UNISYS

BTOS Protected Mode

Programming Guide

Relative to Release
Level 1.0

Priced Item

February 1987
Distribution Code SA
Printed in U S America
5026065

UNISYS

BTOS Protected Mode

Programming Guide

Relative to Release
Level 1.0

Priced Item

February 1987
Distribution Code SA
Printed in U S America
5026065

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the CLASS specified as 2 (S.SW:System Software), the Type specified as 1 (F.T.R.), and the product specified as the seven-digit form number of the manual (for example, 5026065).

About This Guide

Protected mode allows you to address up to 4 Mb of memory on B 28 and B 38 workstations that have BTOS 8.0 installed, limit damage caused by erroneous programs, and dynamically allocate memory more effectively than in real-address mode. This guide describes the protected mode capabilities of the Intel 80286 and 80386 microprocessors. It includes examples specific to BTOS and the B 28 and B 38 workstations.

Who Should Use This Guide

This guide will help you if you are an experienced BTOS programmer. To understand the information in this guide, you must be familiar with BTOS 8.0.

How to Use This Guide

If you are using the protected mode for the first time, you should read section 1. It contains information you will need to understand basic protected mode concepts.

In any case, if you scan the table of contents and review the topics before you start, you may find this guide easier to use. To find definitions of unfamiliar words or mnemonics, use the glossary; to locate specific information, use the Index.

How This Guide is Arranged

This material is divided into sections, with related subjects grouped together. Section 1 describes the basic concepts involved in protected mode operations.

Sections 2 and 3 describe how to write and debug protected mode programs. Section 4 describes the RAM-disk cache facility that the Protected Mode Operating System Service (PMOSS) provides.

Appendixes A and B describe programming interfaces discussed in this guide.

Conventions

The Intel 80386 microprocessor is a proper superset of the 80286 microprocessor. Unless otherwise stated, references to the 80286 microprocessor in this guide include the 80386 microprocessor.

What this guide refers to as a **process**, Intel refers to as a **task.** This guide uses the term **task** to mean an entire executing program, which may contain multiple processes. This guide uses the Intel terminology only when referring to a specific hardware-defined control structure, the task state segment (TSS), which you may think of as a process state segment.

The following conventions are also used in this guide:

Convention	Meaning
LDT	Local Descriptor Table, which PMOSS constructs and maintains for each run file executing in protected mode.
RA	Relative address, which comprises one-half of the linear address. Often referred to as an offset from the segment address.
SA	Segment address, which comprises one-half of the linear address.
SN	A segment address that is in protected mode.
SR	A segment address that is a paragraph number (a real-address mode SA).

When two keys are used together for an operation, their names are hyphenated. For example, **ACTION-GO** means you hold down **ACTION** and press **GO**.

Related Documentation

For information on protected mode addressing with the 80286 microprocessor, refer to the *iAPX 286 Programmer's Reference Manual*, published by Intel Corporation.

For information on protected mode addressing with the 80386 microprocessor, refer to the *80386 Programmer's Reference Manual*, published by Intel Corporation.

For information on the System Performance Accelerator (SPA), refer to the *BTOS System Performance Accelerator (SPA) Installation Guide.*

Contents

Illustrations

Tables

Introduction to Protected Mode

This section summarizes protected mode addressing concepts. It serves as an introduction to the Intel *iAPX 286 Programmer's Reference Manual* and *80386 Programmer's Reference Manual.*

Note: This guide refers to some Intel conventions differently than the way Intel refers to them. For more information, refer to Conventions, in the introduction.

Overview of Protected Mode

Intel microprocessors beginning with 80286 (for B 28 workstations) and including the 80386 (for B 38 workstations) support protected mode operation. Their predecessors, 8086 (for B 21 and B 22 workstations) and 80186 (for B 26 workstations), support the real address mode only.

Protected mode offers the following advantages over real address mode:

- a much larger address space (memory) is available, eliminating the 1 Mb constraint of real address mode
- program execution is subject to protection checks that limit damage from erroneous programs
- on-chip memory management hardware is available, allowing an operating system to dynamically allocate memory more effectively
- with the Protected Mode Operating System Server (PMOSS), the code segments of the system services can be executed in protected mode

Note: PMOSS develops system services that will run on the fully protected mode operating system. It also frees memory space in the real address range.

Two constraints when using protected mode are:

- Because addressing concepts are different and the protection hardware encapsulates programs are in private address spaces, you must modify most programs to make them compatible with protected mode.

- Major enhancements to the operating system are required, in part to support descriptor tables and partially to process state structures that the hardware expects to access directly when it is running in protected mode.

Because of compatibility problems, Unisys B 28 and B 38 workstations support concurrent execution of protected mode and real-mode software.

PMOSS also addresses the second protected mode constraint by enhancing BTOS to manage global and local descriptor tables (GDT/LDT), Task State Segments (TSS), an Interrupt Descriptor Table (IDT), and the memory beyond 1 Mb that is accessible only in protected mode.

Reviewing Segmented Addressing

With Intel microprocessors, instructions do not accept physical addresses as operands; they accept only SA:RA logical addresses. A linear address is formed from two 16-bit parts:

- the segment address (SA)
- the relative address (RA)

Together, these two parts comprise a logical address. When using the Assembler or Debugger, the syntax **SA:RA** writes a logical address.

As each instruction executes, the linear address forms from the logical address and addresses physical memory. There is no alternative use of logical addresses, because there is no way to address physical memory directly with an instruction.

The Intel architecture is referred to as a segmented addressing model because every address is always relative to some SA. Observe the contrast to a linear addressing model such as the Motorola architecture, where instructions accept 32-bit linear addresses rather than SA:RA pairs.

When an SA is a real address mode SA, it may be referred to as an SR to distinguish it from a protected mode SA (which is sometimes called an SN). The term SA applies to either real address or protected mode. In real address mode, all logical addresses are actually SR:RA addresses. In protected mode, they are actually SN:RA addresses. The following text describes the difference between an SR and an SN.

Reviewing Real Address Mode

Real address mode is the only mode in which 8086 and 80186 microprocessors operate. However, 80286 microprocessors execute initially in real address mode when powered-up or reset, but can switch to protected mode (if the operating system software necessary to support protected mode is present).

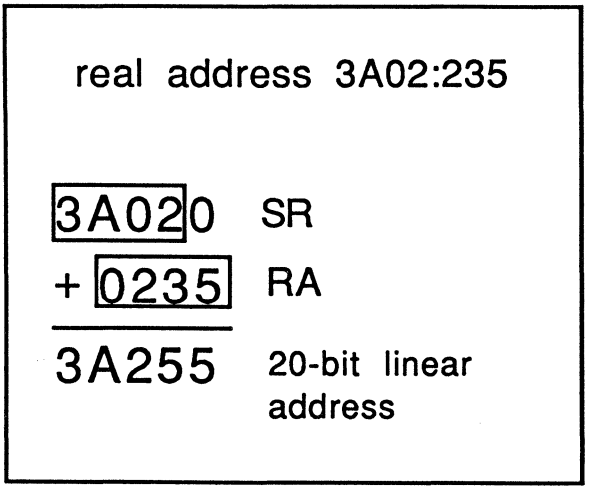
In real mode, to arrive at a 20-bit linear address result, the SR shifts left by four binary places (effectively multiplying by 16) and the RA is added. For example, figure 1-1 shows the logical address 3A02:235 in hexadecimal notation.

The resulting 20-bit quantity can only address 1 Mb of memory (2 exp 20 locations). A 16-byte unit of memory aligned on a 16-byte boundary is called a paragraph. The real address mode SR is a paragraph number, because it denotes a particular 16-byte boundary in the physical address space.

The RA is often referred to as an offset from the SA. In real address mode, the segment registers CS, DS, ES, and SS contain the paragraph numbers corresponding to the base of the current code, data, extra, and stack segments respectively.

These segments are always aligned to start on 16-byte boundaries.

Figure 1-1. **Real Address Mode**



Protected Mode Addresses

The 80286 microprocessor can address up to 16 Mb of memory (2 exp 24) in protected mode, but it requires a 24-bit linear. The 80386 can address up to 4 gigabytes (Gb) of memory (2 exp 32).

No Intel microprocessor, including the 80286, can address more than one Mb in real mode, because only a 20-bit address is formed by the address calculation described in Reviewing Real Address Mode, in this section.

To achieve this longer address compatibly and allow other features of protected mode to be implemented, the same two-part addressing scheme (the segmented addressing model) is used. The address still consists of an SA:RA, but the SA part of the logical address is interpreted differently. The RA part has the same meaning as in real mode.

In protected mode, the 16-bit SA that is held in the CS, DS, ES, or SS register is not a paragraph number; rather, it is an index into a special type of table (called a descriptor table) that is accessible only to the operating system and the hardware. This index is called a selector (SN).

Note: An SN is an SA that is a paragraph number (a real address mode SA).

Paragraph numbers address segments in real mode and selectors address segments in protected mode. For each run file executing in protected mode, PMOSS builds and maintains such a table, called a Local Descriptor Table (LDT) from information that the Linker provides in the header portion of the V6 run file.

Note: This is why you must use the **Bind** command, which produces the new V6 run file format, to link programs that will run in protected mode.

Each code or data segment in the program has a unique selector, which the Linker assigns, and a corresponding unique entry in the LDT. The LDT is an array of these entries, called descriptors, which are eight bytes long and contain segment information. The selector is, in fact, an offset into the LDT, with some additional bits used for special purposes.

When an instruction loads a segment register in a protected mode program, the hardware uses the selector to find the descriptor and retrieve a segment base address from the descriptor. This base address is 24 bits long on the 80286. When an instruction refers to an operand (using the segment register and an RA), the RA is added to the base address to obtain a 24-bit operand address. Unlike real address mode, this base address does not shift; it is not a paragraph number, but a true byte address.

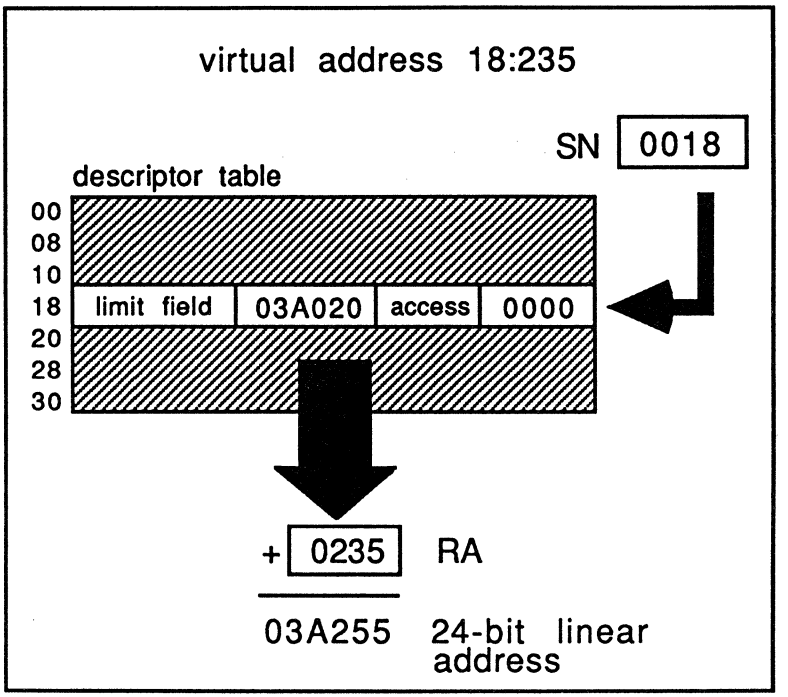
Figure 1-2 shows this process schematically. The example, 18:235, is equivalent to the previous example of real address mode shown in figure 1-1.

This time, the descriptor holds the linear base address, 03A020. Instead of the segment register holding the paragraph number 03A02, it holds a selector value, 18, which is used to index into the descriptor table.

Note: A logical address (SA:RA) consists of two 16-bit parts in both real and protected modes, the only difference being the value of the SA part.

The protected mode SN:RA logical address is referred to as a virtual address because the SN refers only indirectly to memory via a descriptor table entry. By changing the descriptor table entry, the operating system can make the virtual address refer to different physical memory (for example, to move the segment transparently to the program).

Figure 1-2. **Protected Mode**



The real address mode SR:RA logical address is referred to as a real address because it always corresponds to the same physical memory address.

Most incompatibilities between real mode and protected mode arise from this difference between paragraph numbers (SRs) and selectors (SNs), which can be subtle.

The Selector

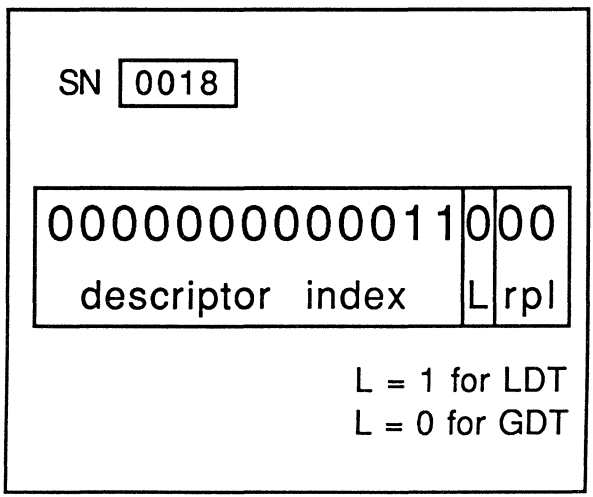
Figure 1-3 shows the anatomy of the 16-bit selector (SN). The high-order 13 bits form the offset that indexes into the descriptor table. The next bit discriminates between two kinds of descriptor tables (LDT and GDT), and the low-order two bits (the Requested Prvilege Level, or RPL) concern a seldom-used feature of the protection scheme. To find the descriptor, the hardware assumes the low-order 3 bits are zero and uses the selector as an offset (since descriptors are 8 bytes long).

The 13 index bits of the selector can generate 2 exp 13, or 8192, possible values. Thus the hardware permits 8192 entries in the LDT and in the GDT. In practice, this means each run file can have a private virtual address space of up to 8192 segments (or other elements).

Descriptor Tables

Two kinds of descriptor tables address segments: Local Descriptor Tables (LDTs) and the Global Descriptor Table (GDT). There must be a separate descriptor table for each run file executing in protected mode, including PMOSS. Each user run file has an LDT. PMOSS has the GDT, which you can think of as PMOSS' own LDT.

Figure 1-3. **Anatomy of a Selector**



At any one time, a particular LDT is the current LDT. Each protected mode process is associated permanently with an LDT when the process is created. (Several processes can share the same LDT if they are part of the same program.) When process switching occurs, the firmware changes the current LDT automatically.

The GDT is special because it is never switched. Regardless of which LDT is in effect, the single GDT is always in effect as well. The GDT is PMOSS' LDT and the descriptors in it are only usable when PMOSS' code is executing, never when user code is executing.

Note: If the system designer wants, the hardware allows user programs to implement GDT descriptors. PMOSS, however, does not usually use this feature.

Because both the GDT and the current LDT are in effect at the same time, PMOSS can use the descriptors in both the GDT and the currently executing user program's LDT. This allows PMOSS to access the memory of user programs.

Because it has its own always-available descriptor table (the GDT), PMOSS can be called at certain entry points, at any time, as a user process subroutine. The operating system services known as kernel primitives and system common procedures are implemented this way.

Note: A protected mechanism known as the call gate limits the user program to calling at legitimate entry points and provides a convenient way to bind the user program to those entry points at program load time.

Descriptor Cache Registers

When one of the registers CS, DS, ES, or SS is loaded with a selector value, the machine fetches the appropriate descriptor from the GDT or LDT and loads it into an associated descriptor cache register. This internal register is not visible to software.

The presence of these invisible registers alleviates the hardware from fetching information from the descriptor table for every memory operand of an instruction. Instead, descriptors are examined only when segment registers are loaded.

