CHAPTER 6
TASK MANAGEMENT

This chapter describes the IA-32 architecture's task management facilities. These facilities are only available when the processor is running in protected mode.

6.1. TASK MANAGEMENT OVERVIEW

A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.

The IA-32 architecture provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another. When operating in protected mode, all processor execution takes place from within a task. Even simple systems must define at least one task. More complex systems can use the processor's task management facilities to support multitasking applications.

6.1.1. Task Structure

A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments (see Figure 6-1). If an operating system or executive uses the processor's privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level.

The TSS specifies the segments that make up the task execution space and provides a storage place for task state information. In multitasking systems, the TSS also provides a mechanism for linking tasks.

NOTE

This chapter describes primarily 32-bit tasks and the 32-bit TSS structure. For information on 16-bit tasks and the 16-bit TSS structure, see Section 6.6., "16-Bit Task-State Segment (TSS)".

A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for the TSS are loaded into the task register (see Section 2.4.4., "Task Register (TR)").

If paging is implemented for the task, the base address of the page directory used by the task is loaded into control register CR3.

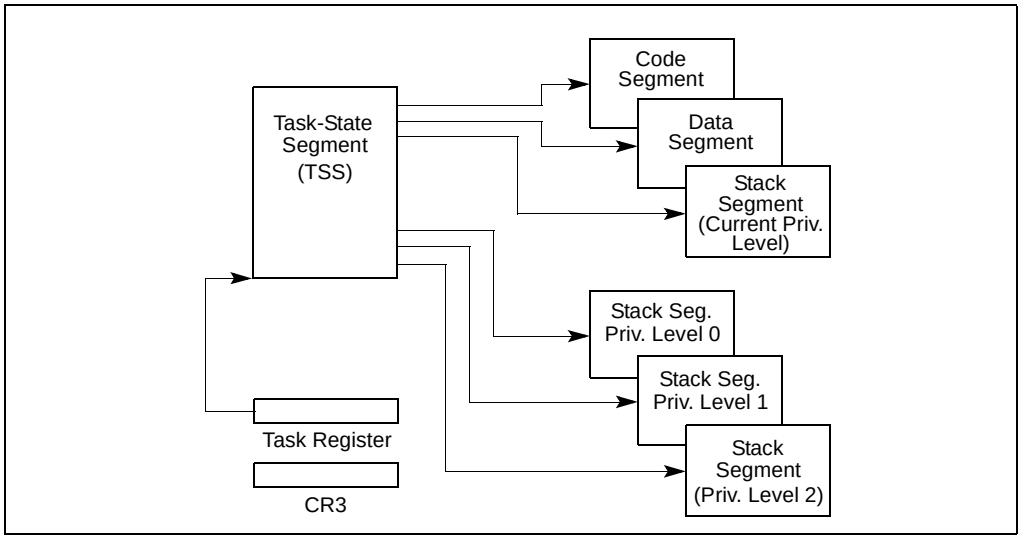

Figure 6-1. Structure of a Task

6.1.2. Task State

The following items define the state of the currently executing task:

The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).

The state of the general-purpose registers.

The state of the EFLAGS register.

The state of the EIP register.

The state of control register CR3.

The state of the task register.

The state of the LDTR register.

The I/O map base address and I/O map (contained in the TSS).

Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).

Link to previously executed task (contained in the TSS).

Prior to dispatching a task, all of these items are contained in the task's TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

6.1.3. Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

A explicit call to a task with the CALL instruction.

A explicit jump to a task with the JMP instruction.

An implicit call (by the processor) to an interrupt-handler task.

An implicit call to an exception-handler task.

A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods of dispatching a task identify the task to be dispatched with a segment selector that points either to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select either the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task to handle an interrupt or exception, the IDT entry for the interrupt or exception must contain a task gate that holds the selector for the interrupt- or exception-handler TSS.

When a task is dispatched for execution, a task switch automatically occurs between the currently running task and the dispatched task. During a task switch, the execution environment of the currently executing task (called the task's state or **context**) is saved in its TSS and execution of the task is suspended. The context for the dispatched task is then loaded into the processor and execution of that task begins with the instruction pointed to by the newly loaded EIP register. If the task has not been run since the system was last initialized, the EIP will point to the first instruction of the task's code; otherwise, it will point to the next instruction after the last instruction that the task executed when it was last active.

If the currently executing task (the calling task) called the task being dispatched (the called task), the TSS segment selector for the calling task is stored in the TSS of the called task to provide a link back to the calling task.

For all IA-32 processors, tasks are not recursive. A task cannot call or jump to itself.

Interrupts and exceptions can be handled with a task switch to a handler task. Here, the processor not only can perform a task switch to handle the interrupt or exception, but it can automatically switch back to the interrupted task upon returning from the interrupt- or exception-handler task. This mechanism can handle interrupts that occur during interrupt tasks.

As part of a task switch, the processor can also switch to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. The page-directory base register (CR3) also is reloaded on a task switch, allowing each task to have its own set of page tables. These protection facilities help isolate tasks and prevent them from interfering with one another. If one or both of these protection mechanisms are not used, the processor provides no protection between tasks. This is true even with operating systems that use multiple privilege levels for protection. Here, a task running at privilege level 3 that uses the same LDT and page tables as other privilege-level-3 tasks can access code and corrupt data and the stack of other tasks.

