Intel x86 Software developers - Segment Descriptor types

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT). These tables contain entries called **segment descriptors**.

Segment descriptors provide the base address of segments well as access rights, type, and usage information.

Each segment has a segment descriptor and each segment descriptor has an associated **segment selector**. A segment selector provides the software that uses it with

- an index into the GDT or LDT (the offset of its associated segment descriptor),
- a global/local flag (determines whether the selector points to the GDT or the LDT), and
- access rights information.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a 16 bit segment selector and a 32-bit offset. The offset part of the logical address is added to the base address for the segment to locate a byte within the segment.

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor's address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

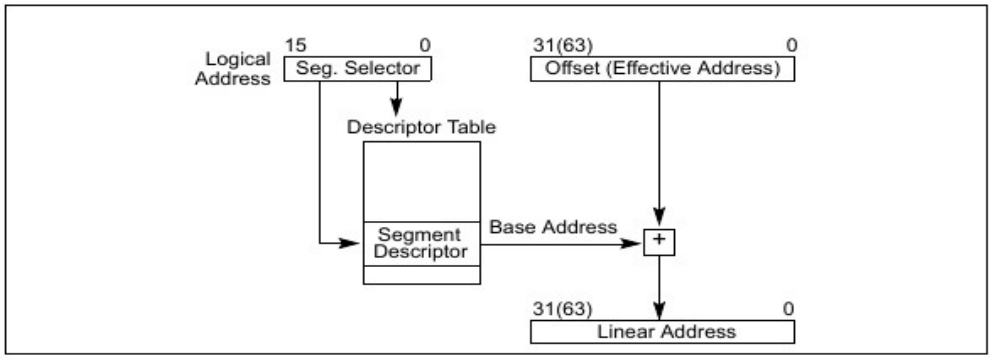


Figure 3-5. Logical Address to Linear Address Translation

Segment Selectors

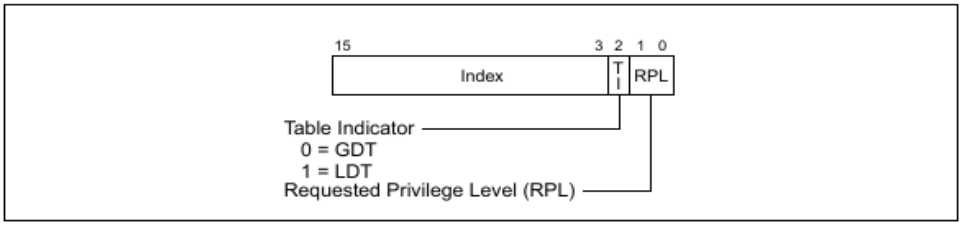
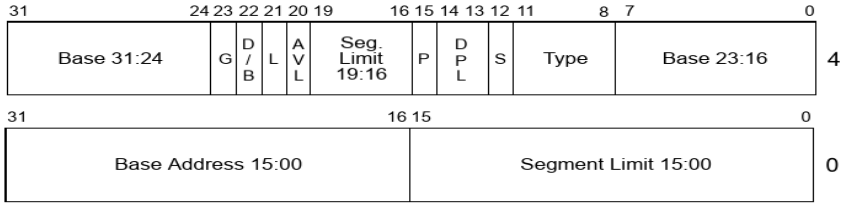


Figure 3-6. Segment Selector

Index	Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).
TI (table indicator)	0 – Selects GDT 1 - LDT
RPL	Specifies the privilege level of the selector.

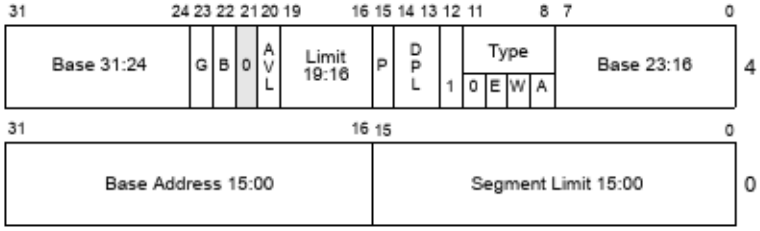
Segment Descriptor



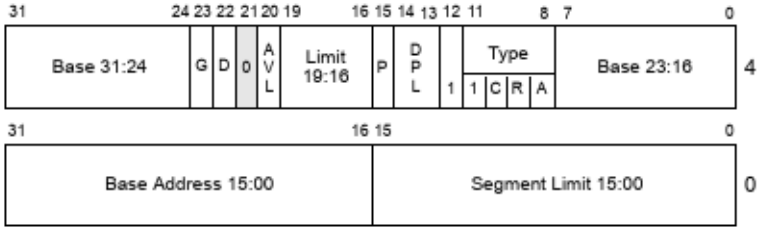
L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

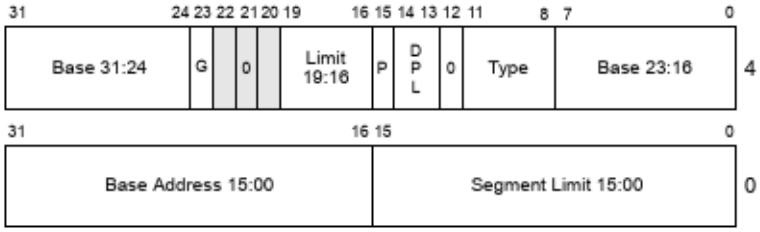
Data-Segment Descriptor



Code-Segment Descriptor



System-Segment Descriptor



A	Accessed	E	Expansion Direction
AVL	Available to Sys. Programmers	G	Granularity
B	Big	R	Readable
C	Conforming	LIMIT	Segment Limit
D	Default	W	Writable
DPL	Descriptor Privilege Level	P	Present

Reserved

<table>
<tr><td>Segment limit</td><td>Specifies the size of the segment.
G = 0 Interpreted in byte units. The segment size range from 1 byte to 1 MBytes.
G = 1 Interpreted in 4KB units. The segment size range from 4 KByte to 4 GBytes.

Expand Up segments — Offset in a logical address range from 0 to segment limit.

Offsets greater than the segment limit generate general-protection exceptions
Expand Down — Offset in a logical address range from segment limit + 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag.

Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions.</td></tr>
<tr><td>Base address</td><td>Defines the location of byte 0 of the segment within the 4 Gbyte linear address space.

Should be aligned to 16 byte boundary. This is not required, but alignment allows programs to maximize performance.</td></tr>
<tr><td>Type</td><td>S = 0 System segment. See Below
S = 1
Data Segment
11 10 9 8
0 E W A
A Accessed
W Write Enabled. 0 – Readonly 1 – Read/Write
E Expansion direction 1 – Expand Down 0 – Expand Up
Code Segment
11 10 9 8
1 C R A
A Accessed
R Read Enabled. 0 – Execute-Only 1 – Execute/Read
C 0 – Non-Conforming 1 – Conforming

A (Accessed) bit in a segment Descriptor: The processor sets this bit whenever it loads a segment selector for the segment into a segment register.

The bit remains set until explicitly cleared. This bit can be used both for virtual memory management and for debugging.

If the segment descriptors in the GDT or an LDT are placed in ROM, the processor can enter an indefinite loop, because it will not be able to set the Accessed bit.

To prevent this: Set the accessed bits for all segment descriptors placed in a ROM.</td></tr>
<tr><td>S</td><td>0 – System segment
1 – Code or Data Segment</td></tr>
</table>

DPL	Specifies the privilege level of the segment. The DPL is used to control access to the segment.
P	**Segment-Preset flag** 0 – Indicated if the segment is not present in memory. Processor generates a segment-not-present exception, when loading segment register with segment selector. 1 – Segment is present in memory.
D/B	**Default operation size/ default stack pointer size and upper bound flag.** **Executable code segment (D flag)** It indicates the default length for effective addresses and operands referenced by instructions in the segment. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default. 0 — 16 bit address 16 or 8 bit operand 1 — 32 bit address 32 or 8 bit operand **Stack segment (B (big) flag) (Expand-Up)** It specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls) 0 — 16 bit stack pointer stored in 16 bit SP register. 1 — 32 bit stack pointer is used, which is stored in 32 bit ESP register. **Stack segment (B (big) flag) (Expand down)** It specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the stack segment is set up to be an expand-down data segment the B flag also specifies the upper bound of the stack segment. 0 — 16 bit stack pointer stored in 16 bit SP register. The upper bound is FFFFH (64 KBytes) 1 — 32 bit stack pointer is used, which is stored in 32 bit ESP register. The upper bound is FFFFFFFFH (4 Gbytes).
G	0 – Segment limit is interpreted in byte units. 1 – segment limit is interpreted in 4-KB units.
L	1 - instructions in this code segment are executed in 64-bit mode. D-bit must be cleared. 0 - nstructions in this code segment are executed in compatibility mode

Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					Description	
Decimal	11	10	9	8	32-Bit Mode	IA-32e Mode
0	0	0	0	0	Reserved	Reserved
1	0	0	0	1	16-bit TSS (Available)	Reserved
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-bit TSS (Busy)	Reserved
4	0	1	0	0	16-bit Call Gate	Reserved
5	0	1	0	1	Task Gate	Reserved
6	0	1	1	0	16-bit Interrupt Gate	Reserved
7	0	1	1	1	16-bit Trap Gate	Reserved
8	1	0	0	0	Reserved	Reserved
9	1	0	0	1	32-bit TSS (Available)	64-bit TSS (Available)
10	1	0	1	0	Reserved	Reserved
11	1	0	1	1	32-bit TSS (Busy)	64-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate	64-bit Call Gate
13	1	1	0	1	Reserved	Reserved
14	1	1	1	0	32-bit Interrupt Gate	64-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate	64-bit Trap Gate

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

Code Segment

1. Code segments can be execute-only or execute/read. An execute/read segment might be used when constants or other static data have been placed with instruction code in a ROM. **In protected mode, code segments are not writable.**
2. A transfer of execution into a more-privileged (lower PL number) ***conforming*** segment allows execution to continue at the current privilege level.
    - A transfer into a ***nonconforming*** segment at a different privilege level results in a general-protection exception, unless a call gate or task gate is used.
    - Handlers for some types of exceptions (such as, **divide error or overflow**) **may be loaded in conforming code segments**.
3. **Execution cannot be transferred** by a call or a jump **to a less-privileged code segment**, **regardless** of whether the target segment is a **conforming or non conforming code segment.**

Data segment

All data segments are **nonconforming** - meaning that they cannot be accessed by less privileged programs or procedures. Unlike nonconforming code segments, however, data segments can be accessed by more privileged programs or procedures without using a special access gate.

Intel x86 Software developers - GDT and Call Gates

GDT

1. No special segment type is needed for GDT. It can exist in the data segment. However LDT must be located in a system segment of the LDT type.
2. The GDT is a data structure in linear address space.

   (That is paging will determine the physical address.)
3. The base linear address and limit must be loadded into the GDTR register.

   (Which also thus holds linear address, that is paging will determine the physical address.).
4. The base address of the GDT should be aligned on an eight-byte boundary to yield the best processor performance.

Enable and Disable Segment protection

Enable	Disable
Setting the ***PE flag*** in register **CR0** causes the processor to switch to protected mode, which in turn enables the segment-protection mechanism.	Once in protected mode, there is no control bit for turning the protection mechanism on or off.

Enable and Disable page level protection

Enable	Disable
Page-level protection is automatically enabled when paging is enabled. Paging is enabled by setting the ***PG flag*** in register ***CR0***.	There is no mode bit for turning off page-level protection once paging is enabled. However, page-level protection can be disabled by: 1. Clear the ***WP flag*** in control register **CR0**. 2. Set the **read/write (R/W)** and **user/supervisor (U/S) flags** for each page-directory and page-table entry. This action makes each page a writable, user page, which in effect disables page-level protection.

Segment level protection

Layout: Segment Descriptor

Limit checking

	Effective limit	Max Effective limit range
G = 0	limit field (20 bits)	0 to F_FFFF (1 MB)
G = 1	limit field (20 bits) * 2^12	FFF (4 KB) to FFFF_FFFF (1 GB) *0 to FFE is also valid as the lower 12 bits are not checked.*

Expand-up:
Valid address space: 0 to Effective limit

Expand-down:
Note: An expand-down segment has maximum size when the segment limit is 0.

	Valid address space
B = 0	Effective-limit + 1 to F_FFFF
B = 1	Effective-limit + 1 to FFFF_FFFF

Validate: In case of expand-down segments, if the effective-limit is determined by the G bit or is the raw bits in limits field.

