M		✓	✓			
- Triadigm Technology - MediaPoint/Teale	Horizontal	Publishing	M		✓	✓			
- Triangle Ernst & Young - Eyes	Horizontal	Financial Mgmt.	M		✓	❑			
- Trilogy Enterprises - BPS IWS Planner's Asst.	Vertical	USA	M		✓	✓			
- Trusted Information Systems - T-MACH	Vertical	USA	M		✓	❑			
- Twinsoft - AS/400	Enabler	Conversion Tool	H		✓	❑			
- Unidata - DBMS	Enabler	Database	H		✓	✓	✓		
- Uniface - Application Generation Tools	Enabler	4 GL	H		✓	✓	✓	✓	
- Universal Instruments - Assembly Equipment	Vertical	USA	M		✓	❑			
- Universal Translated Systems - Lang. Conv.	Vertical	USA	M		✓	✓			
- Urban Analysis Group, The - TRANPLAN	Vertical	USA	M		✓	✓			
- Vector Network Limited - Lanutil for Pathworks	Vertical	Europe	M		✓	✓			
- Vendata - Enforcer 2000	Enabler	Other Enablers	M		✓	✓			
- Ventura - PicturePro	Horizontal	Graphics/Imaging	H		✓	✓			
- Ventura - Publisher	Horizontal	Publishing	H		✓	✓			
- Verbex Voice - Serv. Creation Envir. (SCE)	Enabler	Other Enablers	M		✓	✓			
- Verbex Voice Processing	Enabler	Other Enablers	H		✓	✓	✓		
- Verdix - Meridian - Open ADA	Enabler	Compilers	H		✓				
- Verity, Inc. - TOPIC, (R)	Vertical	USA	M		✓	✓			
- VI Corp. - DataViews	Horizontal	Analysis/Access	M		✓	✓			
- Viewlogic - ECAD Suite	Horizontal	Technical Horiz.	H		✓	❑			
- Viewlogic - PowerView	Horizontal	Technical Horiz.	H		✓	✓			
- VIS, Inc. -- VISIp CICS emultation environ.	Enabler	Conversion Tool	M		✓				
- VisiSoft - VisiNET	Enabler	Sys. Management	M		✓	✓			
- Visix - Galaxy Application Builder	Enabler	CASE Tool	H		✓	✓	✓	✓	
- Vista Control Systems - Vsystem	Vertical	USA	M		✓	✓			
- Visual Numerics - PV Wave	Horizontal	Analysis/Access	H		✓	✓	✓	✓	
- Visual Solutions, Inc. - VisSim (tm)	Vertical	USA	M		✓	✓			
- VXM - Pax-2	Enabler	Sys. Management	M		✓	✓			

- Wall Data - RUMBA - IBM 3270 TE	Enabler	Communications	M		✓	✓	✓	✓
- WANG Image Managmenet Solutions	Horizontal	Graphics/Imaging	M		✓			
- Washington University - Comp Chem & Biomed	Horizontal	Technical Horiz.	M		✓	✓		
- Washington University - Fast Datafinder	Horizontal	Technical Horiz.	M		✓	✓		
- Watcom C/C++	Enabler	Compilers	M		✓			
- Watcom Fortran	Enabler	Compilers	H		✓			
- Waterloo S/W - MapleV	Vertical	USA	M		✓	✓		
- Watermark - Image Enabler	Horizontal	Graphics/Imaging	H		✓	✓	❑	
- Wavefunction - Spartan	Vertical	USA	M		✓	✓		
- Waveteck - Wavetest	Vertical	USA	M		✓	❑		
- Welch Allyn Inc. - Medical Diagnostics	Vertical	USA	M		✓	❑		
- Welch Laboratory, John Hopkins - OO User Intfce.	Vertical	USA	M		✓	✓		
- Welch Medical Applied Research Lab - Genome	Vertical	USA	M		✓	✓		
- Welch Medical Applied Research Lab - Omim	Vertical	USA	M		✓			
- Welcom Software - OPEN PLAN	Horizontal	Groupware/Mail	H		✓	✓		
- Welcom Software - TEXIM	Horizontal	Groupware/Mail	H		✓	✓	✓	
- Welty-Leger Corporation - DCM-Master	Vertical	USA	M		✓	✓		
- Wilson WindowWare - WinBadge	Enabler	CASE Tool	M		✓	✓		
- Wilson WindowWare - WinEdit	Enabler	CASE Tool	M		✓	✓		
- Wind River Cross Compilers	Enabler	Compilers	M		✓			
- Wolfram Research, Inc. - Mathematica	Horizontal	Analysis/Access	M		✓	✓		
- Word Perfect - Word Perfect Office	Horizontal	Groupware/Mail	M		✓			
- Word Perfect - Word Perfect	Horizontal	Publishing	H		✓	❑		
- Workflow System Inc. - FlowLogic (tm)	Vertical	USA	M		✓	✓		
- World Fine Art Exchange - FIMS/FIMS Plus	Vertical	USA	M		✓	✓		
- WRQ - Reflection - HP/VT/ASCII	Enabler	Communications	M		✓	✓	✓	
- Xcellerate Systems - MAC Emulation	Enabler	Conversion Tool	M		✓			
- Xdb	Enabler	Database	M		✓			
- Xtensory - Toolkit	Enabler	Other Enablers	M		✓	✓		
- Xtensory Inc. - XVS-PC	Vertical	USA	M		✓	✓		
- XVT Software - Portability Tool	Enabler	CASE Tool	M		✓			
- Xybion - Path/Tox, Cell Analysis, Microscopy SW	Vertical	USA	M		✓	✓		
- Zinc Application Framework	Enabler	CASE Tool	M		✓	❑		
Total ——————————>	■■■■■				726	504	61	43

32-bit Alpha AXP PALcode for the Windows NT Operating System

Revision 0.1

Author: Joe Notarangelo, decwet::notarangelo
Engineering Manager: Benn Schreiber, univax::bls

NT Systems Group
Digital Equipment Corporation
Bellevue, Washington

Warning:

The interfaces documented in this specification are for provision by processor/PAL providers and the low-level system maintainers.

These interfaces should not be called directly, even in kernel mode. Public interfaces are exported for those functions that would be necessary for execution outside of the lowest levels of the kernel.

As these interfaces are considered private and tightly coupled to the operating system, we anticipate that they will change as necessary to support NT.

Acknowledgments

The following individuals provided advice, information, review, and code examples that contributed to the definition of this specification:

Miche Baker-Harvey	Digital Equipment Corporation, NT Systems Group
Dave Conroy	Digital Equipment Corporation, SEG/AD
David Cutler	Microsoft Corporation, Portable Systems Group
John DeRosa	Digital Equipment Corporation, NT Systems Group
Jeff East	Digital Equipment Corporation, NT Systems Group
Rod Gamache	Digital Equipment Corporation, NT Systems Group
Steve Jenness	Digital Equipment Corporation, NT Systems Group
Jeff McLeman	Digital Equipment Corporation, NT Systems Group
Ed McLellan	Digital Equipment Corporation, SEG/AD
Stephan Meier	Digital Equipment Corporation, SEG/AD
Steve Morris	Digital Equipment Corporation, SEG/AD
Dirk Meyer	Digital Equipment Corporation, SEG/AD
Lou Perazzoli	Microsoft Corporation, Portable Systems Group
Tom Van Baak	Digital Equipment Corporation, NT Systems Group
Rich Witek	Digital Equipment Corporation, SEG/AD *(formerly)*

Revision History

2-Dec-1992	Internal group review
3-Aug-1993	Incorporate changes to reach V1.0

0. Preface

Notational Conventions

The symbolic notation used in this document follows essentially the same conventions as those of the Alpha AXP Architecture SRM and the Alpha AXP Architecture Handbook.

The following additional conventions are also used:

1. The C language convention for specifying radix is followed: all numbers are decimal except those preceded by "0x" which indicates an hexadecimal number. (Octal is not used.)
2. Code examples are listed in an ordered sequence. The code examples may be re-ordered such that the results of the sequence of instructions are not altered. In particular, if an instruction j is listed subsequent to an instruction i and i writes any data that is used by j then i must be executed before j.

Related Documents

The following documents are considered part of this specification. In cases where discrepancies exist between this specification and the following documents, this specification will be considered to supersede the others.

Alpha AXP Architecture SRM V5.0
Alpha AXP Architecture Handbook
NT/Alpha AXP Calling Standard
NT/Alpha AXP Hardware Abstraction Layer Specification
ARC Firmware Specification (what is real name?)

Document Structure

This document is conceptually divided into four major sections: Introduction, Discussion, Reference, and the Appendicies.

Introduction:
The first section, comprising chapters 1 and 2 provide an overview and the context necessary to understand the rest of the specification. They are not actually part of the specification in that they do not specify anything but instead explain.

Discussion
The second section, comprising chapters 3 through 25, discusses each of the areas of functionality that the processor/PAL code must address. These sections describe how different functions work together to satisfy an area of functionality and specify the details for non-call pal functionality.

Reference
Chapters 26 and 27 are reference chapters that specify the details for the call pal functions and the virtual internal processor registers, respectively.

Appendicies
The appendicies specify the particular values and address offsets that are used as constant definitions in the body of the specification (Discussion and Reference).

The call pal functions are a fundamental basis of the specification. Each call pal function is discussed twice: once in the context of the general topic area that the function supports and once in the alphabetized call pal listing that comprises a majority of this specification. A similar approach has been taken with the virtual internal processor registers: they are listed with the topics they support and in a separate section of their own.

Audience

The intended audiences for this document are the following:

1. Alpha AXP processor designers
2. PALcode authors for new processor implementations
3. Maintainers of the Alpha AXP Architecture-specific portions of the Windows NT kernel

In addition, the specification is provided for review of the 32-bit PALcode for architectural correctness and completeness. Readers are expected to be familiar with the Alpha AXP Architecture SRM.

1. Overview/Philosophy/Religion

NT is NT is NT

The Alpha AXP implementation of NT is designed to look and feel like NT on any other architecture. Compatibility with NT was the primary design goal for the NT Systems Group which ported NT to the Alpha AXP architecture. Wherever, a discrepancy was found between the way in which NT handled a problem versus other Alpha AXP implementations, those discrepancies were resolved in NT's favor. The PALcode that is part of the Alpha AXP architecture provided us with a flexible mechanism to mold the processor interface to one that would as simply and cleanly (without performance loss) accommodate NT. The goal was to make the processor appear to have been especially designed for NT, in as much as that was possible. The PALcode support for NT is different from other implementations in that the operating system is designed to run software according to standardized application binary interfaces over which Digital has no influence.

Origins

The PALcode for NT Alpha AXP is based in spirit upon the Alpha AXP/OSF PALcode that was started by Rich Witek. The NT PALcode uses similar mechanisms for exception dispatch and includes many of the same, or very similar, call pals as does the OSF PAL.

The NT PAL diverges from OSF in several fundamental ways:

1. The NT PALcode is a 32-bit implementation
2. The NT PALcode builds NT trap frames when trapping to the kernel
3. The NT PALcode is designed to be platform-independent

1. 32-bit PAL implementation
The NT PALcode supports a 32-bit virtual address space. The page table entries are 32-bits. The 32-bit virtual address space does not take advantage of Alpha AXP's 64-bit virtual address because for now NT is a 32-bit operating system. The 32-bit PTEs do not match the internal Alpha AXP representation and so the PALcode must dynamically reformat the PTEs within the TB miss flows. Currently, NT has set expectations that PTEs are 32-bits -- we used the PALcode to bring Alpha AXP to NT rather than changing NT to run on Alpha AXP machines.

2. NT trap frames
The NT PALcode cooperates closely with the lowest-level kernel exception dispatch by providing saved state in a format that can be used throughout NT. Generally, this means that more state is saved in the NT PAL and that NT status codes are directly derived and forwarded by the PALcode.

3. Platform independence
The NT PALcode is designed to be entirely platform-independent so that one version of the PALcode (for a particular processor implementation) will run on all systems. (More on this topic below.)

Binary Compatibility

The NT Alpha AXP operating system binaries are designed to be binary compatible across different processor implementations. Essentially, this implies that the difference between processors will be entirely hidden by the PALcode for that implementation. This specification then defines the interfaces to which the combination processor-implementation/PALcode must conform to support the binary compatible operating system. The NT Systems Group believes that adherence to this goal will give the Alpha AXP architecture forward compatibility that will enhance Digital's credibility in a commodity

market place. The ability of each new processor to adapt itself to the binary interface via a small layer of software may be a significant advantage for the future of the architecture.

Currently, there are limitations upon the binary compatibility of the operating system for Alpha AXP systems. The two current limitations are implementation page size and multi-processor support. Page size is currently defined as a constant for performance reasons. Page size could be redefined as a variable (though it may be necessary to define an interface routine for accesses from device drivers). The Alpha AXP page size is already larger than most architectures (8K vs. 4K) and so we anticipate that for the next several years a constant 8K page size is a reasonable limitation for binary compatibility. Multi-processor support is also a compile-time decision. Therefore, there will be one kernel for uniprocessor systems and a separate kernel for multi-processor systems. Once again, it would be possible to unite both kernels by forcing the uniprocessor kernel to implement full multi-processor support. We have followed Microsoft's lead by compiling 2 separate kernels.

Platform Independence

The NT PALcode is designed to be entirely platform-independent. The PALcode is viewed more as a component of the kernel (dynamically loaded based on the processor) rather than as embedded micro code specific to the particular platform. The NT Systems Group believes that this will give us several advantages over the life of the program.

First of all, we expect that the market for NT Alpha AXP systems will be an open market. System design teams may build these machines across the world in many different companies. For each processor we (Digital) build we will write a single PALcode to support the operating system. All systems that use the processor will run the same PALcode. There are several advantages that follow from a single PALcode per processor:

1. Operating system PALcode will not be a barrier for system vendors that wish to enter the NT Alpha AXP Market
2. There will be no need to support obscure bugs or performance problems in other vendors PALcode
3. If we ever need to change the binary interface to the PALcode we will not need to coordinate with an arbitrarily long and possibly incomplete list of Alpha AXP system vendors

The NT PALcode is designed to be de coupled from the firmware PALcode environment as well. The NT PALcode image is read from the disk during the boot process just like every other component of the running operating system. One reason for this de coupling is to not burden the firmware with the requirement to run with the "correct" PALcode when booting one of the N operating systems the system might support. The de coupling gives the system team the freedom to implement any manner of PALcode they choose to support the firmware environment (for examples, (1)un-optimized, very simple PALcode or (2)another operating system's PALcode). The firmware PALcode need only support two of the common PAL interfaces: instruction memory barrier and swap pal to allow the operating system to be loaded and the NT PALcode initialized.

Note that some functions and parameters must be implemented on a per-platform basis. Platform-dependent functions are implemented in the HAL (hardware abstraction layer) which is a system-specific library loaded and dynamically linked at boot time. The design of a platform-independent PALcode definition and binary compatible kernel with system-dependent functions in the HAL is based on David Cutler's work for NT on the MIPS architecture.

Other Design Considerations

The PALcode was designed to work smoothly with NT (for example, building an NT trap frame and passing NT status codes) but was also designed to keep the dependencies to a minimum. There are only 2 data structures that the PALcode shares with the operating system: the PCR (processor control region)

and the kernel trap frame definition. Wherever possible, parameters and return values are passed in registers when the kernel and PALcode communicate.

First Implementation

The first implementation of the NT Alpha AXP PALcode is written for the DECchip 21064-aa processor. A seperate specification will document the implementation of the PALcode for the 21064-aa processor. In the future, a specification must be written for each new processor-implementation of the architecture.

2. HAL, Pal, Kernel, OS Loader and Firmware

This section describes how the HAL(Hardware Abstraction Layer), PAL, Kernel, OS Loader and Firmware cooperate in order to provide system-and processor-specific functions so that the kernel may be binary compatible across different platforms and different processor implementations. The description is an overview -- it is beyond the scope of this specification to cover detailed technical information for all of these components. The first components used in the boot sequence, the firmware and OS loader, are responsible for establishing the environment in which the kernel, HAL and PAL will execute.