Use of task management facilities for handling multitasking applications is optional. Multitasking can be handled in software, with each software defined task executed in the context of a single IA-32 architecture task.

6.2. TASK MANAGEMENT DATA STRUCTURES

The processor defines five data structures for handling task-related activities:

Task-state segment (TSS).

Task-gate descriptor.

TSS descriptor.

Task register.

NT flag in the EFLAGS register.

When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

6.2.1. Task-State Segment (TSS)

The processor state information needed to restore a task is saved in a system segment called the task-state segment (TSS). Figure 6-2 shows the format of a TSS for tasks designed for 32-bit CPUs. (Compatibility with 16-bit Intel 286 processor tasks is provided by a different kind of TSS, see Figure 6-9.) The fields of a TSS are divided into two main categories: dynamic fields and static fields.

The processor updates the dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

General-purpose register fields
State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.

Segment selector fields
Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.

EFLAGS register field
State of the EFAGS register prior to the task switch.

EIP (instruction pointer) field
State of the EIP register prior to the task switch.

Previous task link field
Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field

(which is sometimes called the back link field) permits a task switch back to the previous task to be initiated with an IRET instruction.

The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

LDT segment selector field
Contains the segment selector for the task's LDT.

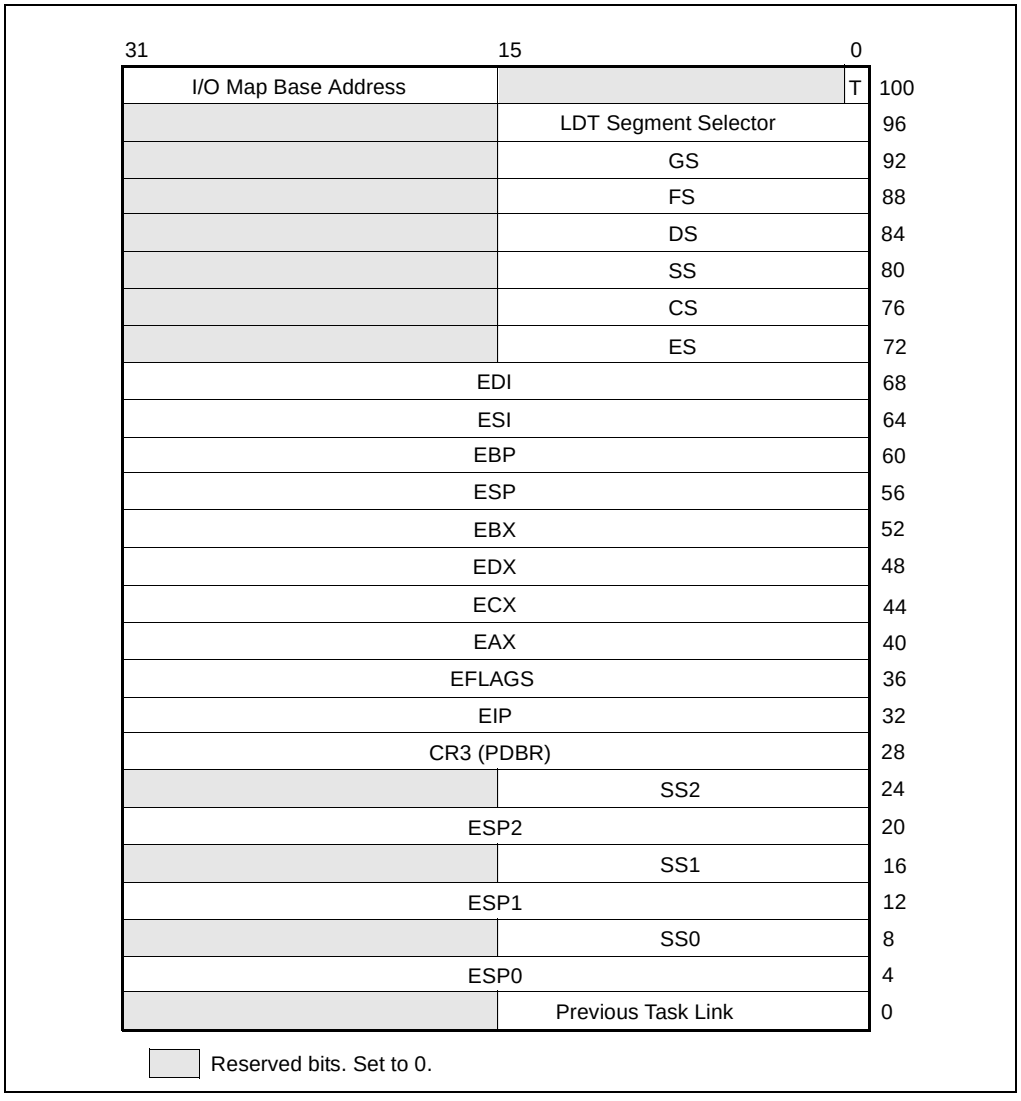

Figure 6-2. 32-Bit Task-State Segment (TSS)

CR3 control register field
Contains the base physical address of the page directory to be used by the task. Control register CR3 is also known as the page-directory base register (PDBR).

Privilege level-0, -1, and -2 stack pointer fields
These stack pointers consist of a logical address made up of the segment selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack (ESP0, ESP1, and ESP2). Note that the values in these fields are static for a particular task; whereas, the SS and ESP values will change if stack switching occurs within the task.