Type Checking

Segment descriptors contain type information in two places:

- ***The S (descriptor type) flag***: Determines if a descriptor is a system or non-system segment.
- ***The type field*** : Determines if the segment is code/data segment and whether write/read is allowed.

Type is checking is done at various times.

1. **When a segment selector is loaded into a segment register**
   - The CS register only can be loaded with a selector for a code segment.
   - System segments cannot be loadedd into data-segment registers (DS, ES, FS and GS).
   - Not readable code segments cannot be loadedd into data-segment registers (DS, ES, FS and GS).
   - Segment selectors of writable data segments can be loaded int othe SS register.
2. **When a segment selector is loaded into the LDTR or task register**
   - LDTR can only be loaded with a selector for an LDT.
   - Task register can only be loaded with a segment selector for a TSS.
3. **When instructions access segments whose descriptors are already loaded into segment registers**
   - No instruction may write into an executable segment.
   - No instruction may write into a data segment if it is not writable.
   - No instruction may read an executable segment unless the readable flag is set.

4. **When an instruction operand contains a segment selector.**
   - A far CALL or far JMP instruction can only access a segment descriptor for
     - a conforming code segment,
     - a nonconforming code segment,
     - a call gate, task gate, or TSS.
   - LLDT instruction must reference a segment descriptor for an LDT.
   - LTR instruction must reference a segment descriptor for a TSS.
   - LAR instruction must reference a segment or gate descriptor for an LDT, TSS, call gate, task gate, code segment, or data segment.
   - LSL instruction must reference a segment descriptor for a LDT, TSS, code segment, or data segment.
   - IDT entries must be interrupt, trap, or task gates.

5. **During certain internal operations**
   On a far call or far jump the processor determines the type of control transfer to be carried out by checking the type field in the segment (or gate) descriptor pointed to by the segment (or gate) selector given as an operand in the CALL or JMP instruction.
     - If the descriptor type is for a code segment or call gate, then a call or jump to another code segment is indicated.
     - If the descriptor type is for a TSS or task gate, a task switch is indicated.
   - **On a call or jump through a call gate**: The processor checks that the segment descriptor pointed to by the gate is for a code segment.
   - **On a call or jump through a task gate**: The processor checks that the segment descriptor pointed to by the task gate is for a TSS.
   - **On a call or jump to a new task by a direct reference to a TSS**: The processor automatically checks that the segment descriptor being pointed to by the CALL or JMP instruction is for a TSS.
   - **On return from a nested task**: The processor checks that the previous task link field in the current TSS points to a TSS.

Null Segment Selector Checking

1. CS and DD segment registers: GP fault is generated when loading a null segment selector.
2. DS, ES, GS, FS registers: A null segment selector can be loaded, attempt to access an address through these registers will cause a GP fault.

Privilege Levels

To carry out privilege-level checks between code segments and data segments, the processor recognizes the three types of privilege levels:

1. **Current privilare level (CPL):**
   - The CPL is the privilege level of the currently executing program or task.
   - It is stored in bits 0 and 1 of the CS and SS segment registers.
   - Normally, the CPL is equal to the privilege level of the code segment from which

instructions are being fetched. DPL is one of the fields in segment descriptor.

- However, the processor changes the CPL when program control is transferred to a nonconforming code segment with a different privilege level.
- Conforming code segments can be accessed, if

Conditon to access a conforming code segment	
CPL >= DPL of the conforming code segment.	DPL sets the numerically lowest privilage level.
For example, if the DPL of a code segment is 1, only programs running at a CPL of 1,2 and 3 can access the segment.	

- CPL is not changed when accessing conforming code segments. Since the CPL does not change, no stack switch occurs.

2. Descriptor privilate level (DPL):

- When the currently executing code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector.
- DPL is interpreted differently, depending on the type of segment or gate being accessed.

Data segment	
MAX(**CPL**,**RPL**) **<= DPL** of the conforming data segment.	DPL sets the numerically highest privilage level.
For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.	

Nonconforming code segment (without using a call gate)	
MAX(**CPL**,**RPL**) **= DPL** of the conforming data segment.	DPL sets the numerically exact privilage level.
For example, if the DPL of a nonconforming code segment is 0, only programs running at a CPL of 0 can access the segment.	

Call gate	
MAX(**CPL**,**RPL**) **<= DPL** of the conforming data segment.	DPL sets the numerically highest privilage level.
For example, if the DPL of a call gate is 1, only programs running at a CPL of 0 or 1 can be able to access the call gate.	

Conforming and nonconforming code segment accessed through a call gate	
MAX(**CPL**,**RPL**) **>= DPL** of the conforming code segment.	DPL sets the numerically lowest privilage level.
For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.	

TSS	
MAX(**CPL**,**RPL**) **<= DPL** of the conforming data segment.	DPL sets the numerically highest privilage level.
For example, if the DPL of a call gate is 1, only programs running at a CPL of 0 or 1 can be able to access the call gate.	

3. **Requested privilage level (RPL):**
   If the RPL of a segment selector is numerically greater than the CPL, the RPL overrides the CPL.
   The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment.

Privilege level checking when loading segment registers.

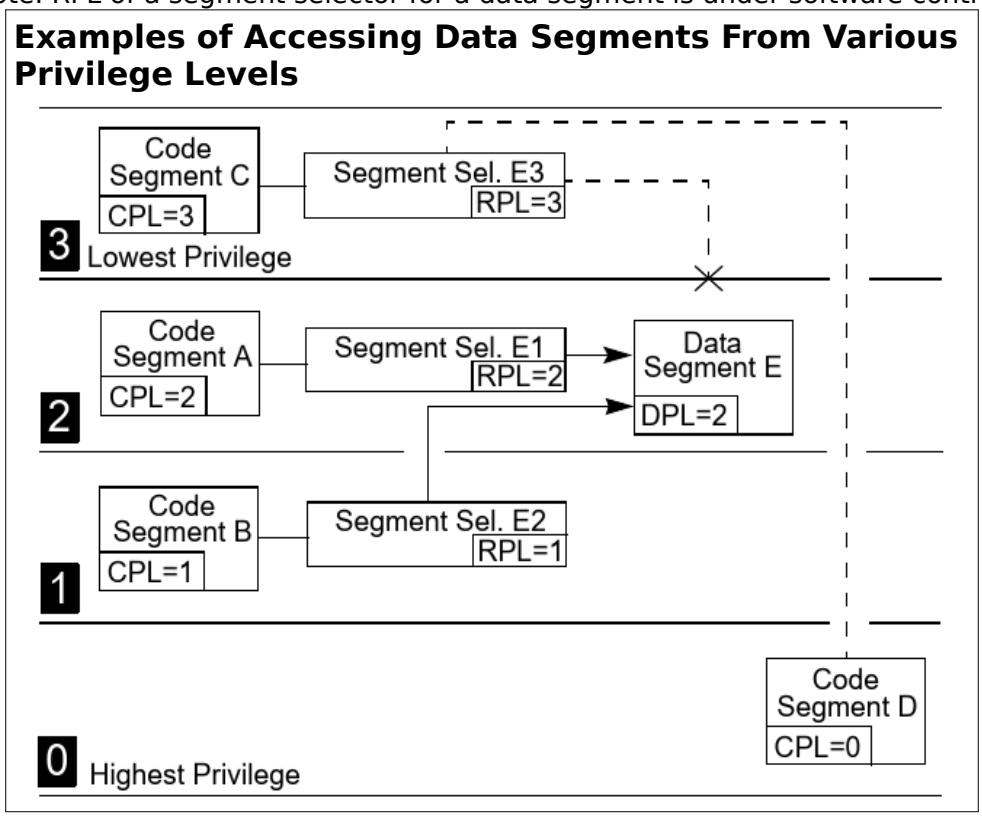
1. Before the processor loads a segmentt selector into a segment register, it performs a privilege check by comparing
   - The CPL of the currently running program or task,
   - The RPL of the segment selector and
   - The DPL of the segment's segment descriptor.

Before processor loads a segment selector into a segment register
DPL >= CPL and DPL >= RPL
If not met, GP fault is generated.

Privilege level checking when accessing data segments

Data segment (same as in page 6)	
MAX(CPL,RPL) <= DPL of the conforming data segment.	DPL sets the numerically highest privilage level.
For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.	

Note: RPL of a segment selector for a data segment is under software control.



Privilege level checking when Accessing Data in Code segments

- Load a data-segment register with a **segment selector for a nonconforming, readable, code segment**.
- *Privilege check of Data segment rules apply.*
- Load a data-segment register with a **segment selector for a conforming, readable, code segment**.
- *Is always valid because the privilege level of a conforming code segment is effectively the same as the CPL, regardless of the DPL*
- **Use a code-segment override prefix (CS)** to read a readable, code segment whose selector is already loaded in the CS register.
- *Is always valid because the DPL of the code segment selected by the CS register is the same as the CPL*

Privilege level checking when Loading the SS Register

Privilege check when loading the SS Register
CPL = RPL of the stack segment selector = DPL of the segment descriptor.
If not met, GP fault is generated.

Privilege level checking when Transferring Program Control Between Code Segments

A JMP or CALL instruction can reference another code segment in any of four ways:

- The **target operand contains the segment selector** for the target code segment.
- The **target operand points to a call-gate descriptor**, which **contains the segment selector for the target code segment**.
- The **target operand points to a TSS**, which **contains the segment selector for the target code segment**.
- The **target operand points to a task gate**, which **points to a TSS, which in turn contains the segment selector for the target code segment**.

1. The **SYSENTER** and **SYSEXIT** instructions are special instructions for making fast calls to and returns from operating system or executive procedures.
2. The **SYCALL** and **SYSRET** instructions are special instructions for making fast calls to and returns from operating system or executive procedures **in 64-bit mode**.

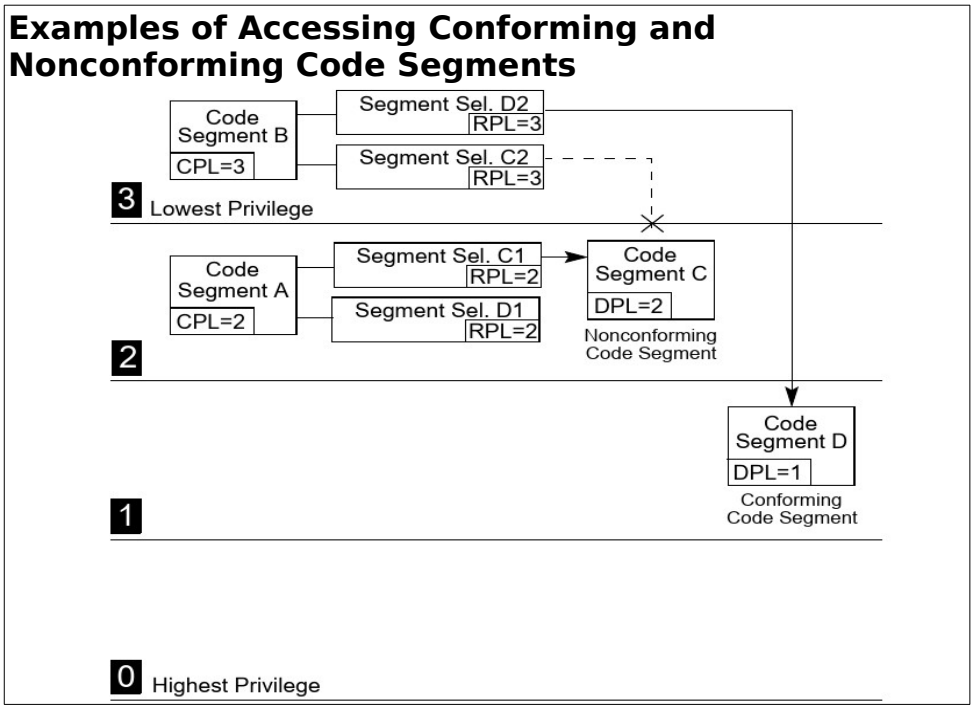
Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, so privilege-level checks are not performed.

Transferring program control without going through a call gate

1. **Accessing Nonconforming Code segments**

Privilege check when accessing Nonconforming code segments
CPL = DPL of the destination code segment descriptor and **RPL =< CPL** *RPL of the segment selector that points to t he nonconforming code segment.*
If not met, GP fault is generated.



2. **Accessing Conforming Code segments**

Privilege check when accessing Conforming code segments (same as page 5)
CPL >= DPL of the destination code segment descriptor RPLs are not checked when accessing conforming code segments
If not met, GP fault is generated.

Conforming Code Segment

- CPL is not changed. Only time CPL is not equal to DPL of the calling segment.
- Since CPL does not change, **no stack switch occurs**.
- Can be used for code modules like **math libraries** and **exception handlers** (which support application programs, but does not access protected system facilites). They are be part of the Operating System.
- Keeping the CPL at the level of a calling code segment, prevents an application from accessing other nonconforming code segments and access to more privileged data.