Note: Loading a segment register is more time-consuming in protected mode than real mode, because of the memory accesses needed to load the descriptor cache register. Ordinary instructions, however, are not more time-consuming.

The descriptor cache register holds the entire contents of the descriptor, which includes other information besides the segment base address. Although descriptors are always eight bytes long, only six bytes are actually used on the 80286.

Faults

Several checks are associated with loading a segment register and its descriptor cache register. There must be an entry in the descriptor table for the selector that was loaded; the system knows the size of the descriptor table (which varies from table to table) and verifies that the selector does not index beyond the end of the table. The system uses information from the descriptor to perform other checks. In addition to the base address, the descriptor contains the size of the segment, the type of segment, and other protection-related information. You cannot access accidentally beyond the end of the segment in protected mode, execute data segments, and write to code segments. You can protect certain data segments from being written and certain code segments from being read.

Not all descriptors describe memory segments. There are other uses for selectors than as components of SN:RA memory addresses. PMOSS uses nonsegment descriptors for many purposes but, except for gate descriptors, they seldom involve user programmers.

If the selector is invalid, a fault results. Faults transfer control automatically to PMOSS. Conceptually, there are two type of faults: restartable faults and exceptions. Restartable faults are, in theory, recoverable; an exception is an error that prevents a program from further executing.

An example of a restartable fault is the not-present fault that occurs when the operating system marks a segment's descriptor to indicate that the segment is not resident, but is swapped out on disk.

A restartable fault lets the operating system:

- read the missing segment into any available free memory
- load the base address in the descriptor
- mark the descriptor present
- return to the interrupted program (which again tries the instruction that faulted, since the value of IP still points to that instruction)

An example of an exception is a bad selector due to a programming error.

Because PMOSS does not support virtual memory, it treats faults as exceptions. Any fault or exception causes the system to enter the Debugger and displays a diagnostic message.

As a result, you cannot load values other than selectors into the ES register. When you do, the hardware tries to fetch the associated descriptor, resulting in a fault — even before you try to use the segment register to reference anything.

The selector value 0 is special. It can be loaded into a segment register, but causes a fault if it is subsequently used in an address calculation. The 0 value allows the passing of, for example, pbPassword = 0 and cbPassword = 0 in a BTOS OpenFile call. Although you can use 0, do not use the value to calculate an address.

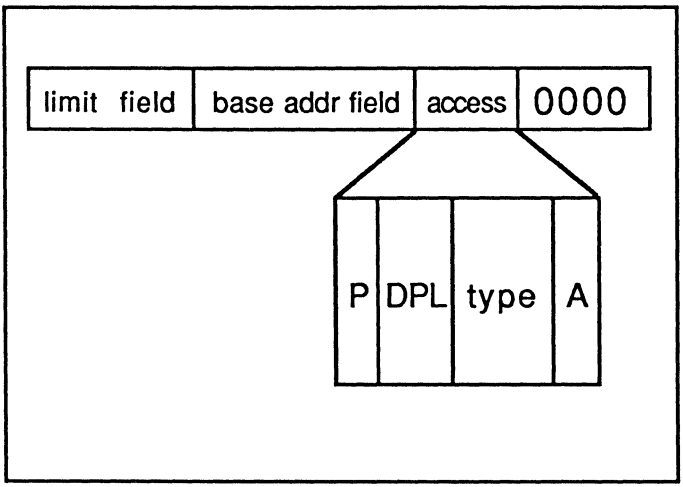
Additional exceptions that can occur later, when using a valid segment register in an address calculation. Trying to address beyond the end of a segment is this type of an exception, called a limit exception.

Descriptor Types

Segment Descriptors

The segment descriptor is involved with SN:RA virtual addresses. Figure 1-4 shows the segment descriptor format used on the 80286.

Figure 1-4 **Segment Descriptor**



The meanings of the various fields are as follows:

Field	Meaning
P	Present bit. This bit, used in virtual memory management, is 0 when the segment is swapped out. When a register is loaded with a selector that fetches this descriptor, if this bit is 0, a fault occurs and the operating system can read the segment in and mark it present.
DPL	Descriptor Privilege Level. This field is used for protection (refer to Protection, in this section).
Type	Several types of code and data segments are possible. This field also identifies the descriptor as a segment descriptor (as opposed to a call gate, for example).
A	Accessed bit. This bit, used by the least recently used (LRU) algorithms in virtual memory management, is set the first time the descriptor loads into a descriptor cache.
Base	The 24-bit base address.
Limit	The maximum RA value that can be used in an SN:RA, where the SN denotes this descriptor. The minimum RA value is zero. A fault occurs if the program tries to access out of segment bounds. The limit is one less than the size of the segment. This field has a different interpretation for some stack segments.

Note: For data segments of a special type called expand-down or grow-down segments, the maximum is always 65535. The limit denotes the minimum allowable RA, which is one less than the minimum legal RA. These segments generally contain stacks, although stacks can also reside in ordinary expand-up segments. For expand-down segments, the base field does not point to the low-order end of the segment; rather, the base plus 65536 points to the high-order end of the segment.

On the 80286 in protected mode, a code segment is implicitly executable and also implicitly not writable. In addition, it may be designated as readable or not readable (for example, to prevent user programs from examining operating system code in a secure system).

A data segment can be designated as writable or not, but cannot be executed.

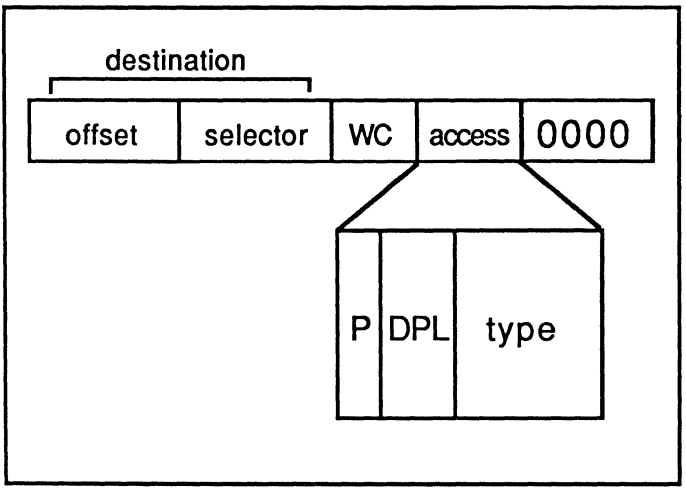
Gate Descriptors

The gate descriptor is a structure that uses indirection to allow programs to call routines whose addresses they cannot know until the program loads. Generally, any program address that lies outside the run file is unknown until load time. For example, a program can call a system common procedure or kernel call in BTOS, the location of which the Linker cannot supply when resolving the reference.

In the protected mode run file, the Linker resolves this reference as a call via a gate. At load time, PMOSS fills in the gate with the SN:RA virtual address of the desired routine. The actual CALL instructions in the calling code segment do not need to be modified at load time.

In the program code, these calls appear to call an SN:RA address, using an ordinary far CALL instruction. When this call executes, the selector obtains the associated descriptor (as before). The 80286 or 80386 examines the descriptor, which turns out to be a gate descriptor (refer to figure 1-5), and uses the destination fields in place of the original SN:RA to reach to the appropriate routine. The SN from the gate, not the original SN, winds up in the CS register. The original RA is ignored, because the gate destination fields provided a new SN:RA address.

Figure 1-5 **Gate Descriptor**



The gate descriptor is also used as part of the protection mechanism. Among other uses, it is a more powerful alternative to the supervisor CALL instruction found on some other processor architectures that feature memory protection. Calls from one run file to another (including operating system calls) use call gates so that entry to the destination program occurs only at certain well-defined entry points that the call receiver establishes. This is why the system ignores the RA supplied by the caller.

The gate descriptor fields are as follows:

Field	Meaning
P	Present bit.
DPL	Descriptor privilege level. This field determines only who can use the gate, not the privilege level at which the called code will run (the DPL of the code segment descriptor to which the gate points determines the privilege level).
Type	Gates that programs use can be call gates or TSS gates. Only PMOSS uses TSS gates. TSS gates also are involved in interrupt processing, as are two additional types of gates: interrupt gates and trap gates.
WC	Word count. Number of words of arguments for procedure being called. This field is only used when the call is to software at a higher level of privilege (such as from a user program to PMOSS).
Selector	The SN of the procedure entry point. Must identify a code segment descriptor with an appropriate DPL. May not identify another call gate descriptor; only one level of call gate indirection is allowed.
Offset	The RA of the procedure entry point.

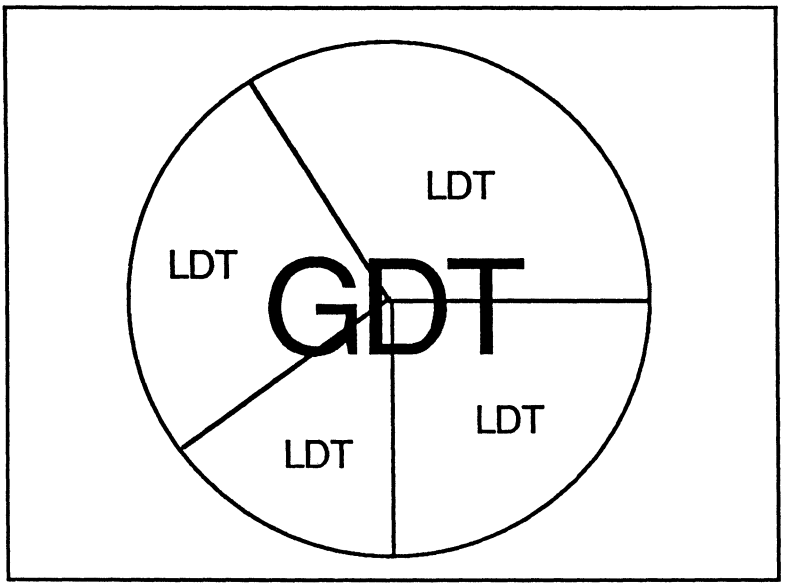
Protection

Protected mode on Intel microprocessors uses two models of protection:

- Protection by separate address spaces, shown in figures 1-6 and 1-7, in which a program is restricted to one virtual address space and cannot another address space location. This protection model provides isolation of one run file from another.
- Protection by privilege level, in which every program executes at one of several levels of authority. This is a ring protection model, shown in figures 1-8 and 1–9. It allows an operating system (like PMOSS) to conveniently protect itself from its clients. A program with a numerically lower privilege level has greater privilege, and can therefore access more restricted locations than a program with a numerically higher privilege.

Both protection models are in effect when in protected mode and both must be satisfied to allow access.

Figure 1-6 **Separate Address Spaces Protection Model**



On 80286 microprocessors, neither model or protection applies to real mode programs, even when PMOSS is installed. However, the physical memory above 992 Kb is inaccessible to real mode programs.

Each pie slice in figures 1-6 and 1-7 represents a separate address space, described by a separate LDT (the first protection model). Figure 1-6 shows the domain of each descriptor table.

A program in one address space is aware of and can describe only those locations for which it can load (and use) selectors. Only the LDT associated with the current process is in effect at one time. The GDT is in effect all the time. Therefore, the process can only address objects for which there exist descriptors in its own LDT or the GDT.

The LDT describes only those objects that it is legitimate for the program to access: the program's code and data, the gates it can use to call other programs, and data that has been received in requests from other programs (if the program is a server).

In PMOSS, the user program generally cannot use the GDT descriptors. PMOSS uses and thinks of the GDT as its own LDT. The existence of this special LDT allows user programs to call PMOSS via call gates, and for PMOSS to access the user program's memory directly (using the user's LDT at the same time as its own LDT, the GDT). Figure 1-7 shows how PMOSS uses the GDT.

Privilege levels regulate calls from user programs to PMOSS (the second protected model). The rings in figure 1-8 correspond to privilege levels. 0 is the level of highest authority or privilege, and 3 is the least privileged level. Each entity in the current address space (everything described by a descriptor in the current LDT or the GDT) is marked with a Descriptor Privilege Level (DPL), which determines its usability.

Figure 1-7 **PMOSS' Simplified Use of Separate Address Space Protection Model**

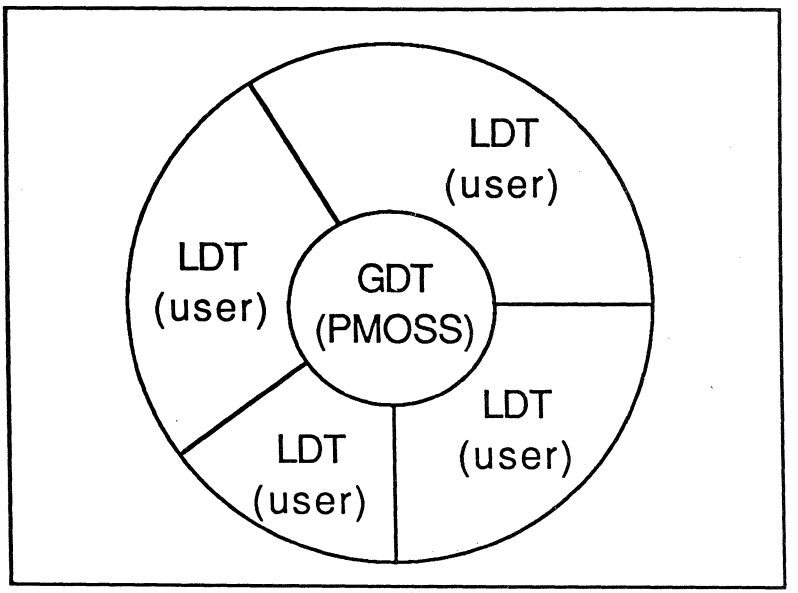
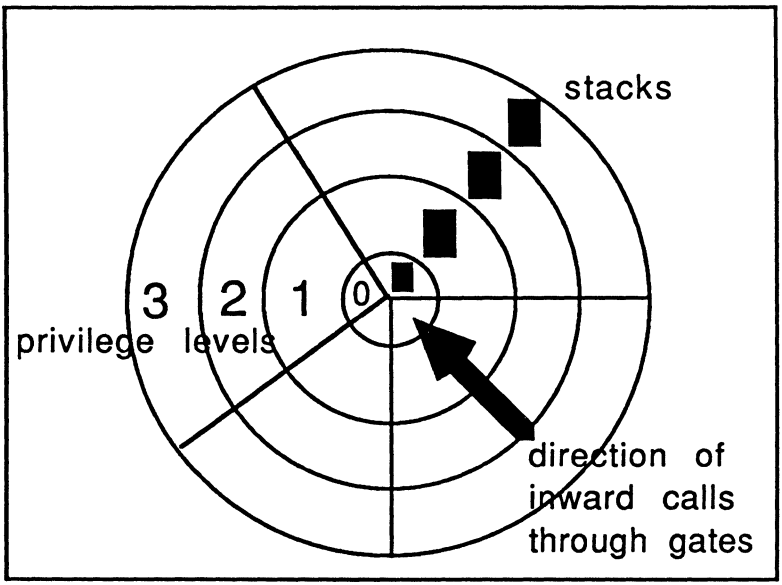


Figure 1-8 **Ring Protection Model**



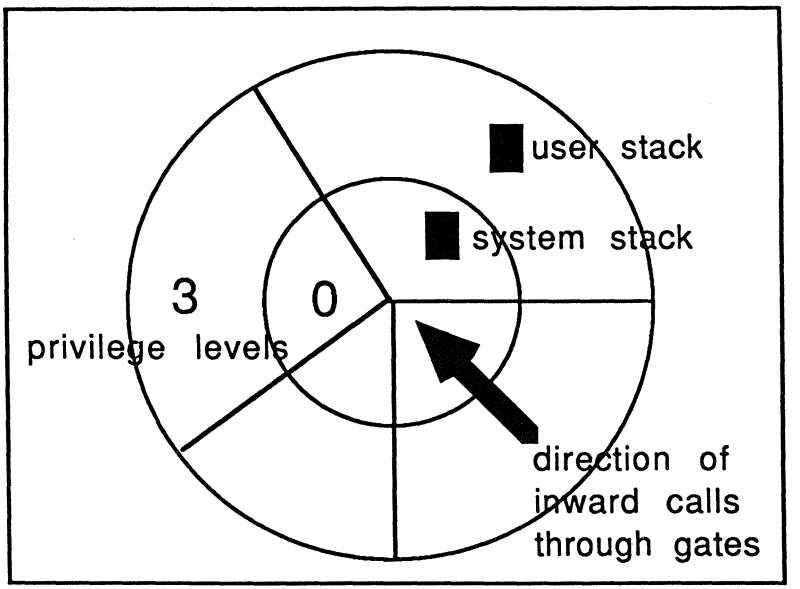
For example, most GDT descriptors are marked 0 in PMOSS, which means that only a program running at privilege level 0 can use them. Most LDT descriptors are marked 3, indicating that any program in the address space can use them.