T (debug trap) flag (byte 100, bit 0)
When set, the T flag causes the processor to raise a debug exception when a task switch to this task occurs (see Section 14.3.1.5., "Task-Switch Exception Condition").

I/O map base address field
Contains a 16-bit offset from the base of the TSS to the I/O permission bit map and interrupt redirection bitmap. When present, these maps are stored in the TSS at higher addresses. The I/O map base address points to the beginning of the I/O permission bit map and the end of the interrupt redirection bit map. See Chapter 12, *Input/Output*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information about the I/O permission bit map. See Section 15.3., "Interrupt and Exception Handling in Virtual-8086 Mode", for a detailed description of the interrupt redirection bit map.

If paging is used, care should be taken to avoid placing a page boundary within the part of the TSS that the processor reads during a task switch (the first 104 bytes). If a page boundary is placed within this part of the TSS, the pages on either side of the boundary must be present at the same time and contiguous in physical memory. The reason for this restriction is that when accessing a TSS during a task switch, the processor reads and writes into the first 104 bytes of each TSS from contiguous physical addresses beginning with the physical address of the first byte of the TSS. It may not perform address translations at a page boundary if one occurs within this area. So, after the TSS access begins, if a part of the 104 bytes is not both present and physically contiguous, the processor will access incorrect TSS information, without generating a page-fault exception. The reading of this incorrect information will generally lead to an unrecoverable exception later in the task switch process.

Also, if paging is used, the pages corresponding to the previous task's TSS, the current task's TSS, and the descriptor table entries for each should be marked as read/write. The task switch will be carried out faster if the pages containing these structures are also present in memory before the task switch is initiated.

6.2.2. TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 6-3 shows the format of a TSS descriptor. TSS descriptors may only be placed in the GDT; they cannot be placed in an LDT or the IDT. An attempt to access a TSS using a segment selector with its TI flag set (which indicates the current LDT) causes a general-protection exception (#GP) to be

generated. A general-protection exception is also generated if an attempt is made to load a segment selector for a TSS into a segment register.

The busy flag (B) in the type field indicates whether the task is busy. A busy task is currently running or is suspended. A type field with a value of 1001B indicates an inactive task; a value of 1011B indicates a busy task. Tasks are not recursive. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

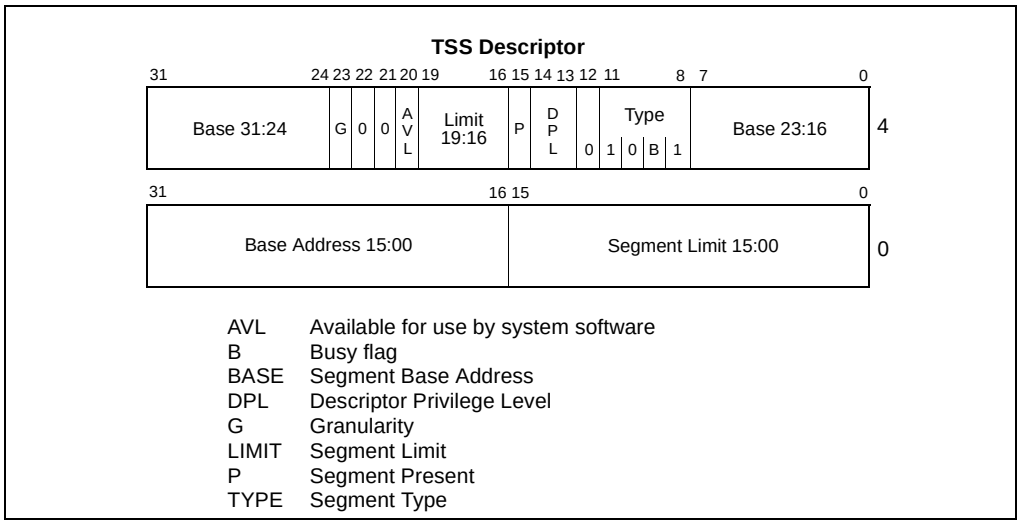

Figure 6-3. TSS Descriptor

The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data-segment descriptors (see Section 3.4.3., “Segment Descriptors”). The limit field must have a value equal to or greater than 67H (for a 32-bit TSS), one byte less than the minimum size of a TSS. Attempting to switch to a task whose TSS descriptor has a limit less than 67H generates an invalid-TSS exception (#TS). A larger limit is required if an I/O permission bit map is included in the TSS. An even larger limit would be required if the operating system stores additional data in the TSS. The processor does not check for a limit greater than 67H on a task switch; however, it does when accessing the I/O permission bit map or interrupt redirection bit map.

Any program or procedure with access to a TSS descriptor (that is, whose CPL is numerically equal to or less than the DPL of the TSS descriptor) can dispatch the task with a call or a jump. In most systems, the DPLs of TSS descriptors should be set to values less than 3, so that only privileged software can perform task switching. However, in multitasking applications, DPLs for some TSS descriptors can be set to 3 to allow task switching at the application (or user) privilege level.

6.2.3. Task Register

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address, 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-4). This information is copied from the TSS descriptor in the GDT for the current task. Figure 6-4 shows the path the processor uses to accesses the TSS, using the information in the task register.