Transferring program control going through a Call Gate

Call gates

- Used to transfer control between privilege levels.
- Used to transfer control between 16-bit and 32-bit segments.
- Can reside in GDT, LDT but not IDT.

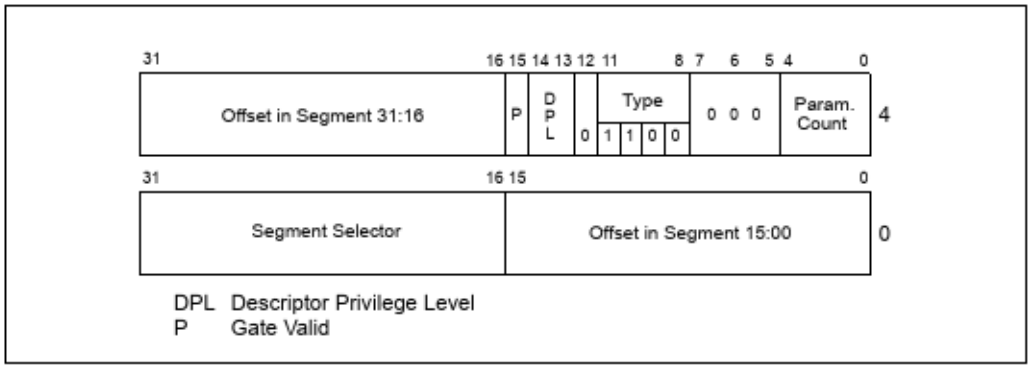


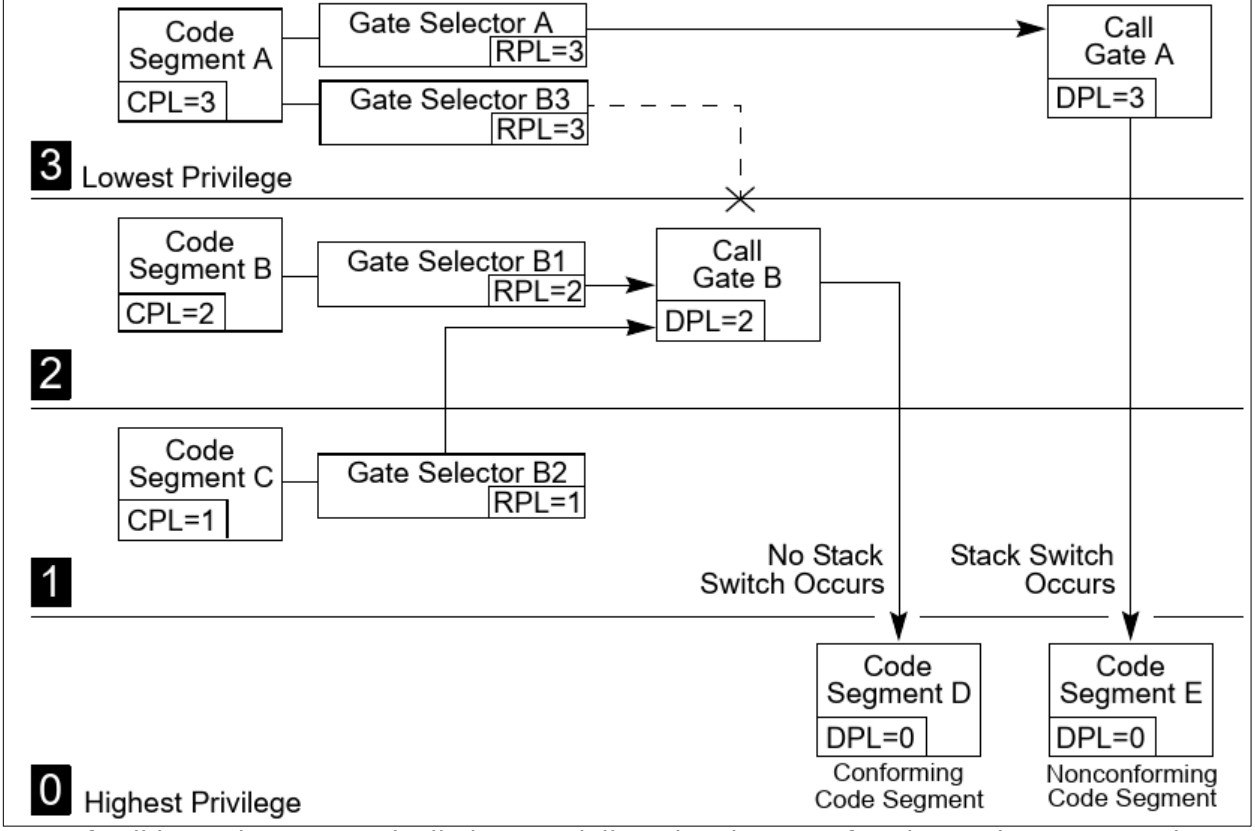
Figure 5-8. Call-Gate Descriptor

If a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.

- It defines the size of values to be pushed onto the target stack:
  - 16-bit gates force 16-bit pushes and
  - 32-bit gates force 32-bit pushes.
- It specifies whether the call-gate descriptor is valid.

Privilege check when accessing through Call Gate		
CALL	**MAX(CPL, RPL) <= DPL** of the call gate **CPL >= DPL** of destination conforming code segment. **CPL >= DPL** of destination nonconforming code segment.	DPL field of the call-gate descriptor specifies the numerically highest privilege level which can access the call gate
JMP	**MAX(CPL, RPL) <= DPL of the call gate** **CPL >= DPL** of destination conforming code segment. **CPL = DPL** of destination nonconforming code segment.	
If not met, GP fault is generated.		

Figure 1: Examples of Accessing Call Gates At Various Privilege Levels



- If call is made to numerically lower privilege level nonconforming code segment, then
  - CPL is lowered to the DPL.
  - Stack switch occurs.

Stack Switching

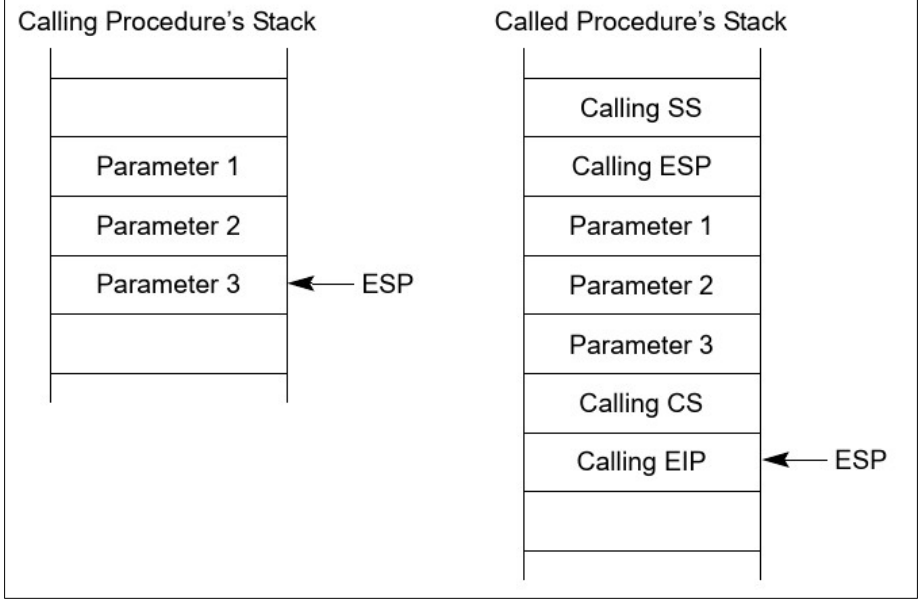
Whenever a call gate is used to transfer program control to a more privileged nonconforming code segment (DPL of the destination code segment is < CPL), the processor automatically, switches to the stack for the destination code segment's privilege level.

- Each task must define 4 stacks: One for Privilege level 3, 2, 1, 0.
- At minimum **there must be 2 stacks defined**, **one or Privilege level 3 and 0**.
- For the currently running task, thte pointers to the privilege level 0, 1, and 2 stacks are stored in the TSS.
- The initial values stored in the TSS are readonly and are only used to create new stack when stack switching occours. The stacks are disposed of when a return is made from the called procedure. The next time the procedure is called, a new stack is created using the initial stack pointers as set in the TSS.
- The pointer for the privilege level 3 is stored in SS and ESP registers. It is stored in the called procedure's stack when stack switch occours and is restored back to the above registers, when control is returned to the caller code at privilege level 3.

- If the operating system does not use the processor's multitasking mechanism, i**t still must create at least one TSS for this stack-related purpose**. (To be used when switchingg to privilege level 0 aka system calls or interrupt handlers)

1. Destination code segemnt --> DPL --> Segment selector and Stack pointer from TSS for that DPL.
2. Reads the Segment selector from GDT and performs
   a) Limit violation checks.
   b) Privilege checks
   c) Type checks.
3. Temporarily stores the current SS and ESP values.
4. Loads the SS and ESP as per TSS entry.
5. Pushes temporarily saved values for SS and ESP onto the new stack.
6. Copies the number of paramers specified in the parameter count of the call gate.
   If count is 0, no parameters are copied.
7. Push current CS and EIP registers.
8. Load segment selector and instruction pointer from call gate to SS and EIP.
9. Begins execution at the new SS:EIP location.

Figure: Stack Switching During an Interprivilege-level Call



Returning from a Called Procedure

- A near return only transfers program control within the current code segment.

Far return to a different privilege level.
CPL < DPL of the return code segment descriptor If the RPL from the return CS register is numerically greater than the CPL, a return across privilege levels occurs.
If not met, GP fault is generated.

1. Check RPL field of the saved CS register to determine if privilege level change is required.
2. Loads CS and EIP registers from the called procedures stack.
3. Increase ESP value based on the number of parameter count from the **called procesures stack**.
4. Load SS and EIP registers with the saved SS and EIP values from called procedures stack.
5. Increase ESP value based on the number of parameter count from the **calling procesures stack.**
6. **(If privilege level switch is required)** Checks the contents of the DS, ES, FS, and GS segment registers. **If any of these registers refer to segments whose DPL is less than the new CPL, the segment register is loaded with a null segment selector.**

Intel x86 Software developers - TSS

26 June 2021

- A task is a unit of work that a processor can dispatch, execute and suspend.
- When the processor is in Protected mode, all processor execution takes place from within a task. Every Operating system must design at least one task.

Task Structure

- A Task is made up of two parts
  - A task execution space
    - Consists of a code segment, a stack segment and one or more data segments.
    - The operating system must provide separate stack for each privilege level.
  - A task-state segment (TSS).
    - TSS specifies the segments that make up the task execution space and provides a storage place for task state information.
    - A task is identified by the segmentt selector for its TSS. So in systems with only one Task, there must be one TSS segment.
    - When a task is loaded for execution, the segment selector, base address, limit and segment descriptor attributes for the TSS is loaded into the Task Register TR.
    - If paging is implemented for the task, the base address of the page directory used by the task is loaded into the control register CR3.
- If protection mechanisms are not used, the processor provides no protection between tasks. It is upto the Operating System to setup protection mechanism for the tasks, either via segmentation (separate LDT) or through paging and virtual memory.

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.