Note: A level 0 program can use not only level 0 descriptors, but also descriptors with numerically greater DPL. This means that, in figure 1-7, a program in the inner circle can see objects in the outer circles, but a program on the periphery may not see into the protected center.

Each process has a separate stack segment for each privilege level. By using call gate selectors, a program can call inwardly to a more privileged level, but only in ways that allowed the system allows. Control switches to the stack for the inner level when this occurs.

PMOSS uses only levels 0 and 3, as shown in figure 1-9. Levels 1 and 2 are reserved for future expansion. Currently, therefore, only two stacks per process are required, not four. Most PMOSS procedures run at level 0. Level 0 includes only code and data structures that are part of PMOSS (including structures that PMOSS creates to manage user programs, such as LDTs).

Figure 1-9 **PMOSS' Simplified Use of Ring Model**



User programs run only at privilege level 3 in PMOSS. Most descriptors in LDTs are marked level 3.

In effect, the way PMOSS currently employs the privilege level mechanism by dividing the world into supervisor (PMOSS) and user castes. PMOSS can use both its own level 0 descriptors and the current user's level 3 descriptors.

The following text summarize the most important protection rules. The protection hardware has many additional features described in the Intel *iAPX 286 Programmer's Reference Manual* and *80386 Programmer's Reference Manual.*

DPL versus RPL

Every descriptor contains a DPL in the range 0 through 3 and every selector contains an RPL in the same range, in the low-order two bits.

The RPL bits of the DS and ES registers are not important in user programs and usually remain at zero, as the Linker set them, even in level 3 programs. The DPL, not the RPL, is the final arbiter of privilege. RPL bits have a special function only PMOSS uses, which is beyond the scope of this discussion.

Current Privilege Level (CPL)

The current privilege level (CPL) is the level at which the process is currently running (essentially, the privilege level of the currently executing code). CPL is stored in the RPL bits of the CS register; it can be displayed by using the Debugger to examine CS. This is true because a process' current privilege level normally is the DPL of the code segment it is executing and, in the case of CS, the RPL bits are copies of the DPL bits in the descriptor for the currently executing code segment.

Note: Except in the special case of conforming code segments (not present in user programs).

The SS register has RPL bits that match the CS register's RPL bits (usually 3 in a BTOS user program). Since only PMOSS usually loads SS, user programs are ordinarily not concerned with the SS RPL bits; however, in medium model programs, they are not the same as the RPL bits of DS, which are normally 0. The DS and SS selector values will be identical, except for their RPL bits in a medium model program; in real mode medium model programs, they are identical in all 16 bits, which is why the new procedure FComparePointer is required.

Switching Privilege Levels with Call Gates

Call gates support only call statements, not jumps or data accesses. As with any descriptor, the call gate must have a DPL at least as great numerically as the caller's CPL. However, the destination code segment (the descriptor that the call gate descriptor refers to) must have a DPL numerically equal to or less than the caller's CPL because only sideways and inward calls are allowed; outward calls are never allowed. The inner (numerically lower) levels are trustworthy and are therefore compromised if they surrender control to an outer level by calling it.

On an inward call, the hardware switches stacks (saving SS and SP and reloading them to point to the high-order end of a stack reserved for the level being called). It also copies parameters automatically from the caller's stack to the new stack, making them accessible to the called procedure in the normal way. The stack switch prevents a situation in which a call to a routine that is known to function correctly fails because the caller's stack was too short. It also leads to the need for multiple stacks for each process. In PMOSS, there are two stacks per process because only two privilege levels - 0 and 3 - are used. Stacks are not switched on sideways calls, even when using a call gate.

A return can be to the same privilege level (sideways) or to one of greater numeric privilege (outwards). The return examines the RPL of the saved CS in the return address to determine whether a stack switch is necessary.

IOPL

The IOPL flag specifies the maximum CPL (numerically) for which I/O and HALT instructions are valid. IOPL is a property of each process (part of the flags word). Under PMOSS, IOPL is always 3 so any process can perform these instructions.

General Protection Faults

Breaking any of the protection rules generates a general protection fault, exception 13. Under PMOSS, general protection faults activate the Debugger.

Processes and Process Switching

Associated with every process is a Task State Segment (TSS). The TSS, shown in figure 1-10, contains the complete register state of the process.

The advantage of the TSS is that it permits extremely rapid process switching, despite the protection boundaries between programs. In protected mode, process switching is implemented in hardware and firmware, in response to interrupts or single instructions.

To use a BTOS example, suppose one program does a SEND to an exchange where a program of higher priority is waiting. The kernel can start the waiting process with a single CALL instruction to SN:RA, where SN is the selector of its TSS (the RA is ignored).

This causes the microprocessor to perform an entire process switch:

- store the entire register state of the current process in the current TS (which is identified by a special hardware register - TR)
- switch TR to point to the new TSS, and set the new TSS' back-link field to point to the old TSS

Figure 1-10. Task State Segment (TSS)

LDTR (LDT selector)
DS
SS
CS
ES
DI
SI
BP
SP
BX
DX
CX
AX
FL (flags)
IP
Initial SS level 2
Initial SP level 2
Initial SS level 1
Initial SP level 1
Initial SS level 0
Initial SP level 0
back-link to prev TSS

- load the entire register state of the new process from the new TSS
- set the NT (nested TSS) flag bit

Execution then continues. The operating system does not have to save and load each register.

Using a structure called a TSS gate, you can arrange for an interrupt to perform a process switch automatically (in effect, an automatic CALL to a TSS).

Conversely, when a BTOS process executes a Wait, the kernel can put away the register state of the process in its TSS and resume execution of the next highest priority (NHP) process using a single instruction that reverses the TSS process switch.

Note: The IRET that reverses the TSS process switch has different semantics than the real mode IRET instruction. A special flag bit, NT, selects the appropriate IRET - either IRET (the familiar instruction) or TRET (for *T*SS *RET*urn) - at any time in protected mode. When the NT bit is 1, IRET does a TRET. User programs do not use either IRET or TRET. Therefore, IRET in protected mode is really two unrelated instructions.

The back link field of the TSS permits nesting of saved states that interrupts or TSS CALL instructions cause. When one interrupt cycle completes, or when a process wants to surrender the processor, execution reverts to the previous TSS via a process switch back to the TSS identified by the back link.

JMP instructions can also be used to cause TSS switches, but PMOSS does not use them.

TSSs have other functions connected with the stack switching operations that call gates perform. This guide is not intended to explain TSSs. For information about TSSs, refer to the Intel *iAPX 286 Programmer's Reference Manual* and the *80386 Programmer's Reference Manual.*

Interrupts

There are two kinds of interrupts: internal and external. Internal interrupts, initiated synchronously as a result of the execution of an instruction, include software interrupts (which occur when an INT instruction executes), exceptions such as interrupt type 4 (division by zero), and faults including protection faults (type 13). External interrupts, initiated by an asynchronous event outside the processor, include I/O interrupts such as disk and real-time clock interrupts, among others.

Every interrupt has an interrupt type number in the range 0 to 255, which is an index into a special descriptor table, the Interrupt Descriptor Table (IDT). There is one IDT per system. Unlike the analogous structure in real mode (the Interrupt Vector Table, or IVT), it need not be located at physical address 0, but can be anywhere in memory (it is based by a special register, the IDTR). The IDT contains only gate descriptors. For a given interrupt type number, the IDT can contain one of three types of gates:

- a TSS gate
- an interrupt gate
- a trap gate

The type of gate and its contents determine how the interrupt is handled.

When an interrupt is routed through a TSS gate, the hardware switches to the new process indicated by a TSS selector contained in the TSS gate. The process switch is identical to the one that occurs with a TSS CALL instruction (refer to Processes and Process Switching, in this section). All the register state of the interrupted process is saved automatically and the NT bit is set. After the Interrupt Service Routine (ISR) executes, a special instruction switches back and resume the interrupted process.

Note: This instruction is the conceptual TRET type of the IRET instruction (IRET when NT = 1).

If the entry in the IDT for a given interrupt type is in interrupt or trap gate, the effect as if an automatic CALL via a call gate executed. Fewer registers are saved automatically; this type of gate may therefore be faster for very simple ISRs. Like real mode interrupts, interrupt and trap gates push the flag word on the stack then clear the NT bit.

Note: Clearing the NT bit records that the last interrupt was the interrupt gate or trap gate type, rather than the TSS gate type. After the ISR executes, the same special instruction (IRET) resumes the interrupted procedure. However, since NT = 0, IRET reverses the effect of the call-gate-like interrupt or trap gate, rather than performing a full process switch. Because IRET has two entirely different semantics, depending on the current setting of the NT bit, the same ISR can be installed using either type of IDT mechanism without changing its code. Alternatively, it can be installed in a real mode system.

Guide to Compatible Programming

Programs compatible with protected mode can execute on any Intel microprocessor in real address mode or on the 80286 and subsequent Intel microprocessors in protected mode. This section describes how to create compatible programs or adapt existing programs to be compatible.

This section also describes general rules for protected mode programming.

80286 Real Mode Issues

The following incompatibilities arise between the B 28 and B 38 (80286) workstations and the B 26 and B 27 (80186) workstations, even when the former operate exclusively in real mode:

- differences in I/0 port addresses (which should be corrected by using GetPStructure)
- timing races with certain peripheral chips due to the 80286 faster execution speed and instruction pipelining (which should be corrected by adding delaying instructions)

Guidelines for Addressing Schemes

These guidelines arise from the differences between segment addresses SAs in real mode (paragraph numbers, called SRs) and protected mode (selectors, called SNs).

You can use real mode SRs as short (16-bit) versions of long pointers, if the objects they address are aligned on 16-byte boundaries. In a certain sense, SRs form a 16-byte-granular linear address space (of up to 1 Mb). In real mode, you may want to subtract two SRs, or add a value to an SR to get another SR.

Protected mode selectors do not form a linear address space because selectors are merely indexes to objects, not object addresses. In protected mode, you never want to add or subtract SNs because there is no relationship between the values of SN and the location of segments in the linear address space.

In real mode, when segments are contiguous in memory, you can detect that contiguity by comparing pointers. For example, the segment at 5000:0 of length 20h is contiguous with the segment at 5002:0 because
50000h + 20h + 50020h + Oh.

In protected mode, there is no relationship between the ascending numerical value of selectors and the ordering of segments in memory. A user program cannot know which segments are adjacent. In fact, the operating system reserves the right to move them from time to time without informing the user program.

Observe the following for addressing:

- Do not use contiguous data objects larger than 64 Kb.

  For example, in real mode, the compiled BASIC runtime takes over all remaining memory in the partition for heap space (potentially more than 64 Kb). Heap nodes are paragraph-aligned and are addressed by use of SRs. Therefore, a future compiled BASIC requires some reimplementation of its heap logic to make it compatible with protected mode.

  In protected mode, multisegment objects are not necessarily contiguous in memory. Also, selectors cannot be generated arithmetically.

  Compatible programs must use a multisegment rather than a contiguous data object and the segments must be addressed internally using offsets, not paragraph numbers.

- Use compatible memory management interfaces.

  New memory management requests support programs that are targeted for execution in both real and protected mode:

  AllocAreaSL

  ExpandAreaLL

  ExpandAreaSL

  QueryBigMemAvail

  ShrinkAreaLL

  ShrinkAreaSL

Refer to appendix A for information on these new memory management requests, as well as revised descriptions of the following previously existing memory management requests:

AllocAllMemorySL

AllocMemoryLL

AllocMemorySL

DeallocMemoryLL

DeallocMemorySL

QueryMemAvail

Programs should not generally depend on the descending or ascending nature of segment addresses that memory management operations return.

Avoid using AllocMemory calls for very small, numerous, fixed-length segments. This is a common, harmless practice in real mode because there is no system memory overhead for each segment. However, in protected mode, a 16-byte segment, for example, consumes more system memory for its overhead than for the segment itself.

- Do not use SAs as paragraph-aligned (short) pointers.

  Some programs convert addresses of objects that are known to be paragraph-aligned into short pointers. A short pointer is an SR made by combining the SR and RA of a long pointer arithmetically. The implied RA of a short pointer is 0. This technique does not work in protected mode.

  Compatible programs that require zero offsets for dynamically allocated memory should continue to use AllocMemorySL; however, they cannot depend on the contiguity of segments.

  Compatible programs that do not care about zero offsets but do care about contiguity between separately allocated chunks of memory should use AllocAreaSL and ExpandAreaSL. This provides up to 64 Kb of contiguous memory.

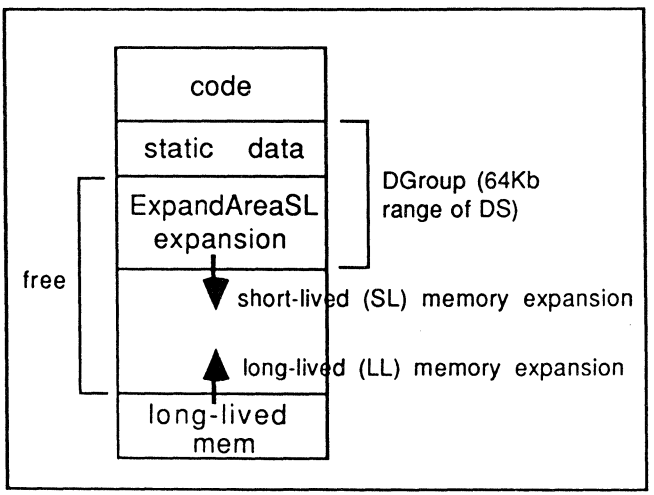
- Use ExpandAreaSL to allocate DS-relative memory.

  The Linker's DS allocation feature lets you dynamically increase the size of the statically allocated DGroup segment (in which all statically allocated data reside in a medium model program). It does so by locating the statically created data at the high-order end of the 64 Kb range of the DS register, so that the segment may be expanded downwards dynamically (assuming that the program is linked so that DGroup is at the low-order end of the program image). Figure 2-1 shows this arrangement. It permits memory dynamically allocated by the program to be addressable by DS-relative offsets. (Refer to the BTOS *Linker/Librarian Programming Reference Manual* for a further explanation of DS allocation.)

  When using DS allocation in protected mode, the Linker and PMOSS cooperate to allocate an expand-down segment for DGroup. Compatible PL/M-86 programs must declare doubleword types that are not virtual addresses as DWORD and incorrect hard-coded occurrences pointer. Pascal programs must use the integer4 type. C programs must use long.

  Compatible assembly code must not load ES (or other segment registers) with data. Use DX:AX to return a DWORD function result, and ES:BX only for a POINTER result.

Figure 2-1 **BTOS Partition Using DS Allocation**



- Do not base program variables on system common pointers located at absolute addresses — use GetPStructure instead.

  Under PMOSS, many system data structures are the same as for the underlying version of BTOS, but they must be accessed through calls to GetPStructure rather than through pointers initialized to low-memory paragraph-number addresses.

  The obsolete way of accessing a system common data structure is to declare a pointer at an absolute address in the scat and base a variable (usually another pointer) on it. You can still use these absolute addresses as arguments to GetPStructure, but do not use them to initialize pointers directly. For more information, refer to Access to System Data Structures, in appendix B.