Firmware

The firmware contributes the following important components to the boot sequence:

1. Maintains the environment in which the OS loader executes (provides I/O services, PAL functions for imb and swppal)
2. Creates the configuration database: devices, memory size etc.
3. Reads the OS Loader from the disk and executes it

OS Loader

The OS Loader is a linking loader that reads the component operating system images from the disk, performs necessary relocation, and binds the dynamically linked images together. The OS Loader loads the appropriate HAL and PAL based on the configuration information provided by the firmware. In addition, the OS Loader will load the appropriate boot drivers as read from the operating system configuration files. The OS Loader also builds the loader parameter block structure by using information provided by the firmware. The loader parameter block includes configuration information (processor, system, device and memory configuration) and per-processor data structures. Once the operating system components are loaded, the OS Loader jumps to the beginning of the kernel to begin execution of the operating system. The OS Loader will load the operating system PALcode on a 64K byte aligned address. The kernel will activate the operating system PALcode by executing the swppal instruction.

PALcode

The PALcode is specific to a particular processor implementation and must hide the internal workings of the processor from the kernel. The PAL performs TB management and first level exception dispatch among other functions. The PALcode for a particular processor may include per-processor functions but they must only be called by the HAL.

Hardware Abstraction Layer (HAL)

The HAL provides the system-specific layer between the kernel and the system hardware. The HAL provides interfaces for the following types of operations:

1. Interrupt handling, including dispatch and acknowledge
2. DMA control
3. Timer support
4. Low-level I/O support
5. Cache coherency

If a processor implementation requires PAL intervention to support any of these functions then it may provide call pals to supply the needed services. *These processor-specific functions must be provided in a system-independent manner.*

Kernel

The kernel is a binary compatible image that can run on any Alpha AXP processor and on any Alpha AXP system. The kernel does its job by using the standard interfaces provided for the PAL and the HAL and by reading the configuration information. The interfaces that the kernel expects from the PALcode are documented in this specification. The HAL interfaces are specified in a separate document.

3. Initialization

From the perspective of the PALcode environment there are 4 phases of initialization:

1. Internal processor state established before the PALcode runs
2. Initialization of the internal processor state by the PALcode
3. Initialization callbacks used by the kernel to prepare the PAL to handle exceptions
4. Initialization of the interrupt tables so that standard interrupt support can be used

Pre-PALcode Initialization

The PALcode expects the processor and system to set to a known good state by firmware before the PALcode entry point is called. The firmware must initialize any internal processor registers that contain system-specific parameters such as timing or memory size information. This is necessary because the PAL is entirely independent of the system. The firmware must guarantee that all caches are coherent with main memory before calling the PAL and that the memory system has been fully initialized.

> *Implementation Note (Hardware):*
> *If system configuration information is written to write-only IPRs, then those configuration IPRs cannot have any control bits that need to be written by the platform-independent operating system PALcode. If this were done, then the firmware will have to pass the configuration information in internal processor state on a per-implementation basis. Hardware designers should consider allowing configuration registers to be read as well as written to allow the platform-independent layer to have visibility to the full internal processor state.*

PALcode Initialization

The PALcode will be called at its first instruction which will be the base of the PALcode image. This will be the reset vector for the PAL. The PALcode is not called with any parameters and essentially must act as a standard subroutine for purposes of initialization. Namely, the PAL is free to destroy volatile general purpose integer and floating registers but must preserve the non-volatile register state across the call. The non-volatile state that must be preserved is listed in the register definition table in an appendix. The PALcode must accomplish the following initialization:

1. Deassert all interrupt requests and disable all interrupt enables (this includes software, hardware and asynchronous trap interrupts)
2. Set the processor to kernel mode
3. Set the processor status register (PSR) such that: interrupts are enabled, interrupt request level is set to high level (7), and mode = kernel
4. Flush all translation buffers
5. Establish all required super-page mappings: 32-bit i- and d-stream, and 43-bit d-stream mappings

6. Set the PAL_BASE register to the base address of the PALcode image
7. Set the interrupt level table so that all interrupt enables are off for all interrupt levels
8. All architected internal processor registers initialized to their specified initialization values
9. Any implementation-specific initialization that is required, for example unlocking error registers
10. Set the PREVIOUS_PAL_BASE register to the previous value of the PAL_BASE register

When the PALcode has completed its initialization it resumes execution at the address passed in the ra (return address) register.

Kernel callback initialization of PALcode

The kernel uses privileged initialization call pals to call back into the PALcode with the initialization values that will allow exceptions to be handled properly between the PAL and the kernel. Two privileged call pals are provided for this initialization: initpal and wrentry.

Initpal is used to establish per-processor context for the PAL, system-permanent context, and per-thread context for the initialization thread. The per-processor context established for the PAL are the processor control registers (PCR) and the processor control block (PRCB). These addresses are passed to the PAL as standard arguments. The addresses must be 32-bit super-page addresses. The system-permanent context passed to initpal is the kernel global pointer (gp) which is passed via the integer general purpose register gp. The initialization thread data passed in initpal are the page directory page, the initial kernel stack pointer, and the initialization thread address. The page directory page and thread address are passed as standard parameters while the kernel stack pointer is passed in the general purpose integer register sp. The PALcode will also initialize the PAL information section of the PCR as part of the initpal call pal function.

The kernel uses the wrentry call pal to register the kernel exception entry points with the PALcode. The wrentry call pal must be called once for each kernel exception entry point. Each call includes the exception entry point address and the number of the exception class it will handle.

Interrupt Table Initialization

The Interrupt tables in the PCR are system-specific and so are not initialized until HAL initialization. Until these tables are initialized the PALcode will be using interrupt tables that are initialized such that all interrupts are disabled. An implementation may choose to cache some portion of the interrupt tables within the processor. If an implementation does cache the interrupt tables then it must provide implementation-specific call pal(s) to allow the HAL to resynchronize the cached tables with the values written to the PCR.

4. Per-processor data areas

The operating system has two per-processor data structures. The processor control registers (PCR) is a one-page data structure that is used to store information that may be specific to a particular architecture. This type of information would be data that is shared between the PAL, the HAL, and/or the architecture-specific portions of the kernel. The processor control block (PRCB) is a software structure that would include data that would not be accessed by the PAL or HAL. The PRCB is a standard structure and therefore need not require an entire page.

Processor Control Registers(PCR)

The PCR contains many fields that are of importance to the PALcode. In the first place, a 2K region in the PCR is reserved for PALcode use, this is the only per-processor data region available to the PAL. The PCR also includes the interrupt level table (ILT) which maps the interrupt enable masks for each

possible interrupt request level. The PALcode may continually read these masks from the PCR or may read them once and cache them inside the processor. The PCR also includes the interrupt dispatch table (IDT) which contains the address of an interrupt handler for each possible interrupt vector. The interrupt mask table (also in the PCR) maps each possible pattern of interrupt requests to the highest priority interrupt vector and the corresponding synchronization level. The PCR also contains the panic stack pointer. In addition, the PALcode is responsible for initialization the PAL base address field and several PALcode revision fields within the PCR. An Appendix is included to describe the fields that would be of interest to the PALcode and to define their offsets and sizes within the PCR.

The address of the PCR is accessible via the privileged call pal rdpcr. The value returned is the address supplied to the PALcode in the initpal call pal. Initpal is the only mechanism supplied to write the PCR address.

Some of the data items that conceptually belong in the PCR are accessible by call pal functions. These functions allow atomic access to data that is logically in the PCR. In addition, these values can be retrieved with a single call pal thereby eliminating the need to use the rdpcr call pal followed by a load instruction. The operating system isolates all accesses to these data items by macros and routines so that the kernel may choose the best manner to access this data (see fixed PCR issue below). The data items that logically are part of the PCR are the current thread, the DPC flag, and the Processor Control Block (PRCB). The thread environment block (TEB) could also be logically stored in the PCR.

Processor Control Block (PRCB)

The privileged call pal rdprcb is supplied to allow the kernel to quickly retrieve the pointer to the current processor's processor control block. Like the PCR, the address returned by rdprcb is written only by the initpal call pal.

PALcode Version Control

The PALcode is responsible for populating version information in the PCR. Some of the version information is for maintenance and debugging purposes, while other version information is used by the kernel for check-pointing with the PALcode. The PalMajorVersion, PalMinorVersion, and PalSequenceVersion are provided for maintenance and debugging. The PALcode is responsible for populating these fields but the values are implementation-specific. The PalMajorSpecification and PalMinorSpecification are used by the kernel for check-pointing with the PALcode. This document defines the current major and minor PALcode specification version numbers. The PALcode will populate the specification fields with the version numbers that correspond to the version of this specification to which the PALcode image complies. Minor revisions within the same major revision will be backward compatible. The kernel will read the PalMajorSpecification and determine if it is compatible with the version of the PALcode. If the kernel is not compatible (the PalMajorSpecification is greater than the kernel's expected PAL major specification) then the kernel will run-down in a controlled manner.

5. Memory Management

Virtual Address Space

NT/Alpha AXP as a 32-bit implementation has a 32-bit virtual address space represented in the following table:

Figure 1. Virtual Address Map

Address Range (32 bits)	Permission	Description
0x00000000 :0x7fffffff	User and Kernel	General user address space
0x80000000 : 0xbfffffff	Kernel	Non-mapped kernel space (32-bit super-page)
0xc0000000 : 0xc1ffffff	Kernel	Mapped, page table space
0xc2000000 : 0xffffffff	Kernel	Mapped, general kernel space

The address map takes advantage of the 32-bit super-page feature of the Alpha AXP architecture. If the implementation of the 32-bit super-page is not done in hardware then it must be implemented in software (PALcode). The entire 1GB address space mapped by the 32-bit super-page must be valid at all times for both instruction fetch and data access.

> *Implementation Note (Hardware):*
> *It is strongly recommended that implementations include a hardware mapping of the 32-bit super-page for both instruction and data stream.*

I/O Space Address Extension

The Alpha AXP kernel implementation of Windows NT takes advantage of the architecture's 64-bit address space to provide a non-mapped extended address for I/O space. The extended address space uses the 43-bit super-page that is available in the Alpha AXP architecture. The super-page allows kernel mode access to an address space with a pre-determined translation, therefore, these accesses never require page table mapping nor will they ever cause a translation buffer miss. The extended super-page provides non-mapped access to a 41-bit physical address space. The extended address space is important because the bus mapping scheme that has been designed for industry-standard buses involves using a shifted physical address where the lower address bits are used to determine the byte enables. The implication is that the effective page size has become smaller. Without the non-mapped super-page I/O accesses Alpha AXP systems would suffer a performance disadvantage because of the need to write many more page table entries and to fill many more translation buffer misses. The extended address space is desirable because the likely physical address space for an Alpha AXP implementation will be 34 bits or more and the 32-bit super-page can only allow accesses to 30 bits of physical address space. The extended address space is the only exception to the 32-bit virtual address map described above. The extended address space is intended for I/O access only and can only be used in kernel mode.

Table 1 I/O Address Extension Address Map

Address Range (64 bits)	Permission	Description
0xfffffc0000000000 : 0xfffffdffffffffffffff	Kernel	Non-mapped kernel mode I/O extension

Canonical Virtual Address Format

All virtual addresses, with the exception of the large super-page addresses, must be in canonical longword form. The PALcode must check the faulting virtual addresses in the first level miss flows and raise an exception if the addresses are not canonical longwords. The check is required because the

processor may generate 64-bit addresses that are not canonical longwords but the common memory management code only knows about 32-bit addresses and so could not necessarily identify or signal the exception to the offending code. The PALcode cannot simply resolve the miss by using only the lower 32 bits. When the faulting instruction is re-executed it will again attempt to access the non-canonical address. If a virtual address fails the canonical form test then the PALcode will raise a general exception (see below).

Page Tables

Page table entries (PTEs) provide the translation from virtual addresses to their physical addresses. The physical address, in the form of a page frame number (PFN), protection information, and performance hints are included in the PTE. The virtual address is related to a page table entry solely based upon the position of the PTE within a set of page tables.

There are two methods that may be used to traverse the page tables to retrieve the corresponding PTE for a given virtual address. The first is to view the page tables as a single-level virtually contiguous table. The second is to view the page tables as a two-level, physical table.

A virtual address must be viewed in the following way for a single-level, virtual traversal:

Figure 2 Virtual Address (virtual view)

31 .. N	N-1 .. 0
Virtual Page Number (VPN)	Byte offset within page

where:

2**N = implementation page size

To access the corresponding PTE for a *VA* virtual address using the single level, virtual method use the following algorithm:

```
Va  ← BYTE_ZAP( Va, 0xf0 )                    ! clear upper bits in case Va is sign-extended
vpn ← RIGHT_SHIFT( Va, PAGE_SHIFT )  ! get virtual page number
pte_va ← VIRTUAL_PTE_BASE + ( vpn * 4)  ! 4 bytes per pte, offset + base
pte ← (pte_va)                                    ! get pte
```

where:

VIRTUAL_PTE_BASE = 0xc0000000
PAGE_SHIFT = N

In cases where the virtual access method cannot be used (for example if the pte address is itself not valid) the two-level physical method can be used to find the corresponding pte for a virtual address. The key to traversing the page tables physically is the pointer to the page directory (PPDR). The PPDR is maintained on a per-process basis whenever process context is swapped. The PPDR is input to the PALcode as a PFN but it is expected that it will be stored internally as a physical address. The PPDR is the address of the page directory (PDR) page that forms the first level of the page tables. The first level of the page tables will easily fit within a single page. Each entry in the PDR is called a PDE (page directory entry). A PDE maps one page of PTEs.

A virtual address must be viewed in the following way for a two-level, physical traversal of the page tables:

Figure 3 Virtual Address (physical view)

31 .. N+P	N+P-1 .. N	N-1 .. 0
Page Directory Index (PDI)	Page Table Index (PTI)	Byte offset within page

where:

2**N = implementation page size
2**P = ptes per page = page size / 4

To access the corresponding PTE for a *VA* (virtual address) using the two-level, physical traversal, use the following algorithm:

```
Va ← BYTE_ZAP( Va, 0xf0 )                          ! clear upper bits in case Va is sign-
extended
pde_index  ← RIGHT_SHIFT( Va, PDE_SHIFT )    ! get pde number
pde_offset ← pde_index * 4                          ! 4 bytes per pde, index * 4 ← byte offset
pde_pa ← PPDR + pde_offset                          ! offset + base
pde ← (pde_pa)                                        ! get page directory entry
pte_pfn ← pde<PFN>                                    ! get pfn of pte page from pde
pte_page ← LEFT_SHIFT( pte_pfn, PAGE_SHIFT ) ! get physical address of pte page
pte_index ← VA<PTI>                                 ! extract page table index from virtual
                                                              ! address
pte_offset ← pte_index * 4                          ! calculate offset, 4 bytes per pte
pte_pa ← pte_page + pte_offset                      ! address ← base + offset
pte ← (pte_pa)                                        ! read the pte
```

where:

PDE_SHIFT = N + P
PAGE_SHIFT = N

Page directory entries are themselves page table entries and so they have the same format. There are some implications for DTB implementation because the PDEs establish a recursive mapping for addresses within the PTE address space. The implications and a description of the recursive mapping are described in an Appendix.

The format for a PTE in NT/Alpha AXP is:

Figure 4 Page Table Entry

31..9	8..7	6..5	4	3	2	1	0
PFN	SFW	GH	G	R	D	O	V

Table 2 Page Table Entry Fields

Field	Description
PFN	Page Frame Number
SFW	Reserved for software (operating system)
GH	Granularity hints
G	Global translation hint (address space match)
O	Owner, 0 = kernel access only, 1 = user access permitted
D	Dirty, 0 = page is not dirty, 1 = page is dirty
R	Reserved
V	Valid, 0 = translation not valid, 1 = valid translation

Notes regarding the PTE fields:

1. The G or global bit is a hint to an implementation that the indicated translation is global for all processes and that the translation need not be invalidated when an implicit tbiap is executed. Since the global bit is considered a hint it is not necessary for an implementation to use this field.
2. The D or dirty bit is implemented for Alpha AXP as the inverse of FOW (fault on write). The dirty bit serves double duty by causing faults for the first write to a page. Dirty serves as a write protect bit and as a marker allowing the operating system to track dirty pages.
3. The O or owner bit indicates if user-mode is allowed access to this page, either for instruction fetch or data access. Kernel mode code has implied access to all pages that have a valid translation.
4. The GH or granularity hint bits provide a method for mapping translations larger than the standard implementation page size. These large pages must be both virtually and physically aligned. Once again, the granularity hints are hints only and may be ignored by an implementation. The granularity hints define the translation in terms of a multiple of the page size where the multiplier = 8**N, where N is the granularity hint value (0..3).