- A Task is identified by a segment selector that points to a Task Gate or the TSS for the Task.
  - Using a CALL or JMP to dispatch a task: The selector in the instruction can point to the TSS directly or a Task Gate, thatt holds the selector for the TSS.
  - IDT entry for an interrupt or Exception handler must contain Task Gate only.
- For **all IA-32 processors**, tasks are not recursive. **A task cannot call or jump to itself.**
- **When tasks are used for Interrupt and exception handling**, the processor performs a task switch to handle the interrupt or exception and automatically switches back to the interrupted task upon returning from the interrupt-handler task

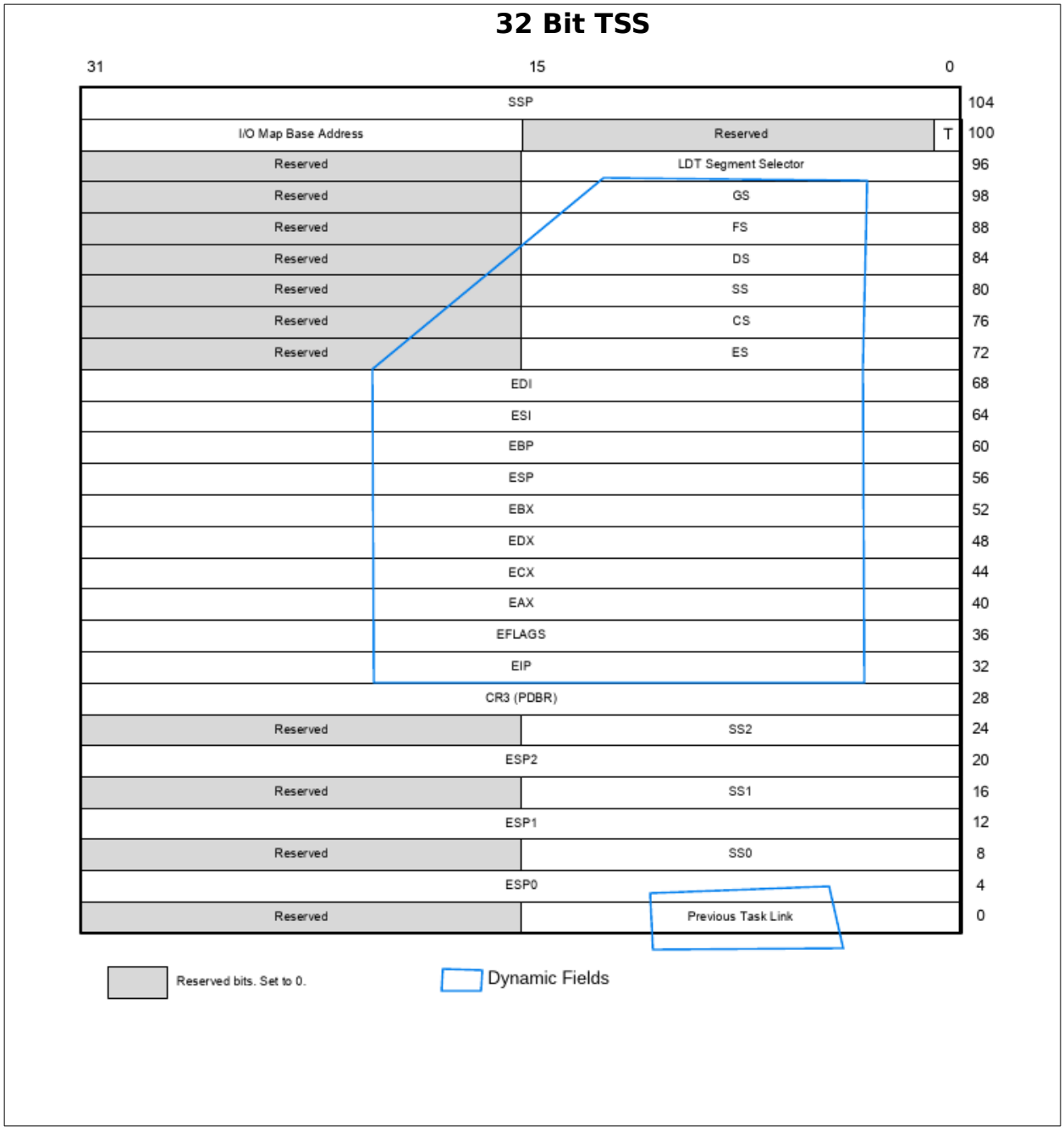
or exception-handler task.

- Both TSS descriptors and Task Gate reside in the GDT.
- 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.
- 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.
- **There are as many TSS segments descriptors in GDT as there are tasks. As e**ach task is identified by a TSS segment.

  When a Task switch happens from Task A to Task B. The context of Task A is saved in the TSS of Task A, and Task B starts from the context as saved in the TSS of Task B.
- Each TSS should have only one TSS descriptor that points to it.

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).
- The fields of a TSS are divided into two main categories:
  - **dynamic fields** : The processor updates dynamic fields when a task is suspended during a task switch. *These fields are inside the blue polygon in the diagram below.*
  - and **static fields** : The processor reads the static fields, but does not normally change them.



Dynamic Fields:

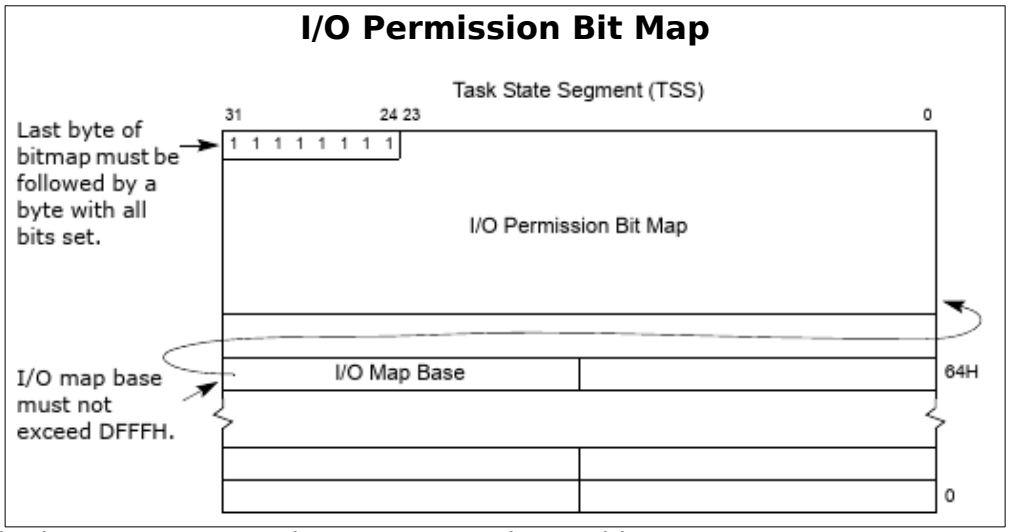
- **General-purpose register fields** — State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.
- **Segment selector fields** — Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.
- **EFLAGS** register field — State of the EFLAGS 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 by using the IRET instruction.

Static Fields:

- **LDT segment selector** field — Contains the segment selector for the task's LDT.
- **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.
- **Shadow Stack Pointer (SSP)** — Contains task's shadow stack pointer. The shadow stack of the task should have a supervisor shadow stack token at the address pointed to by the task SSP (offset 104). This token will be verified and made busy when switching to that shadow stack using a CALL/JMP instruction, and made freewhen switching out of that task using an IRET instruction.
- **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.

  The **I/O permission bit map in the TSS** can be used to **modify the effect of the IOPL** on I/O sensitive instructions, **allowing access to some I/O ports by less privileged programs or tasks.**

I/O Permissions for IN, INS, OUT, OUTS, CLI and STI instructions.
CPL =< IOPL of the current task or program. This above check fails processor checks the I/O Permission bit map in the TSS of the current task, to determine if access to a particular I/O port is allowed.
If not met, GP fault is generated.



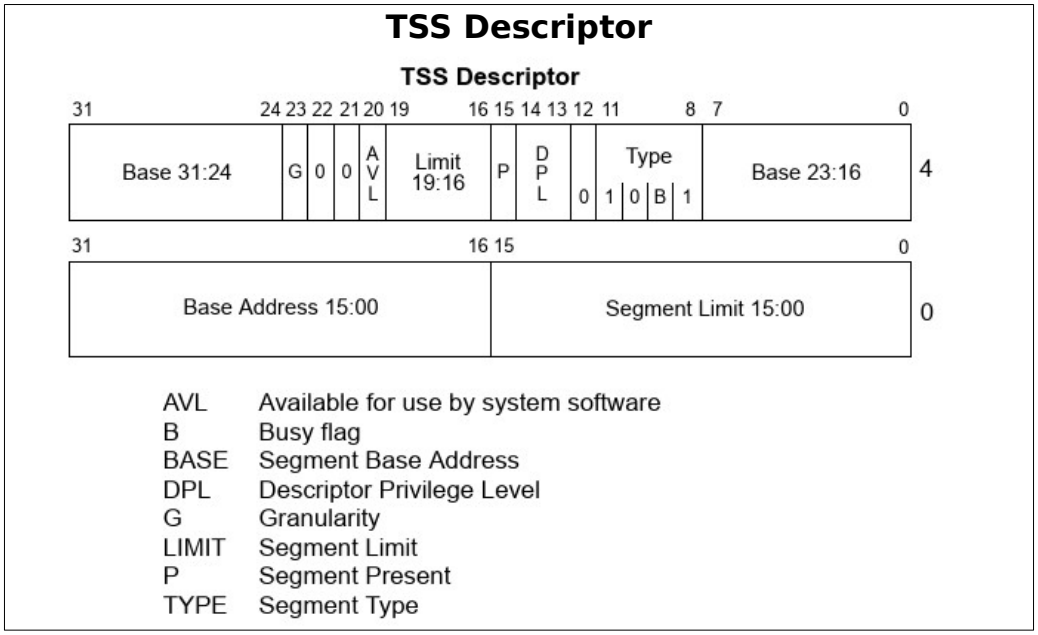
Each bit in the map corresponds to an I/O port byte address.

For example, the control bit for I/O port address 29H in the I/O address space is found at bit position 1 of the sixth byte in the bit map.

If paging is used:

- Avoid **placing a page boundary in the part of the TSS that the processor reads during a task switch** (the first 104bytes). **The processor may not correctly perform address translations if a boundary occurs in this area.** *During a task switch, the processor reads and writes into the first 104 bytes of each TSS* (using contiguous physical addresses beginning with the physical address of the first byte of the TSS). So, after TSS access begins, if part of the 104 bytes is not physically contiguous, the processor will access incorrect information without generating a page-fault exception.
- Pages corresponding to the **previous task's TSS**, **the current task's TSS**, and **the descriptor table entries for each** all should be **marked as read/write**.
- **Task switches are carried out faster if the pages containing these structures are present in memory** before the task switch is initiated.

TSS Descriptor



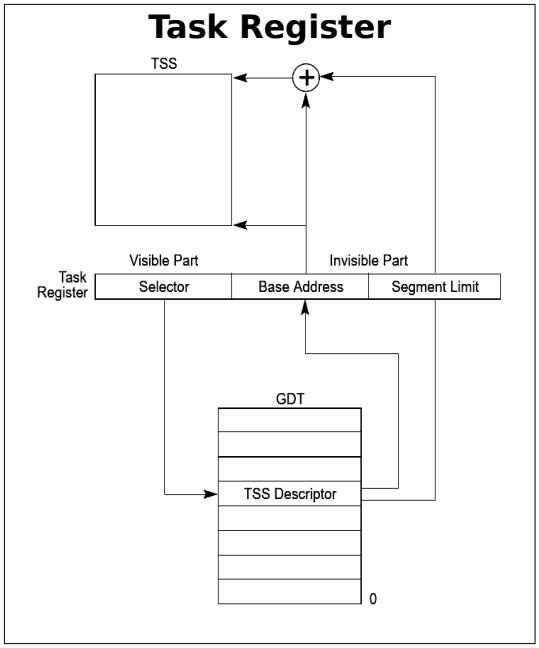
- The TSS, like all other segments, is defined by a segment descriptor.
- TSS descriptors can only be placed in the GDT; they cannot be placed in LDT or the IDT.
- A **general-protection exception** is generated if an attempt is made to **load a segment selector for a TSS into a segment register**.
- The **busy flag (B)** in the type field indicates whether the task is busy.

B = 1	Busy, i.e Suspended or currently Running
B = 0	Inactive.

  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.
- The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data segment descriptors.
- **Limit:**
  - **When the G flag is 0** in a TSS descriptor for a 32-bit TSS, the **limit field must have a value equal to or greater than 67H**, (67H = 0d103) one byte less than the minimum size of a TSS.
  - A larger limit is required if an I/O permission bit map is included or if the operating system stores additional data.

Access to TSS segemnt. Same as Data segment	
CPL <= DPL of the conforming data segment.	DPL sets the numerically highest privilage level.
!! What is the function of RPL here?	

Task Register



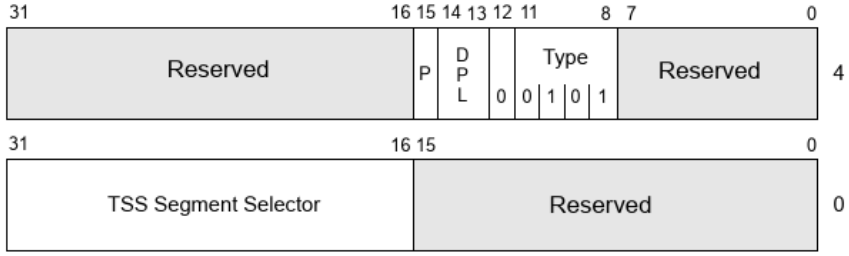
- The task register holds the following for the TSS of the current task.
  - 16-bit segment selector and
  - the entire segment descriptor (32-bit base address, 16-bit segment limit, and descriptor attributes.
- The task register has a **visible part**
  - that can be read and changed by software.
  - The segment selector in the visible portion points to a TSS descriptor in the GDT.
- And an **invisible part**
  - maintained by the processor and is inaccessible by software.
  - 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.
- The **LTR** (load taskregister) and **STR** (store task register) instructions **load and read the visible portion** of the task register.
- **LTR** is a privileged instruction that may be executed only when the CPL is 0.
- On power up or reset of the processor, segment selector and base address are set to the default value of 0; the limit is set to FFFFH.