  Pointers contained within system structures are usually real mode pointers, even when the structure is correctly accessed using the GetPStructure. Normally, therefore, protected mode programs cannot retrieve and use pointers from system data structures.

- You must use a special operation when performing pointer comparison operations.

  A pointer is a PL/M-86 type that is equivalent to Pascal's ADS type (refer to the *BTOS Pascal Compiler Programming Reference Manual*). With BTOS 8.0 and PMOSS, call FComparePointer to compare pointers in protected mode. For example, instead of using **If p1=p2**, use **If FComparePointer (p1, p2)**.

  Note: In protected mode, FComparePointer is the only valid way to compare pointers.

  For most applications use FComparePointer with the bCompareMode parameter set to mode 0 or 1 (refer to FComparePointer, in appendix A).

- Code segments must have classname code.

  Protected mode code segments must have the classname code. Different types of descriptors are built for code and data segments; data segments cannot be executed and code segments cannot be written (usually, both can be read).

Assembly language routines that omit the code classname from the segment directive work in real mode, but fault in protected mode.

- Writable code segments are not permitted.

  Compatible programs cannot have writable variables in the code segment or self-modifying code.

  Compatible programs can, however, use SetSegmentAccess to change segment type. For example, COBOL must do so to load an intermediate file (as data) and execute it (as code).

- Use SetPStructure when modifying system structures.

  In real mode, as some programs do, you can directly modify operating system structures (especially per partition data structures that the operating system maintains).

  In protected mode, access to operating system data structures is only possible using GetPStructure, which returns a pointer based on a read-only descriptor for most structures. Therefore, GetPStructure alone is not sufficient when you need to modify this kind of structure.

  PMOSS supports the SetPStructure call for user programs that have a legitimate reason for modifying specific fields in certain data structures. SetPStructure is not present in BTOS real mode BTOS and therefore must be used only in protected mode. (Refer to SetPStructure and FProtectedMode, in appendix A.)

- Avoid timing loops.

  Protected mode compatible programs cannot contain timing loops that depend on instruction execution speed. Certain instructions execute slowly in protected mode (refer to the *iAPX 286 Programmer's Reference Manual* and the *80386 Programmer's Reference Manual*).

- Use only compatible instructions.

  Programs that are to remain B 21 and B 22 compatible as well as B 26, B 28, and B 38 compatible can use only the 8086 instruction set. Programs that are to be compatible only among the various B 26, B 28, and B 38 processors can use the full 80186 instruction set, but none of the 80286 extensions.

  Unisys-supplied compilers normally use only the 8086 instruction set. SRT PL/M-86 use the 80186 instructions (but not the 80286) if the $MOD186 directive is used. These programs can be recognized with the Debugger because each procedure includes ENTER and LEAVE instructions near its entry and exit points, respectively.

  You cannot use the PUSH SP instruction because it differs between the 80186 and the 80286.

- You can still encode system calls using CS:IP hack.

  Compatible programs can continue to encode trap instructions using alternate CS:IP addends to arrive at the same 20-bit address.

  This is how to call BTOS for procedural requests, kernel calls, and system common procedure calls. Use exactly the same CS:IP values in programs compatible with protected mode as have always been used in real mode programs.

  The Linker, BTOS, and PMOSS support these encoded instructions in both real and protected mode programs for compatibility, but are supported only for compatibility purposes in protected mode. Since these addresses fault in protected mode, they are altered before executing. You cannot use CS:IP hack in protected mode.

  The virtual code segment facility (the overlay manager) cannot use the CS:IP hack in protected mode.

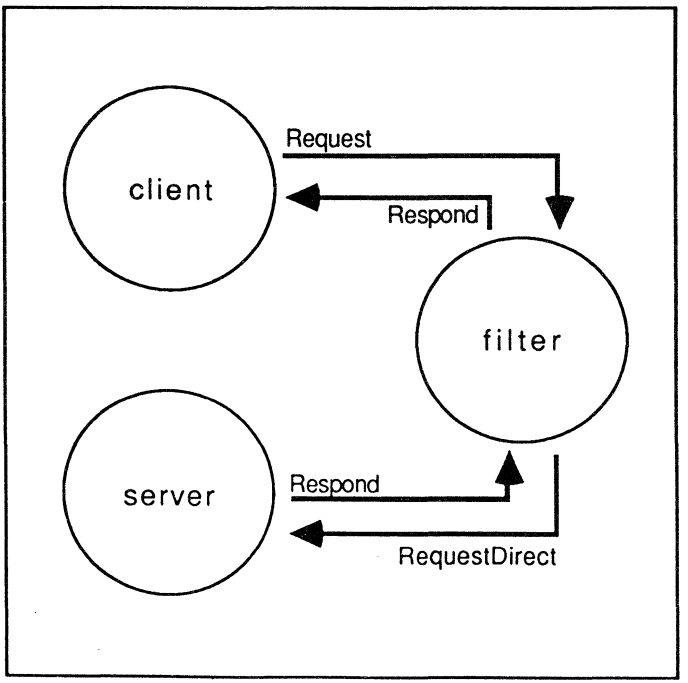
- Filter processes may require a change.

  Servers that filter (intercept) requests destined for other servers are called filter processes. A common use of filter processes is to enhance the semantics of an existing request without changing an existing server. Since BTOS is a set of servers, you can use the filter process technique to enhance BTOS by allowing BTOS to load run files in protected mode.

  Filter processes commonly use one of two techniques for passing through requests to the original server. One of these techniques does not work in protected mode in its original form and therefore requires the new kernel primitive, ForwardRequest, to make filter processes using those techniques compatible with protected mode.

  Servers that pass through requests by issuing a second request primitive (or RequestDirect) continue to work in protected mode, without change. This type of two-way filter process, shown in figure 2-2, intercepts the request on its way to the original server and also on its return (the response).

Figure 2-2 **Two-way Filter Process (no change required for protected mode)**



The following is the sequence of typical events of a two-way filter process:

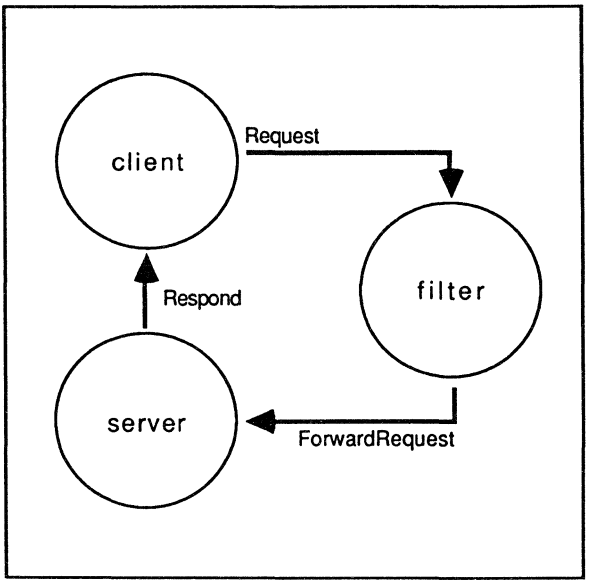
- Client process issues Request.
- Filter processes proceeds from its Wait.
- Filter process changes exchResp field to its own exchange and issues RequestDirect to original server's exchange, then Wait.
- Original server proceeds from its Wait.
- Original server issues Respond.
- Filter processes proceeds from its Wait.
- Filter process changes exchResp field back to client exchange and issues Respond.
- Client process proceeds from its Wait.

Servers that intercept the request only on its way to the original server, but not on its return trip to the client, traditionally use the Send primitive (or occasionally, the RequestDirect primitive) to forward the request to the original server. This type of one-way filter process, shown in figure 2-2, does not work in protected mode unless modified.

The following is the traditional sequence of events:

- Client process issues Request.
- Filter processes proceeds from its Wait.
- Filter process issues Send (or RequestDirect) to original server's exchange.
- Original server proceeds from its Wait.
- Original server issues Respond.
- Client process proceeds from its Wait.

Figure 2-3 **One-way Filter Process (change as shown for protected mode)**



To make the traditional sequence compatible with protected mode, use the new kernel primitive ForwardRequest in place of Send or RequestDirect. This new primitive (described in appendix A) is similar to Send in that no response is expected; by issuing it, the filter process discharges its responsibility for the request as if it performed a Respond. Like RequestDirect, its argument must always be a pointer to a request block.

- Using the Send and PSend kernel primitives are subject to restrictions.

  Use the pMsg operand of Send and PSend only to send an actual pointer. In particular, the SN of the pMsg must be a valid segment selector or the null selector (zero). There are no restrictions on the RA value when the SN is zero, so you can use Send and PSend rather than a pointer to send two bytes of arbitrary data.

  Otherwise, you can use Send and PSend exactly as they are in real mode, provided that the sender and receiver are part of the same task. That is, two processes sharing the same LDT can communicate using Send, or an interrupt service routine can use PSend to communicate with a process using the same LDT.

You can only use Send between tasks (when one or both is a protected mode task) if the receiving task already has addressability to the segment that pMsg references. That is, the receiving task must either be a real mode task or already possess a descriptor for the segment that the SN of the pMsg identifies. For example, the following sequence is legal in protected mode:

- Process A issues Request.
- Process B receives the request. The system builds alias descriptors in the LDT of process B for the request block and a pb/cb, then process B proceeds from its Wait.
- Process B issues one or more Send process A, where the pMsg operand is a pointer to the request block, a pb from the request block, or a pointer to something wholly contained within one of the pb/cb fields (in other words, a pointer to something for which process A is known to already have a descriptor).
- Process A receives each pMsg which, upon receipt, is translated to a pointer containing the selector of the descriptor that process A already has in its LDT.
- Process B issues Respond and the alias descriptors are removed from its LDT.

After this, process B cannot issue a Send command using pointers related to that request block.

The restriction occurs because alias descriptors are constructed only upon receiving block sent with Request, RequestDirect, or ForwardRequest. When a pMsg from Send is received, an appropriate descriptor must already exist. (The system finds the descriptor, if it exists.)

This restriction is necessary because there is no paired equivalent to Send (equivalent to Respond) that enables the system to know when to deallocate the alias descriptor, if one was created.

Only interrupt service routines, use PSend when communicating with a process. Both the interrupt service routine and the process are usually in the same task. Therefore, this restriction for using intertask pMsg does not occur with PSend.

PMOSS imposes additional restrictions on the use of Send and PSend.

- Load the SS and CS registers properly to avoid a fault.

  Assembly language code that loads SS, or any code that causes CS to be loaded with a value other than a previous value of CS (for example, by manually storing a segment address on the stack and then returning to it), must adhere to special rules of RPL usage.

  The Request Privilege Level (RPL) bits are bits 0 and 1 of the selector. The Linker emits selectors that have zero in these bits, even though most programs run at privilege level 3.

  In fact, the RPL bits use only the operating system. The DPL bits in the descriptor, not the RPL bits in the selector, protect against unauthorized use of descriptors. The RPL bits have no effect when they are zero, or when the program using the descriptor is running at level 3. When nonzero, they weaken the selector. For example, a program running at level 0 faults if the RPL of the selector is numerically greater than the DPL of the descriptor. The RPL bits are provided so that operating system procedures occasionally can tag selectors they receive (as arguments from the user program) with the user's privilege level. This ensures that the operating system procedure will not successfully access data to which the user is not entitled. This occurs even when the operating system procedure is running at a higher privilege level, which would legalize these accesses if the RPL bits were zero.

  When selectors are loaded into DS or ES, the RPL bits remain as the Linker set them (zero). The RPL bits of these segment registers do not usually represent the DPL of the segment to which they refer. (To determine the DPL with the Debugger, use the **CODE-V** command.)

However, when CS is loaded during a far CALL or JMP instruction, the DPL of the destination code segment is placed in the RPL bits of the CS register. Although the Linker puts 0 in the RPL bits of all segment addresses (including those in CALL instructions), after a CALL to a user procedure executes, the CS register's RPL bits are 3, not 0.

This occurs so the RET instruction can determine if the return is to be to the same privilege level or not. This requirement exists because RET, like CALL, can be interlevel. Interlevel CALLs are only possible using call gates.

At any time, the RPL bits of the CS register (the current CS) are by definition the current privilege level (CPL), the privilege level at which the processor is executing now.

The SS register's RPL bits are required to match the CS register's RPL bits exactly. Therefore, code that loads SS, like code that manufactures saved CS values, carefully must set these RPL bits to the DPL of the stack segment before loading SS, to avoid a fault. Moreover, the DPL of the stack segment must match CPL exactly.

- NIL pointer problems can occur as a result of certain coding sequences.

  In protected mode, it is valid to put 0 in a segment register, but not valid to use it to address data. A segment register with 0 in it is out of action temporarily.

  Trying to reference data at paragraph number 0 in real mode usually causes an error in a user program.

  However, the following construct may be valid in real mode, but fault in protected mode due to the kind of code that the compiler generates:

```
DECLARE
      pFoo    POINYRT
    , foo     BASED pFoo WORD
    ;

IF (pFoo <> 0) AND (foo = 1), THEN
      DO;
        ...
      END;
```

In PL/M-86, the above construct allows the compiler to test pFoo and foo in either order. However, the access to foo must never occur if pFoo is nil; otherwise a general protection fault occurs. In this case, the code should be rewritten as:

```
IF pFoo <> 0 THEN
     IF foo = 1 THEN
       DO;
          ...
       END;
```

- Pascal programs may require a workaround for a fault that results from an incompatible code generation practice.

  Unisys' Pascal compiler can generate code that violates the limit restriction in protected mode when accessing a segment.

  This occurs because Pascal sometimes does full-word operand fetches from memory, even when the object being accessed is a byte; it then discards the high-order byte. When the object being accessed is the last byte in a segment, a limit fault occurs. This form of code generation is improper in protected mode.

  The problem can occur in servers that are written in Pascal and try to read a byte located at the very end of a pb/cb field passed to them by a client. These fields are always tightly encapsulated segments to the server, even through they may not be located at the end of the client's data segment because PMOSS gives the server its own descriptor for each pb/cb field when the request arrives.

  One workaround is to expand the segment or field by padding it with an additional byte. With the pb/cb, this means increasing the cb by 1. Another workaround is to access the last word of the structure, rather than the last byte, and then manually shift to obtain the last byte.

  Note: For more information on this restriction refer to the *BTOS Pascal Compiler Programming Reference Manual.*

Linking

Version 6 Run File Format

Programs compatible with protected mode must use version 6 run file format, which is designed to be loaded in either real mode or protected mode. It supports protected mode while preserving single run file compatibility with real mode.

PMOSS never tries to load version 4 run files in protected mode.

Note: In the **SystemBuild** field of the BIND command, you must specify PM for protected mode or PMOSS for PMOSS (refer to the *BTOS Linker/Librarian Programming Reference Manual*).

The BIND command (an alternate command case invocation of the Linker version 8.0 and later) produces version 0 run file format. The LINK command produces version 4 run files. V6Link is a synonym for BIND.

Version 6 run file format is acceptable to:

- BTOS workstations 5.0 and later
- XE520s that support MS8, which allows the XE520 to accept version 6 run file format

Earlier BTOS versions accept only version 4 run file format, which is not compatible with protected mode.

Marking the Run File

PMOSS will not load a V6 run file in protected mode unless it has been marked executable-in-protected-mode by setting the high-order bit of the *verAlt* field at offset 20th in the runfile header.

Future versions of the Linker (the BIND command) support an option to accomplish this marking.

Currently, however, you must use the following Debug File command sequence to mark the run file after linking:

1 With the system at the Executive level, type **Debug File.**

2 Press **RETURN.**

The Debug File command form appears with the cursor in the **File name** field.

3 Type **MyFile.run.**

4 Press **RETURN.**

The cursor moves to the **[Write?]** field.

5 Type **yes.**

6 Press **RETURN.**

The cursor moves to the **[Image mode?]** field.