Translation Buffer Management

The PALcode must also provide call pals to manage the cached translations maintained in the translation buffers. The following call pals are provided: tbis, tbisasn, dtbis, and tbia.

Tbis invalidates a translation for a single virtual address. Its purpose is to invalidate a single translation for a specific virtual address passed as a parameter to tbis. Tbis invalidates the translation in all TBs within the processor.

Tbisasn invalidates a translation for a single virtual address for a specified address space number. The address space number may or may not be the address space number for the currently executing thread. Tbisasn invalidates the translation in all TBs within the processor.

Dtbis invalidates a single data stream translation for a specified address. It is designed for those cases when the operating system can determine that the translation is not used in the instruction stream. Implementations may take advantage of the fact that dtbis is invalidating a data stream only translation by avoiding the requirement to invalidate instruction stream translations in both, potentially, an ITB and a virtual ICACHE.

Tbia invalidates all page table translations in all TBs within the processor. (The translations invalidated are limited to "page table translations" because it is possible that an implementation has used fixed TB entries to implement one or more of the required super-pages. These fixed translations are considered "hard-wired" by the operating system and must be valid at all times.

Note, that tbiap (translation buffer invalidate for all processes) is not included explicitly in the PALcode. Tbiap is used to invalidate translations for all processes but may spare translations with the global (ASM) bit set. The tbiap operation is an implicit side-effect of swapping from one process to another (essentially one address space to another). For implementations that do not include ASNs, the tbiap will be implicit for every address space swap. Processors that implement ASNs need only issue the tbiap when ASNs are reused. ASN reuse is indicated to the PALcode by a parameter passed to the swpctx and swpprocess call pal functions.

6. Processes and threads

NT is designed as a multi-threaded operating system with multiple threads executing within the same process. Each thread has its own processor context, user-mode stack and kernel stack. Memory and the address space are shared across all threads in the same process.

The PALcode "knows" nothing about the structure of threads or processes in NT. The PALcode implements the means to swap from one thread context to another and to allow a thread to attach to the address space of another process. The state to accomplish these operations is passed entirely in registers.

Additionally, the PALcode maintains two internal registers that allow threads to query state about the currently executing thread. A unique value identifying the current thread is written when the thread context is swapped. This unique thread value is accessible by the privileged call pal rdthread. In addition, a user-accessible value is supplied when thread context is swapped. The user-accessible value is a pointer to the thread environment block (TEB) for the new thread. The TEB is obtained via the rdteb unprivileged call pal. Once again, the PALcode knows nothing about the structure of the TEB, it simply returns the written value when context is switched.

The swpctx call pal swaps from the context on one thread to another. The following parameters are passed to swpctx:

1. Initial kernel stack pointer
2. PTEs for the first two kernel stack pages
3. Unique thread value
4. Thread environment block pointer
5. PFN of the directory table base page for the new process
6. ASN for the new process

Swpctx must switch to the new kernel stack for the new thread. The first parameter is the initial kernel stack pointer and it is written to the internal processor register IKSP. The next parameters are the PTEs that map the first two pages of the kernel stack for the new thread. The use of these PTEs is optionally -- they are provided so that an implementation may guarantee the translations are valid in order to avoid the likely fault when swpctx returns and the kernel stack is accessed. The kernel does not guarantee that these translations are not already valid in the translation buffers.

The unique thread value and the pointer to the TEB have been described above. Note that these values are only written at context switch time as they are kernel maintained. The implicit assumption is that these values for a particular thread cannot change while that thread is executing.

The additional parameters to swpctx allow switching to a new process address space. The pfn of the directory table base page is an overloaded parameter -- it is used to indicate if the process needs to be swapped. The pfn is set to a negative value in the kernel if the previous thread and the new thread are in the same process (address space). If the two threads are in the same process then there will be no need to swap the address space. When the two threads are in the same process the pfn will be set to a negative value and the value for the ASN parameter is undefined. If the two threads are in different processes then the pfn will be greater than zero and will be used to write the PDR internal processor register. When the pfn is valid (greater than zero) then the ASN must be valid also and will be used to write the ASN internal processor register.

Swapping to a new process address space involves establishing a new directory pointer to the page table base page for the new process and possibly performing translation buffer operations. If the ASN is equal to the maximum address space number for the implementation then the PALcode must perform an invalidation operation for each cached translation in the translation buffers and virtual caches that does not have the ASM (address space match) bit set.

The swpprocess (swap process) call pal is also provided to allow a thread to attach to an another process. Swpprocess requires only two parameters: the PFN of the new directory table base page and the new ASN. Swpprocess performs the same address space swapping operation as does swpctx when the pfn of the page directory page is valid.

7. Caches and cache coherency

Alpha AXP implementations may include caches that may not be kept coherent with main memory. Currently there is an architected common way to make the instruction execution stream coherent with main memory via the imb (Instruction Memory Barrier). Imb is a non-privileged call pal that guarantees that subsequently executed instructions will be fetched coherently with respect to main memory. Code that modifies the instruction stream, either through writes or by DMA from an I/O device must issue the imb instruction to ensure the instruction stream becomes coherent.

The PALcode does not architect a method for future processors to make potentially incoherent data streams coherent. The first implementation processor maintains data stream coherency. For any systems that require the need to make the data stream coherent or to flush a write-back cache, native code can be implemented via the standard HAL interfaces. However, if a future implementation does not permit (for whatever reason) native code to be able to either make the data stream coherent or flush internal write-back data caches then that implementation must provide a call pal to do so. The interface to such a call pal will effect the HAL for those systems that use the implementation only.

8. Stacks

Two special-mode stacks are provided for executing in kernel mode: the kernel stack and the DPC stack. Each thread is allocated its own pages for a kernel stack. The DPC (Deferred Procedure Call) stack is a processor-wide stack upon which all DPCs are executed.

The initial kernel stack pointer (IKSP) points to the top of the kernel stack currently active kernel stack for the current thread. Two call pals are provided to access the IKSP, rdksp to read the IKSP and swpksp to atomically read the current IKSP and write a new one. The kernel stack is the 2 pages of virtual address space below the IKSP for a thread, where the IKSP points to the byte beyond the top of the 2 pages.

In addition to the kernel stack and the DPC stack a panic stack is provided to allow the operating system to remain coherent when it crashes.

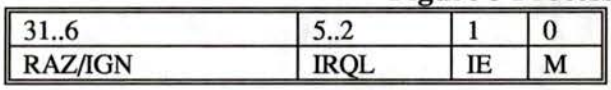
The DPC stack and the panic stack must remain valid for the lifetime of the system. The kernel stack for the currently executing thread must also be valid. Software must guarantee that the kernel stack pointer remains 16-byte aligned as per the NT/Alpha AXP calling standard.

9. Processor Status

Two components define processor status for NT/Alpha AXP: the processor status register and the DPC flag.

The processor status register (PSR) is defined below:

Figure 5 Processor Status Register



31..6	5..2	1	0
RAZ/IGN	IRQL	IE	M

Table 3 Processor Status Register Fields

Field	Type	Description
M	RW	Process mode, 0 = kernel mode, 1 = user mode
IE	RW	Interrupt enable, 0 = interrupts disabled, 1 = interrupts enabled
IRQL	RW	Interrupt request level, 0 - 7

The IRQL field is used to provide interrupt priority levels in the range of 0..7 where 0 would indicate all interrupts are enabled and 7 would indicate no interrupts are enabled.
The IE field is a global interrupt enable that can be used to turn interrupts on and off without changing the IRQL.
The M field describes the current processor privilege mode: user (unprivileged) or kernel (privileged). The privilege mode of the processor defines the instructions that can be executed and then the memory protection that will be used according to the following table:

Table 4 Processor Privilege Map

Operation	Privileged	Unprivileged
super-page access	yes	no
page protection	access to all pages	access to only those pages with the Owner bit = 1
privileged call pal instructions	yes	no

Two privileged call pals are provided to access the PSR, rdpsr to read the PSR and wrpsr to write the PSR.

The DPC flag indicates if the processor is currently executing a Deferred Procedure Call (DPC). The DPC flag is defined below:

Figure 6 DPC_FLAG

31..1	0
SBZ	DPC

Table 5 DPC_FLAG Fields

Field	Type	Description
DPC	RW	DPC flag, 0 = not currently executing a DPC, 1 = currently executing a DPC

Two privileged call pals are provided to access the DPC flag, rddpcflag to read the DPC flag and wrdpcflag to write the DPC flag.

10. Firmware interfaces

The firmware PALcode environment is de coupled from the operating system PALcode. Two PAL interfaces are provided to permit the operating system to transition to the firmware PALcode context: the halt and swppal call pals.

The halt call pal is used to perform a controlled transition to firmware. Halt essentially follows the semantics for a restart to the ARC firmware environment with the addition of Alpha AXP support for switching to the firmware PALcode. The halt function accomplishes the following tasks:

1. Retrieve the restart block pointer from the PCR
2. Verify the restart block
3. Save the general purpose register state in the restart block
4. Save the architected internal process register state in the restart block
5. Save the RESTART_ADDRESS in the restart block
6. Retrieve the firmware restart address from the PCR
7. Restore the PAL base to the previous PAL base
8. Complete the restart block by updating the boot status and the checksum
9. Restart execution at the firmware restart address passing a pointer to the restart block in the a0 register

The restart block is expected to be initialized by the firmware. The pointer to the restart block is passed by the firmware through the OS Loader to the kernel in the loader parameter block. The kernel writes the restart block pointer into the PCR during startup. The restart block pointer must be a 32-bit super-page address.

The firmware environment is responsible for allocating memory for the entire restart block including the Alpha AXP-specific saved state area. The firmware is also responsible for initializing the restart block as per the ARC specification.
It would be nice if the firmware and the OS loader could cooperate to have the restart block within the PCR (the space is available) this might save a page of memory per-processor.

The PALcode will verify the restart block by insuring that the restart block signature is valid and that the restart block and saved state area lengths are of sufficient size to contain the state the PALcode will save. If the PALcode determines that the restart block is not valid then an alternate restart will be initiated.

The alternate restart will be for the PALcode to restore the previous PAL base to the PAL_BASE register and to transfer control to the previous PAL base in PAL mode.

The general register state saved will include all 32 integer registers and all 32 floating point registers. In addition, the floating point control register will also be saved. The internal processor register will be store in its architected format so that it may be interpreted in the firmware environment. In addition, remaining space will be allocated so that the total size of the restart block is 1K bytes. The additional space will be usable for per-implementation data.

The RESTART_ADDRESS is stored in the saved state area to allow return from halt via the restart call pal function. The HAL is responsible for populating the Version, Revision, and RestartAddress fields of the restart block header. Restart will be described in greater detail below.

The PALcode will capture the previous PALcode environment when it is first initialized. The previous PALcode base address will be read from the PAL_BASE register and written to the PREVIOUS_PAL_BASE register. When the processor executes the halt function it will restore the previous PALcode environment by writing the value in the PREVIOUS_PAL_BASE register to the PAL_BASE register.

> *Implementation Note (Hardware):*
> *There are several restrictions imposed on the hardware design to support this model for switching PALcode environments:*
> 1. *The currently active PALcode must be settable by writing the base address of the PALcode image to an internal processor register*
> 2. *No implementation can require an alignment of greater than 64K for the base of the PALcode*
> 3. *The internal processor register used to set the base of the PALcode must be readable for each byte which is writable*

The firmware restart address is the address to which the PALcode will transfer control upon completion of the halt. The firmware restart address is passed from the firmware through the OS Loader to the kernel and stored in the PCR as is the restart block pointer. The firmware restart address is read from the PCR and written to the RESTART_ADDRESS register with implementation-specific (but well-defined) interpretation.

The restart call pal function is provided to undo the work that has been by a halt and allows the processor to restart execution. The restart function must simply perform the inverse of the tasks that were performed in the halt.

The tasks and sequence required for performing a halt and restart are described below:

1. Firmware allocates restart block, initializing signature, length, id fields and the pointer to next restart block, restart block pointer and firmware restart address are passed to the kernel
2. HAL populates the Version and Revision fields during HAL initialization
3. Some external event triggers an halt or reboot or power-fail
4. The appropriate HAL routine populates the RestartAddress field of the restart block with the address of the HAL restart routine
5. The HAL executes the halt instruction
6. The PAL saves processor state, including the RESTART_ADDRESS register (which is the address in the HAL of the instruction after the halt call pal instruction)
7. The PAL transfer to the firmware environment
8. The firmware initializes a restart by calling the HAL restart routine (via the address in the restart block header)
9. The HAL uses the swppal instruction to restore the operating system PALcode environment
10. The HAL uses the restart call pal function to restore complete processor state

11. The PAL restores state and then returns execution to the instruction after the halt call pal in the HAL
12. The HAL completes the restart

The swppal call pal is a flexible interface that allows kernel code to transition to any PAL environment as opposed to halt which limits the caller to transition to the previous PAL environment. The swppal call pal is a subset of the swppal implemented for VMS and OSF. Swppal supports only the address mode: swppal will resume execution at the address provided in the call pal.

11. Exception dispatch

Overview

When the processor encounters an exception it traps to the PALcode which provides preliminary exception dispatch for the operating system. Some exceptions (e.g. TB miss) may be entirely handled by the PALcode without the intervention of the operating system.

The PALcode is designed to provide a simple and efficient dispatching method to the operating system for those exceptions that require operating system action. In general the following operations characterize exception dispatch:

1. Switch to kernel mode (if in user mode).
2. Allocate a trap frame on the kernel stack.
3. Save the necessary processor state in the trap frame.
4. Prepare arguments to the kernel exception handler using the standard argument registers where possible.
5. Set the processor state for executing the kernel (establish the stack pointer so it points to the kernel stack, establish the global pointer to point to the kernel global area).
6. Restart execution at the address of the kernel exception handler registered for the class of exception that was encountered.

The details of the actions required for each specific type of exception are described below in a separate section for each exception type.

Exception Classes

The PALcode categories each of the possible exceptions into one of the following classes:

1. Memory Management Exceptions
2. Interrupt Exceptions
3. System Service Calls
4. General Exceptions
5. Panic Exceptions

Memory Management Exceptions will be raised for two types of exceptions:

1. Translation not valid faults: accesses to addresses which do not have a valid translation for the currently executing context
2. Access violations: accesses to addresses for which the currently executing context does not have permission for the access

Interrupt Exceptions will be raised when a software or hardware interrupt is asserted and enabled.

System Service Calls are not exceptions per-se but are handled as such to allow non-privileged code to request and receive privileged services. System Services may be requested from both non-privileged and privileged modes (user and kernel mode respectively).

The General Exception class is the catch-all category for all of the other exceptions that may be raised by non-privileged code:

1. Illegal instruction execution
2. Unaligned memory access
3. Arithmetic exceptions
4. Software exceptions
5. Breakpoints
6. Subsetted instruction execution

Panic Exception is the class reserved for conditions from which execution cannot reliably be continued. The following general cases of panic exceptions are anticipated:

1. Invalid kernel stack (including overflow and underflow).
2. Unrecoverable machine checks.
3. Unexpected exceptions from PALcode.

Rfe (return from exception)

The rfe (return from exception) call pal is provided to allow the operating system to return from an exception. Rfe may also be used to transition from kernel mode to user-mode startup code. Note that two of the classes of exceptions will not use rfe to return to the previously executing context: System Service Exceptions and Panic Exceptions. A special call pal, retsys, is used for returning from System Service Calls. Retsys is necessary because a System Service exception has different semantics with regard to the processor state saved than any of the other exceptions. Panic Exceptions do not return as they precipitate a controlled crash of the operating system.