Task-Gate Descriptor

- A task-gate descriptor provides an indirect, protected reference to a task.
- It can be placed in the GDT, an LDT, or the IDT. Unlike TSS segment, which can only be placed in GDT.
- 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.

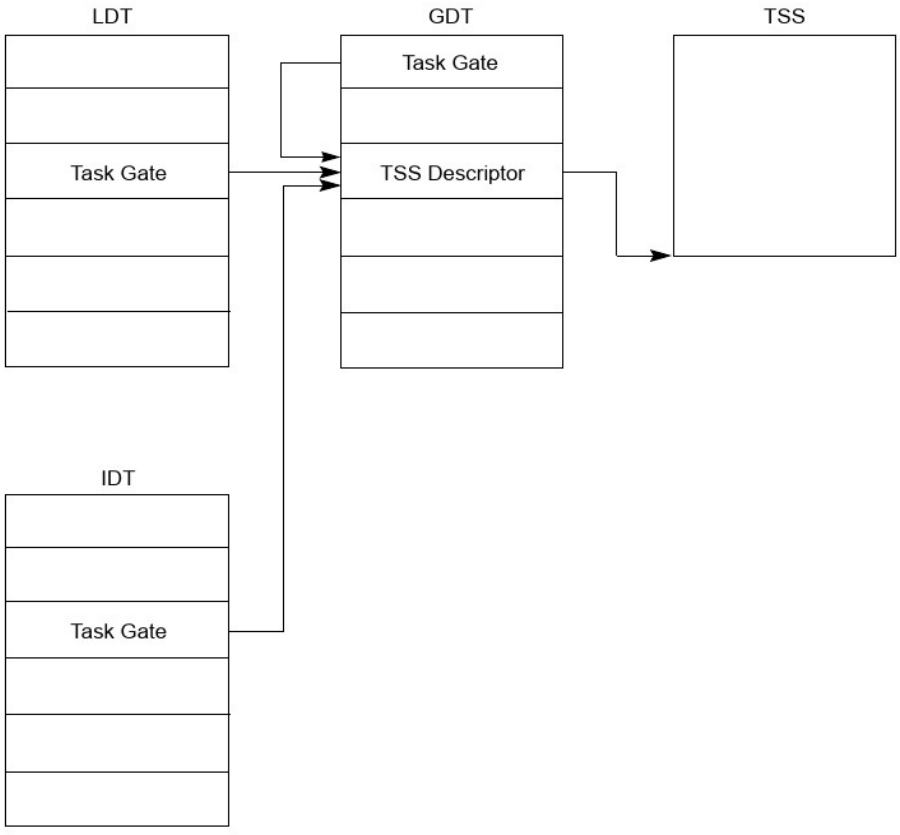
Privilege check when accessing through Task Gate		
CALL / JMP	**MAX(CPL, RPL) <= DPL** of the task gate descriptor. DPL of the destination TSS descriptor is not used.	DPL field of the task-gate descriptor specifies the numerically highest privilege level which can access the task gate

Task-Gate Descriptor



DPL Descriptor Privilege Level
P Segment Present
TYPE Segment Type

Task Gates Regerencing the same Task



Intel x86 Software developers - Interrupt and Exception Handling

28 June 2021

- **Interrupts** occur at random times during the execution of a program, in response to signals from hardware or software executing the INT n instruction.
- **Exceptions** occur when the processor detects an error condition while executing an instruction, such as division by zero.
- The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a **machine-check exception** to be generated when internal hardware errors and bus errors are detected.

Excpetion And Interrupt Vectors

- Every exception and interrupt condition is assigned a unique identification number, called a **vector number**.
  - Vector numbers is **0 to 255**
  - Vector numbers in the range **0 through 31** are **reserved** by Intel for architecture defined exceptions and interrupts.
  - Vector numbers in the range **32 to 255** are designated as **user-defined interrupts.**

External Interrupts:

- External interrupts are received **through pins on the processor** or **through the local APIC**.
- The primary interrupt pins are the **LINT[1:0] pins** (on *Pentium 4, Intel Xeon, P6 family, and Pentium* processors)
  - Local APIC is **enabled:**

    LINT[1:0] pins can be programmed through the APIC's **local vector table** (LVT) to be associated with any of the processor's exception or interrupt vectors.
  - Local APIC is **disabled:**

    LINT[1:0] pins are configured as **INTR** and **NMI** pins. In Intel486 processor and earlier Pentium processors, there were dedicated INTR and NMI pins.

    INTR: Asserting the INTR pin signals the processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A.

    NMI: Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt **vector 2**.

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

How Interrupts are handled when APIC in enabled:

The processor's local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC's pins can be directed to the local APIC through the system bus or the APIC serial bus.

The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC.

Maskable Hardware Interrupts:

- Any external interrupt that is delivered by the **INTR** pin or through the **local APIC** is called a **maskable hardware interrupt**.
- The **IF** flag in the **EFLAGS** register permits all maskable hardware interrupts to be masked as a group.
- Maskable hardware interrupts that can be delivered through the **INTR pin**:

  0 through 255.
- Maskable hardware interrupts that can be delivered through the **local APIC**:

  16 through 255 (i.e excluding the software interrupts or exceptions).

Software-Generated Interrupts:

- Software interrupts can be generated by the INT n instruction.

  For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.
- Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction.
- The **IF** flag in **EFLAGS** registrer **do not mask/disable** a software interrupt.

Software-Generated Exceptions:

Can be generated using one the below instructions.

- INTO
- INT1
- INT3:

  Causes Breakpoint exception to be generated.
- BOUND

The INT n instruction can be used to emulate exceptions in software; but the processor does not push an error code on the stack. The exception handler will still attempt to pop an error code from the stack while handling the exception. The handler will thus pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

Program-Error Exceptions:

- **Faults** —

  Can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity.

  When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction.

  The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.

  The processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

  Note: Faults are not always restartable. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. However, this situation when occurs, is not restartable. The Operating System must detect such class of exceptions and must terminate the causing application.

- **Traps** —

  It is reported immediately following the execution of the trapping instruction.

  Traps allow execution of a program or task to be continued without loss of program continuity.

  The return address for the trap handler points

  - To the instruction after the trapping instruction.
  - However, if trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer.

    For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction.

- **Aborts** —

  It does not always report the precise location of the instruction causing the exception.

  Does not allow a restart of the program or task that caused the exception.

  Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Machine-Check Exceptions:

The P6 family and Pentium processors provide an internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions.

When a **machine-check error** is detected, the processor signals a **machine-check exception** (**vector 18**) and returns an error code.

Masking Exceptions and Interrupts When Switching Stacks:

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, StackTop
```

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but

before the ESP register has been loaded, stack space becomes inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

The processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction.

However Intel recommends that software use the **LSS** instruction to load the SS register and ESP together.

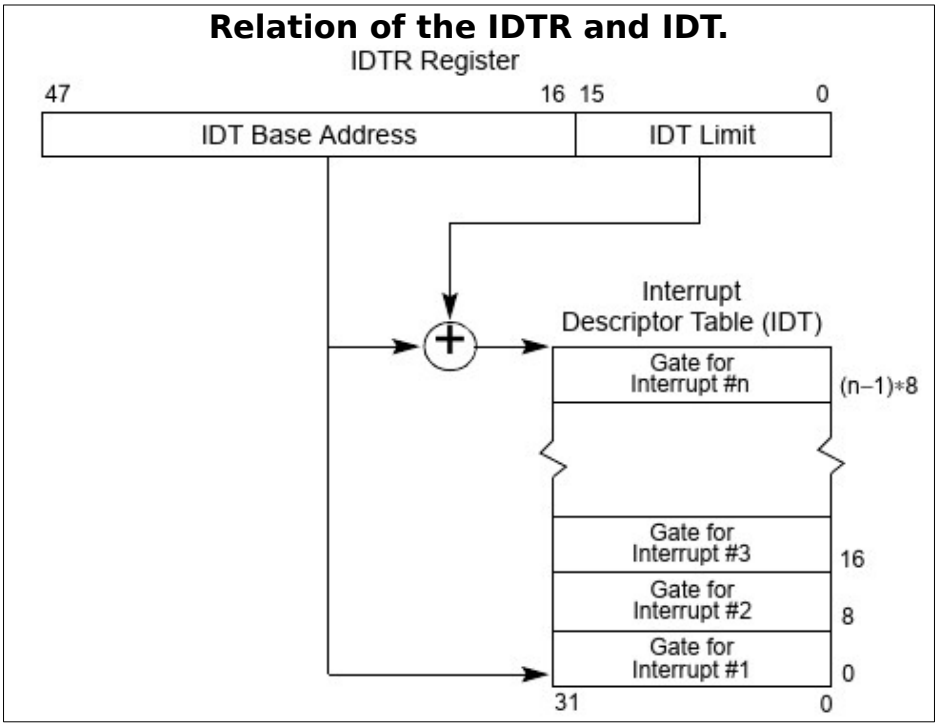
Interrupt Descriptor Table (IDT)

- The IDT, like GDT, LDT, is an array of 8-byte descriptors (in protected mode).
- Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors.
- All **empty descriptor** slots in the IDT should have the **present flag** for the descriptor **set to 0**.
- The base addresses of the **IDT** should be **aligned** on an **8-byte boundary** to maximize performance of cache line fills.
- The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte.

  Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, 8N – 1).
- Last Valid Byte is (8N-1) bytes from the starting location.

  Where N is the number if IDT entries.
- The IDT reside anywhere in the **linear address space**.

IDT Register (IDTR)



- **LIDT: L**oad IDT register.
  - This instruction can be executed only when the CPL is 0.
- **SIDT: S**tore IDT register

IDT Descriptors

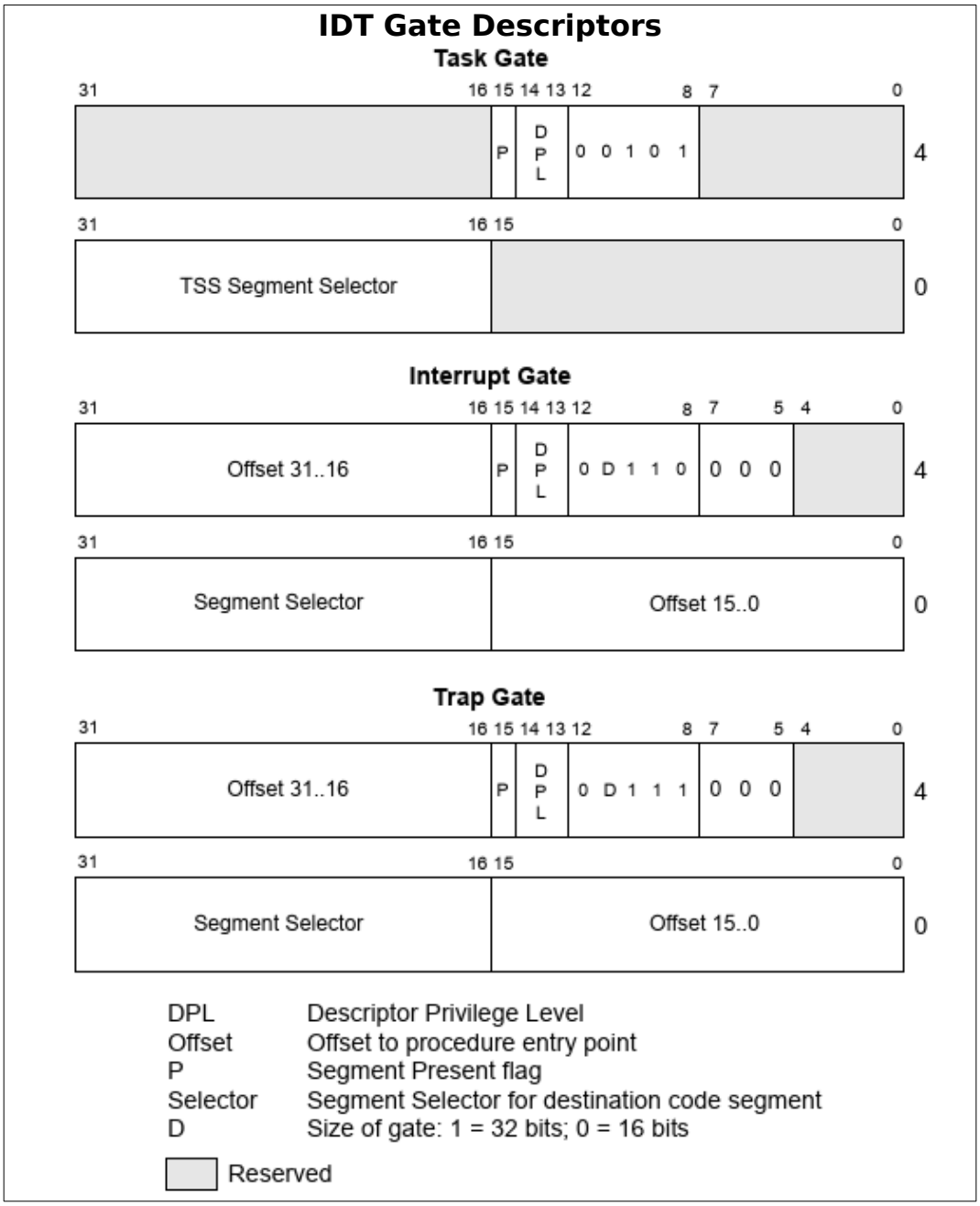
The IDT may contain any of three kinds of gate descriptors:

- **Task-gate descriptor**
- **Interrupt-gate descriptor**

  Interrupt and trap gates are very similar to call gates.

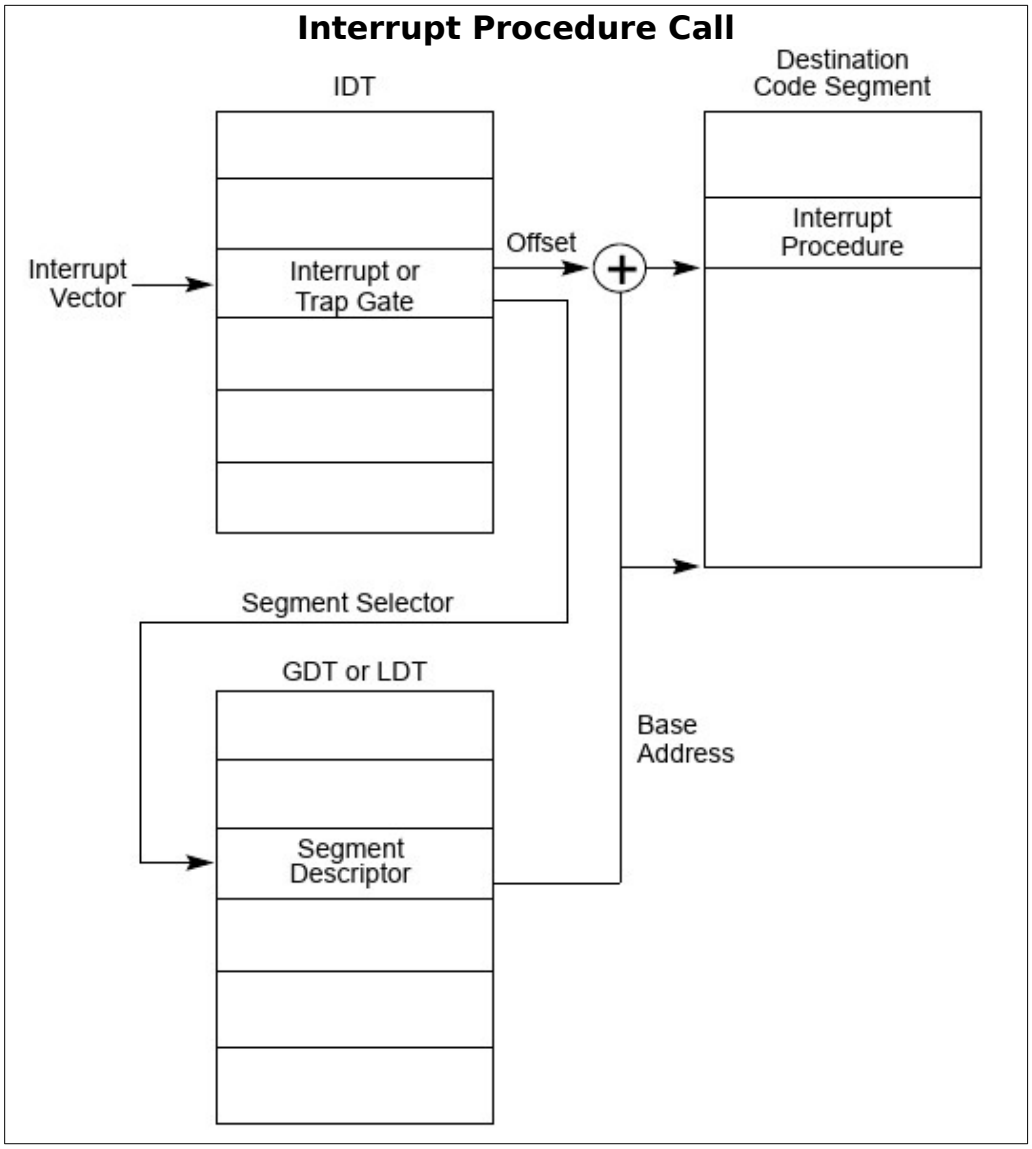
  They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution in an exception- or interrupt-handler code segment.
- **Trap-gate descriptor:**

  The format is the same as that of a task gate used in the GDT or an LDT.



- When accessing an exception/interrupt handler through either an **a trap gate**,
  - TF, VM, RF, and NT flags in the EFLAGS register are Cleared.
- When accessing an exception- or interrupt-handling procedure through an **interrupt gate**.
  - TF, VM, RF, and NT flags in the EFLAGS register are Cleared.
  - IF flag is Cleared. This prevents other interrupts from interfering with the current interrupt handler.

Exception and Interrupt Handling



When the processor performs a call to the Exception or Interrupt handler procedure:

- If the handler procedure is going to be **executed** at a numerically **lower privilege level,** a stack switch occurs. When the **stack switch occurs**:

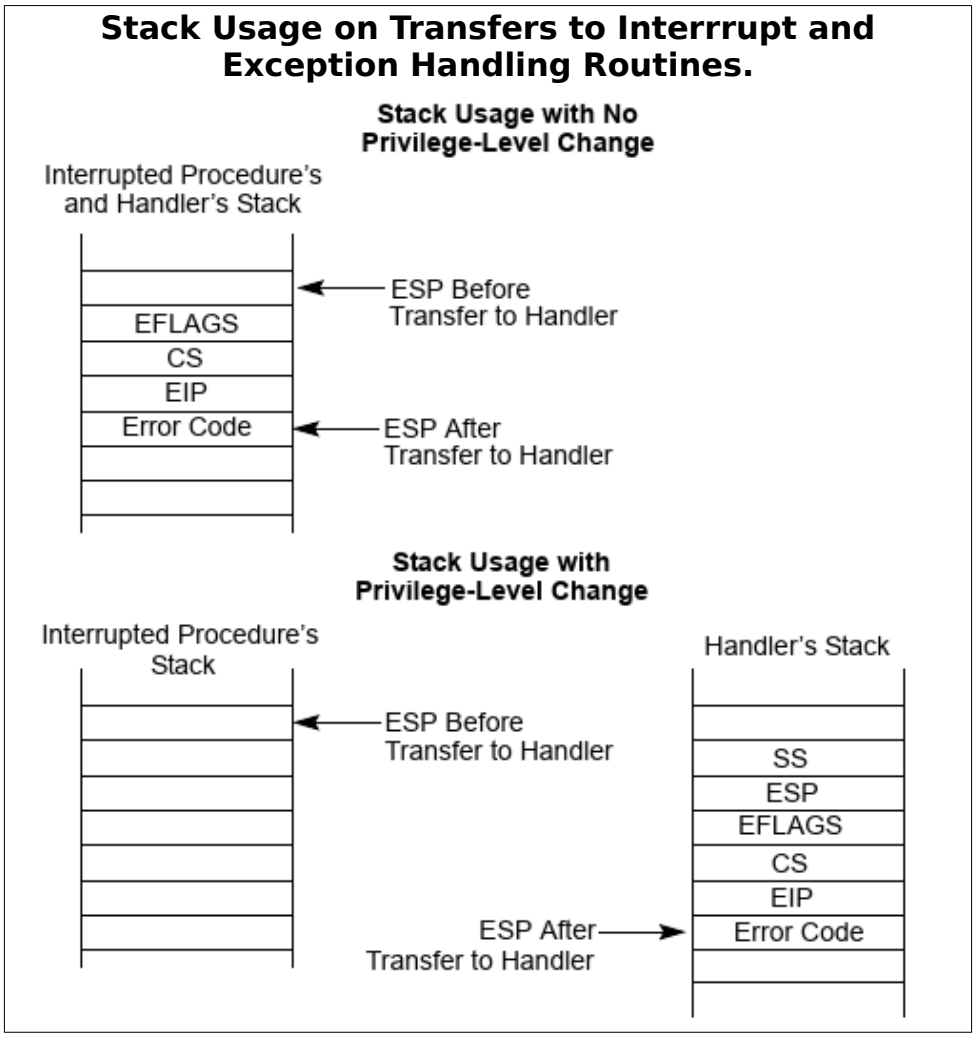
  a) The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task (by DPL of the code segment descriptor, not DPL of interrupt/exception IDT decriptor).

  On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.

  b) The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack.

  c) If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.

- If the handler procedure is going to be **executed** at the **same privilege level** as the interrupted procedure:
  a) The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack.
  b) If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.



- **IRET** instruction is used to return from an exception or interrupt hanfler procedure.
- The **IRET** instruction is similar to the **RET** instruction **except** that it **restores** the saved flags into the **EFLAGS** register.
  - The **IOPL** field of the EFLAGS register is restored only if the CPL is 0.
  - The IF flag is changed only if the CPL is less than or equal to the IOPL.
- If a **stack switch** occurred when calling the handler procedure, the **IRET** instruction **switches back** to the interrupted procedure's stack on the return.

Transfer to an exception/interrupt hander procedure.	
CPL >= DPL of the code segment. Code segment DPL = 0. Can be accessed from any CPL.	DPL sets the numerically lowest privilage level.
General protection fault is generated if validation fails.	

Transfer to an exception/interrupt hander procedure when generated by INT n, INT3 or INTO instruction.
CPL <= DPL of the gate descriptor.
General protection fault is generated if validation fails.

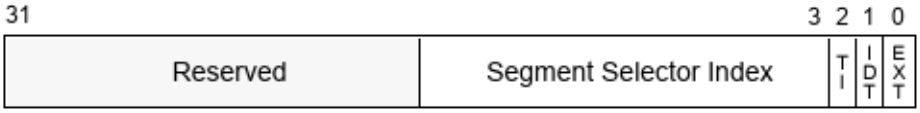
- Both the above conditions is checked when processor transfer to a interrupt/exception handler. Take the below example from MeghaOS.

INT 0x40			
Interrupt Gate DPL	Code Segment DPL	CPL	Result
2	0	3	GP fault is generated. • **CPL >= DPL** of Code segmet: Valid • **CPL <= DPL** of Gate: Invalid
3	0	3	Transfered to the interrupt procedure. • **CPL >= DPL** of Code segmet: Valid • **CPL <= DPL** of Gate: Valid

- RPL is not checked, as there exits no RPL for interrupt/exception vectors.
- For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.
- If the handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program.

Error Code

When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler.



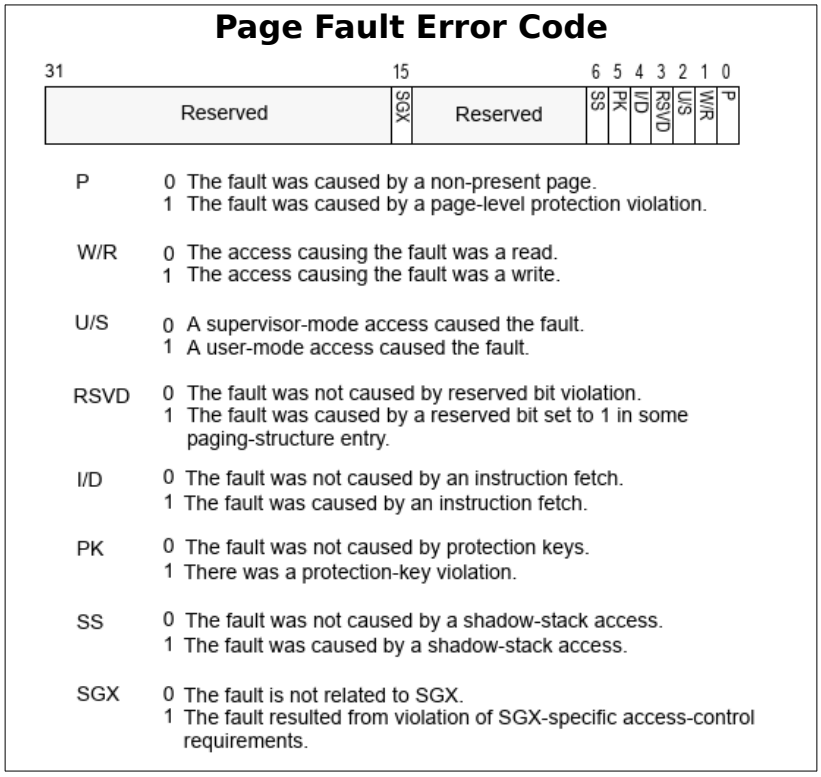
EXT External event (bit 0) — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.5 The bit is cleared if the exception occurred during delivery of a software interrupt (INT n, INT3, or INTO).

IDT Descriptor location (bit 1) — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.

GDT/LDT (bit 2) — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

- Error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Note: For Page Fault, the error code, pushed on the Stack is differnt from the format mentioned above.



Intel x86 Software developers - Registers

18 September 2021

CR4

Bit	Name	Description
0	VME	**Virtual 8086 Mode Extensions** - If set, enables support for the virtual interrupt flag (VIF) in virtual-8086 mode.
1	PVI	**Protected-mode Virtual Interrupts** - If set, enables support for the virtual interrupt flag (VIF) in protected mode.
2	TSD	**Time Stamp Disable** - If set, RDTSC instruction can only be executed when in ring 0, otherwise RDTSC can be used at any privilege level.
3	DE	**Debugging Extensions** - If set, enables debug register based breaks on I/O space access.
4	PSE	**Page Size Extension** - If unset, page size is 4 KiB, else page size is increased to 4 MiB If PAE is enabled or the processor is in x86-64 long mode this bit is ignored.
5	PAE	**Physical Address Extension** - If set, changes page table layout to translate 32-bit virtual addresses into extended 36-bit physical addresses.
6	MCE	**Machine Check Exception** - If set, enables machine check interrupts to occur.
7	PGE	**Page Global Enabled** - If set, address translations (PDE or PTE records) may be shared between address spaces.
8	PCE	**Performance-Monitoring Counter enable** - If set, RDPMC can be executed at any privilege level, else RDPMC can only be used in ring 0.
9	OSFXSR	**Operating system support for FXSAVE and FXRSTOR instructions** - If set, enables Streaming SIMD Extensions (SSE) instructions and fast FPU save & restore.
10	OSXMMEXCPT	**Operating System Support for Unmasked SIMD Floating-Point Exceptions** - If set, enables unmasked SSE exceptions.
11	UMIP	**User-Mode Instruction Prevention** - If set, the SGDT, SIDT, SLDT, SMSW and STR instructions cannot be executed if CPL > 0.
12	LA57	If set, enables 5-Level Paging.
13	VMXE	**Virtual Machine Extensions Enable**
14	SMXE	**Safer Mode Extensions Enable**
16	FSGSBASE	Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.
17	PCIDE	**PCID Enable** - If set, enables process-context identifiers (PCIDs).
18	OSXSAVE	**XSAVE and Processor Extended States Enable**
20	SMEP	**Supervisor Mode Execution Protection Enable** - If set, execution of code in a higher ring generates a fault.
21	SMAP	**Supervisor Mode Access Prevention Enable** - If set, access of data in a higher ring generates a fault.
22	PKE	**Protection Key Enable**

EGLAGS

Bit	Name	Description	Category	1	0
FLAGS					
0	CF	Carry flag	Status	Carry	No Carry
1		Reserved, always 1 in EFLAGS			
2	PF	Parity flag	Status	Parity	Parity Odd

				Even	
3		Reserved			
4	AF	Adjust flag	Status	Auxiliary Carry	No Auxiliary Carry
5		Reserved			
6	ZF	Zero flag	Status	Zero	Not Zero
7	SF	Sign flag	Status	Negative	Positive
8	TF	Trap flag (single step)	Control		
9	IF	Interrupt enable flag	Control	Enable Interrupt	Disable Interrupt
10	DF	Direction flag	Control	Down	Up
11	OF	Overflow flag	Status	Overflow	Not Overflow
12-13	IOPL	I/O privilege level (286+ only), always 1on 8086 and 186	System		
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System		
15		Reserved, always 1 on 8086 and 186, always 0 on later models			
EFLAGS					
16	RF	Resume flag (386+ only)	System		
17	VM	Virtual 8086 mode flag (386+ only)	System		
18	AC	Alignment check (486SX+ only)	System		
19	VIF	Virtual interrupt flag (Pentium+)	System		
20	VIP	Virtual interrupt pending (Pentium+)	System		
21	ID	Able to use CPUID instruction (Pentium+)	System		
22-31		Reserved	System		

- The **status flags** (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions.

- The **system** and **control** flags in the EFLAGS register control operating-system or executive operations. They should not be modified by application programs.

CR0

bit	label	description
0	pe	protected mode enable
1	mp	monitor co-processor
2	em	emulation
3	ts	task switched
4	et	extension type
5	ne	numeric error
16	wp	write protect
18	am	alignment mask
29	nw	not-write through
30	cd	cache disable
31	pg	paging

CR2

bit	label	description
0-31	pfla	page fault linear address

GDTR

bits	label	description
0-15	limit	(size of GDT) - 1
16-47	base	starting address of GDT

LDTR

bits	label	description
0-15	limit	(size of LDT) - 1
16-47	base	starting address of LDT

IDTR

bits	label	description
0-15	limit	(size of IDT) - 1
16-47	base	starting address of IDT

Intel x86 Software developers - Paging

16 September 2021

- Segmentation converts logical addresses to linear addresses.
- Paging translates each linear address to a physical address.
- Software enables paging by
  - Loading CR3 with the physical address of the first paging structure that the processor will use for linear-address translation and
  - then using MOV to set CR0.PG.
- Paging can be enabled only if protection is enabled (CR0.PE = 1).

Paging Modes

The values of CR4.PAE, CR4.LA57, and IA32_EFER.LME determine which paging mode is used:

Paging mode	CR0.PG	CR4.PAE	CR4.LA57	A32_EFER.LME	Linear address width	Physical address width	Page sizes
None	0	n/a	n/a	n/a	32	32	n/a
32 bit	1	0	n/a	0	32	Upto 40	4 KB, 4 MB
PAE	1	1	n/a	0	32	Upto 52	4 KB, 2 MB
4-level	1	1	0	1	48	Upto 52	4 KB, 2 MB 1 GB
5-level	1	1	1	1	57	Upto 52	4 KB, 2 MB 1 GB

- Linear-address width. The size of the linear addresses that can be translated.
- Physical-address width. The size of the physical addresses produced by paging.
- Page size. The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.
- Regardless of the current paging mode, software can disable paging by clearing CR0.PG with MOV to CR0.

Paging mode modifiers

Reg	Bit	0	1
CR0	WP	Supervisor-mode code can write to read-only access linear addresses.	Supervisor-mode code cannot write to read-only access linear addresses.
	PG	Paging is disabled	Paging is enabled.