7 Type **yes.**

8 Press **GO.**

The prompt **Debugger 8.0 (File Mode)** appears and the cursor appears next to the % on the line below the prompt.

9 Type **20.**

10 Press **Right Arrow.**

An arrow (→) appears, followed by a four-digit number, which is part of the run file header.

Note: The old value of the verAlt field is usually 0001, as shown. If another value is displayed, the run file is probably a V4 run file.

11 Type **8001.**

12 Press **RETURN.**

The cursor moves to the %.

13 Press **FINISH.**

The system returns to the Executive level.

14 Type **Debug File.**

15 Press **RETURN.**

The Debug File command form appears with the cursor in the **File name** field.

16 Type **MyFile.run.**

17 Press **RETURN.**

The cursor moves to the **[Write?]** field.

18 Type **yes**.

19 Press **GO**.

The prompt **Debugger 8.0 (File Mode)** appears and the cursor appears next to the % on the line below the prompt.

20 Press **FINISH**.

Note: Unless the run file is already marked, you must use the Debug File utility twice: the first time to modify one bit of the header and the second time without modifying anything. This is necessary because when you specify **[Image mode?]**, the system does not recalculate the run file checksum when you finish. Activating the Debug File utility a second time, with **[Write?]** specified but **[Image mode?]** left blank, stores the correct run file checksum.

Contiguous Code and Data

All code segments and all data segments, respectively, must be contiguous in the run file.

PMOSS requires all the code segments in a run file to be physically adjacent. Similarly, all the static data segments must be adjacent to each other. The code and data portions of the run file can occur in either order, however.

The Linker tries to order code and data segments in separate, contiguous regions in the run file. However, improperly classed segments can prevent the Linker from ordering segments properly.

Failing to keep all the code segments together can prevent PMOSS from reclaiming the command to compare the memory consumption of the program when it is running in real mode versus protected mode. The used number should differ by the total amount of code in the program. (These numbers reflect only consumption of real mode memory.)

If they do not (for example, if no code space was saved in protected mode), examine the Linker map for noncontiguous code or data. Pay particular attention to ??SEG segments that the Assembler generates. The ordering of the segment does not matter if it has a length of zero, but can cause a problem if the segment has anything in it. These segments are emitted automatically when code or data appears outside of any segment/ends pair in an assembly language program.

Refer to the *BTOS Linker/Librarian Programming Reference Manual* to resolve any problems with segment ordering.

Remedies for Incompatibilities

The following text describes remedies for a program that cannot conform to the guidelines described in this section.

Checking for Protected Mode at Run Time

Occasionally, there may be good reason for you to use two different algorithms in real and protected modes. You can package these programs as a single run file by including both algorithms with run-time checks in the code to choose between them.

Refer to FProtectedMode, in appendix A, which you can use to make the run-time check.

PMOSS Limitations

If making a program compatible with protected mode seriously impairs its usefulness as a real mode program, you can port the program to protected mode as a separate run file to preserve the appearance of a single program. This technique requires renaming of the real mode program and that the protected mode version replace it. The protected mode program calls FProtectedMode after executing and if it finds itself in real mode, chains to the real mode run file. This technique handles both the case of a processor that cannot support protected mode and a system on which PMOSS is not running.

You might find this technique useful, for example, as an interim measure enabling a program to continue using some facility that increases performance or decreases program size in real mode, but which protected mode does not support, such as the virtual code segment facility (overlays).

Naming Conventions

Unisys source code uses conventional prefixes and suffixes for variable and function names. These conventions make source code easily readable by acting as an extension to the type system of the language. Unisys documentation of procedural interfaces also uses these prefixes to name parameters.

The previously existing prefixes listed below are interpreted as indicated with respect to protected mode concepts.

Prefix	Meaning
sa	Segment address. Deliberately ambiguous; can be either an sn or an sr prefix (described below). Used when the code is to operate in either real or protected mode, or anywhere that the distinction between real and protected mode operation is not relevant to understanding the variable's role.
ra	Relative address. An offset from a segment address (sa). Exactly the same meaning as sa.
p	Pointer (logical address). An sa:ra pair. Deliberately ambiguous in the same sense as sa.

The following new prefixes deal with protected versus real mode distinctions.

Prefix	Meaning
sn	Selector. Refers specifically to the protected mode form of segment address. Used only when it is important to understand that the variable is not a paragraph number.
sr	Paragraph number. Refers specifically to the real mode form of segment address. Used only when it is important to understand that the variable is a paragraph number.
pn	An sn:ra pair (virtual address).
pr	An sr:ra pair (real address).
sg	A global (GDT) selector. More specific than an sn. Used when it is important to understand that the variable is a GDT selector, as opposed to just any selector.
sl	A local (LDT) selector. More specific than an sn. Used when it is important to understand that the variable is an LDT selector, as opposed to just any selector.

pg	An sg:ra pair (global virtual address).
pl	An sl:ra pair (local virtual address).
la	Linear address. On the 80286, a DWORD containing a 24-bit physical address and a high-order zero byte. A linear address is stored in the base field of a segment descriptor. You cannot use a linear address directly in an instruction, but the Debugger accepts linear addresses (any numeric address that does not contain a : is interpreted as an la).

New Machine Instructions

80286 Instructions

80286 instructions are used in PMOSS, but not in user programs that must remain compatible with real mode. To be compatible with protected mode, user programs do not require a new version of the Assembler that supports these instructions.

The operating system Debugger accompanying BTOS 8.0 supports 80286 instructions. The Debug File utility that is part of standard language development software 8.0 does not, however. The Debug File utility usually disassembles 80286 instructions as ILLEGAL.

There are 80286 instructions in CTOS.lib 8.0 that conditionally execute only when running on 80286-based CPU modules.

PMOSS is built from source partially by using an enhanced Assembler that supports the new 80286 instructions. This version of the Assembler is not a part of 8.0 standard language development software.

80186 Instructions

PMOSS also uses 80186 instructions (instructions found on the 80186 and 80286, but not the 8086). BTOS 8.0 and standard language development software 8.0. fully support these instructions.

Debugging Protected Mode Programs

Overview

Debugger versions 8.0 and later support debugging of protected mode programs. This section explains the new Debugger facilities and how to use them.

This section describes differences you will encounter when using the Debugger. For more information on the Debugger, refer to the *BTOS Debugger Programming Reference Manual.*

Most important Debugger 8.0 operations work with protected mode programs. The Debugger facilitates portation of existing servers to protected mode.

In addition to supporting familiar operations such as setting and clearing breakpoints and checking the contents of registers in protected mode, the Debugger adds protected mode Task State Segments (TSS) to the **CODE-S** display of processes and exchanges. It also supports a new command, **CODE-V**, which displays the contents of the descriptor corresponding to a selector (refer to Descriptors: CODE-V, in this section).

The PR Value and Its Meaning

PR is a Debugger variable that identifies the process currently being examined. Only the Debugger user needs to know about PR. Unlike other registers, PR can be changed without affecting the outcome of execution; however, PR can affect Debugger commands. The meaning of PR differs between real and protected modes.

In real mode, before PMOSS is installed, the command **CODE-S** displays a summary of information about all processes and exchanges in the system. The first column on the left in this display has the heading **id**. These entries are the process identification numbers for all processes of which the system (real mode BTOS) is aware. The **id** value is an index into a list of processes, so the values do not become very large; 00h - 2Fh are typical examples.

In real mode, any of these process numbers can be assigned to PR. PR thus identifies a process and:

- implies a load offset to which the Debugger is to relocate any subsequently loaded symbol file (when the **CODE-F** command is used)
- indicates which set of saved registers the Debugger should display or modify when you display or modify hardware registers
- indicates which process should be single-stepped when using the **CODE-X** command

When PMOSS is installed, PR can be assigned either a real mode process number or a TSS selector. A TSS selector corresponds to the descriptor that contains the address of the Task State Segment (TSS) of a protected mode process. The values of these TSS selectors are generally much larger than the values of the real mode process identification numbers; 5A0 or 608 are typical examples.

Note: Although user programs do not use TSS selectors, PMOSS uses some TSS selectors that are small; for example, the PMOSS process TSS is 18h.

When a breakpoint is taken, the system automatically sets PR to the process that encountered the breakpoint (except for a **CODE-I** breakpoint). The same is true of faults and exceptions.

In the Debugger, PR not only identifies a process but also determines whether the debugging is for a real mode program or a protected mode program.

The first sign of a change in PR is the Debugger prompt:

- An asterisk appears when debugging a real mode process (except when inside BTOS or at interrupt level).
- A solid square prompt appears when debugging a protected mode process (a TSS).

The second effect of this enhancement is the way **CODE-F** behaves:

- When a symbol file is loaded while PR is set to a real mode process, the segment addresses associated with the symbols are paragraph numbers, which are relocated to correspond to the load offset (base address) where the program's code was loaded.

  Note: When debugging a real mode program, some programs may need to specify an explicit numeric load offset as a parameter to the **CODE-F** command (though the Debugger can often manage without this parameter). When debugging a protected mode program, a load offset is never required, and should never be used, or the Debugger assumes it is dealing with a real mode symbol file (for example, that symbol addresses are paragraph numbers).

- When a symbol file is loaded while PR is set to a TSS, the segment addresses associated with the symbols become selectors. No relocation is necessary because the selectors are not relative to actual memory addresses, since they are really indexes into the program's LDT. Every program has its own LDT, which the Debugger finds by looking in the TSS (that is, by knowing PR).

The third effect is the way that SA:RA addresses of the are decoded:

- When PR points to a real mode process, the Debugger interprets the SA portion of the address as a paragraph number (an SR).
- When PR points to a TSS, the Debugger interprets the SA as a selector (an SN).

Note: Numeric addresses that do not contain a colon (:) are interpreted the same way, regardless of the PR setting. These addresses refer to linear addresses, such as absolute memory locations on the 80286.

For example, the address 0 always refers to the first byte in physical memory, regardless of PR. If PR points to a real mode process, the address 0:0 also refers to this same byte. However if PR points to a TSS, the address 0:0 is invalid and, if used, generates the message **Non-existent memory** if used because 0 is not a valid selector.

Note: Zero is never a valid selector; in protected mode, this distinguished, reserved selector value is called the null selector and never refers to an actual segment. This is a hardware-enforced, rather than a software, convention.

Looking at Processes: CODE-S

If PMOSS is installed and no other protected mode program is loaded, the command **ACTION-A** causes the system to enter the Debugger.

Sending the command **CODE-S** shows the display of real mode processes (PCBs) and exchanges, followed by a new display: the GDTR register, IDTR register, and the protected mode Task State Segments (TSS). At this point, all the TSS segments are internal PMOSS processes or exception handlers.

To choose a TSS selector from the display of protected mode processes, use the following procedure:

1 Set PR equal to the SgTss value.

2 Press **RETURN**.

A solid square Debugger prompt appears, showing that the Debugger mode is protected mode.

3 If you are debugging PMOSS, load the associated symbol file in the usual way, type **[sys]<sys>PMOSS.sym**.

4 Press **CODE-F**.

The GDTR register locates the base of the Global Descriptor Table (GDT). Although this information may be useful, you can use the **CODE-V** command to conveniently examine part or all of the GDT (refer to Descriptors: CODE-V, in this section).

Entering the Debugger

Use caution when you work with protected mode programs and enter the Debugger with the **ACTION-A** and **ACTION-B** commands. Although you can use these commands to activate the Debugger at any time, the value of PR may not be the expected one. Even worse, trying to set PR to the appropriate value may not provide access to the process' current registers and stack. Unfortunately, TSS stores the state of the protected mode process only after a:

- breakpoint
- CODE-**X**, **CODE-GO**, or INT 3 instruction
- fault

Therefore, when you use **ACTION-A** or **ACTION-B** to enter the Debugger, the values in the TSS you select (by setting PR) may be old.

You can enter the Debugger at the outset of the run by using **CODE-GO** to begin execution rather than **GO**.

After you enter the Debugger and receive the square prompt, load the symbol file and place breakpoints in your protected mode program in the usual way.

There is also a new way to enter the Debugger in protected mode: the fault. A fault is a condition that prevents the hardware from completing an instruction, such as a protection violation. In PMOSS, all faults cause the system to enter the Debugger and display an appropriate message.

Not all faults are errors. Faults also occur in virtual memory systems when the Debugger tries to access a segment that is not in memory; this fault is not an error, but a signal for the operating system to read the segment from disk and restart the program. PMOSS does not use faults this way. Programs that are not debugged will fault at or soon after the error. Therefore, one of the protected mode benefits is how it facilitates debugging.

Accessing 80286 Registers

You can enter registers common to the 80286, 80186 and 8086 using the commands in the *BTOS Debugger Programming Reference Manual.* In addition, the Debugger lets you access the registers peculiar to the 80286 (MSW, SS0, etc.), including all the TSS fields.

Note: The Debugger does not support the additional 80386 registers.

Mnemonics

To display or modify a register, use the two-letter mnemonic register name that the Debugger recognizes.

The following list shows the new register mnemonics:

Intel mnemonics	Debugger mnemonics
MSW	MS
TR	TR
LDTR	LD
SS0	S0
SP0	P0
SS1	S1
SP1	P1
SS2	S2
SP2	P2
Back-link field of TSS	LK

The GDTR and IDTR values have no Debugger mnemonics, but appear when you use the **CODE-S** command.

Warnings

The Debugger shows incorrect IDTR and TR values and prevents the TR from being modified.

When PR is set to a TSS, the Debugger fetches from the TSS all the registers except GDTR, IDTR, MSW, and TR. If you change a register, the Debugger stores the new value in the TSS. Thus, if PMOSS is not currently using the TSS for storing register values, the registers appear incorrectly and commands to modify them do not take effect. In PMOSS, this may happen when you use **ACTION-A** or **ACTION-B** to enter the Debugger. It never happens when you enter the Debugger by:

- taking a breakpoint
- an INT 3 instruction
- an exception or fault
- the **CODE-GO** command
- single-stepping (the **CODE-X** command)

Finding the BTOS Process

Each Task State Segment (TSS) is associated with a BTOS real mode Process Control Block (PCB). The TSS is used when the process is in protected mode and the PCB is used for real mode.

You may want to examine the current TSS and determine the PCB associated with it because when a process waits at an exchange, it waits in real mode; therefore, the process is a PCB that is waiting, exactly as in normal real mode debugging.

You can find the PCB for your protected mode process any time you are at a square prompt by using the following procedure:

1 Type **s0**.

2 Press **Right-Arrow**.

A four-digit value appears.

3 Type **nnnn:0ffca**, where **nnnn** is the four-digit value that appeared in step 1.

4 Press **Right Arrow.**

The value that appears should be the same as the TSS number (the current PR value). If it is not, or if the message **Non-existent memory** appears, you are looking at a non-user TSS (a special PMOSS internal TSS) and cannot use this procedure to find the PCB.

If it is the value of PR, press **Down-Arrow.**

The value that appears (at offset 0FFCCh) is the process' default response exchange number. The process may or may not be waiting at this exchange.

To find the PCB, press **CODE-S**; the value appears in the exch column of the PCB display.

Breakpoints

You can set breakpoints in the usual way. You can set up to 16 breakpoints, but only up to six breakpoints can be broken at once.

Note: Reaching a seventh breakpoint causes a system crash. However, multiple concurrent breakpoints can occur only when multiple processes with breakpoints set in them are running concurrently.

CODE-B Breakpoints

You use the **CODE-B** command to set breakpoints in processes (just as you do in real mode).

If you reach a **CODE-B** breakpoint and enter the Debugger, the message **Break at [address] in process [n]** appears. In a protected mode process, the prompt is square, indicating that the process number is a TSS selector.

To clear breakpoints, use the **CODE-C** command.

CODE-I Breakpoints

You set and clear breakpoints in interrupt service routines using the **CODE-I** and **CODE-C** commands, respectively.