Rfe is documented in detail in the call pal portion of this specification. Briefly, rfe reverses the effect of an exception by restoring the original processor state from the trap frame on the kernel stack. In addition, rfe accepts a parameter that allows it to set software interrupt requests for the execution context that is about to be reestablished.

Trap Frames

Trap frames are allocated on the kernel stack for all classes of exceptions in PALcode. The PALcode will also partially populate the trap frame; the fields populated are based upon the exception being handled. The kernel stack must be guaranteed to remain aligned on a 16 byte boundary as per the NT/Alpha AXP calling standard. The trap frame itself will be guaranteed to be a multiple of 32-bytes in size. The PALcode may over-align the kernel stack pointer when allocating the trap frame in order to improve memory throughput, with consideration for the extra memory being consumed. The trap frame will be structured so that writes will aggregate. The register values stored in the trap frame are 64-bit values - this is required as the register set is 64-bits and may contain 64-bit values (as opposed to canonical longwords).

Can we allocate a smaller trap frame for system service calls just to conserve space?

12. Interrupt handling

The NT PALcode supports two software interrupt levels and an implementation-specific limit of hardware interrupt sources. The NT PALcode supports 8 levels of interrupt priority known as interrupt request levels (IRQL). The supported IRQLs are numbered 0..7.

The platform independence of interrupt dispatch is accomplished via three tables: Interrupt Level Table, Interrupt Mask Table, and the Interrupt Dispatch Table.

Interrupt Level Table (ILT)

The Interrupt Level Table consists of 8 entries, indexed 0..7. Each table entry corresponds to an IRQL by its index within the table. The value of each entry is an enable mask that indicates which interrupt sources are to be enabled within the processor for the corresponding IRQL. One full longword is reserved for each table entry. The interpretation of the bits within the enable mask is processor-specific.

> *Implementation Note (Software):*
> *The Interrupt Level Table is probably the most important optional set of data that could be cached within the processor. Implementations should consider implementing a call pal that causes the ILT to be re-read and re-cached within the processor. Also note, some processors may have an effectively hard-wired ILT (in such a case, the HAL will have no influence over which interrupts are enabled for each IRQL).*

Interrupt Mask Table (IMT)

The Interrupt Mask Table relates a mask value of requested interrupts to both an interrupt vector and a synchronization IRQL. The table is an implicit interrupt priority resolver as only one interrupt vector can be assigned for each request mask. The table is divided into 2 sub-tables as below:

Table 6 Interrupt Mask Table

Index Range	Interrupt source description
0..3	Software (2 sources)
4..131	Hardware

Each entry in the table is a longword that consists of 2 word values: the interrupt vector number and the synchronization level. The usage of the software portion of the table is strictly defined and consistent across all processor implementations. The software entries are used only if no hardware interrupts are pending. The entries must be initialized so that DPC software interrupts are higher priority than APC software interrupts. The expected initialization of the software portion of the table is defined below:

Table 7 Software Entries of the IMT

Index	Synchronization Level	Vector
0	PASSIVE_LEVEL = 0	Passive release vector
1	APC_LEVEL = 1	APC Dispatch vector
2	DISPATCH_LEVEL = 2	DPC Dispatch vector
3	DISPATCH_LEVEL = 2	DPC Dispatch vector

Usage of the hardware portion of the IMT is designed to be flexible. Each implementation must define a relation *f* that defines a mapping of requested and enabled hardware interrupt sources to entries in the IMT. The relation *f* is implementation-specific but *f* must be a function in the mathematical sense (for each input there is a single unambiguous result). All interrupts other than software interrupts will be considered hardware interrupts. Hardware interrupts could include external interrupt signals, performance counter interrupts, and correctable read interrupts.

Interrupt Dispatch Table (IDT)

The Interrupt Dispatch Table (IDT) has an entry for each possible interrupt vector. The possible interrupt vectors are in the range 0..255. Each entry is a longword pointer which is the virtual address of the interrupt dispatch routine for the vector which corresponds to the index of the entry within the table.

Interrupt Dispatch

Interrupt dispatch within the PALcode will proceed through the following steps:

```
irr ← currently requested interrupt mask (from internal processor state)
ier ← currently enabled interrupt mask (from current IRQL)
irm ← irr AND ier        ! mask of requested and enabled interrupt sources
CASE                     ! retrieve value from interrupt mask table
    Any Hardware Interrupt Pending :      ! any hardware interrupt source is pending
        index = f(irm)
        sirql ← (IMT<{index*4}>)<SynchronizationIRQL>    ! get synchronization IRQL
        vector ← (IMT<{index*4}>)<InterruptVector>       ! get interrupt vector
    Any Software Interrupt Pending
        sirql ← (IMT<{irm*4}>)<SynchronizationIRQL>      ! get synchronization IRQL
        vector ← (IMT<{irm*4}>).InterruptVector          ! get interrupt vector
    Otherwise:               ! software interrupt should be set
        Passive release, restart execution
    ENDCASE
Set processor to sirql IRQL
if( processor interrupt )
        { acknowledge the interrupt }
endif
```

Issue: This does not work this way today (the PAL does less and the kernel does more) More on this below in the issues section.

Once synchronization level has been set and the interrupt service routine has been determined the PALcode builds a trap frame and dispatches to the kernel interrupt exception handler passing in the interrupt vector. The specification for this is the following:

```
previousPsr ← PSR
If PSR<Mode> ← User
        PSR<Mode> ← Kernel    ! set processor to kernel mode
        tp ← IKSP - TrapFrameLength     ! establish trap pointer
  Else
        tp ← sp - TrapFrameLength       ! establish trap pointer
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrFir(tp) ← ExceptionPC
TrPsr(tp) ← previousPSR
TrServiceStack(tp) ← tp
TrIntRa(tp) ← ra
sp ← tp
TrServiceStack(tp) ← sp
fp ← sp
gp ← KGP
a0 ← vector
a1 ← PCR
a3 ← previousPSR
RestartAddress ← INTERRUPT_ENTRY
Restart processor
```

All other general purpose register values must be preserved across interrupt dispatch.

The kernel will use the rfe instruction to restart the interrupted code sequence.

Interrupt Acknowledge

Interrupts will be acknowledged according to their origin. Internal processor interrupts, such as software interrupts and performance counters, will be acknowledged by the PALcode. System-level interrupts will be acknowledged in the interrupt dispatch routines.

Synchronization functions

The NT PALcode provides 3 functions to allow the kernel to effect the processor's current interrupt enable state: swpirql, di and ei.

Swpirql swaps the current interrupt request level of the processor. Swpirql takes as a parameter the new IRQL for the processor and returns the previous IRQL.

Di disables all interrupts *without changing the current IRQL.* Ei enables interrupts at the currently set IRQL. The usage of these functions and the existence of the interrupt enable bit in the PSR are used as a global interrupt enable for all interrupts.

Software Interrupt Requests

The PALcode includes an architected internal processor register for controlling software interrupt requests: the SIRR (Software Interrupt Request Register). The format of the SIRR is as below:

Figure 7 Software Interrupt Request Register

31 .. 2	1	0
RAZ	DPC	APC

Table 8 Software Interrupt Request Register Fields

Field	Type	Description
APC	RW	APC software interrupt requested.
DPC	RW	DPC software interrupt requested.

The NT PALcode provides 2 functions for effecting the state of software interrupt requests: ssir and csir.

Ssir sets software interrupt requests. Ssir takes as a parameter the software interrupts to be set. The software interrupt requests levels to be requested are indicated by the bits set in the parameter register. The NT PALcode support only 2 interrupt levels so only the 2 least significant bits of the parameter register are used.

Csir clears (or deasserts) software interrupt requests. Csir takes as a parameter the software interrupts to be cleared. The software interrupt requests levels to be cleared are indicated by the bits set in the parameter register. The NT PALcode support only 2 interrupt levels so only the 2 least significant bits of the parameter register are used.

13. System Service Exceptions

System Service Calls are initiated via the callsys call pal. Callsys essentially has the semantics of a standard routine call, arguments are passed in the argument registers and on the stack, volatile registers are considered free and non-volatile registers must be preserved across the call. (See appendices for the calling standard register definitions.) In addition, to the standard calling sequence callsys passes the number of the desired system service in the return value register v0. System Service Calls may be made from both user and kernel modes. The details for the callsys call pal are provided in the call pal listing section; a brief overview is provided here.

Callsys switches to kernel mode, if necessary, and allocates a trap frame on the kernel stack. The volatile registers may be used freely by the PALcode. The argument registers must be preserved through the call pal. Standard control information such as the previous PSR is stored in the trap frame. Callsys then restarts execution at the kernel system service exception entry passing the previous mode as a parameter in the t0 register.

Retsys is provided to return from a System Service back to the caller. For the most part it is very similar to the rfe call pal with the following exceptions:

1. Retsys need not restore the argument registers a0..a3 from the trap frame
2. Retsys need not preserve volatile register state
3. Retsys returns to the address in the ra register at the point of the callsys rather than the faulting instruction address (the ra was written to the faulting instruction address by callsys).

A detailed description of retsys is included in the call pal listing section of the specification.

14. Memory Management Exception handling

The NT PALcode will recognize 2 classes of memory management faults: translation not valid faults and access violations. Translation not valid faults are detected when a page table entry for a virtual address has the valid bit cleared. The invalid page table entry could be either a first or second level table entry. Access violations are detected by the hardware when the processor attempts to access a virtual address and that type of access is not permitted according to the protection mask in the page table entry that maps the translation for the virtual address.

The PALcode will dispatch to the kernel in the exact same manner for each of these 2 classes of exceptions according to the following description:

```
previousPSR ← PSR
IF PSR<Mode> ← User
        PSR<Mode> ← Kernel
        tp ← IKSP - TrapFrameLength
  ELSE
        tp ← sp - TrapFrameLength
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntRa(tp) ← ra
TrIntGp(tp) ← gp
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrFir(tp) ← ExceptionPC
TrPsr(tp) ← previousPSR
sp ← tp
TrServiceStack(tp) ← sp
fp ← sp
gp ← KGP
a0 ← 1 if faulting operation was a store, 0 otherwise
a1 ← faulting virtual address
a2 ← previousPSR<Mode>
a3 ← previousPSR
RestartAddress ← MEM_MGMT_ENTRY
restart the processor
```

All other general purpose registers must be preserved across the memory management exception dispatch.

If the kernel can resolve the fault it will use the rfe instruction to restart the faulting thread so as to re-issue the instruction that faulted. Otherwise, the kernel will raise the appropriate exception.

15. Panic Exception handling

There are 3 general classes of severe problems or panics that the NT PALcode may recognize. Severe problems are not recoverable -- the operating system cannot continue executing normally. Panic

exception handling is used to shutdown the machine in a controlled manner that will assist in debugging the problem. With the exception of hardware errors, panic exceptions are not expected to occur in the production operating system. In fact, some of the checks that detect panics may be disabled when the production PALcode ships.

The PALcode will raise a panic exception to the kernel and will describe the condition that causes the panic with a bugcheck code. When the kernel receives a panic exception it will enter the kernel debugger if the kernel debugger is enabled.

The 3 classes of panic exceptions are:

1. Unrecoverable processor or system hardware errors
2. Kernel stack corruption
3. Unexpected exceptions in PALcode

Unrecoverable Hardware Errors

The PALcode may be able to continue execution when the hardware exhibits catastrophic and unrecoverable error conditions. The types of conditions that would be unrecoverable are processor-specific. If such a condition occurs the PALcode will raise a panic exception with the bugcheck code set to DATA_BUS_ERROR.

Kernel Stack Corruption

The PALcode may recognize the following types of kernel stack corruption: invalid kernel stack, kernel stack overflow and kernel stack underflow. The kernel stack for an executing thread must always be valid, if the processor faults when accessing the kernel stack and the page tables indicate the kernel stack address is not valid then the PALcode will raise a panic exception. The PALcode will also check for kernel stack underflow and overflow and will raise a panic exception if either condition is detected. (Kernel stack checking may be disabled for production releases - maybe, either (1)must be left on or (2)define distinctions between debug pal and production pal, or (3)punt this -- i prefer 1 or 2). The kernel stack is the 2 pages of virtual address space below the IKSP for a thread, where the IKSP points to the byte beyond the top of the 2 pages. When raising a kernel stack corruption, the PALcode will set the bugcheck code to PANIC_STACK_SWITCH.

Unexpected PAL exceptions

When the PALcode detects an exception caused by PALcode that is unexpected, the PALcode may raise a panic exception. It is anticipated that these conditions will either indicate a bug in the PALcode or that the processor is no longer correctly executing. The PALcode will raise the bugcheck code TRAP_CAUSE_UNKNOWN.

The PALcode will build a trap frame for the kernel before it dispatches. The PALcode will also fill in the exception record that exists within the trap frame. The PALcode may optionally populate the exception information longwords with information that will assist in debugging the problem. Only ErExceptionInformation<0..3> may be used. It is highly recommended that the PALcode populate the first longword with a bugcheck subcode that identifies the particular error condition detected by the PALcode. The PALcode will strive to maintain all possible register state in order to assist in debugging.

Panic Exception Dispatch

The PALcode will follow the following operations when dispatching a panic exception to the kernel:

```
previousPSR ← PSR
IF PSR<Mode> ← User
        PSR<Mode> ← Kernel
ENDIF
panicStack ← PcPanicStack(PCR)! get the panic stack
tp ← panicStack - TrapFrameLength          ! allocate trap frame on panic stack
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntRa(tp) ← ra
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrPsr(tp) ← previousPSR
TrFir(tp) ← ExceptionPC
ErExceptionCode(tp) ← bugcheck code
ErExceptionAddress(tp) ← ExceptionPC
ErExceptionFlags(tp) ← zero
ErExceptionRecord(tp) ← zero
ErNumberParameters(tp) ← zero
sp ← tp
TrServiceStack(tp) ← sp
fp ← sp
gp ← KGP
a0 ← bugcheck code
RestartAddress ← PANIC_ENTRY
Restart processor
```

All other general purpose register must be preserved across the panic exception dispatch.

16. General Exception handling

General Exception is classification for all of the other exceptions that may be raised by hardware or software. These exceptions are handled in approximately the same manner in the PALcode and in exactly the same manner in the lowest level kernel exception dispatch.

The following exceptions have been grouped together under the General Exception rubric:
1. Arithmetic Exceptions
2. Unaligned Access Exceptions
3. Illegal Instruction Exceptions
4. Machine Check Exceptions
5. Breakpoints
6. Software Exceptions
7. Subsetted IEEE Instruction Exceptions

A general exception builds a trap frame on the kernel stack and populates the exception record within the trap frame and then dispatches to the kernel general exception entry point. The differences between each of these types of exceptions are the population of the exception record and the meaning of the faulting

instruction field within the trap frame. The values for each specific exception are detailed in the sections that follow.

General Exceptions: Common Dispatch

The common dispatch for all general exceptions takes the following steps:

```
previousPSR ← PSR
IF PSR<Mode> = User
        PSR<Mode> ← Kernel
        tp ← IKSP - TrapFrameLength
ELSE
        tp ← sp - TrapFrameLength
ENDIF
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntRa(tp) ← ra
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrPsr(tp) ← previousPSR
TrFir(tp) ← ExceptionPC
sp ← tp
TrServiceStack(tp) ← sp
fp ← sp
gp ← KGP
a0 ← tp + TrExceptionRecord          ! a0 ← pointer to exception record
a3 ← previousPSR
```

All other general purpose registers must be preserved across the general exception dispatch.

17. Arithmetic Exceptions

Arithmetic exceptions for the Alpha AXP architecture are imprecise, meaning that the processor may not signal an exception until some arbitrary number of instructions after the instruction that caused the exception. Special handling is required in the kernel and compiler to deterministically raise the appropriate exceptions to user programs. These topics are covered in greater detail elsewhere (SRM, Handbook). Important to this specification is the definition of the ExceptionPC that is written to the TrFir offset of the trap frame. The exception PC written into the trap frame is the virtual address of the first instruction after the faulting instruction that has not yet executed.