CR4	PSE	32-bit paging can use only 4-KByte pages.	32-bit paging can use both 4-KByte pages and 4-MByte pages
	PAE	Disables PAE Paging	Enables PAE Paging
	PGE	Disabled Global Pages, no translations are shared across address spaces.	Enabled Global pages, specified translations may be shared across address spaces
	PCIDE	Disables process-context identifiers (PCIDs).	Enables process-context identifiers (PCIDs) for 4-level paging and 5-level paging. PCIDs allow a logical processor to cache information for multiple linear-address spaces.
	SMEP	Software operating in supervisor mode **can** fetch instructions from linear addresses that are accessible in user mode.	Allows pages to be protected from supervisor-mode instruction fetches. Software operating in supervisor mode **cannot** fetch instructions from linear addresses that are accessible in user mode.
	SWAP	software operating in supervisor mode **can** access data at linear addresses that are accessible in user mode.	software operating in supervisor mode **cannot** access data at linear addresses that are accessible in user mode. Software can override this protection by setting EFLAGS.AC flag.
	PKE		4-level paging and 5-level paging associate each linear address with a protection key. The PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written.
	PKS		The IA32_PKRS ***MSR*** does the same for supervisor-mode linear addresses.
	CET		Enables control-flow enforcement technology, Certain memory accesses are identified as shadow-stack accesses and certain linear addresses translate to shadow-stack pages
IA32_EFER ***MSR***	NXE		Enables execute-disable access rights for PAE paging, 4-level paging, and 5-level paging. Instruction fetches can be prevented from specified linear addresses.

32 Bit Paging

- 32-bit paging may map linear addresses to either 4-KByte pages or 4-MByte pages.
- 32-bit paging translates 32-bit linear addresses to 40-bit physical addresses. Linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.
- 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain.
  - If that entry maps a page, the page frame is 22 Bytes = 4 Mbytes. Thus no page page is needed for 4 MB pages.
  - In our case each entry of the 1st paging structure (page directory) points to a 2nd paging structure (a page table), which uses 10 more bits to locate an entry into this structure (the page table); 12 bits remain. Thus the page frame is 12 bits = 4 KBytes

Format of CR3 and Paging structure entries with 32-Bit paging

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Item
Address of page directory																				-							PCD	PWT	-				CR3
Address of page table																				-				PS 0	-	A	PCD	PWT	U/S	R/W	P	1	PDE 4 KB
Bits 31:22 of address of 4 MB page frame										Reserved (must be 0)										Bits 39:32 of address			PAT	-		G	PS 1	D	A	PCD	PWT	U/S	R/W	P	1	PDE 4 MB
Address of 4KB page frame																				-			G	PAT	D	A	PCD	PWT	U/S	R/W	P	1	PTE 4 KB

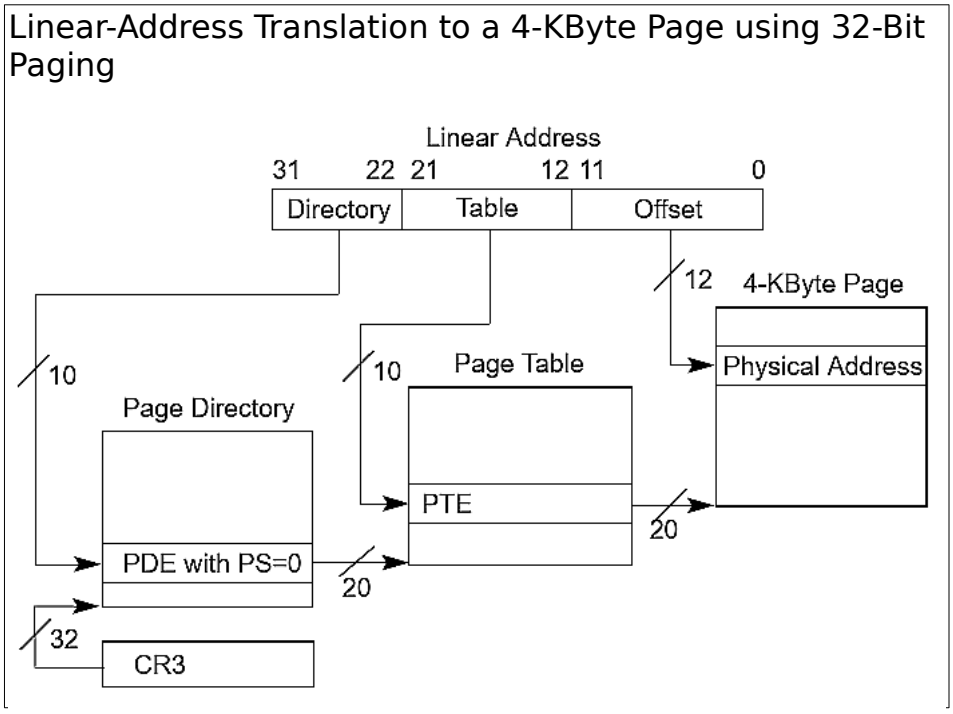
Note: *CR3 has 64 bits on processors with Intel-64 architecture. These bits are ignored with 32-bit paging.*

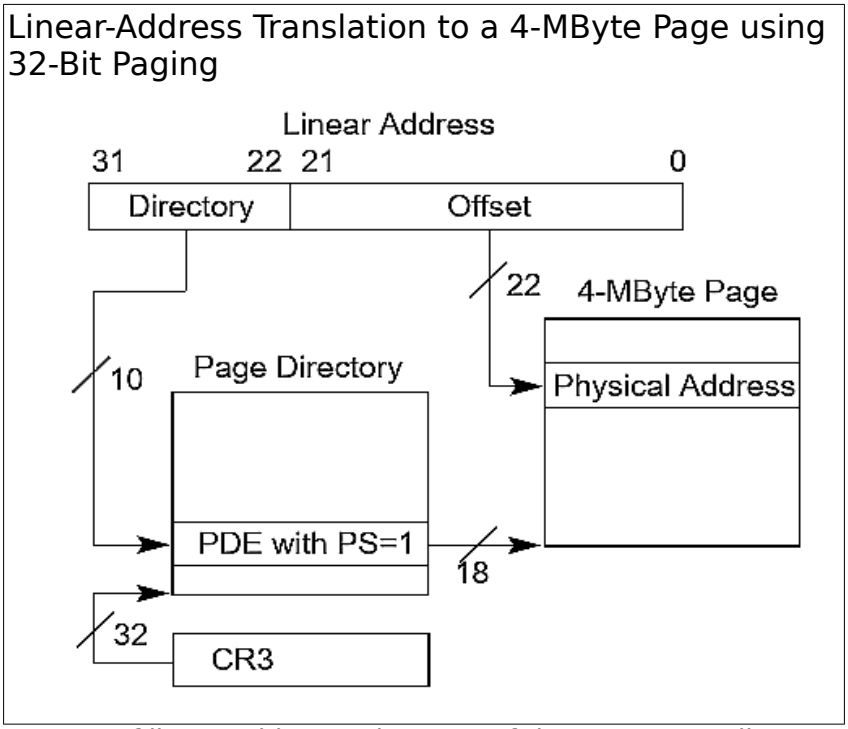
Use of CR3 with 32-Bit Paging

Bit	Contents
3 (PWT)	Page-level write-through. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead. Controls the write-through/writeback caching policy on the whole page directory. Since the internal cache of the I page ntel486 processor is a write-through cache, it is not affected by the state of the PWT flag. Also used to indirectly determines the memory type used to access the page directory during linear-address translation. (see Section 4.9)
4 (PCD)	Page-level cache disable. If the bit is set, the page (which contains the page directory) will not be cached. Otherwise, it will be. Also used to indirectly determines the memory type used to access the page directory during linear-address translation. (see Section 4.9)
31:12	Physical address of the 4-KByte aligned page directory used for linear-address translation

The **memory type** of a memory access refers to the type of caching used for that access.

Address translation in 32 bit paging





4MByte pages: 31:22 of linear address select one of the 1024 page directory entries. The 4 Mbyte PDE outputs 39:22 of the physical address for the page frame. The lower 22 bytes come from the remaining 22:0 of the linear address. But because we are still working with 32 bits of linear address, 4 Gbyte is the maximum addressable. That is why page table is not needed for this mode – PDE directly maps page frames not individual page tables.

Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead. Controls the write-through/writeback caching policy on the whole page table (page with contains the page table). Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag. Also used to indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable. If the bit is set, the page (which contains the page table) will not be cached. Otherwise, it will be. All the IA-32 processors that have on-chip caches also provide the PCD (page-level cache disable) flag in page table and page directory entries. This flag allows caching to be disabled on a page-by-page basis. Also used to indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
7 (PS)	If CR4.PSE = 1, must be 0 (If set to 1, this entry maps a 4-MByte page; see Table 4-4). If CR4.PSE = 0 ignored.
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit	Contents
0 (P)	Present, Must be 1 to map a 4KB page.
1 (R/W)	Read/write; if 0, writes is not be allowed to the 4-KByte page referenced by this entry. Read is always allowed?
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry.
3 (PWT)	Page-level write-through. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead. Controls the write-through/writeback caching policy on the whole page. Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag. Also used to indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable. If the bit is set, the page (page frame) will not be cached. Otherwise, it will be. All the IA-32 processors that have on-chip caches also provide the PCD (page-level cache disable) flag in page table and page directory entries. This flag allows caching to be disabled on a page-by-page basis Also used to indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global; ignored otherwise. If set (along with CR4.PGE), prevents the TLB from updating the address in its cache if CR3 is reset
31:12	Physical address of the 4-KByte page referenced by this entry.

Access Rights

- Every access to a linear address is either a supervisor-mode access or a user-mode access.
- Accesses made while CPL < 3 are supervisor-mode accesses.
- Accesses made while CPL = 3 are user-mode accesses.
- **implicit supervisor-mode**

  Accesses to some of data structures are supervisor-mode accesses regardless of CPL:

  - Accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor.
  - Accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception.
  - Accesses to the task-state segment (TSS) as part of a task switch or change of CPL.
- **explicit supervisor-mode** are all other accesses made while CPL < 3.
- **supervisor-mode address:** U/S flag (bit 2) is 0 in at least one of the paging-structure entries.
- **user-mode address**: U/S flag (bit 2) is 0 in all of the paging-structure entries.

Access and Dirty flags

These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Accessed flag : Is set whenever the processor **uses a paging-structure entry** as part of linear-address translation.

Dirty flag: Is set whenever there is a **write to a linear address**.
The Dirty flag is set in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are “**sticky**,” meaning that, once set, the processor does not clear them; only soft-ware can clear them.

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10).

This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected linear address (see Section 4.10.4.3).

Caching Translation Information

The Intel-64 and IA-32 architectures may accelerate the address-translation process by caching data from the paging structures on the processor.

Because the processor does not ensure that the data that it caches are always consistent with the structures in memory, it is important for software developers to understand how and when the processor may cache such data.

Appendix A - Alignment requirements

GDT

The base address of GDT must be aligned on an eight-byte boundary to yield the best processor performance.

Stack Pointer

The stack pointer for a stack segment should be aligned on 16-bit or a 32-bit boundaries – Based on the D flag (stack uses this flag to determine how much to increment or decrement the stack pointer on a push or pop operation).

Pushing a 16-bit value onto a 32-bit wide stack can result in stack misalignment (no longer aligned to a 32-bit boundary). One exception to this rule is when the contents of a segment register (a 16-bit segment selector) are pushed onto a 32-bit wide stack. Here the processor automatically alignes the stack pointer to the next 32-bit boundary.

Misaligning a stack pointer can cause serious performance degradation and in some cases program failure.

IDT

The base address of IDT should be aligned to 8-byte boundary to maximize performance of cache line fills.

TSS

Avoid placing a page boundary in the part of the TSS that the processor reads during a task switch (the first 104 bytes). The processor may not correctly perform address translations if a boundary occurs in this area.

Other than this, I did not find any other alignment requirement for TSS base address.

Higher half paging in Megha OS

Note: *This is a very early note. I put this here as it focuses on the basics of higher half kernel implementation.* ***Do not take the addressses mention here as final, they will not match what we have currently***

How to use page tables to map Virtual Addresses to Physical Addresses?

Choose a mapping

1. Kernel Physical Address (kpm):
   Location where the kernel is loaded, or could be taken as 0000 address.
2. Kernel Virtual Address (kvm):
   Location you want to use to get to the kernel from user programs. This is the higher virtual address you want for your kernel. Memory **at or above** `kvm` is available for the kernel, and memory **less** than `kvm` is available for the user programs.

kpm = 0x1000, kvm = 0xC0000000

This is possible, but we will not be able to access memory lower than 0x1000. This is therefore,
not a viable mapping. (There will be no mapping of physical addresses from 0000 to 0x0FFF in
the page tables)

kpm = 0x1000, kvm = 0xC0001000

kpm = 0x0000, kvm = 0xC0000000

You can see that the above two cases, the mapping remains the same i.e. Same virtual address maps to the same physical address. Because mappings are the same, we will have the same paging structures and tables.

Page directories and CR3 register

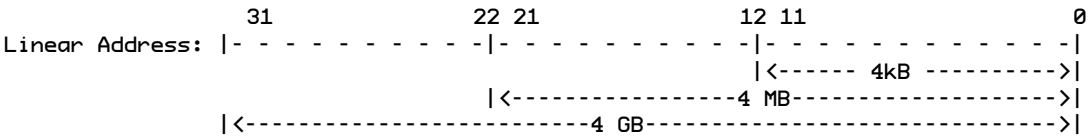
Choose any 4KB aligned physical address to store the Page directory entries.

In general depending on the requirement, you may need to allocate storage form 1 to 1024 such
directory entries.