The task register has both a visible part (that can be read and changed by software) and an invisible part (that is maintained by the processor and is inaccessible by software). The segment selector in the visible portion points to a TSS descriptor in the GDT. The processor uses the invisible portion of the task register to cache the segment descriptor for the TSS. Caching these values in a register makes execution of the task more efficient, because the processor does not need to fetch these values from memory to reference the TSS of the current task.

The LTR (load task register) and STR (store task register) instructions load and read the visible portion of the task register. The LTR instruction loads a segment selector (source operand) into the task register that points to a TSS descriptor in the GDT, and then loads the invisible portion of the task register with information from the TSS descriptor. This instruction is a privileged instruction that may be executed only when the CPL is 0. The LTR instruction generally is used during system initialization to put an initial value in the task register. Afterwards, the contents of the task register are changed implicitly when a task switch occurs.

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level, to identify the currently running task; however, it is normally used only by operating system software.

On power up or reset of the processor, the segment selector and base address are set to the default value of 0 and the limit is set to FFFFH.

6.2.4. Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task. Figure 6-5 shows the format of a task-gate descriptor. A task-gate descriptor can be placed in the GDT, an LDT, or the IDT.

The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. The RPL in this segment selector is not used.

The DPL of a task-gate descriptor controls access to the TSS descriptor during a task switch. When a program or procedure makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. (Note that when a task gate is used, the DPL of the destination TSS descriptor is not used.)

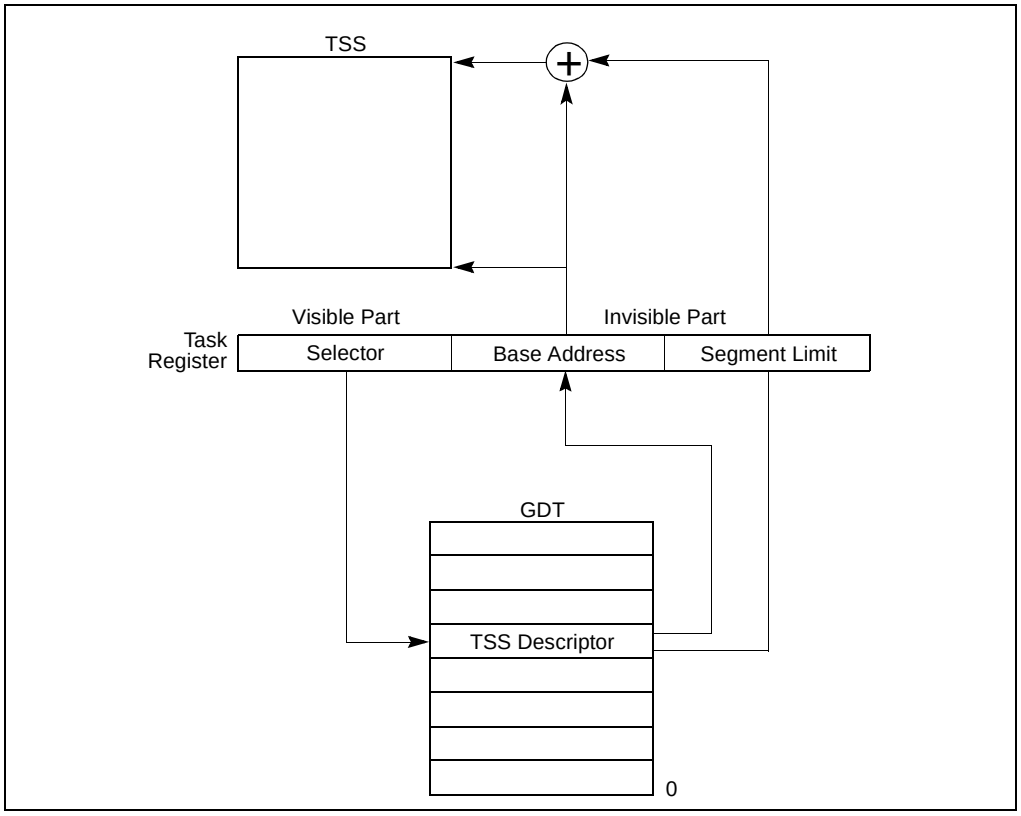

Figure 6-4. Task Register

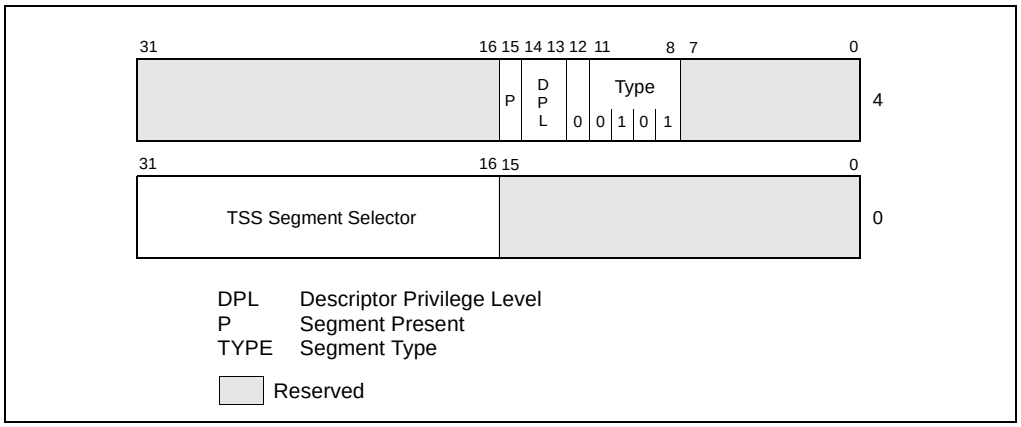

Figure 6-5. Task-Gate Descriptor

A task can be accessed either through a task-gate descriptor or a TSS descriptor. Both of these structures are provided to satisfy the following needs:

The need for a task to have only one busy flag. Because the busy flag for a task is stored in the TSS descriptor, each task should have only one TSS descriptor. There may, however, be several task gates that reference the same TSS descriptor.