Arithmetic traps write the following information into the exception record of the trap frame:

ErExceptionCode(er) ← STATUS_ALPHA_ARITHMETIC
ErExceptionInformation<0>(er) ← FLOATING_REGISTER_MASK
ErExceptionInformation<1>(er) ← INTEGER_REGISTER_MASK
ErExceptionInformation<2>(er) ← EXCEPTION_SUMMARY
ErNumberParameters(er) ← 3
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

where:

er = exception record pointer

The Floating register masks indicate which floating point registers were destinations of instructions that caused an exception. A one in the corresponding position for a register indicates that the register was the destination of an instruction that faulted. A zero indicates that the register was not the destination of an instruction that faulted. The definition of the correspondence between the floating registers and the bits in the mask is defined as follows:

Figure 8 FLOAT_REGISTER_MASK

31	30	29..2	1	0
F31	F30	F29..F2	F1	F0

The Integer register masks indicate which integer registers where destinations of instructions that caused an exception. A one in the corresponding position for a register indicates that the register was the destination of an instruction that faulted. A zero indicates that the register was not the destination of an instruction that faulted. The definition of the correspondence between the integer registers and the bits in the mask is defined as follows:

Figure 9 INTEGER_REGISTER_MASK

31	30	29..2	1	0
R31	R30	R29..R2	R1	R0

The format of the EXCEPTION_SUMMARY register is as follows:

Figure 10 EXCEPTION_SUMMARY

31..7	6	5	4	3	2	1	0
RAZ	IOV	INE	UNF	OVF	DZE	INV	SWC

Table 9 EXCEPTION_SUMMARY Fields

SWC	Software Completion	The software completion option /S was selected for all of the faulting instructions.
INV	Invalid Operation	One or more of the operands of a floating point operation was an illegal value.
DZE	Division by zero	Floating point divide attempt with a divisor of zero.
OVF	Overflow	Result of floating operation overflowed the destination exponent.
UNF	Underflow	Result of floating operation underflowed the destination exponent.
INE	Inexact result	Result of floating operation caused loss of precision.
IOV	Integer overflow	Result of integer operation overflowed the destination's precision.

18. Unaligned Accesses Exceptions

Unaligned access exceptions are reported to and handled by the kernel. Currently, the NT kernel has implemented the following policies regarding unaligned access fix-ups:

1. Kernel unaligned accesses will not be fixed and when instead cause a bugcheck
2. User-mode fix-ups are optional on a per-thread basis. The default state will be that user-mode unaligned accesses will *not* be fixed up and will instead cause the kernel to raise an exception to the user-mode program.

Unaligned access exceptions are precise, therefore, the address written to the faulting instruction offset of the trap frame will be the virtual address of the load or store instruction that accessed the unaligned address. .

The PALcode will write the following information into the exception record of the trap frame for an unaligned access exception:

ErExceptionCode(er) ← STATUS_DATATYPE_MISALIGNMENT
ErExceptionInformation<0>(er) ← Faulting opcode
ErExceptionInformation<1>(er) ← Destination register
ErExceptionInformation<2>(er) ← Unaligned virtual address
ErNumberParameters(er) ← 3
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

where:
er = exception record pointer

19. Illegal Instruction Exceptions

The following types of illegal operations will be raised as illegal instruction exceptions by the PALcode:

1. Attempt to execute an instruction with an opcode reserved for Digital's use (opcode DEC)
2. Attempt to execute an instruction with an unimplemented call pal code point
3. Attempt to execute a privileged call pal from user (unprivileged) mode
4. Attempt to execute an instruction with an illegal operand
5. Attempt to execute an unimplemented/subsetted instruction

Note: Instructions with illegal operands will cause illegal instruction exceptions to be raised only if the processor raises an exception for these operations.

Illegal instruction exceptions are precise; the faulting address written into the trap frame will be the virtual address of the instruction that caused the exception.

The PALcode will write the following information into the exception record of the trap frame for an illegal instruction exception:

ErExceptionCode(er) ← STATUS_ILLEGAL_INSTRUCTION
ErNumberParameters(er) ← 0
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

where:

er = exception record pointer

20. Non-Canonical Virtual Address Exceptions

If the PALcode detects that a faulting virtual address is not a canonical longword then the PALcode will raise a general exception. The implementation is required to test for the non-canonical format for both instruction stream and data stream translation buffer fills. For data stream faults the faulting address written to the trap frame will be the virtual address of the instruction that caused the reference to the invalid address. Instruction stream invalid addresses present a more difficult problem because the exception address itself is invalid and cannot be properly interpreted by 32-bit NT. In the case of instruction stream virtual addresses the ra (return address) register - 4 will be written to the faulting address field of the trap frame. The ra register is used because it is very likely to yield a sane address within the correct program that faulted. In addition, the ra-4 is the most probable faulting address as the most likely instruction to have caused the fault is: jsr ra, (rx). Certainly it is not perfect.

The PALcode will write the following information into the exception record of the trap frame for a non-canonical virtual address fault:

ErExceptionCode(er) ← STATUS_INVALID_ADDRESS
ErExceptionInformation<0>(er) ← va<63..32>
ErExceptionInformation<1>(er) ← va<31..0>
ErNumberParameters(er) ← 2
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

where:

er = exception record pointer

21. Machine Check Handling

Machine checks are initiated when the hardware detects an hardware error condition. The hardware error condition may be correctable or uncorrectable. Machine checks are not the only way that systems may indicate hardware errors. Systems may choose to signal errors via hardware interrupts. Hardware error interrupts will be delivered to the kernel as standard interrupts where they may be hooked by the HAL for system-specific processing.

There are 5 classes of machine checks recognized by the NT PALcode:
1. Catastrophic errors
2. Processor uncorrectable errors
3. Processor corrected errors
4. System uncorrectable errors
5. System correctable errors

Catastrophic Errors
Catastrophic errors are those errors that indicate that the machine is left in a state where execution cannot be reliably restarted. Errors may be catastrophic if they indicate that the hardware cannot be trusted to execute properly or if the state of data within the system cannot be determined. The conditions that will be considered catastrophic are processor-implementation specific. It is likely however, that uncorrectable machine checks taken while a machine check handler is executing will be considered catastrophic.

IS THIS WHERE WE DO THE MACHINE CHECK HALT?

Processor Uncorrectable Errors
Processor uncorrectable errors are that class of errors that are detected by the processor and exhibit data errors that cannot be reliably corrected. The actual errors that fit this class are processor implementation defined.

Processor Corrected Errors
Processor corrected errors are data errors detected by the processor that can be reliably corrected. The PALcode may or may not have to intervene in order the correct the errors.

System Uncorrectable Errors
System uncorrectable errors are errors that are detected by the system hardware and have not been corrected.

System Correctable Errors
System correctable errors are detected by the system hardware and have been corrected so that incorrect data has not been read into the processor.

The general model for machine check handling in NT/Alpha AXP has the following flow:
1. The PALcode corrects the error if possible
2. The PALcode sets the machine to a known state from which restart is possible (if reliable restart is not possible then the PALcode will raise an unrecoverable hardware error via a panic exception).
3. The PALcode builds a logo frame describing the detected error.
4. The PALcode sets processor IRQL appropriately (see below).
5. The PALcode dispatches a general exception to the kernel.
6. The kernel will forward the exception to the HAL for handling.
7. If the HAL is able to handle the exception then execution resumes
8. If the HAL is unable to handle the exception then
    a. If correctable error, the kernel will ignore the error and resume execution
    b. If uncorrectable error and the previous mode was user-mode the kernel will raise the exception to the user thread
    c. If uncorrectable error and the previous mode was kernel mode then the kernel will initiate a bugcheck to shutdown execution

The machine check error summary register is used to indicate and control the current state of machine check handler for the processor. A description of the MCES register follows:

Figure 11 Machine Check Error Summary

31..6	5	4	3	2	1	0
Reserved	DMCK	DSC	DPC	PCE	SCE	MCK

Table 10 Machine Check Error Summary Fields

Field	Type	Description
DMCK	RW	Disable all machine checks.
DSC	RW	Disable System Correctable error reporting
DPC	RW	Disable Processor Correctable error reporting
PCE	RW	Processor Correctable Error reported
SCE	RW	System Correctable Error reported
MCK	RW	Machine Check (uncorrectable) reported

All machine checks (correctable and uncorrectable) are maskable via the DMCHK bit in the MCES register. This bit is provided with the intention that it be used to debug systems only. The initial value is implemenation-specific but wherever possible will attempt to preserve the state of machine check enables from the previous PALcode environment during initialization.

The correctable errors (both system and processor) are maskable via the MCES internal processor register. Correctable errors are disabled in PAL initialization and must be explicitly enabled by the HAL. Correctable errors are delivered from the PALcode to give the HAL a chance to log the errors. It is optional but recommended that the HAL take advantage of this opportunity. The PALcode will build a logout frame with per-processor information that will assist the HAL in logging the error.

Uncorrectable errors (both system and processor) are likely to be unrecoverable. The machine check exception is raised to the HAL to give an opportunity for per-platform error handling. Uncorrectable errors will be delivered immediately upon detection. The PALcode will create a logout frame with per-processor information to assist the HAL in handling the error condition.

PALcode will populate the exception record with following the following values for a machine check:

ErExceptionCode(er) ← DATA_BUS_ERROR
ErExceptionInformation<0>(er) ← machine check type
ErExceptionInformation<1>(er) ← pointer to logout frame
ErNumberParameters(er) ← 2
ErExceptionFlags ← 0
ErExceptionRecord ← 0

IS THERE A RETRYABLE FLAG AS WELL?

where:

1. er is the exception record pointer, it is anticipated that frequently this will be the a0 register which is set to point to the exception record on entry on the kernel for all general exceptions.
2. machine check type is determined by the following table

Figure 11 Machine Check Types

Machine check	Machine check type code
Processor Uncorrectable	1
Processor Correctable	2
System Uncorrectable	3
System Correctable	4

3. virtual address of the logout frame is a 32-bit super-page address and the logout frame has a per-processor format

Machine checks differ from all other general exceptions in that they effect and are effected by the current processor IRQL. Corrected machine checks raise IRQL to 6 before dispatching to the kernel. Uncorrected machine checks raise IRQL to 7. Where possible, corrected machine checks will only be delivered if the current processor IRQL is below 7. Correctable machine checks recognized when IRQL = 7 or when interrupts are disabled will be deferred until IRQL falls below 7 and interrupts are enabled. Uncorrectable machine checks will be delivered immediately regardless of the current IRQL.

The draina (drain aborts) privileged call pal is provided to allow software to force completion of all previously executed instructions such that the previous instructions cannot cause machine checks to be signaled while any instructions subsequent to the draina are executed. Note that due to the per-processor possibilities for draina the drain operation may require per-processor native code support as well.

22. Breakpoints and Debugger Support

Breakpoint instructions will raise general exceptions. NT/Alpha AXP supports several different breakpoint instructions. Many of these breakpoints are implemented to support the kernel debugger on NT and essentially are special subroutine calls. The exact semantics of these calls are not important to the PALcode -- all breakpoints are handled in the same manner and are distinguished only by the breakpoint type that is written into the exception record.

All breakpoints are implemented as unprivileged call pals which leaves the policy decision as to whether the breakpoint can be taken in the current mode to the kernel.

The following table lists the breakpoint opcodes and their corresponding breakpoint types:

Table 12 Breakpoint Types

Breakpoint Opcode Mnemonic	Breakpoint Type	Description
bpt	USER_BREAKPOINT	user breakpoint
kbpt	KERNEL_BREAKPOINT	kernel breakpoint
callkd	passed in v0	call kernel debugger

The faulting instruction address for all breakpoints will be the virtual address of the breakpoint call pal instruction.

PALcode will complete the exception record for Breakpoints as follows:

ErExceptionCode(er) ← STATUS_BREAKPOINT
ErExceptionInformation<0>(er) ← breakpoint type
ErNumberParameters(er) ← 0
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

where:
er = exception record pointer

23. Software Exceptions

Software may raise exceptions via the unprivileged call pal gentrap (generate trap). The gentrap instruction is used to raise exceptions recognized in (possibly)user-mode software for conditions such as divide by zero. (The Alpha AXP architecture does not provide an integer divide instruction; division is accomplished by specialized divide routines.)

The gentrap call pal takes a single parameter which is preserved but not interpreted by the PALcode. The gentrap parameter is written into the exception record where it will be interpreted by the kernel exception handler. Gentrap uses the status: STATUS_ALPHA_GENTRAP as an exception code. The kernel exception dispatcher will interpret the gentrap parameter to determine the appropriate NT status to raise to the currently executing thread.

The faulting address for a gentrap exception will be the virtual address of the executed gentrap call pal instruction.

The PALcode will write the following information into the exception record for a gentrap:

ErExceptionCode(er) ← STATUS_ALPHA_GENTRAP
ErExceptionInformation<0>(er) ← gentrap parameter (a0 register upon execution of gentrap)
ErNumberParameters(er) ← 1
ErExceptionFlags ← 0
ErExceptionRecord ← 0

where:
er = exception record pointer

24. Floating point

Implementation Note (Hardware)
NT /Alpha AXP requires implementation of IEEE floating point in each processor implementation.

VAX floating point format is not supported for NT/Alpha AXP.

Subsetted IEEE floating point instructions, that is those not implemented in hardware, will be raised by the PALcode as illegal instruction exceptions.

We do not support FEN (floating enable) faults in NT. Floating point instructions are always enabled.

25. Debug vs. Free PALcode

Each implementation is required to supply 2 versions of the PALcode to the operating systems group: debug PALcode and free, or production, PALcode. The debug PALcode is a functional super-set of the production PALcode which is specified in this document. The debug PALcode includes extra counters for performance evaluation and additional sanity checks. We cannot burden the production PALcode with the performance loss necessary to implement these features. The debug PALcode will be used in the laboratory only.

The debug PALcode contains the following additional features:
1. Kernel stack underflow/overflow checking
2. Special I/O address checking
3. Event counters

Kernel Stack Checking

Whenever, the debug PALcode allocates a trap frame and the previous mode was kernel mode it must check for kernel stack underflow and overflow. Underflow occurs when the kernel mode sp is greater than the initial kernel stack pointer (IKSP) for the thread. Two pages of kernel stack are allocated for each thread. Overflow is detected whenever the kernel mode stack pointer would be less than (IKSP - 2 * PAGE_SIZE). Kernel stack underflow and overflow are indicated with a panic exception (documented in the panic exception section).

I/O Address Checking

Alpha AXP systems that use standard buses and drivers cannot provide direct access to I/O space addresses (as would Intel-based systems). Instead, the Alpha AXP systems provide access to I/O space by allowing the standard device drivers to use address handles, provided by the HAL, that may be treated

as standard I/O virtual addresses for all operations except the I/O accesses themselves. The I/O accesses must be performed by specialized routines in the HAL that are able to convert the address handles to the actual virtual addresses used for the I/O space accesses. The HAL will use the range of numbers 0xa0000000 : 0xbfffffff to represent these address handles whenever possible. This range of numbers falls into the upper half of the 32-bit super-page address range. The debug PALcode will disable the 32-bit super-page in hardware and provide support for the lower half of the 32-bit super-page in PALcode (the range of addresses 0x80000000: 0x9fffffff). Addresses in the range 0xa0000000: 0xbfffffff will be treated as standard addresses and, since they will not be mapped, will cause memory management faults (translation not valid). This support in the PALcode will allow easy and precise trapping of device driver code that attempts to access I/O addresses directly without using the intended access routines provided by the HAL.

Event Counters

The debug PALcode will provide software counters to count significant events within the PALcode. The PALcode will also provide a privileged call pal to allow kernel-mode code to read the counters: rdcounters. Rdcounters is documented in the privileged call listing section of this document. The events counted are implementation-specific but must include the following: a separate counter for each of the different call pal functions, TB miss counts, and interrupt counts. The format of the data returned by rdcounters is also implementation-specific, however, all counters must be 64-bit counters.

26. Call pal listings

The call pal functions for NT/Alpha AXP generally follow the NT/Alpha AXP calling standard, that is, arguments are passed in the argument registers: a0 - a5, and return values are returned in the value register: v0. In some cases exact adherence to the calling standard is inconvenient and/or unnecessary. The call pals also incorporate the following rules into their own calling standard:

1. Only the argument registers (a0 - a5) are considered volatile
2. All parameters are passed in registers
3. The return address register is not used

The argument registers are used as volatile registers because often they contain parameters to the call pals. In strict accordance to the calling standard the temporary registers: t0 - t12 could also be considered volatile in the call pal functions but they are not. The reason the temporaries are not considered volatile is that the call pals generally don't need that many free registers and it is convenient in assembly language, from which the call pals are most frequently called, to be able to assume that the temporaries are preserved across the call pal.