However, the following special restrictions apply when debugging protected mode interrupt service routines with PMOSS:

- These breakpoints can be taken, but it is not possible to proceed (**CODE-P** or **GO**) or single-step (**CODE-X**) afterwards.
- Upon taking the breakpoint, the value of the Debugger's LD register (the LDTR) is erroneous. You must change this value manually to the correct one before code or data may be examined.

Note: Real mode interrupt service routines fully support **CODE-I** breakpoints.

You can display the correct value of LD any time the process that created the interrupt service routine is running — for example, by starting the program with the **CODE-GO** command. Remember this value when setting a **CODE-I** breakpoint and set LD to it immediately when the **CODE-I** breakpoint occurs.

Note: The PR value appearing at the breakpoint differs from that of the original process because all interrupt service routines use a special system TSS when running. Manually change LD only, not PR.

The IP value can also be incorrect (usually one greater than it should be, making it appear that execution was interrupted in mid-instruction). You will have better results trying to disassemble the code at the breakpoint works if you first back up IP to the actual IP value of the breakpoint.

After you examine code and data at the **CODE-I** breakpoint, reboot the workstation.

Descriptors: CODE-V

If you press **CODE-V** without supplying an argument, the Debugger displays the entire GDT, IDT, and current LDT. The IDT display, however, is meaningless in this version of the Debugger. You can access the IDT while debugging through GDT descriptor 10h. To display an IDT descriptor, display the six bytes of memory at **10:n**, where **n** is the interrupt vector number times eight.

If you supply a selector value and press **CODE-V**, the Debugger displays the descriptor that corresponds to the supplied selector. You can also supply any symbol or virtual address; the **CODE-V** command uses the SA portion of the SA:RA only.

An 80286 descriptor is an eight-byte structure, of which only six bytes are used. The format of the display is different for different types of descriptors.

Segment Descriptors

To display a segment descriptor, use the following procedure:

1 Type **InitRqTables**, where **InitRqTables** is a procedure name in this example (an internal PMOSS procedure).

2 Press **CODE-V**.

The following segment descriptor information appears:

```
iSn     sn      base    limit   ar      p
0060    0300    0FC1F0  02F2    9B      0
code,non-conforming,readable
```

The fields that appear are:

iSn	the most-significant 13 bits of the selector is the array index into the descriptor table (considered an array of eight-byte structures)
sn	the selector value you entered with the low-order 2 bits (the RPL) zeroed; that is, the offset of the descriptor in the descriptor table, plus 4 if the descriptor table is the LDT, as opposed to the GDT (the example above is a GDT descriptor)
base	the linear address (which is the absolute physical address on the 80286) of the segment, except for the expand-down segment
limit	the segment size, minus one (except for an expand-down segment)
ar	the access rights byte, in its entirety, with the text appearing to the right of the descriptor fields disassembling this byte (for more information on how this affects the 80286 and 80386 microprocessors, refer to the *iAPX 286 Programmer's Reference Manual* and *80386 Programmer's Reference Manual*, respectively)
p	the Descriptor Privilege Level (DPL), which is bits 5-6 of **ar**

Gate Descriptors

To display a gate descriptor, use the following procedure:

1 Type **CreateProcess**, where **CreateProcess** is a kernel call reached by a call gate from a user program.

Press **CODE-V**.

The following gate descriptor information appears:

```
iSn     sn      sn : ra       wc    ar    p
0072    0394    02E8:0C48     02    E4    3     call gate
```

The fields that differ from the segment descriptor display are:

sn:ra	the destination address of the gate (**sn** refers to yet another descriptor)
wc	word count — in a call gate only, the number of words to be copied from the top of the caller's stack to the new stack if the call is to a more privileged level (which corresponds to the number of argument words in the procedure call)

Effect of Call Gates on Debugging

If a procedure is reached through a gate, trying to disassemble the procedure code displays the message **Non-existent memory**. For example, typing the procedure name **CreateProcess** and pressing **MARK** produces this message because CreateProcess is a kernel call reached via a gate.

Typing **CreateProcess** and pressing **CODE-V** displays the call gate. Then, typing the sn:ra value shown in the gate and pressing **MARK** disassembles the first instruction of the procedure:

2E8:0C48 MARK PUSH DS

As with all kernel calls and system common procedures, CreateProcess is a large model procedure. (Most user programs use the medium model of segmentation.)

PMOSS uses the PL/M-86 $MOD186 option, which instructs the compiler to use the instructions not available on processors prior to the 80186. The most common 80186 instructions you may see while debugging are ENTER and LEAVE, found near the entry and exit of PMOSS internal procedures. These replace familiar MOV BP,SP, and PUSH BP instructions.

Behavior at a Fault

A fault occurs when a protected mode hardware check detects a condition that prevents execution from continuing (refer to Faults, in section 1).

To clear a fault condition, you can exit by pressing **ACTION-FINISH** if the program has not executed ConvertToSys. If a server program executed ConvertToSys, however, you must reboot the system.

The message **restatable condition** indicates restartable faults. You can patch around this fault in some circumstances by correcting the data that caused the fault (usually, an invalid selector value). These faults can also repeat. If you proceed from a fault (**CODE-P** or **GO**) without correcting the problem, the same fault occurs again.

The Debugger recognizes faults or DEBUG (INT 3) instructions that occur inside PMOSS because the CS value denotes a GDT selector (bit 2 = 0). This event may not necessarily indicate a PMOSS bug. When the user process is executing, it may call the Debugger because an invalid argument (for example, an invalid request block or pointer operand) passed to PMOSS.

To find the cause of this event, check for an internal PMOSS error condition. When PMOSS gets an unexpected error code from an inner procedure, it issues a DEBUG (INT 3) to enter the Debugger and the message **Debugger call at ...** appears. Debugger entry occurs in a PMOSS procedure called CheckErc (a special PMOSS version of this procedure, not the standard version from CTOS.lib).

Use the following procedure to load the PMOSS symbol file and check the CS:IP value to determine if the Debugger was entered from CheckErc:

1 Type **[Sys]<Sys>PMOSS.sym**.

2 Press **CODE-F**.

3 Type **cs:ip**.

4 Press **MARK**.

If the procedure that called the Debugger is named CheckErc, the value in AX is the error code. This represents an internal PMOSS error or an invalid argument that PMOSS could not detect.

If a fault or a Debugger call from some place other than CheckErc occurs, the problem may still not be a PMOSS internal error. Trace the stack back to where execution left the user program and entered PMOSS (usually via a call gate). To do so, you must understand the call gate mechanism.

When control passes through a call gate to a more privileged level, the stack switches. To perform stack tracing when a fault occurs inside PMOSS, examine memory beginning at SS:SP; do not use **CODE-T,** which does not:

- work for most PMOSS procedures because they are large models
- know how to follow an interlevel return back to the user stack

Debugging PMOSS Interrupt Service Routines

PMOSS supports only two types of protected mode interrupt service routines (ISRs):

- raw RS-232 comm ISRs
- programmable interval time (PIT) ISRs

You cannot activate the Debugger when a fault occurs in an ISR because:

- the nonresident portion of the Debugger is not yet in memory
- to bring the Debugger into memory, you must reenable interrupts to use the disk process

Note: If you reenable the interrupts in an ISR before reaching the end of the ISR (and the operating system issues an end-of-interrupt instruction to the device and interrupt controller), the system does not respond to your commands.

To avoid this situation, PMOSS does not enter the Debugger when a fault or fatal error occurs during an ISR. Instead, it calls CRASH with error code 14108 (if a fault occurred) or with a fatal error code (refer to the *BTOS Status Codes Reference Manual*).

You can make the Debugger work properly by installing PMOSS and then locking the Debugger in memory before a fault occurs. You can then use the Debugger when a fault or fatal error occurs.

Allowing the System to Enter the Debugger

There are two ways to have the system enter the Debugger when installing PMOSS:

- Using the RUN command, type **PmAgent.run** in the **Run file** field and **no yes** in the **[Param 4]** field.
- Using the Command File Editor, add the following fourth line to the INSTALL PROTECTED MODE command:

  [Enter debugger (during installation, on isr fault)?]

  and type **no yes** before activating the INSTALL PROTECTED MODE command.

The **yes** selection in the RUN and INSTALL PROTECTED MODE commands lets you debug faults and fatal errors that occur in PMOSS interrupt service routines.

The **no** selection prevents PMOSS from entering the Debugger when you install PMOSS; otherwise, PMOSS enters the Debugger twice:

- once after the system enters protected mode, but before it performs the upper-memory test
- again after the system performs the memory test, relocating Pmoss.img to the upper megabytes

Note: Enter PMOSS in the Debugger only when debugging PMOSS.

Locking the Debugger in Memory

After you allow the system to enter the Debugger, you lock the Debugger in memory before a fault occurs.

There are three ways you can lock the Debugger in memory, depending on whether:

- servers fault during installation, or you are using protected mode programs
- servers fault after you install them, and you do not need to run the Executive or an application program on the workstation
- you want to use the Executive or an application program with **CODE-I** breakpoints

Using Protected Mode Programs and Servers That Fault during Installation

The following procedure describes how to lock the Debugger in memory when using protected mode programs or servers that fault when you install them.

This procedure requires **CODE-I** breakpoints, which prevent the system from ending the primary partition (for example, the Executive). If this occurs, you must wait until after the server installs before setting the breakpoint, since a termination occurs when the server calls ConvertToSys and then Exit.

If you type Executive commands that load run files while the **CODE-I** breakpoint is in effect, you may get a 401 error code or the system may not respond because the Executive tries to terminate. (You can alternatively use the procedure described in Using the Executive or Application Program with the CODE-I Breakpoint, in this section.)

To lock the Debugger in memory when using protected mode programs or servers that fault during installation, use the following procedure:

1 Press **CODE-GO** to run the affected program.

2 Insert a **CODE-I** breakpoint anywhere in real or protected mode code that will never execute.

   If a fault occurs, the system enters the Debugger and reports the fault. You can then use the Debugger, but cannot use symbols.

3 If you must use the Executive or another application, but you receive a 401 error, press **ACTION-A.**

   The system activates the Debugger.

4 Press **CODE-C.**

   The system removes the **CODE-I** breakpoint.

5 Press **GO.**

   The system exits the Debugger.

6 Use the Executive, if you desire.

7 Set the **CODE-I** breakpoint again.

   The system reenters the Debugger.

Note: If a fault occurs while you perform this procedure, the system will not respond to your commands.

Servers That Fault After Installation

The following procedure describes how to lock the Debugger in memory when using servers that fault after you install them. Use this procedure when you do not need to run the Executive or an application program on the workstation to cause the fault.

This procedure requires **CODE-I** breakpoints, which prevent the system from ending the primary partition (for example, the Executive). If this occurs, you must wait until after the server installs before setting the breakpoint, since a termination occurs when the server calls ConvertToSys and then Exit.

If you type Executive commands that load run files while the **CODE-I** breakpoint is in effect, you may get a 401 error code or the system may not respond because the Executive tries to terminate. (You can alternatively use the procedure described in Using the Executive or Application Program with the CODE-I Breakpoint, in this section.)

If a fault occurs after installing a server, use the following procedure to lock the Debugger in memory:

1 After installing the server, wait for the system to return to the Executive level.

2 Press **ACTION-A**.

The system enters the Debugger.

3 Set the **CODE-I** breakpoint at a location that will not execute.

Using the Executive or Application Program with the CODE-I Breakpoint

You can use the Executive or an application program while the CODE-I breakpoint is in effect by running the programs under BTOS Context Manager (refer to the *BTOS Context Manager Administration Guide*). However, you must have:

- enough memory for your servers
- enough memory for BTOS Context Manager, with 80 Kb reserved for the Debugger
- at least one application partition of sufficient size

To use the Executive or application program with CODE-I breakpoints, use the following procedure:

1 Using the BTOS Context Manager Configuration File Editor, reserve at least 80 Kb of memory for the Debugger (refer to the *BTOS Context Manager Administration Guide*).

2 Install BTOS Context Manager.

3 Set the **CODE-I** breakpoint.

The Debugger locks into the reserved memory and does not interfere with the Executive or any other application.

SPA Mover Interface

PMOSS is compatible with, replaces, and subsumes the functionality of the SPA Mover Server (Mover.run). This allows PMOSS to have as clients the RamDisk Server (RamDisk.run) and BTOS Context Manager, as well as other clients of the SPA request interface. PMOSS uses the identical request interfaces that Mover.run uses.

However, PMOSS uses these interfaces differently than SPA uses them, which may have implications for some clients.

Although correctly written client programs do not notice the difference, you can write client programs that use the Mover Server interface and work with the SPA Mover.run, but not with PMOSS' implementation of the same interface. This section explains how to write programs that work in either environment.

For more information on SPA, refer to the *System Performance Accelerator (SPA) Installation Guide*. For more information on BTOS Context Manager, refer to the *BTOS Context Manager Administration Guide*.

Procedural Interfaces

SPA does not provide a procedural interface to the Mover Server requests. Clients are required to build, send, and wait for their own request blocks.

This restriction is due only to the choice of request code numbers (in an odd-level request code range where the automatic procedural interface is not available). PMOSS adds new request codes that are semantically equivalent to the old codes (in a new, even-level range where a procedural interface is automatically provided). PMOSS also recognizes the old codes, for compatibility with existing clients.

Appendix A describes the following Mover Segment request interfaces:

- AllocMoverSegment: Obtains a variable-length segment of memory for use as a cache.
- MovbMoverSegment: Moves bytes between two locations, either or both of which can be in a previously allocated cache memory segment or in real mode memory.
- DeallocMoverSegment: Frees a previously allocated cache memory segment.
- QueryVersionMoverSegment: Obtains the Mover Server version number.

Mover Segments

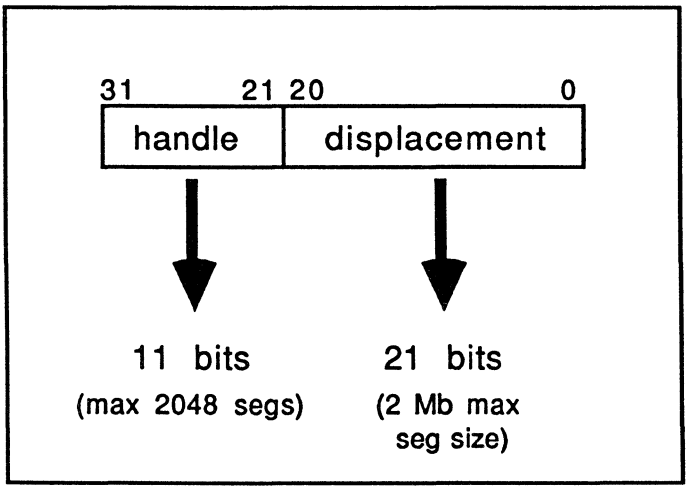
PMOSS performs various checks that SPA does not perform. For example, SPA lets the client write to any part of protected mode memory, without checking that the client has already allocated the memory.

PMOSS uses mover segments, which let clients read or write only those portions of protected mode memory that they have previously been allocated. Each AllocMoverSegment request creates one such segment. Mover segments are similar to the program segments that protected mode programs use, except:

- Individual mover segments can be up to 2 Mb in size.
- Client programs cannot access mover segments directly. Like SPA, mover segments can be accessed only via the MovbMoverSegment request.
- Addresses of mover segments, or objects within mover segments, are called mover addresses (refer to figure 4-1). Mover addresses are not SA:RA addresses that programs use. They are 32 bits long, and consist of an 11-bit handle and a 21-bit displacement. Mover addresses are not physical addresses (unlike SPA). They are a new kind of logical address compatible, within certain limits, with the physical addresses that SPA uses.

The AllocMoverSegment request returns a mover address (with PMOSS) or a physical address (with SPA). Compatible programs must not depend on one or the other address form.

Figure 4-1 **32-bit Address**



A PMOSS mover address uses the high-order 11 bits for the handle. Each AllocMoverSegment request returns a unique handle, but two successive requests may not return consecutive handles. AllocMoverSegment never returns a handle of zero; the handle zero is reserved and has a special meaning when used in the MovbMoverSegment request.