The need to provide selective access to tasks. Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor's DPL. A program or procedure that does not have sufficient privilege to access the TSS descriptor for a task in the GDT (which usually has a DPL of 0) may be allowed access to the task through a task gate with a higher DPL. Task gates give the operating system greater latitude for limiting access to specific tasks.

The need for an interrupt or exception to be handled by an independent task. Task gates may also reside in the IDT, which allows interrupts and exceptions to be handled by handler tasks. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

Figure 6-6 illustrates how a task gate in an LDT, a task gate in the GDT, and a task gate in the IDT can all point to the same task.

6.3. TASK SWITCHING

The processor transfers execution to another task in any of four cases:

The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.

The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.

An interrupt or exception vector points to a task-gate descriptor in the IDT.

The current task executes an IRET when the NT flag in the EFLAGS register is set.

The JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all generalized mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).

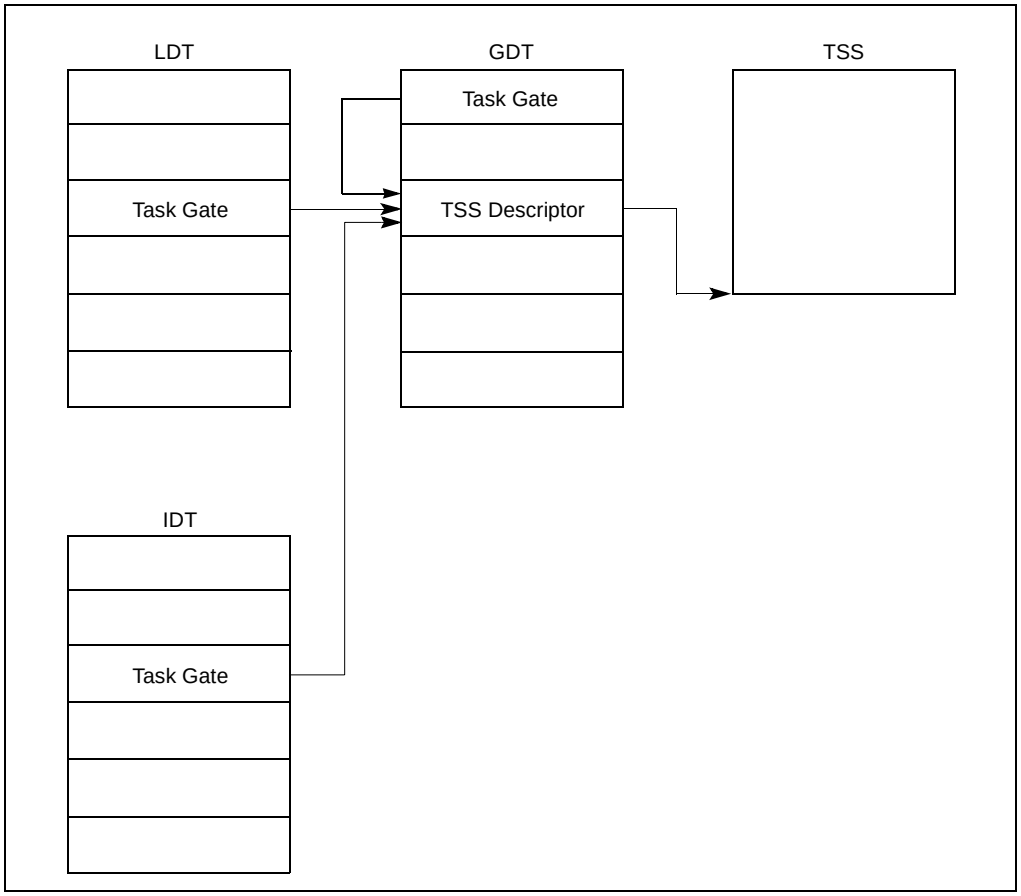

Figure 6-6. Task Gates Referencing the Same Task

2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for interrupts generated by the INT *n* instruction), and the IRET instruction are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT *n* instruction, the DPL is checked.
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).

5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt, the busy (B) flag is left set. (See Table 6-2.)
7. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
8. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).
9. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor sets the NT flag in the EFLAGS image stored in the new task's TSS; if initiated with an IRET instruction, the processor restores the NT flag from the EFLAGS image stored on the stack. If initiated with a JMP instruction, the NT flag is left unchanged. (See Table 6-2.)
10. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
11. Sets the TS flag in the control register CR0 image stored in the new task's TSS.
12. Loads the task register with the segment selector and descriptor for the new task's TSS.