All parameters to the call pal functions are passed in registers. If the number of parameters exceeds the available number of argument registers then additional temporary registers will be used as arguments. This precludes the need for callers to be build an appropriate stack frame for call pals with more than 6 parameters.

The RETURN_ADDRESS register indicates the next execution address when the PALcode exits. Upon entry to each of the call pal functions the RETURN_ADDRESS register is considered to contain the address of the instruction immediately following the call pal instructions.

A range of privileged call pals has been reserved for processor-implementation specific call pals that allow specialized communication between the HAL and the PALcode.

Privileged Call PAL functions

Figure 13 Privileged Call PALs

Mnemonic	Description	Opcode
halt	halt the processor	0x00
restart	restart the processor	0x01
draina	drain aborts	0x02
initpal	initialize the PALcode	0x04
wrentry	write system entry	0x05
swpirql	swap IRQL	0x06
rdirql	read current IRQL	0x07
di	disable interrupts	0x08
ei	enable interrupts	0x09
swppal	swap PALcode	0x0a
ssir	set software interrupt request	0x0c
csir	clear software interrupt request	0x0d
rfe	return from exception	0x0e
retsys	return from system service call	0x0f
swpctx	swap privileged thread context	0x10
swpprocess	swap privileged process context	0x11
rdmces	read machine check error summary	0x12
wrmces	write machine check error summary	0x13
tbia	translation buffer invalidate all	0x14
tbis	translation buffer invalidate single	0x15
dtbis	data translation buffer invalidate single	0x16
tbisasn	translation buffer invalidate single for asn	0x17
rdksp	read initial kernel stack	0x18
swpksp	swap initial kernel stack	0x19
rdpsr	read processor status register	0x1a
rdpcr	read PCR (processor control registers)	0x1c
rdthread	read the current thread value	0x1e
wrperfmon	write performance monitoring values	0x20
rdcounters	read PAL event counters (Debug only)	0x30
rdstate	read internal processor state	0x31

Opcodes 0x38 - 0x3f are reserved for processor implementation-specific call pal functions.

All other opcodes are reserved for Digital's use.

csir

Clear software interrupt request.

Parameters:

a0 = Software interrupt requests to clear.

Return Value:

None.

Description:

Draina is used to drain all aborts, including machine checks from the current processor. Draina guarantees that no abort will be signaled for an instruction issued before the draina while any instruction issued subsequent to the draina is executing.

GPR State Change:

None.

IPR State Change:

SIRR ← a0<1..0>.

Operation:

```
if ( PSR<Mode> EQ User ) then
        {initiate illegal instruction exception}
endif
if ( a0<1> EQ 1 ) then
        SIRR<DPC> ← 0
endif
if ( a0<0> EQ 1 ) then
        SIRR<APC> ← 0
endif
```

Exceptions:

Illegal Instruction, Machine Checks

di

Disable all interrupts.

Parameters:

None.

Return Value:

None.

Description:

Di disables all interrupts by clearing the IE (interrupt enable bit) in the PSR. IRQL is unaffected. Interrupts may be re-enabled via the ei call pal function.

GPR State Change:

None.

IPR State Change:

PSR<IE> ← 0.

Operation:

```
if ( PSR<Mode> EQ User ) then
        {initiate illegal instruction exception}
endif
PSR<IE> ← 0.
```

Exceptions:

Illegal Instruction, Machine Checks

draina

Drain all aborts including machine checks.

Parameters:

None.

Return Value:

None.

Description:

Draina is used to drain all aborts, including machine checks from the current processor. See the Alpha AXP SRM Common Architecture for a complete definition (of this so vaguely defined function that it is in fact worthless).

GPR State Change:

None.

IPR State Change:

None.

Operation:

```
if ( PSR<Mode> EQ User ) then
        {initiate illegal instruction exception}
endif
{ implementation-specific drain }
```

Exceptions:

Illegal Instruction, Machine Checks

dtbis

Data translation buffer invalidate single.

Parameters:

a0 = virtual address of translation to invalidate.

Return Value:

None.

Description:

Dtbis is used to invalidate a single data stream translation. The translation for the virtual address in a0 must be invalidated in all data translation buffers and in all virtual data caches.

GPR State Change:

a0 - a5 are unpredictable.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ invalidate all translations in the data stream for the virtual address in a0  }
```

Exceptions:

Illegal Instruction, Machine Checks

ei

Enable interrupts..

Parameters:

None.

Return Value:

None.

Description:

Ei enables interrupts for the IRQL set in the PSR by setting the IE bit in the PSR.

GPR State Change:

None.

IPR State Change:

PSR<IE> ← 1.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
PSR<IE> ← 1.
```

Exceptions:

Illegal Instruction, Machine Checks

halt

Halt the operating system and return to the boot environment.

Parameters:

None.

Return Value:

None.

Description:

Halt stops the operating system from executing and returns execution to the boot environment. Halt is responsible for completing the ARC Restart Block before returning to the boot environment. The PALcode must accomplish 2 tasks to restore the boot environment: re-establish the boot environment PALcode and restart execution in the boot environment at the Firmware Restart Address.

GPR State Change:

All registers are unpredictable.

IPR State Change:

PAL_BASE ← PREVIOUS_PAL_BASE.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
RestartBlockPointer ← PcRestartBlock(PCR)
{ if cannot verify restart block, restart previous PALcode }
{ save general purpose register state in saved state area }
{ save internal processor register state in saved state area, includes OS PAL base address }
{ save implementation-specific data in saved state area }
{ set the saved state length in restart block }
{ set PowerFailFinished in BootStatus of restart block }
{ compute and store Checksum for restart block }
PAL_BASE ← PREVIOUS_PAL_BASE.
{ imb }
{ tbia }
{ di }
RESTART_ADDRESS ← PcFirmwareRestartAddress(PCR)
```

Exceptions:

Illegal Instruction, Machine Checks

initpal

Initialize PAL data structures with operating system values.

Parameters:

a0 = page directory entry page, super-page 32 address
a1 = initial thread value
a2 = initial teb value
a3 = maximum kernel stack size
sp = initial kernel stack pointer
gp = kernel global pointer

Return Value:

None.

Description:

Initpal is called early in the kernel initialization sequence to establish values for IPRs that are needed for trap and fault handling. Some of these values (KGP, PCR, PRCB) are initialized once and persist throughout the run-time of the operating system.

GPR State Change:

a0 - a3 are unpredictable.

IPR State Change:

IKSP ← sp
PDR ← a0
KGP ← gp
PCR ← a1
PRCB ← a2
THREAD ← a3

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
PDE     ← a0
PCR     ← a1
PRCB    ← a2
THREAD ← a3
IKSP    ← sp
KGP     ← gp
PcPalBaseAddress(PCR) ← PAL_BASE
PcPalMajorVersion(PCR) ← PalMajorVersion
PcPalMinorVersion(PCR) ← PalMinorVersion
PcPalSequenceVersion(PCR) ← PalSequenceVersion
PcPalMajorSpecification(PCR) ← PalMajorSpecification
PcPalMinorSpecification(PCR) ← PalMinorSpecification

where:
        PalMajorVersion, PalMinorVersion, PalSequenceVersion are implementation-defined
```

PalMajorSpecification, PalMinorSpecification are the revision numbers of this document to which the PALcode image compiles

Exceptions:

Illegal Instruction, Machine Checks

rdcounters DEBUG ONLY

Read the current values of the software event counters within the PALcode.

Parameters:

a0 = pointer to counter record.

Return Value:

None.

Description:

Rdcounters causes the PALcode to write the state of its internal software event counters into the counter record pointed to by the address passed in the a0 register. The format and content of the software event counter record are implementation-specific.

For production PALcode this code point will be treated as an illegal instruction.

GPR State Change:

None.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ dump event counter values to the counter record }
```

Exceptions:

Illegal Instruction, Machine Checks

rdirql

Read the current IRQL from the PSR.

Parameters:

None.

Return Value:

v0 = current IRQL.

Description:

Rdirql is used to read the current IRQL from the PSR.

GPR State Change:

v0 ← PSR<IRQL>.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← PSR<IRQL>.
```

Exceptions:

Illegal Instruction, Machine Checks

rdksp

Read initial kernel stack pointer for the current thread.

Parameters:

None.

Return Value:

v0 = initial kernel stack pointer.

Description:

Rdksp returns the value of the initial kernel stack pointer for the currently executing thread.

GPR State Change:

None.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← IKSP
```

Exceptions:

Illegal Instruction, Machine Checks

rdmces

Read the machine check error summary register.

Parameters:

None.

Return Value:

v0 = machine check error summary.

Description:

Rdmces returns the contents of the machine check error summary register (MCES).

GPR State Change:

None.

IPR State Change:

v0 ← MCES.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← MCES
```

Exceptions:

Illegal Instruction, Machine Checks

rdpcr

Read the processor control registers base address.

Parameters:

None.

Return Value:

v0 = processor control registers base.

Description:

Rdpcr returns the value of the processor control registers.

GPR State Change:

v0 ← PCR.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← PCR
```

Exceptions:

Illegal Instruction, Machine Checks

rdpsr

Read the current PSR.

Parameters:

None.

Return Value:

v0 = current PSR.

Description:

Rdpsr returns the value of the current PSR.

GPR State Change:

v0 ← PSR

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← PSR
```

Exceptions:

Illegal Instruction, Machine Checks

rdstate

Read the current internal processor state.

Parameters:

a0 = pointer to internal processor state record.

Return Value:

None.

Description:

Rdstate causes the PALcode to write the internal processor state to the internal processor state record pointed to by the address passed in the a0 register. The format and content of the internal processor state record are implementation-specific.

For production PALcode this code point will be treated as an illegal instruction.

GPR State Change:

None.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ dump internal processor state to the processor state record }
```

Exceptions:

Illegal Instruction, Machine Checks

rdthread

Read the thread value for the current thread.

Parameters:

None.

Return Value:

v0 = thread value for the currently executing thread.

Description:

Rdthread returns the value of the currently executing thread.

GPR State Change:

v0 ← THREAD.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← THREAD
```

Exceptions:

Illegal Instruction, Machine Checks

restart

Restart the operating system from the restart block.

Parameters:

None.

Return Value:

None.

Description:

Restart restores saved processor state and resumes execution of the operating system.

GPR State Change:

All registers are unpredictable.

IPR State Change:

All registers are unpredictable.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
RestartBlockPointer ← PcRestartBlock(PCR)
{ restore general purpose register state from saved state area }
{ restore internal processor register state from saved state area, includes OS PAL base address }
{ restore implementation-specific data from saved state area }
{ set RestartFinished in BootStatus of restart block }
{ compute and store Checksum for restart block }
RESTART_ADDRESS ← RbReturnAddress(RestartBlockPointer)
```

Exceptions:

Illegal Instruction, Machine Checks

retsys

Return from trap or interrupt.

Parameters:

a0 = previous PSR.
a1 = new software interrupt requests
fp = pointer to Trap Frame

Return Value:

v0 = system service completion status.

Description:

Retsys returns from a system service call exception by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, retsys accepts a parameter to allow it to set software interrupt requests that have become pending while the exception was handled.

GPR State Change:

ra ← TrIntRa(TrapFrame).
gp ← TrIntGp(TrapFrame).
fp ← TrIntFp(TrapFrame).
sp ← TrIntSp(TrapFrame).
t0 - t12, a0 - a5 are unpredictable.

IPR State Change:

PSR ← a0.
SSIR ← a1<1..0>.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
if ( a1<1> EQ 1 ) then
        SIRR<DPC> ← 1
endif
if ( a1<0> EQ 1 ) then
        SIRR<APC> ← 1
endif
ra ← TrIntRa(TrapFrame).
gp ← TrIntGp(TrapFrame).
fp ← TrIntFp(TrapFrame).
sp ← TrIntSp(TrapFrame).
RESTART_ADDRESS ← TrIntFir(TrapFrame)
PSR ← a0
```

Exceptions:

Illegal Instruction, Machine Checks, Invalid Kernel Stack.

rfe

Return from trap or interrupt.

Parameters:

a0 = previous PSR.
a1 = new software interrupt requests.
fp = pointer to TrapFrame

Return Value:

None.

Description:

Rfe returns from exceptions by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, rfe accepts a parameter to allow it to set software interrupt requests that have become pending while the exception was handled.

GPR State Change:

a0 ← TrIntA0(TrapFrame).
a1 ← TrIntA1(TrapFrame).
a2 ← TrIntA2(TrapFrame).
a3 ← TrIntA3(TrapFrame).
ra ← TrIntRa(TrapFrame).
gp ← TrIntGp(TrapFrame).
fp ← TrIntFp(TrapFrame).
sp ← TrIntSp(TrapFrame).

IPR State Change:

PSR ← a0.
SSIR ← a1<1..0>.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
if ( a1<1> EQ 1 ) then
        SIRR<DPC> ← 1
endif
if ( a1<0> EQ 1 ) then
        SIRR<APC> ← 1
endif
temp ← a0
a0 ← TrIntA0(TrapFrame).
a1 ← TrIntA1(TrapFrame).
a2 ← TrIntA2(TrapFrame).
a3 ← TrIntA3(TrapFrame).
ra ← TrIntRa(TrapFrame).
gp ← TrIntGp(TrapFrame).
fp ← TrIntFp(TrapFrame).
sp ← TrIntSp(TrapFrame).
```

RESTART_ADDRESS ← TrIntFir(TrapFrame)
PSR ← temp

Exceptions:

Illegal Instruction, Machine Checks, Invalid Kernel Stack.

swpirql

Swap the current IRQL (Interrupt Request Level).

Parameters:

a0 = new IRQL.

Return Value:

v0 = previous IRQL..

Description:

Swpirql swaps the current IRQL by setting the processor so that only interrupts that are permitted are enabled for the new IRQL. Swpirql updates the IRQL field of the PSR and returns the previous IRQL.

GPR State Change:

v0 ← PSR<IRQL>.

a0 - a3 are unpredictable.

IPR State Change:

PSR<IRQL> ← a0.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← PSR<IRQL>.
PSR<IRQL> ← a0.
```

Exceptions:

Illegal Instruction, Machine Checks

swpksp

Swap the initial kernel stack pointer for the current thread.

Parameters:

a0 = new initial kernel stack pointer.

Return Value:

v0 = previous initial kernel stack pointer.

Description:

Swpksp returns the value of the previous initial kernel stack pointer and writes a new initial kernel stack pointer for the currently executing thread.

GPR State Change:

v0 ← IKSP

IPR State Change:

IKSP ← a0.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
v0 ← IKSP
IKSP ← a0
```

Exceptions:

Illegal Instruction, Machine Checks

swppal

Swap the currently executing PALcode.

Parameters:

a0 = base address of new PALcode.

Return Value:

None.

Description:

Swppal swaps the currently executing PALcode by transferring to the base address of the new PALcode image (provided in a0) in PAL mode.

GPR State Change:

All are unpredictable.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ imb }
{ tbia }
{ di }
{ jump to address in a0 in PAL mode }
```

Exceptions:

Illegal Instruction, Machine Checks

swpprocess

Swap process context (swap address space).

Parameters:

a0 = page frame number(PFN) of new page directory pointer (PDR).
a1 = address space number(ASN) of new process.

Return Value:

None.

Description:

Swpprocess swaps the privileged process context by changing the address space for the currently executing thread. The address space change is accomplished by establishing a new ASN and a new PDR. In addition, swpprocess is responsible for executing an implicit tbiap whenever the ASN wraps. The tbiap is executed by invalidating all translations and virtual cache blocks that do not have the Global (or Address Space Match) bit set. Address space number wrapping is signaled whenever the address space number is equal to the maximum address space number (MaxASN) for the implementation.

GPR State Change:

a0 - a3 are unpredictable.