PMOSS always returns zeroes in the low-order 21 bits (the displacement), but the client must not depend on this because SPA does not usually return zeroes in these bits.

Validation Checks

The MovbMoverSegment request takes a source and destination operand, each consisting of a mover address and a byte count. Either operand can refer to real mode memory or cache memory (and both operands can refer to the same kind of memory). PMOSS performs the following checks on this request:

- For each operand, the mover address handle must be either zero, indicating that the displacement part refers to real mode memory, or an existing mover segment handle.
- If the operand refers to real mode memory (zero handle), the displacement and count must specify that all the indicated bytes lie in the range 0 to 992 Kb.
- If the operand refers to a mover segment (nonzero handle part), the userNum field of the MovbMoverSegment request block must match the userNum field in the AllocMoverSegment request block previously used to allocate the mover segment. This is the case when using the procedural interface, provided the allocation and the move are issued from the same partition.
- Each operand that refers to a mover segment can specify part or all of only one mover segment. The request may not span mover segments, even when the segments are allocated to the same userNum (because they are not necessarily adjacent physically).
- Any alignment is allowed, but using odd mover addresses slows byte transfer.

New Procedural Interfaces

This section lists new BTOS interfaces added to support protected mode compatibility. This section also describes certain existing interfaces to explain their behavior in protected mode.

AllocAllMemorySL

AllocAllMemorySL (pcParagraphRet, ppSegmentRet): ercType

Description

AllocAllMemorySL creates a short-lived segment and allocates all free memory available to the program.

In protected mode, if the size of allocated memory is greater than 64 Kb, the region beyond 64 Kb is not addressable using the returned memory address.

For this reason, programs that run in both real and protected modes should use the following short-lived memory allocation operations:

- AllocMemorySL
- AllocAreaSL
- Expand9AreaSL

Procedural Interface

AllocAllMemorySL (pcParagraphRet, ppSegmentRet): ercType

where

pcParagraphRet is the memory address of a word where the count of bytes available (divided by 16) returns.

ppSegmentRet is the memory address into which the 4-byte memory address of the allocated segment returns. The offset, which is always 0, returns in the low-order 2 bytes. The segment address returns in the high-order 2 bytes.

Request Block

scParagraphMax is always 2 and spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	46
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	2	2
24	ppSegmentRet	4	
28	spSegmentMax	4	4

AllocAreaSL

AllocAreaSL(cBytes, ppSegmentRet): ercType

Description

AllocAreaSL creates a short-lived segment and allocates memory of the specified size at the end of the segment.

This operation differs from AllocMemorySL in that the offset portion of the allocated segment's address is not necessarily zero. (Refer to AllocMemorySL, in this section.)

In protected mode, this operation also allocates a new expand-down data segment descriptor.

Programs that run in both real and protected modes should use AllocAreaSL if the program allocates additional memory in the short-lived segment. A program should use ExpandAreaSL to allocate additional memory.

Procedural Interface

AllocAreaSL(cBytes, ppSegmentRet): ercType

where

cBytes is the count of bytes to be allocated.

ppSegmentRet is the memory address into which the 4-byte memory address of the allocated segment returns.

The offset, which is not necessarily 0, returns in the low-order 2 bytes. The segment address returns in the high-order 2 bytes.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	278
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocMemoryLL

AllocMemoryLL (cBytes, ppSegmentRet): ercType

Description

AllocMemoryLL creates a long-lived segment and allocates memory of the specified size at the beginning of the segment.

In protected mode, this operation also allocates a new expand-up data segment descriptor.

Programs that run in both real and protected modes should use ExpandAreaLL to allocate additional memory within the segment that AllocMemoryLL creates.

Procedural Interface

AllocMemoryLL (cBytes, ppSegmentRet): ercType

where

cBytes is the desired segment size.

ppSegmentRet is the memory address into which the 4-byte memory address of the allocated segment returns. The offset, which is always 0, returns in the low-order 2 bytes. The segment address returns in the high-order 2 bytes.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
	nRespPbCb	1	1
4	userNum	2	
6	exchResp		
8	ercRet	2	
10	rqCode	2	44
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocMemorySL

AllocMemorySL (cBytes, ppSegmentRet): ercType

Description

AllocMemorySL creates a short-lived segment and allocates memory of the specified size at the beginning of the segment. AllocMemorySL differs from AllocAreaSL in that the offset portion of the allocated segment's address is always 0. (Refer to AllocAreaSL, in this section.)

In protected mode, this operation also allocates a new expand-up data segment descriptor.

Programs that run in both real and protected modes should use this operation only if additional memory will not be allocated in the segment. A program should use AllocAreaSL if additional memory will be allocated.

Procedural Interface

AllocMemorySL (cBytes, ppSegmentRet): ercType

where

cBytes is the desired segment size.

ppSegmentRet is the memory address into which the 4-byte memory address of the allocated segment returns. The offset, which is always 0, returns in the low-order 2 bytes. The segment address returns in the high-order 2 bytes.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	42
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocMoverSegment

AllocMoverSegment (qb, pMaRet): ercType

Description

AllocMoverSegment allocates a variable-length segment of protected mode memory for caching. It is typically used in real mode programs to escape the 1 Mb address space limitation of real address mode, although it is valid in either mode.

The requested segment can be up to 2 Mb in size if there is sufficient protected mode memory available. The segment is uninitialized memory.

The segment allocated is not directly addressable. You must use the MovbMoverSegment request to write and read the segment.

The System Performance Accelerator (SPA) Mover Server or PMOSS implements this operation. One of these servers must be installed to use the operation.

Procedural Interface

AllocMoverSegment (qb, pMaRet): ercType

where

qb is the desired segment size, in bytes (a 4-byte quantity, in the range zero to 200000h).

pMaRet is the address of the 4-byte memory location into which the mover segment address returns.

Note: PMOSS supports the procedural interface, but SPA does not. SPA clients must build their own request blocks.

Request Block

sMaMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4124*
12	qb	4	
16	pMaRet	4	
20	sMaMax	2	4

* PMOSS supports the alias request code 32949, but SPA does not. This is the request code that the procedural interface uses.

DeallocMemoryLL

DeallocMemoryLL (pSegment, cBytes): ercType

Description

DeallocMemoryLL deallocates the specified long-lived segment containing memory of the specified size. Segments must be deallocated in a sequence exactly opposite to how they were allocated (that is, last allocated, first deallocated).

In protected mode, this operation deallocates the segment descriptor.

Procedural Interface

DeallocMemoryLL (pSegment, cBytes): ercType

where

pSegment is the memory address of the segment to deallocate. The offset portion must be 0. pSegment should be the same memory address that the corresponding AllocMemoryLL operation returned.

cBytes is the size, in bytes, of the segment to deallocate. cBytes should be the same value that passed to the corresponding AllocMemoryLL operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	45
12	cBytes	2	
14	pSegment	4	

DeallocMemorySL

DeallocMemorySL (pSegment, cBytes): ercType

Description

DeallocMemorySL deallocates the specified short-lived segment containing memory of the specified size. Segments must be deallocated in a sequence exactly opposite to how they were allocated (that is, last allocated, first deallocated).

In protected mode, this operation deallocates the segment descriptor.

Procedural Interface

DeallocMemorySL (pSegment, cBytes): ercType

where

pSegment is the memory address of the segment to deallocate. The offset portion must be 0. pSegment should be the same memory address that the corresponding AllocMemorySL operation returned.

cBytes is the size (in bytes) of the segment to deallocate. cBytes should be the same value that passed to the corresponding AllocMemorySL operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	43
12	cBytes	2	
14	pSegment	4	

DeallocMoverSegment

DeallocMoverSegment (qb, pMaRet): ercType

Description

DeallocMoverSegment frees a variable-length segment of protected mode memory that AllocMoverSegment allocated.

The SPA Mover Server or PMOSS implements this operation. One of these servers must be installed to use the operation.

For compatibility with all versions of the SPA and PMOSS, the exact same mover address that AllocMoverSegment returns must be used to deallocate the segment. The qb must also be identical to the one used with AllocMoverSegment. It is incompatible to free a portion of a mover segment.

Procedural Interface

DeallocMoverSegment (ma, qb): ercType

where

ma is the 4-byte mover segment address that a previous AllocMoverSegment request returns.

qb is the segment size, in bytes (a 4-byte quantity, in the range zero to 200000h).

Note: PMOSS supports the procedural interface, but SPA does not. SPA clients must build their own request blocks.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4125*
12	ma	4	
16	qb	4	

* PMOSS supports the alias request code 32954, but SPA does not. This is the request code that the procedural interface uses.

ExpandAreaLL

ExpandAreaLL(cBytes, sa, pRaRet): ercType

Description

ExpandAreaLL allocates additional memory of the specified size within the specified long-lived segment. A prior call to AllocMemoryLL creates the specified segment.

Programs that run in both real and protected modes should use the ExpandAreaLL operation to allocate additional memory in a long-lived segment.

Procedural Interface

ExpandAreaLL(cBytes, sa, pRaRet): ercType

where

cBytes is the amount, in bytes, by which the segment will expand. The system returns status code 400 (**Memory not available**) if the resulting segment is larger than 64 Kb.

sa is the segment address (high-order 2 bytes of a memory address) of the segment to be expanded.

pRaRet is the memory address of a word in which the offset (low-order 2 bytes of a memory address) of the newly allocated memory returns.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	280
12	cBytes	2	
14	sa	2	
16	reserved	2	
18	pRaRet	4	
22	sRaRetMax	2	2

ExpandAreaSL

ExpandAreaSL(cBytes, sa, pRaRet): ercType

Description

ExpandAreaSL allocates additional memory of the specified size within the specified short-lived segment. A prior call to AllocAreaSL creates the specified segment. The segment grows by expanding downward (toward lower offsets).

Programs that run in both real and protected modes should use ExpandAreaSL to allocate additional memory in a short-lived segment.

Procedural Interface

ExpandAreaSL(cBytes, sa, pRaRet): ercType

where

cBytes is the amount, in bytes, by which the segment will expand. The system returns status code 400 (**Memory not available**) if the resulting segment is larger than 64 Kb.

sa is the segment address (high-order 2 bytes of a memory address) of the segment to be expanded.

pRaRet is the memory address of a word in which the offset (low-order 2 bytes of a memory address) of the newly allocated memory returns.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	279
12	cBytes	2	
14	sa	2	
16	reserved	2	
18	pRaRet	4	
22	sRaRetMax	2	2

FComparePointer

FComparePointer(p1, p2, bCompareMode): FlagType

Description

FComparePointer returns TRUE if the pointers p1 and p2 are equal. The bCompareMode parameter defines the meaning of pointer equality:

0 Returns TRUE only if p1 and p2 have the same 32-bit binary value.

1 Returns TRUE if p1 and p2 address the same byte location in the linear address space.

For example, in real mode, the pointers 10:8 and 0:108 are equal. In protected mode, the pointers 88:4 and 9C:2A0 are equal only if the linear base address stored in the GDT segment descriptor identified by selector 88h, plus 4, equals the linear base address stored in the LDT descriptor identified by selector 9Ch, plus 2A0h. (Refer to the note.)

This comparison is expensive. Use it only when there is a possibility of alias addresses.

2 If in real mode, returns TRUE only if p1 and p2 have the same 32-bit binary value.

In protected mode, returns TRUE if p1 and p2 have the same binary value, except for possible differences in the RPL field of the selectors.

This is the typical meaning of pointer equality.

Procedural Interface

FComparePointer(p1, p2, bCompareMode): FlagType

where

p1 is the first pointer for comparison.

p2 is the second pointer for comparison.

bCompareMode determines how the test for equality is made.

Request Block

FComparePointer is an object module procedure.

Note: This version of FComparePointer does not fully implement the bCompareMode = 1 option in protected mode. Normalized comparison of real mode paragraph numbers is supported, but protected mode treats bCompareMode = 1 like bCompareMode = 2. This should not affect programs since no facilities exist in PMOSS for a protected mode user program to create two alias pointers visible to the same task.

ForwardRequest

ForwardRequest(exch, prq): ercType

Description

Filter processes use the ForwardRequest primitive to forward a request block to another server for further processing. Use it only in filter processes that intercept a request on its way to the original server, but not on its return trip to the client. In real mode, these filter processes traditionally use the Send or RequestDirect primitive to forward the request to the original server. To make this program compatible with protected mode, use ForwardRequest in place of Send or RequestDirect. ForwardRequest is similar to Send in that no response is expected; by issuing it, the filter processes discharges its responsibility for the request as if it had done a Respond. It is similar to RequestDirect in that its argument must always be a pointer to a request block.

Procedural Interface

ForwardRequest(exch, pRq): ercType

where

exch is the exchange where the request block is to be sent.

pRq is the request block memory address.

Request Block

ForwardRequest is a Kernel primitive.

FProcessorSupportsProtectedMode

FProcessorSupportsProtectedMode: FlagType

Description

FProcessorSupportsProtectedMode returns TRUE on an 80286 or subsequent microprocessor (a processor capable of protected mode execution).

FProcessorSupportsProtectedMode returns FALSE on an 8086 or 80186 microprocessor.

The actual mode in which the processor is executing (real or protected) has no effect on the result.

FProcessorSupportsProtectedMode does not indicate whether PMOSS is installed or not; it indicates the microprocessor's capabilities only.

Procedural Interface

FProcessorSupportsProtectedMode: FlagType

Request Block

FProcessorSupportsProtectedMode is an object module procedure.

FProtectedMode

FProtectedMode: FlagType

Description

FProtectedMode returns TRUE if the microprocessor is executing in protected mode. It returns FALSE if the microprocessor is executing in real mode.

You can use FProtectedMode on any Intel microprocessor. FProtectedMode always returns FALSE on microprocessors that do not support protected mode execution.

Procedural Interface

FProtectedMode: FlagType

Request Block

FProtectedMode is an object module procedure.

MovbMoverSegment

MovbMoverSegment (maFrom, maTo, cb): ercType

Description

MovbMoverSegment reads and/or writes data in a mover segment (a cache segment) that a AllocMoverSegment request created.

With this request, you can move bytes:

- from real mode memory to a mover segment
- from a mover segment to real mode memory
- within a mover segment
- between two mover segments
- between two real mode locations

The SPA Mover Server or PMOSS implements this operation. One of these servers must be installed to use the operation.

Procedural Interface

MovbMoverSegment (maFrom, maTo, cb): ercType

where

maFrom is the source mover address.

maTo is the destination mover address.

cb is the number of bytes to move. (The maximum number of bytes that can be moved in one request is 65535, even though the maximum mover segment size is larger.)

Note: PMOSS supports the procedural interface, but SPA does not. SPA clients must build their own request blocks.

The following rules must be followed to ensure compatibility with SPA and PMOSS.

The maFrom and maTo operands can refer to a real mode memory address, a mover segment address, or an address within a mover segment. They can be of the same or different types.

For real mode addresses, the operand is a 20-bit physical address and the high-order 12 bits must be zero.

For mover addresses, the operand can be the mover segment address that an AllocMoverSegment request returns, or such an address plus a displacement to any byte location within the mover segment.

Do not assume that the AllocMoverSegment value is a physical address. The only property of such values common to all SPA and PMOSS implementations is that they are never zero in the high-order 12 bits (so you can never confuse them with real mode memory addresses).

Operands cannot span mover segments. The displacement (if any) plus the cb operand must not overlap beyond the end of a mover segment. Two AllocMoverSegment requests always return two separate mover segments (even if issued consecutively), which cannot be treated as a combined segment.

Odd maFrom or maTo addresses slows byte transfer in some implementations.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4123*
12	maFrom	4	
16	maTo	4	
20	cb	2	

* PMOSS supports the alias request code 32948, but SPA does not. This is the request code that the procedural interface uses.