NOTE

At this point, if all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 12, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch. If an unrecoverable error occurs after the commit point (in steps 13 and 14), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task. If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 5, "Interrupt 10—Invalid TSS Exception (#TS)", for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

13. Loads the new task's state from its TSS into processor. Any errors associated with the loading and qualification of segment descriptors in this step occur in the context of the new task. The task state information that is loaded here includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment descriptor parts of the segment registers.
14. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

Table 6-1 shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other IA-32 processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

Table 6-1. Exception Conditions Checked During a Task Switch

Condition Checked	Exception[1]	Error Code Reference[2]
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP	New Task's TSS
TSS descriptor is present in memory.	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#TS (for IRET)	New Task's TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS	New Task's TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid [3].	#TS	New Task's LDT
Code segment DPL matches segment selector RPL.	#TS	New Code Segment
SS segment selector is valid [2].	#TS	New Stack Segment
Stack segment is present in memory.	#SF	New Stack Segment

Table 6-1. Exception Conditions Checked During a Task Switch (Contd.)

Stack segment DPL matches CPL.	#TS	New stack segment
LDT of new task is present in memory.	#TS	New Task's LDT
CS segment selector is valid [3].	#TS	New Code Segment
Code segment is present in memory.	#NP	New Code Segment
Stack segment DPL matches selector RPL.	#TS	New Stack Segment
DS, ES, FS, and GS segment selectors are valid [3].	#TS	New Data Segment
DS, ES, FS, and GS segments are readable.	#TS	New Data Segment
DS, ES, FS, and GS segments are present in memory.	#NP	New Data Segment
DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).	#TS	New Data Segment

NOTES:

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SF is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).

The TS (task switched) flag in the control register CR0 is set every time a task switch occurs. System software uses the TS flag to coordinate the actions of floating-point unit when generating floating-point exceptions with the rest of the processor. The TS flag indicates that the context of the floating-point unit may be different from that of the current task. See Section 2.5., "Control Registers", for a detailed description of the function and use of the TS flag.

6.4. TASK LINKING

The previous task link field of the TSS (sometimes called the "backlink") and the NT flag in the EFLAGS register are used to return execution to the previous task. The NT flag indicates whether the currently executing task is nested within the execution of another task, and the previous task link field of the current task's TSS holds the TSS selector for the higher-level task in the nesting hierarchy, if there is one (see Figure 6-7).

When a CALL instruction, an interrupt, or an exception causes a task switch, the processor copies the segment selector for the current TSS into the previous task link field of the TSS for the new task, and then sets the NT flag in the EFLAGS register. The NT flag indicates that the previous task link field of the TSS has been loaded with a saved TSS segment selector. If software uses an IRET instruction to suspend the new task, the processor uses the value in the previous task link field and the NT flag to return to the previous task; that is, if the NT flag is set, the processor performs a task switch to the task specified in the previous task link field.

NOTE

When a JMP instruction causes a task switch, the new task is not nested; that is, the NT flag is set to 0 and the previous task link field is not used. A JMP instruction is used to dispatch a new task when nesting is not desired.

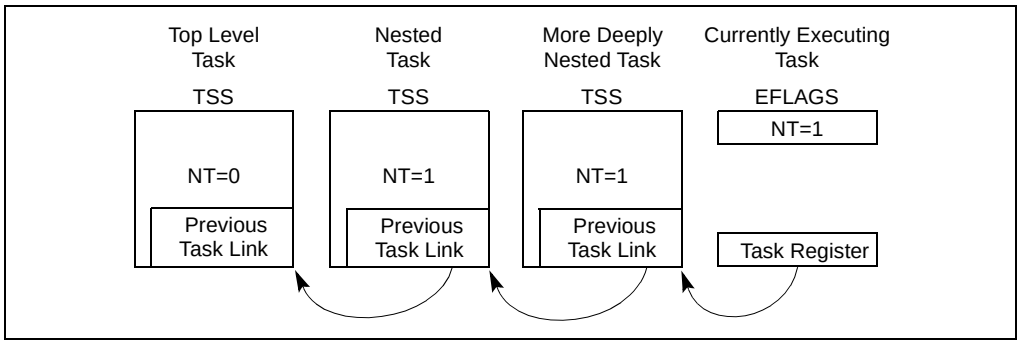

Figure 6-7. Nested Tasks

Table 6-2 summarizes the uses of the busy flag (in the TSS segment descriptor), the NT flag, the previous task link field, and TS flag (in control register CR0) during a task switch. Note that the NT flag may be modified by software executing at any privilege level. It is possible for a program to set its NT flag and execute an IRET instruction, which would have the effect of invoking the task specified in the previous link field of the current task's TSS. To keep spurious task switches from succeeding, the operating system should initialize the previous task link field for every TSS it creates to 0.

Table 6-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag

Flag or Field	Effect of JMP instruction	Effect of CALL Instruction or Interrupt	Effect of IRET Instruction
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	No change.	Flag is set.	Restored to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS.	No change.
Previous task link field of old task.	No change.	No change.	No change.
TS flag in control register CR0.	Flag is set.	Flag is set.	Flag is set.