IPR State Change:

PDR ← a0.
ASN ← a1.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
temp  ← SHIFT_LEFT( a0, PAGE_SHIFT )
temp ← temp OR Super32Base
PDR ← temp.
ASN ← a1.
if ( ASN EQ MaxASN ) then
        { invalidate all translations and virtual cache blocks that do not have the Global bit set }
endif

where:
        2**PAGE_SHIFT = implementation page size
        Super32Base = base of 32-bit super-page = 0x80000000
```

Exceptions:

Illegal Instruction, Machine Checks

swpctx

Swap thread context.

Parameters:

a0 = va of initial kernel stack pointer for new thread.
a1 = pte for first kernel stack page of new thread.
a2 = pte for second kernel stack page of new thread.
a3 = new thread value.
a4 = new thread environment block pointer.
a5 = new address space PDR page frame number or a negative number
t0 = ASN for new address space

Return Value:

None.

Description:

Swpctx swaps the privileged portions of thread context. Thread context is swapped by establishing the new thread's kernel stack and writing the new thread value and new TEB pointer. The ptes for the new kernel stack are passed to allow the PALcode to optionally establish the kernel stack mappings during the context switch. If the PALcode does not use the ptes then the kernel stack translations will be demand-faulted. Note, if the kernel stack is a 32-bit super-page address then the PALcode must ignore the ptes and not perform the mapping. Swpctx may also swap the address space (or process) for the new thread. If the new thread is in the same process (address space) as the previous thread then the kernel will pass a negative value for the PDR page frame number indicating that the address space need not be switched. If the PDR page frame number is zero or a positive number then the PDR PFN and the ASN will be used to swap the address space just as if swpprocess had been executed.

GPR State Change:

a0 - a5 are unpredictable.

IPR State Change:

IKSP ← a0.
THREAD ← a3.
TEB ← a4.
PDR ← a5 (possibly).
ASN ← t0 (possibly).

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
IKSP ← a0.
THREAD ← a3.
TEB ← a4.
if (  a0 GE 0xc0000000 ) then       ! NOT super-page 32 kernel stack
        { optional: map kernel stack (first insuring that kernel stack translations are not already
        valid) }
endif
if ( a5 GE 0 ) then      ! swap address space
        temp ← SHIFT_LEFT( a5, PAGE_SHIFT )
```

```
            temp ← temp OR Super32Base
            PDR ← temp.
            ASN ← t0.
            if ( ASN EQ MaxASN ) then
                    { invalidate all translations and virtual cache blocks that do not have the
                    Global bit set }
            endif
    endif

    where:
            2**PAGE_SHIFT = implementation page size
            Super32Base = base of 32-bit super-page = 0x80000000
```

Exceptions:

Illegal Instruction, Machine Checks

ssir

Set software interrupt request.

Parameters:

a0 = Software Interrupt Requests to set.

Return Value:

None.

Description:

ssir set software interrupt requests by setting the appropriate bits in the SIRR.

GPR State Change:

a0 - a3 are unpredictable.

IPR State Change:

SIRR ← a0<1..0>

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
if ( a0<1> EQ 1 ) then
        SIRR<DPC> ← 1
endif
if ( a0<0> EQ 1 ) then
        SIRR<APC> ← 1
endif
```

Exceptions:

Illegal Instruction, Machine Checks

tbia

Translation buffer invalidate all.

Parameters:

None.

Return Value:

None.

Description:

Tbia invalidates all translations and virtual cache blocks within the processor.

GPR State Change:

a0 - a5 are unpredictable.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ invalidate all translations and virtual cache blocks within the processor }
```

Exceptions:

Illegal Instruction, Machine Checks

tbis

Translation buffer invalidate single.

Parameters:

a0 = virtual address of translation to invalidate.

Return Value:

None.

Description:

Tbis is used to invalidate a single virtual translation. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches.

GPR State Change:

a0 - a5 are unpredictable.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ invalidate all translations for the virtual address in a0, invalidate in all translation buffers and
in all virtual caches }
```

Exceptions:

Illegal Instruction, Machine Checks

tbisasn

Translation buffer invalidate single for asn.

Parameters:

a0 = virtual address of translation to invalidate.
a1 = address space number

Return Value:

None.

Description:

Tbisasn is used to invalidate a single virtual translation for a specified address space. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches.

GPR State Change:

a0 - a5 are unpredictable.

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
{ invalidate all translations for the virtual address in a0, invalidate in all translation buffers and
in all virtual caches }
```

Exceptions:

Illegal Instruction, Machine Checks

wrentry

Write kernel exception entry routine.

Parameters:

a0 = address of exception entry routine, super-page 32 address
a1 = exception class value

Return Value:

None.

Description:

Wrentry is used to register kernel exception handling routines for an exception class. The address in a0 is registered for the exception class corresponding to the exception class value in a1. The kernel must use wrentry to register an exception handler for each of the exception classes. The relation between the exception classes and the exception class values is captured in the following table:

Table 14 Exception Class Values

Exception Class	Exception Class Value
Panic Exceptions	0
Memory Management Exceptions	1
Interrupt Exceptions	2
System Service Exceptions	3
General Exceptions	4

GPR State Change:

a0 - a3 are unpredictable.

IPR State Change:

*_ENTRY ← a0

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
case a1 begin
        0:
                PANIC_ENTRY ← a0
                break;
        1:
                MEM_MGMT_ENTRY ← a0
                break;
        2:
                INTERRUPT_ENTRY ← a0
                break;
        3:
                SYSCALL_ENTRY ← a0
                break;
```

```
            4:
                    GENERAL_ENTRY ← a0
                    break;
            otherwise:
                    {initiate panic exception}
    endcase;
```

Exceptions:

Illegal Instruction, Machine Checks, Panic Exception

wrmces

Write the machine check error summary register.

Parameters:

a0 = new values for the machine check error summary register.

Return Value:

None.

Description:

Wrmces writes new values for the MCES register. Some of the fields of the register are direct writes while others are writes to reset (clear) bits already set.

GPR State Change:

None.

IPR State Change:

MCES ← a0.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
MCES<DSC> ← a0<4>
MCES<DPC> ← a0<3>
if ( a0<2> EQ 1 ) then
        MCES<PCE> ← 0
endif
if ( a0<1> EQ 1 ) then
        MCES<SCE> ← 0
endif
if( a0<0> EQ 1 ) then
        MCES<MCK> ← 0
endif
```

Exceptions:

Illegal Instruction, Machine Checks

Unprivileged Call PAL functions

Figure 15 Unprivileged Call PALs

Mnemonic	Description	Opcode
bpt	breakpoint trap	0x80
callsys	call system service	0x83
imb	instruction memory barrier	0x86
gentrap	generate trap	0xaa
rdteb	read TEB (thread environment block)	0xab
kbpt	kernel breakpoint trap	0xac
callkd	call kernel debugger	0xad

Reserved unprivileged Call PAL Opcodes

0x81
0x9e
0x9f
0xae
0xaf

bpt

Breakpoint trap (standard user-mode breakpoint).

Parameters:

None.

Return Value:

None.

Description:

Bpt raises a breakpoint general exception to the kernel setting a breakpoint type of USER_BREAKPOINT.

GPR State Change:

See General Exception Dispatch and Breakpoint handling.

IPR State Change:

See General Exception Dispatch and Breakpoint handling.

Operation:

See General Exception Dispatch and Breakpoint handling.

Exceptions:

Illegal Instruction, Machine Checks, Kernel Stack Invalid

callkd

Call kernel debugger.

Parameters:

v0 = Kernel breakpoint type.

Return Value:

None.

Description:

Callkd raises a breakpoint general exception to the kernel setting the breakpoint type with the value supplied in v0. This function is used to implement special calls to the kernel debugger (kd).

GPR State Change:

See General Exception Dispatch and Breakpoint handling.

IPR State Change:

See General Exception Dispatch and Breakpoint handling.

Operation:

See General Exception Dispatch and Breakpoint handling.

Exceptions:

Machine Checks, Kernel Stack Invalid

callsys

System Service Call.

Parameters:

None, a0 - a5 preserved through call, v0 = system service number.

Return Value:

None.

Description:

Callsys raises a system service exception to the kernel. The system call has the software semantics of a standard procedure call so that all volatile registers are considered free. callsys switches to kernel mode if necessary and builds a trap frame on the kernel stack and then enters the kernel at the kernel system service exception handler.

GPR State Change:

t0 ← PSR.

IPR State Change:

PSR<MODE> ← KernelMode

Operation:

```
previousPSR ← PSR
if( PSR<MODE> EQ UserMode ) then
        PSR<MODE> ← KernelMode
        tp  ← IKSP - TrapFrameLength
else
        tp  ← sp - TrapFrameLength
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntRa(tp) ← ra
TrIntGp(tp) ← gp
TrFir(tp) ← ra
TrPsr(tp)  ← previousPSR
gp ← KGP
sp ← tp
TrServiceStack(tp) ← sp
fp ← tp
t0 ← previousPSR<MODE>
RESTART_ADDRESS  ← SYSCALL_ENTRY
```

Exceptions:

Illegal Instruction, Machine Checks, Kernel Stack Invalid

gentrap

Generate a trap.

Parameters:

a0 = trap reason code.

Return Value:

None.

Description:

Gentrap generates a software general exception which will raise an exception to the current thread. The trap reason code will be used to generate the NT status/exception code raised to the current thread. Gentrap is expected to be used to raise software detected exceptions such as bound check errors or overflow conditions.

GPR State Change:

See General Exception Dispatch and Software Exception handling.

IPR State Change:

See General Exception Dispatch and Software Exception handling.

Operation:

See General Exception Dispatch and Software Exception handling.

Exceptions:

Machine Checks, Kernel Stack Invalid

imb

Instruction Memory Barrier.

Parameters:

None.

Return Value:

None.

Description:

Imb guarantees that all subsequent instruction stream fetches will be coherent with respect to main memory. Imb must be issued before executing code in memory that has modified (either by stores from the processor or DMA from an I/O processor).

GPR State Change:

None.

IPR State Change:

None.

Operation:

{ make processor instruction stream coherent with main memory }

Exceptions:

Machine Checks, Kernel Stack Invalid

kbpt

Kernel breakpoint trap.

Parameters:

None.

Return Value:

None.

Description:

Kbpt raises a breakpoint general exception to the kernel setting a breakpoint type of KERNEL_BREAKPOINT.

GPR State Change:

See General Exception Dispatch and Breakpoint handling.

IPR State Change:

See General Exception Dispatch and Breakpoint handling.

Operation:

See General Exception Dispatch and Breakpoint handling.

Exceptions:

Machine Checks, Kernel Stack Invalid

rdteb

Read the thread environment block pointer for the currently executing thread.

Parameters:

None.

Return Value:

v0 = current TEB

Description:

Rdteb returns the thread environment block (TEB) pointer for the currently executing thread.

GPR State Change:

v0 ← TEB

IPR State Change:

None.

Operation:

v0 ← TEB

Exceptions:

Machine Checks, Kernel Stack Invalid

27. Architected Internal Processor Registers

The following internal processor registers are defined across all implementations. It is implementation-dependent how these registers are implemented within the processor.

Table 16 Internal Processor Registers

IPR Name	Description
ASN	Address Space Number of current process
GENERAL_ENTRY	General exception class kernel handler address
IKSP	Initial Kernel Stack Pointer
INTERRUPT_ENTRY	Interrupt exception class kernel handler address
KGP	Kernel Global Pointer
MCES	Machine Check Error Summary
MEM_MGMT_ENTRY	Memory Management exception class kernel handler address
PAL_BASE	PAL image base address
PANIC_ENTRY	Panic exception class kernel handler address
PCR	Processor Control Registers base address
PDR	Page Directory base address
PSR	Processor Status Register
RESTART_ADDRESS	Restart execution address
SIRR	Software Interrupt Request Register
SYSCALL_ENTRY	System Service exception class kernel handler address
TEB	Thread environment block base address
THREAD	Thread unique value (kernel thread address)

ASN

Address Space Number

31 .. 16	15 .. 0
RAZ	ASN

Field	Type	Initial Value	Description
ASN	WO	0	Address space number for current process

Description:

The Address Space Number is a process tag that may be used by the processor to qualify each virtual translation. When the translations are qualified it will not be necessary for the processor to flush all virtual translations for previous processes when performing a context swap or process swap. If a processor does not implement ASNs then its MaximumASN will equal zero and zero will be passed for all context and process swaps. The ASN is provided in the call pal functions swpctx and swpprocess.

GENERAL_ENTRY

General Exception Class kernel handler entry address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Kernel entry address, 32-bit super-page format.

Description:

The GENERAL_ENTRY register contains the entry address of the kernel exception handler for the General class of exceptions. GENERAL_ENTRY is written via the wrentry call pal function.

IKSP

Initial Kernel Stack Pointer.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Initial kernel stack address.

Description:

The IKSP points to the top of the kernel stack for the currently executing thread. IKSP is read via the rdksp call pal function and written via the swpksp call pal function. IKSP is also written in swpctx and during system initialization via initpal.

INTERRUPT_ENTRY

Interrupt Exception Class kernel handler entry address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Kernel entry address, 32-bit super-page format.

Description:

The INTERRUPT_ENTRY register contains the entry address of the kernel exception handler for the Interrupt class of exceptions. INTERRUPT_ENTRY is written via the wrentry call pal function.

KGP

Kernel Global Pointer.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	WO	Unpredictable	Kernel global pointer (gp) value.

Description:

The KGP is the global pointer value for the kernel. The PALcode restores the kernel global pointer to the general purpose register gp whenever dispatching to a kernel exception handler. The KGP is written via the initpal call pal function.

MCES

Machine Check Error Summary

31..5	4	3	2	1	0
RAZ	DSC	DPC	PCE	SCE	MCK

Field	Type	Initial Value	Description
MCK	RW	0	Machine check (uncorrectable) in progress.
SCE	RW	0	System correctable error in progress.
PCE	RW	0	Processor correctable error in progress.
DPC	RW	1	Disable processor correctable error reporting. 1 = disable 0 = enable
DSC	RW	1	Disable system correctable error reporting. 1 = disable 0 = enable

Description:

The Machine Check Error Summary register is used to report and control the current state of machine check handling. MCES is read via the rdmces call pal function and written via the wrmces call pal function. The DPC and DSC bits control whether correctable errors are reported and are fully read/write bits. The MCK, SCE, and PCE are reporting bits that can be reset by writing a '1' to their corresponding positions with the wrmces call pal function. The reporting bits can only be set by the PALcode machine check handler.

MEM_MGMT_ENTRY

Memory Management Exception Class kernel handler entry address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Kernel entry address, 32-bit super-page format.

Description:

The MEM_MGMT_ENTRY register contains the entry address of the kernel exception handler for the Memory Management class of exceptions. MEM_MGMT_ENTRY is written via the wrentry call pal function.

PAL_BASE

PAL image base address.

PA_BITS .. K	K-1 .. 0
ADDR	RAZ

where:

PA_BITS = physical address bits for the implementation
2**K = minimum PAL byte alignment for the implementation

make note that OS Loader will use 64K boundaries

Field	Type	Initial Value	Description
ADDR	RW	PAL image base	Physical address of currently active PAL image.

Description:

The PAL_BASE register controls which PALcode image is current active. PAL_BASE is the physical address of the base of the currently active PALcode image. The hardware will vector into the appropriate PALcode handlers as offsets from the base in the PAL_BASE register. The offsets for each handler and the type of handler are implementation-specific, except for the reset vector. The reset vector is the PALcode initialization vector and must begin at offset 0 within the PALcode image.

PANIC_ENTRY

Panic Exception Class kernel handler entry address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Kernel entry address, 32-bit super-page format.

Description:

The PANIC_ENTRY register contains the entry address of the kernel exception handler for the Panic class of exceptions. PANIC_ENTRY is written via the wrentry call pal function.

PCR

Processor Control Registers.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Base address of the processor control registers page. The address must be in 32-bit super-page format.

Description:

The PCR is the base address of the processor control registers page. The address must be a 32-bit super-page address. The processor control registers is a page of per-processor data. The PCR is written via the initpal call pal function and read via the rdpcr call pal function.

Page Directory Page

31 .. 0


ADDR

Field	Type	Initial Value	Description

ADDR	RW	Unpredictable.	Base address of the page directory page in 32-bit super-page format.