```
CR3 [31:12] = Physical Address of the first page directory / 0x1000.
```

In the above example, we may place the page directory at physical location 0x1000.
Store the same in CR3.

Address translation in paging



[11:0] Linear address:

Selects byte within a 4 kB page.

[21:12] Linear address:

Selects which page. Indexes into the page table to get the start address of the 4kB page.

Value in [21:12]	Page selected. Subsequently the start address of the 4 Kbyte page.
0x00	Selects 1st page table entry and gets the start address of the 1st page. (from the current page table)
0x3FF	Selects the 1024st page table entry and gets the start address of the 1024st page. (from the current page table)

[31:22] Linear address

Selects which page table. Indexes into the page directory table to get the start of the page table.

Value in [31:22]	Page table selected. Subsequently the start address of the page table.
0x00	Selects 1st directory entry and gets the start address of the 1st page table. (from the current page table)
0x300	Selects 769st directory entry and gets the start address of the 769st page table. (from the current page table)

Considerations for higher half kernel mapping.

Consider we have

- kvm = 0xC0001000
- kpm = 0x00001000
- We want the map VGA physical memory 0xB8000 to the 1024th page.

What will be the setup?
Note: Physical addresses are shown in **bold**.

- Placement of the page directory table is arbitory. Lets say we place it at physical location **0x1000**.
- We place the 1st page table at location **0x2000** (4 kB bytes from the start of page directory table).
- In our scheme, 0xC0001000 maps to **0x1000**. Which is same as 0xC0000000 mapping to **0x0000**.
- kvm of 0xC0000000 mapping to **0x0000**, means that every virtual address will be equal to or higher than 0xC0000000.
  For example: **0x200** of physical location is 0xC0000200 in virtual address.
- As mentioned above [31:22] bits of the logical/virtual address, selects the page directory.
  If 0xC0000000 is our lowest address, then, 0x301st (index = 0x300) page directory entry will be selected first. In other words, every address from 0xC0000000 will index page directories from 768 to 1023.

Index	**Value in [31:12] of the directory or page table entry**	
Page directory (Base : 0x1000)		
0	2	Identity mapping. 0x0000 maps to **0x0000**
768	2	Higher half mapping. 0xC0000000 maps to **0x0000**
Page table (Base : 0x2000)		
0	0	Normal mapping. 0^{th} page maps to 0x000 and so on. Note that here mapping to **0xB8000** may also exist. (not taken into account below)
…	…	
1023	B8	1023^{rd} page maps to **0xB8000**

The identity mapping will be required in real life, because the paging setup code is still mapped at the physical kernel load address. We will eventially remove the identity mapping when the control has moved to a portion of code that is mapped at the higher address. Untill that time, the identity mapping is required.

Calculation of logical address given physical address.

In general, application programs, should not worry about physical addresses. If however, such
calculation is required, then, this is mostly done during development and such address is hard coded.

1. Search page tables, for an page table entry equal to `physical address/0x1000` value.
   I = Take the page entry index.
2. M = address of the start of this page table.
3. Search page directory table for an directory entry equal to `M/0x1000`.
   J = Take the directory index.

Logical address:

```
[31:22] : J
[21:12] : I
[11:0 ] : any offset.
```

Mathematically this can be done by: `J * 1024 * 4096 + I * 4096 + offset.`

Because there are two directory entires that point to the same page table, each page frame will have two addresses. In our case also there will be two addresses for the 1023rd page.

Directory entry 0:
Logical address for **0xB8000** = `0 * 1024 * 4096 + 1023 * 4096 = 0x003FF000.`

```
[31:22] : 0
[21:12] : 1023
[11:0 ] : 0
```

Directory entry 768:
Logical address for **0xB8000** = `768 * 1024 * 4096 + 1023 * 4096 = 0xC03FF000.`

```
[31:22] : 768
[21:12] : 1023
[11:0 ] : 0
```

So `0xC03FF000` and `0x003FF000` are both valid virtual addresses that point to the physical **0xB8000**.
Before the kernel_main is called, the identity mapping will be removed, so there will be only one mapping to **0xB8000**.