6.4.1. Use of Busy Flag To Prevent Recursive Task Switching

A TSS allows only one context to be saved for a task; therefore, once a task is called (dispatched), a recursive (or re-entrant) call to the task would cause the current state of the task to be lost. The busy flag in the TSS segment descriptor is provided to prevent re-entrant task switching and subsequent loss of task state information. The processor manages the busy flag as follows:

1. When dispatching a task, the processor sets the busy flag of the new task.
2. If during a task switch, the current task is placed in a nested chain (the task switch is being generated by a CALL instruction, an interrupt, or an exception), the busy flag for the current task remains set.
3. When switching to the new task (initiated by a CALL instruction, interrupt, or exception), the processor generates a general-protection exception (#GP) if the busy flag of the new task is already set. (If the task switch is initiated with an IRET instruction, the exception is not raised because the processor expects the busy flag to be set.)
4. When a task is terminated by a jump to a new task (initiated with a JMP instruction in the task code) or by an IRET instruction in the task code, the processor clears the busy flag, returning the task to the “not busy” state.

In this manner the processor prevents recursive task switching by preventing a task from switching to itself or to any task in a nested chain of tasks. The chain of nested suspended tasks may grow to any length, due to multiple calls, interrupts, or exceptions. The busy flag prevents a task from being invoked if it is in this chain.

The busy flag may be used in multiprocessor configurations, because the processor follows a LOCK protocol (on the bus or in the cache) when it sets or clears the busy flag. This lock keeps two processors from invoking the same task at the same time. (See Section 7.1.2.1., “Automatic Locking”, for more information about setting the busy flag in a multiprocessor applications.)

6.4.2. Modifying Task Linkages

In a uniprocessor system, in situations where it is necessary to remove a task from a chain of linked tasks, use the following procedure to remove the task:

1. Disable interrupts.
2. Change the previous task link field in the TSS of the pre-empting task (the task that suspended the task to be removed). It is assumed that the pre-empting task is the next task (newer task) in the chain from the task to be removed. Change the previous task link field to point to the TSS of the next oldest task in the chain or to an even older task in the chain.
3. Clear the busy (B) flag in the TSS segment descriptor for the task being removed from the chain. If more than one task is being removed from the chain, the busy flag for each task being remove must be cleared.
4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to insure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

6.5. TASK ADDRESS SPACE

The address space for a task consists of the segments that the task can access. These segments include the code, data, stack, and system segments referenced in the TSS and any other segments accessed by the task code. These segments are mapped into the processor's linear address space, which is in turn mapped into the processor's physical address space (either directly or through paging).

The LDT segment field in the TSS can be used to give each task its own LDT. Giving a task its own LDT allows the task address space to be isolated from other tasks by placing the segment descriptors for all the segments associated with the task in the task's LDT.

It also is possible for several tasks to use the same LDT. This is a simple and memory-efficient way to allow some tasks to communicate with or control each other, without dropping the protection barriers for the entire system.

Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

If paging is enabled, the CR3 register (PDBR) field in the TSS allows each task can also have its own set of page tables for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

6.5.1. Mapping Tasks to the Linear and Physical Address Spaces

Tasks can be mapped to the linear address space and physical address space in either of two ways:

One linear-to-physical address space mapping is shared among all tasks. When paging is not enabled, this is the only choice. Without paging, all linear addresses map to the same physical addresses. When paging is enabled, this form of linear-to-physical address space mapping is obtained by using one page directory for all tasks. The linear address space may exceed the available physical space if demand-paged virtual memory is supported.

Each task has its own linear address space that is mapped to the physical address space. This form of mapping is accomplished by using a different page directory for each task. Because the PDBR (control register CR3) is loaded on each task switch, each task may have a different page directory.

The linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

With either method of mapping task linear address spaces, the TSSs for all tasks must lie in a shared area of the physical space, which is accessible to all tasks. This mapping is required so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear address space mapped by the GDT also should be mapped to a shared area of the physical space; otherwise, the purpose of the GDT is defeated. Figure 6-8 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

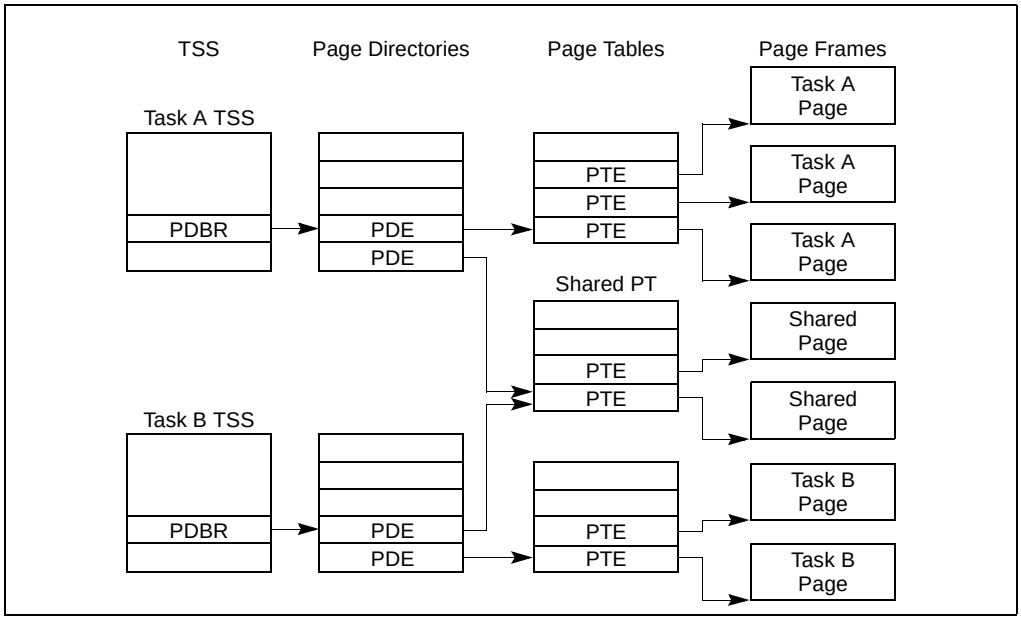

Figure 6-8. Overlapping Linear-to-Physical Mappings

6.5.2. Task Logical Address Space

To allow the sharing of data among tasks, use any of the following techniques to create shared logical-to-physical address-space mappings for data segments:

Through the segment descriptors in the GDT. All tasks must have access to the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear-address space that are mapped into an area of the physical-address space common to all tasks, then all tasks can share the data and code in those segments.