Description:

The PDR is the base address of the page directory page. The page directory page contains all of the first-level page table entries (page directory entries or PDEs). As such, the page directory page defines an address space for a process. The PDR is written whenever the address space is swapped via the call pal functions swpctx (sometimes) and swpprocess (always). The PDR is also written with the initpal call pal function.

PSR

Processor Status Register

31 .. 5	4 .. 2	1	0
RAZ	IRQL	IE	MODE

Field	Type	Initial Value	Description
MODE	RW	0 = KernelMode	Processor mode: privileged (kernel) or unprivileged (user) 0 = KernelMode 1 = UserMode
IE	RW	1 = enabled	Interrupt Enable 0 = Interrupts disabled, 1 = Interrupts enabled
IRQL	RW	7 = HIGH_LEVEL	Interrupt Request Level (0 .. 7) where ascending values are higher priority

Description:

The PSR controls the privilege state and interrupt priority of the processor. There are 2 processor privilege states: kernel (privileged) and user (unprivileged). When the processor is privileged it has access to the super-page address spaces and may execute the privileged call pal instructions. In addition, the processor may access virtual addresses mapped by any and all valid page table entries. In unprivileged mode, the processor has no access to the super-page address spaces (access violations) or the privileged call pal functions (illegal instructions). In addition, the processor may only use virtual addresses mapped by PTEs that have OWNER = User(1). The interrupt request level controls which interrupt sources are enabled. The interrupt enable bit is a global enable bit then can be used to disable all interrupts. The PSR can be read with the rdpsr call pal function and written with the rfe and retsys call pal functions. In addition, call pal functions are provided to access individual fields within the PSR. The swpirql call pal function atomically reads and writes the IRQL field of the PSR. The rdirql call pal function reads the current IRQL field of the PSR. The di and ei functions are provided to, respectively, clear and set the IE field of the PSR.

This explanation also belongs in the Processor Status section.

RESTART_ADDRESS

Restart execution address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	WO	Unpredictable.	The address to resume execution when PALcode is exited.

Description:

The RESTART_ADDRESS is the address where the processor will resume execution when the PALcode exits. The RESTART_ADDRESS is considered to contain the virtual address + 4 of the call pal instruction for all call pal functions.

Include this comment when describing call pal listing format.

SIRR

Software Interrupt Request Register

31 .. 2	1	0
RAZ	DPC	APC

Field	Type	Initial Value	Description
APC	RW	0	APC software interrupt requested.
DPC	RW	0	DPC software interrupt requested.

Description:

The SIRR indicates requested software interrupts. The SIRR is also considered the source for software interrupt requests. The ssir call pal function is used request software interrupts. The csir call pal function is used to clear software interrupt requests. In addition, software interrupt requests in the SIRR will be cleared by the PALcode during interrupt dispatch.

SYSCALL_ENTRY

System Service Exception Class kernel handler entry address.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	Unpredictable	Kernel entry address, 32-bit super-page format.

Description:

The SYSCALL_ENTRY register contains the entry address of the kernel exception handler for the System Service class of exceptions. SYSCALL_ENTRY is written via the wrentry call pal function.

TEB

Thread Environment Block.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	0	The address of the user thread environment block.

Description:

The TEB is the pointer to the user thread environment block. The TEB is written with each swpctx call pal and read with the rdteb call pal function.

THREAD

Thread Environment Block.

31 .. 0
ADDR

Field	Type	Initial Value	Description
ADDR	RW	0	The address of the kernel thread structure.

Description:

The THREAD is the pointer to the currently executing kernel thread structure. The THREAD is written with each swpctx call pal and read with the rdthread call pal function.

28. Appendices

Appendix A. Status code and bugcode values

Status Codes:

Symbol	Value
STATUS_DATATYPE_MISALIGNMENT	0x80000002
STATUS_ALPHA_GENTRAP	0xc00000aa
STATUS_ALPHA_ARITHMETIC	0xc0000092
STATUS_ILLEGAL_INSTRUCTION	0xc000001d
STATUS_BREAKPOINT	0x80000003
STATUS_INVALID_ADDRESS	0xc0000141

Bugcodes:

Symbol	Value
DATA_BUS_ERROR	0x2e
PANIC_STACK_SWITCH	0x2b
TRAP_CAUSE_UNKNOWN	0x12

Appendix B. PCR definitions and offsets

pointer to restart block for ARC restart block?

Need another appendix for Saved State offsets within the restart block.

Appendix C. Trap Frame definitions and offsets

Symbolic Name	Offset	Size
TrIntSp	0x0	Quadword
TrPsr	0x8	Longword
TrFir	0x10	Quadword
TrServiceStack	0x18	Quadword
TrIntA0	0x20	Quadword
TrIntA1	0x28	Quadword
TrIntA2	0x30	Quadword
TrIntA3	0x38	Quadword
TrIntFp	0x40	Quadword
TrIntGp	0x48	Quadword
TrIntRa	0x50	Quadword
TrIntV0	0x58	Quadword

Appendix D. Calling standard register usage

General purpose integer registers

Register Number	Symbolic Name	VOLATILITY	DESCRIPTION
r0	v0	volatile	return value register
r1 - r8	t0 - t7	volatile	temporary registers

r9 - r14	s0 - s5	non-volatile	saved registers
r15	s6/fp	non-volatile	saved register/frame pointer
r16 - r21	a0 - a5	volatile	argument registers
r22 - r25	t8 - t11	volatile	temporary registers
r26	ra	weird	return address register
r27	t12	volatile	temporary register
r28	at	volatile	assembler temporary register
r29	gp	non-volatile	global pointer
r30	sp	non-volatile	stack pointer
r31	zero	constant	RAZ / writes ignored

General purpose floating point registers

Register Number	Volatility	Description
f0	volatile	return value register (real part)
f1	volatile	return value register (imaginary part)
f2 - f9	non-volatile	saved registers
f10 - f15	volatile	temporary registers
f16 - f21	volatile	argument registers
f22 - f30	volatile	temporary registers
f31	constant	RAZ / writes ignored

Appendix E. Performance priority table

This table is a rough picture of the performance prioritization for each of the call pals and likely hardware vectors in the PALcode. Certainly, the relative usage of each of the flows will vary on an application by application basis but the table below gives a starting point for PALcode designers to make tradeoffs in their specific implementations. The numbers below are based on actual counts retrieved from a running NT system.

The ratings are based on a relative performance of 1..100. With increasing numbers indicating increasing frequency of execution. If a flow or call pal function is not listed in the table than its effective rating could be considered a 0.

Hardware vector/Call PAL function	Rating
Interrupts	5
I-stream TB miss	50
D-stream TB miss (native)	25
D-stream TB miss (PAL)	5
rfe	5
retsys	5
swpirql	95
di	5
ei	5
ssir	1
csir	1
tbis	1
rdpsr	10
swpctx	1

swpprocess	1
rddpcflag	5
wrdpcflag	5
rdpcr	5
rdprcb	5
rdthread	50
callsys	5
rdteb	10

More complete measurements must be made under differing loads before we can publish the previous table.

Appendix F. Implications of recursive TB Mappings

The recursive virtual mapping used by NT has an implication for data translation buffer implementation. The implication is that it is possible for 2 exactly identical translations to be written in the DTB during the same miss handling sequence. If the DTB cannot correctly operate with 2 identical translations then the PALcode must include additional checks to prevent the condition from occurring.

Note: I believe that this condition also occurs for both VMS and OSF/1 implementations on Alpha AXP.

The description of memory management will not be completely re-iterated here. Refer to the memory management section for details of the page table layout.

The page tables can either be viewed as a virtual contiguous single-level table or as a two-level table that must be traversed physically. When viewed as a two-level table the first level is a single page called the PDR (for page directory). Each entry in the PDR, called a PDE, provides the first-level translation so that the TB fill code can find the page table page that contains the PTE with the translation for the faulted virtual address. All page table pages are mapped by a PDE in the PDR.

The page tables are recursive in that the PDR is a standard page table page and it is virtually mapped in the single-level virtual page table. Therefore, there exists one PDE that maps the PDR itself. The PDE that maps the PDR in a 2 level lookup is also the PTE that maps the PDR for the single level virtual mapping. This special PDE is called the root PTE or RPTE.

Assume that the processor implementation has 2 data stream TB miss flows: one for the misses taken in native mode and one for the misses taken in PALmode (the 21064-aa is such processor). For the case when a native-mode virtual access is made to the PDR page, obvious PALcode will take the following flows:

Native Miss Flow	PAL Miss Flow
1. {get va for pte that maps the faulted va: VA}	
2. {get the pte using its va} ldl rx, 0(ry) where ry ← va of pte	
	3. { ldl rx, 0(ry) from PALmode faulted }
	4. { resolve this fault by making the va of

the missed pte valid }

5. { translation for RPTE is written into the DTB }

6. {re-execute the load that failed since the va of the pte is now valid}

7. { load completes, rx ← RPTE }

8. { write the translation for the faulting va, VA, into the DTB }

9. { RPTE is now in the DTB twice }

10. { re-execute the original native-mode instruction that faulted when accessing VA }

Since there is only one pte, RPTE, that will exhibit this behavior, the PALcode could check the faulting pte address in the second-level fill routine to special case for RPTE. Naturally, it would be preferable to not slow down even the second-level fill flow, however, this is a processor-implementation decision.

D. Dump State

The state written by the PALcode for the rdstate call pal is described by the following structure written in C language format (UQUAD is a 64-bit unsigned value).

```
struct _PROCESSOR_STATE_21064{
        UQUAD ItbPte[12];
        UQUAD Iccsr;
        UQUAD Ps;
        UQUAD ExcSum;
        UQUAD PalBase;
        UQUAD Hirr;
        UQUAD Sirr;
        UQUAD Astrr;
        UQUAD Hier;
        UQUAD Sier;
        UQUAD Aster;
        UQUAD DtbPte[32];
        UQUAD MmCsr;
        UQUAD Va;
        UQUAD AltMode;
        UQUAD PalTemp[32];
        UQUAD DcStat;
        UQUAD BiuStat;
        UQUAD BiuAddr;
        UQUAD FillAddr;
        UQUAD FillSyndrome;
} PROCESSOR_STATE_21064, *PPROCESSOR_STATE_21064;
```

The fields correspond directly to the internal processor registers.

E. Counter State

The counter state written the PALcode for the 21064 is defined by the following C language structure definition (where UQUAD is an unsigned 64-bit value):

```
struct _COUNTERS_21064{
        UQUAD MachineCheckCount;
        UQUAD ArithmeticExceptionCount;
        UQUAD InterruptCount;
        UQUAD ItbMissCount;
        UQUAD NativeDtbMissCount;
        UQUAD PalDtbMissCount;
        UQUAD ItbAcvCount;
        UQUAD DtbAcvCount;
        UQUAD UnalignedCount;
        UQUAD OpcdecCount;
        UQUAD FenCount;
        UQUAD HaltCount;
        UQUAD RestartCount;
        UQUAD DrainaCount;
        UQUAD InitpalCount;
        UQUAD WrentryCount;
        UQUAD SwpirqlCount;
        UQUAD RdirqlCount;
        UQUAD DiCount;
        UQUAD EiCount;
        UQUAD SwppalCount;
        UQUAD SsirCount;
        UQUAD CsirCount;
        UQUAD RfeCount;
        UQUAD RetsysCount;
        UQUAD SwpctxCount;
        UQUAD SwpprocessCount;
        UQUAD RdmcesCount;
        UQUAD WrmcesCount;
        UQUAD TbiaCount;
        UQUAD TbisCount;
        UQUAD DtbisCount;
        UQUAD RdkspCount;
        UQUAD SwpkspCount;
        UQUAD RdpsrCount;
        UQUAD RdprcbCount;
        UQUAD RdthreadCount;
        UQUAD RddpcflagCount;
        UQUAD WrdpcflagCount;
        UQUAD RdcountersCount;
        UQUAD RdstateCount;
        UQUAD InitpcrCount;
        UQUAD WrperfmonCount;
        UQUAD BptCount;
        UQUAD CallsysCount;
```

```
        UQUAD ImbCount;
        UQUAD GentrapCount;
        UQUAD RdtebCount;
        UQUAD KbptCount;
        UQUAD DbgstopCount;
        UQUAD DbgprintCount;
        UQUAD DbgpromptCount;
        UQUAD DbgloadCount;
        UQUAD DbgunloadCount;
} COUNTERS_21064, *PCOUNTERS_21064;
```

wrperfmon

Write performance counter interrupt control information.

Parameters:

a0 = performance counter (0 or 1)
a1 = enable/disable (1 = enable, 0 = disable)
a2 = count control
a3 = performance counter mux

Return Value:

v0 = previous enable state for this performance counter

Description:

Wrperfmon is used to write control information for the 2 performance counters within the 21064-aa processor. The first parameter selects which performance counter will be selected. The second parameter controls if the performance counter is to be enabled or disabled. If it is disabled then the performance counter will cease to interrupt when its counter overflows and the last 2 parameters will be ignored. If the performance counter is enabled then the last 2 parameters will be written directly to the ICCSR internal processor register. The count control parameter will be written to the appropriate PCx (PC0 or PC1) field of the ICCSR. This parameter controls the overflow count for the performance counter (the performance counter causes and interrupt when it overflows). The final parameter will be written to the appropriate PCMUXx (PCMUX0 or PCMUX1) field of the ICCSR. The performance counter mux controls which event is counted by the performance counter. See the 21064-aa specification for programming information.

GPR State Change:

v0 ← previous enable state for the selected performance counter.
a0 - a3 are unpredictable..

IPR State Change:

None.

Operation:

```
if ( PSR<MODE> EQ User ) then
        {initiate illegal instruction exception}
endif
case a0 begin
        0:      ! performance counter 0
                if( a1 = 0 ) then
                        v0 ← previous enable state performance counter 0
                        { disable performance counter interrupt 0 }
                        break;
                endif
                v0 ← previous enable state performance counter 0
                { enable performance counter interrupt 0 }
                ICCSR<PC0> ← a2
                ICCSR<PCMUX0> ← a3
                break;

        1:      ! performance counter 1
                if( a1 = 0 ) then
```

```
                    v0 ← previous enable state performance counter 1
                    { disable performance counter interrupt 1 }
                    break;
                endif
                v0 ← previous enable state performance counter 1
                { enable performance counter interrupt 1 }
                ICCSR<PC1> ← a2
                ICCSR<PCMUX1> ← a3
                break;

        otherwise:
                { ignore the request }
    endcase;
```

Exceptions:

Illegal Instruction, Machine Checks

3. SRM conflicts

a. unaligned access assumptions
The Alpha Architecture handbook states on page 4-2 that "General-purpose layered and application software that executes in User mode may assume that certain loads ... and certain stores ... of unaligned data are emulated by system software". NT applications cannot make such an assumption. Unaligned emulation is an operating system policy, NT implements the policy that unaligned emulation is selectable on a per-thread basis with the default selection having fix-ups turned off.

b. rdunique/ wrunique/bugchk
The SRM claims that these opcodes must be recognized by opcode and mnemonic but their effect is implementation dependent. The NT PALcode will raise an illegal instruction exception if any of these opcodes is executed.

Bugchk is not needed in NT because NT supplies structured mechanisms for raising an exception. It is not desired for NT because it would provide a user facility that is not source compatible with the other NT implementations. Any code that would wish to use such a facility as bugchk should use the common interfaces.

Rdunique and wrunique are also not needed in NT because NT has multi-threadedness built in with complete with a thread environment block and thread local storage. Any code that would wish to use rdunique/wrunique should use NT common interfaces to implement the required effect as these call pals will not be available in other architectures.

4. To do list

a. Determine correct trademark terms to use: Alpha AXP OK? Can we shorten to NT for the discussions?

b. Make the non-call pal flows look like the call pal code flows.

c. Change name of interrupt tables:
interrupt level table -> interrupt enable table
interrupt mask table -> interrupt vector and synchronization table

d. Separate tables and figures in their own tables of contents

e. Provide table of contents for the call pals and the internal processor registers

g. Condense IPRs by having multiples per page??? (leave call pals as is for readability)

h. Add restricted distribution stuff.