QueryBigMemAvail

QueryBigMemAvail(pqRet): ercType

Description

QueryBigMemAvail returns the size, in bytes, of free memory that is available to the program. Size is a function of the following:

- physical memory size
- limits specified in the run file containing the QueryBigMemAvail operation
- limits specified at partition creation time

The following operations allocate free memory:

AllocAllMemorySL
AllocAreaSL
AllocMemoryLL
AllocMemorySL
ExpandMemoryLL
ExpandAreaSL

Programs that operate in both real and protected modes should use this operation rather than QueryMemAvail because the latter operation reports a maximum size of only 1 Mb.

Procedural Interface

QueryBigMemAvail(pqRet): ercType

where

pqRet is the memory address where the 4-byte amount of memory available returns.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	283
12	reserved	6	
18	pqRet	4	
22	sqMax	2	4

QueryMemAvail

QueryMemAvail (pcParagraphRet): ercType

Description

QueryMemAvail returns the size,in 16-byte paragraphs, of free memory that is available to the program.

The following operations allocate free memory:

AllocAllMemorySL
AllocAreaSL
AllocMemoryLL
AllocMemorySL
ExpandMemoryLL
ExpandAreaSL

Programs that operate in both real and protected modes should use QueryBigMemAvail rather than this operation because QueryMemAvail report a maximum size of only 1 Mb.

Procedural Interface

QueryMemAvail (pcParagraphRet): ercType

where

pcParagraphRet is the memory address of a word where the count of bytes available (divided by 16) returns.

Request Block

scParagraphMax is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	48
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	4	2

SetPStructure

SetPStructure(wStructCode, ph, oField, pb, cb): ErcType

Description

SetPStructure provides controlled write-access to selected fields of certain system data structures that user programs can modify legitimately.

Object module procedures in CTOS.lib use SetPStructure primarily to perform system-related functions. In most cases, it is unnecessary for a user program to use SetPStructure directly.

In real mode, a user program can modify and examine a system data structure once a pointer to the structure has been acquired using GetPStructure. This is not possible in protected mode because GetPStructure returns pointers based on read-only descriptors (for most structures) in keeping with the protected mode's primary obejective: protecting system structures against accidental damage.

SetPStructure provides write-access on a field-by-field basis. You can use SetPStructure to modify only certain fields and validate the values placed in these fields.

You must use a separate call to SetPStructure for setting each field to be modified, except for adjacent pb/cb fields, which are modified by a single call to SetPStructure. In this way, SetPStructure is unlike its counterpart, GetPStructure, which returns a pointer to the entire system data structure.

Procedural Interface

SetPStructure(wStructCode, ph, oField, pb, cb): ErcType

wStructCode identifies the system data structure. This parameter is identical to the corresponding parameter of GetPStructure.

ph is the partition handle. If ph = 0, the operation applies to the calling partition.

oField is the offset, within the system data structure, of the field to be modified. This parameter must correspond to the first byte of a field. (For information on which fields are supported, refer to SetPStructure Cases Supported, in appendix B.)

pb a pointer. This parameter's meaning depends on the type of field being addressed.

cb a count of bytes. This parameter's meaning depends on the type of field being addressed.

SetPStructure is extended to support access to additional fields and new system data structures as necessary, without introducing new procedural interfaces. Because of this generality, the pb and cb parameters are interpreted differently, depending on the type of field being modified:

p The system data structure field is a 32-bit SA:RA logical address.

The pb parameter is the pointer value to be placed in the structure (not the address of the value). The cb parameter is unused.

sa The field is a 16-bit segment address (in real mode, a paragraph number; in protected mode, a selector).

The SA portion of the pb parameter (the high-order 16 bits) replaces the field. The RA portion of pb (the low-order 16 bits), and cb, are unused.

pb/cb A pb and cb pair occur as adjacent, related fields in a system data structure.

The pb and cb operands replace the fields. This is the only instance in which a single call to SetPStructure modifies two fields.

The oField parameter should be the offset of the pb field.

sb The field takes a variable-length string value. The first byte of the field is the current length of the string (excluding the length byte).

The pb parameter is the address of the new string value (without a length byte). The new string value copies to the field, beginning at the second byte of the field. The cb parameter is the length of the string value and placed in the first byte of the field; it cannot exceed the maximum size of the field minus one.

other Any other fixed-length field, including byte, word, dword, or larger fixed-length fields.

The pb parameter is the address of the new value (not the value itself) and cb must exactly match the size of the field.

Request Block

SetPStructure is a system common procedure.

ShrinkAreaLL

ShrinkAreaLL(p, cBytes): ercType

Description

ShrinkAreaLL deallocates memory of the specified size within the specified long-lived segment. Memory must be deallocated in a sequence exactly opposite to how it was allocated (that is, last allocated, first deallocated).

ShrinkAreaLL differs from DeallocMemoryLL in that the offset portion of the memory pointer can be nonzero.

Use ShrinkAreaLL for a program that runs in both real and protected modes if the program deallocates memory in a long-lived segment.

Procedural Interface

ShrinkAreaLL(p, cBytes): ercType

where

p is the memory address of the memory to be deallocated.

cBytes is the count of bytes of memory to be deallocated.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	282
12	cBytes	2	
14	p	4	

ShrinkAreaSL

ShrinkAreaSL(p, cBytes): ercType

Description

ShrinkAreaSL deallocates memory of the specified size within the specified short-lived segment. Memory must be deallocated in a sequence exactly opposite to how it was allocated (that is, last allocated, first deallocated).

ShrinkAreaSL differs from DeallocMemorySL in that the offset portion of the memory pointer may be nonzero.

Use ShrinkAreaSL for a program that runs in both real and protected modes if the program deallocates memory in a short-lived segment.

Procedural Interface

ShrinkAreaSL(p, cBytes): ercType

where

p is the memory address of the memory to be deallocated.

cBytes is the count of bytes of memory to be deallocated.

Request Block

Size Offset	Field	(bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	281
12	cBytes	2	
14	p	4	

Summary of GetPStructure Interfaces

Access to System Data Structures

The GetPStructure system common procedure lets programs obtain pointers to certain system data structures.

GetPStructure is the only ways to obtain this access in protected mode. You cannot obtain a pointer directly from the System Common Address Table (SCAT) at an absolute address in low memory. However, GetPStructure accepts offsets into this table (as well as small-integer codes of its own) as a parameter identifying the desired pointer.

Limitations in Protected Mode

The GetPStructure interface is documented in the *BTOS Reference Manual.* This description applies to both real and protected mode. Table B-1 shows how access is limited in protected mode. There are several ways in which access is limited in protected mode:

- Not all the structCode cases of GetPStructure are implemented in protected mode. Unsupported codes (shown in table B-1) return an error code.
- The pointers that GetPStructure returns usually include a selector that references a read-only descriptor, so the user program cannot modify the system data structure. (Exceptions that return read/write descriptors are footnoted in table B-1.) A new system common procedure, SetPStructure, must be used to set fields in read-only system data structures. SetPStructure is valid only for appropriate fields.
- The descriptor base and limit fields permit access only to the designated data structure, not to adjacent data structures in the same segment. For example, a pointer to an object traditionally located in BTOS' DGroup (such as the system date-time structure) cannot address other objects in BTOS' DGroup (such as the array of file control blocks), even for read-only access. Also, refer to the Size column of table B-1.

However, when GetPStructure returns a pointer that addresses an offset, or an array of offsets, the user can assume that these offset(s) are located in the same segment as the object(s) they point to, and that the descriptor that GetPStructure returns encloses both the offset(s) and the object(s) to which they point.

GetPStructure Codes

The Code(S) column of table B-1 shows the wStructCode parameter value for GetPStructure.

The Size column shows the number of bytes accessible using the pointer that is returned.

Unless otherwise noted, all descriptors are read-only expand-up data segments.

SetPStructure Cases Supported

The SetPStructure interface is described in appendix A.

SetPStructure is defined for only those fields to which object module procedures in BTOS 8.0 required write-access. SetPStructure allows additional fields to be supported in the future, without adding new procedural interfaces.

Table B-2 shows the supported SetPStructure cases.

Table B-1 **GetPStructure Cases Supported**

Code(s)	Size	Structure
0	171	Extended Partition Desc[1]
1	UNSUPPORTED	OS Video Character Map
2 or 244h	*	VCB (Video Control Block)[2]
3 or 250h	304	ASCB
4	*	VLPB
5	1489	BCB (Batch Control Block)
6	UNSUPPORTED	Type-ahead Buffer
7	UNSUPPORTED	RgPVidMemLine
8	UNSUPPORTED	RgLineMap
9	UNSUPPORTED	Partition Swap Status
10	UNSUPPORTED	RgUserReadCount
11	UNSUPPORTED	RgPDebuggerState
12	UNSUPPORTED	GraphicsInfo
13	*	ECB (Event Control Block)
14	84	NGEN Port Structure
20	32	RgPRcLookUp (request table)
21	32	RgPRcLookUpBase (ditto)
22	32	RgRcMax (ditto)
32	42	PARD (Partition Descriptor)[3]
240h	6	System Date/Time Structure[4]
2C8h	37	System Configuration Block

* Variable-sized. System builds an appropriate descriptor based on the size of the structure.

[1] First four fields only

[2] Writable descriptor

[3] BTOS 8.0 does not support code 32 in real mode. Both modes support GetPartitionStatus.

[4] This selector does not include the entire BTOS DGroup, only the 6-byte date/time structure.

Table B-2 **SetPStructure Cases Supported**

Code	Struct	Offset	Field	Size
3	ASCB	10	pbMsgRet/cbMsgRet	6
13	ECB	2	qMailId	4
		6	sbNodeMail	13
32	PARD	16	sbPartitionName	13

Glossary

Alias Descriptor. An alias descriptor allows one user program (for example, a server) to see an object in another program's segment. PMOSS constructs the alias descriptor in the server's LDT for this purpose.

Call Gate. A call gate is a protection mechanism that limits the user program to calling at legitimate entry points and provides a convenient way to bind the user program to those entry points at program load time.

Current Privilege Level (CPL). The Current Privilege Level is the level at which the process is currently running (essentially, the privilege level of the currently executing code), which is stored in the CS register and can be displayed by using the Debugger to examine the CS register.

Descriptor Cache Register. The descriptor cache register is an internal register associated with the CS, DS, ES, and SS segment registers, but does not appear to the software. It holds the entire contents of the descriptor, including segment base address and other information.

Descriptor Privilege Level (DPL). The Descriptor Privilege Level marks each entity in the current address space (everything described by a descriptor in the current LDT or GDT), which determines its usability.

Descriptors. Descriptors are eight bytes long and contain various information about a segment. They are basically an offset into the Logical Descriptor Table, with some additional bits used for special purposes.

Descriptor Table. A descriptor table is a special type of table that only the operating system and hardware can access. In protected mode, the CS, DS, ES, and SS registers hold a 16-bit SA that is an index into the descriptor table.

Exception. An exception is an error that prevents the program from executing.

External Interrupt. An external interrupt is initiated by an asynchronous event outside the processor. External interrupts are I/O interrupts such as disk and real-time clock interrupts.

Fault. A fault is a condition (such as a protection violation) that prevents the hardware from completing an instruction.

Gate Descriptor. A gate descriptor is a structure that uses indirection to allow programs to call routines whose addresses they cannot know until the program is loaded.

General Protection Faults. General protection faults are generated by breaking any of the protection rules, which activate the Debugger.

Global Descriptor Table (GDT). The Global Descriptor Table is a special table that is PMOSS' LDT and the descriptors in it are used only when PMOSS' code is executing, never when user code is executing. The single Global Descriptor Table is never switched; it is always in effect, no matter what Local Descriptor Table is in effect.

Internal Interrupt. An internal interrupt is initiated synchronously as a result of an instruction executing. Internal interrupts include software interrupts, (which happen when an INT instruction executes), exceptions (such as interrupt type 4), and faults (including protection faults).

Interrupt Descriptor Table (IDT). An Interrupt Descriptor Table is a special descriptor table that contains the interrupt type numbers corresponding to every interrupt. There is one Interrupt Descriptor Table per system.

IOPL Flag. The IOPL flag specifies the maximum CPL (numerically) for which I/O and HALT instructions are valid.

Limit Exception. A limit exception is an exception that occurs when using a valid segment register in an address calculation (such as when you try to address beyond the end of a segment).

Linear Address. A linear address is formed from the logical address as an instruction executes and then addresses physical memory.

Linear Address Model. A linear address model refers to an architecture (such as the Motorola architecture) in which instructions accept 32-bit linear addresses (instead of SA:RA pairs, as with the segmented addressing model).

Logical Address. A logical address is composed of the segment address (SA) and the relative address (RA). You use the SA:RA syntax to write a logical address when using the Debugger or Assembler.

Local Descriptor Table (LDT). The Logical Descriptor Table is an array of descriptors. PMOSS constructs and maintains the Logical Descriptor Table for each run file executing in protected mode.

Paragraph. A paragraph is a 16-byte unit of memory aligned on a 16-byte boundary.

Paragraph Number. A paragraph number is a real address mode SR, which denotes a particular 16-byte boundary in the physical address space.

Protected Mode Operating System Server (PMOSS). Protected Mode Operating System Server installs on BTOS and lets you write system services that are compatible with protected mode (the 3 Mb over the 1 Mb of real mode).

Real Address. The real address is the real address mode SR:RA logical address because it always corresponds to the same physical memory address.

Real Address Mode. Real address mode is the mode in which 8086 and 80186 microprocessors operate all the time. 80286 microprocessors operate in real address mode when powered-up or reset, but can switch to protected mode if the operating system software supports protected mode.

Relative Address (RA). The relative address comprises one-half of the linear address. It is often referred to as an offset from the segment address.

Restartable Fault. A restartable fault is a fault (such as a not-present fault) that is, in theory, recoverable.

Ring Protection Model. A ring protection model affords protection by privilege level, in which every program executes at one of several levels of authority. It is intended to allow an operating system (such as PMOSS) to conveniently protect itself from its clients.

Segment Address (SA). The segment address comprises one-half of the linear address.

Segmented Address Model. A segmented address model refers to the Intel architecture in which every address is always relative to some segment address (SA).

Segment Registers. Segment registers contain the paragraph numbers corresponding to the vase of the current code (CS), data (DS), extra (ES), and stack segments (SS). These segments are always aligned to start on 16-byte boundaries.

SN. The SN is a segment address that is in protected mode.

SR. The SR is a segment address that is a paragraph number (a real-address mode SA).

System Performance Accelerator (SPA). The System Performance Accelerator is an installed system service that improves the response time for workstations performing file-system operations and also provides a caching mechanism.

Task State Segment (TSS). The Task State Segment is associated with every process and contains the complete register state of the process. It permits extremely rapid process switching, despite protection boundaries between programs.

TSS Gate. A TSS gate lets you arrange for an interrupt to perform a process switch automatically (in effect, an automatic Call to a TSS).

Virtual Address. The virtual address is the protected mode SN:RA logical address because the SN refers only indirectly to memory via a descriptor table entry.

Index

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

V

W

Title: _____

Form Number: _____ Date: _____

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: _____

_____

_____

_____

Name _____

Title _____

Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code

Title: _____

Form Number: _____ Date: _____

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: _____

_____

_____

_____

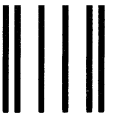
Name _____

Title _____

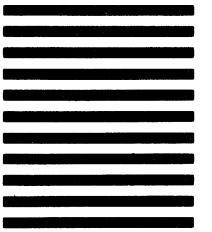
Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

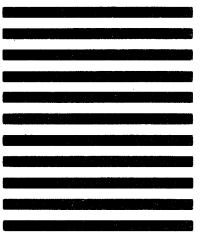
FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Unisys Corporation
1300 John Reed Court
City of Industry, CA 91745 USA

ATTN: Corporate Product Information

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Unisys Corporation
1300 John Reed Court
City of Industry, CA 91745 USA

ATTN: Corporate Product Information

Burroughs

Printed in U.S.A.