Through a shared LDT. Two or more tasks can use the same LDT if the LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments that are mapped to a common area of the physical address space, the data and code in those segments can be shared among the tasks that share the LDT. This method of sharing is more selective than sharing through the GDT, because the sharing can be limited

to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared segments.

Through segment descriptors in distinct LDTs that are mapped to common addresses in the linear address space. If this common area of the linear address space is mapped to the same area of the physical address space for each task, these segment descriptors permit the tasks to share segments. Such segment descriptors are commonly called aliases. This method of sharing is even more selective than those listed above, because, other segment descriptors in the LDTs may point to independent linear addresses which are not shared.

6.6. 16-BIT TASK-STATE SEGMENT (TSS)

The 32-bit IA-32 processors also recognize a 16-bit TSS format like the one used in Intel 286 processors (see Figure 6-9). It is supported for compatibility with software written to run on these earlier IA-32 processors.

The following additional information is important to know about the 16-bit TSS.

Do not use a 16-bit TSS to implement a virtual-8086 task.

The valid segment limit for a 16-bit TSS is 2CH.

The 16-bit TSS does not contain a field for the base address of the page directory, which is loaded into control register CR3. Therefore, a separate set of page tables for each task is not supported for 16-bit tasks. If a 16-bit task is dispatched, the page-table structure for the previous task is used.

The I/O base address is not included in the 16-bit TSS, so none of the functions of the I/O map are supported.

When task state is saved in a 16-bit TSS, the upper 16 bits of the EFLAGS register and the EIP register are lost.

When the general-purpose registers are loaded or saved from a 16-bit TSS, the upper 16 bits of the registers are modified and not maintained.

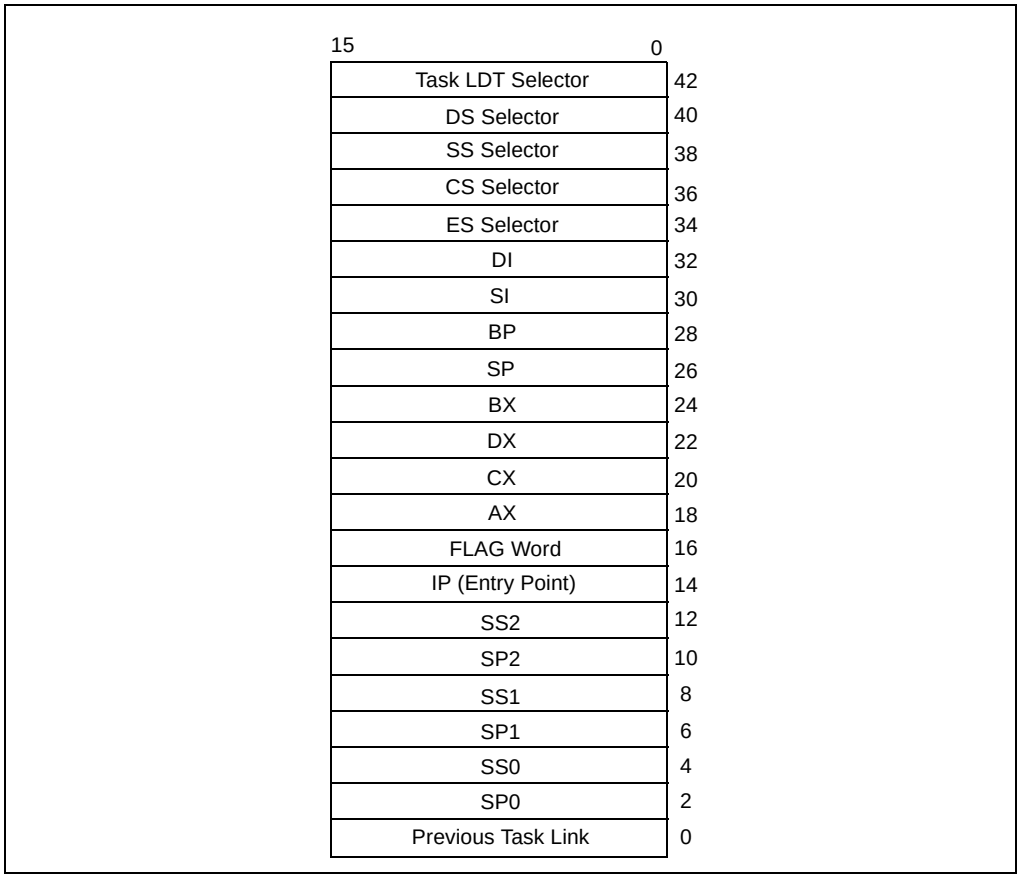

Figure 6-9. 16-Bit TSS Format

7

Multiple-Processor Management