MEMORY MANAGEMENT

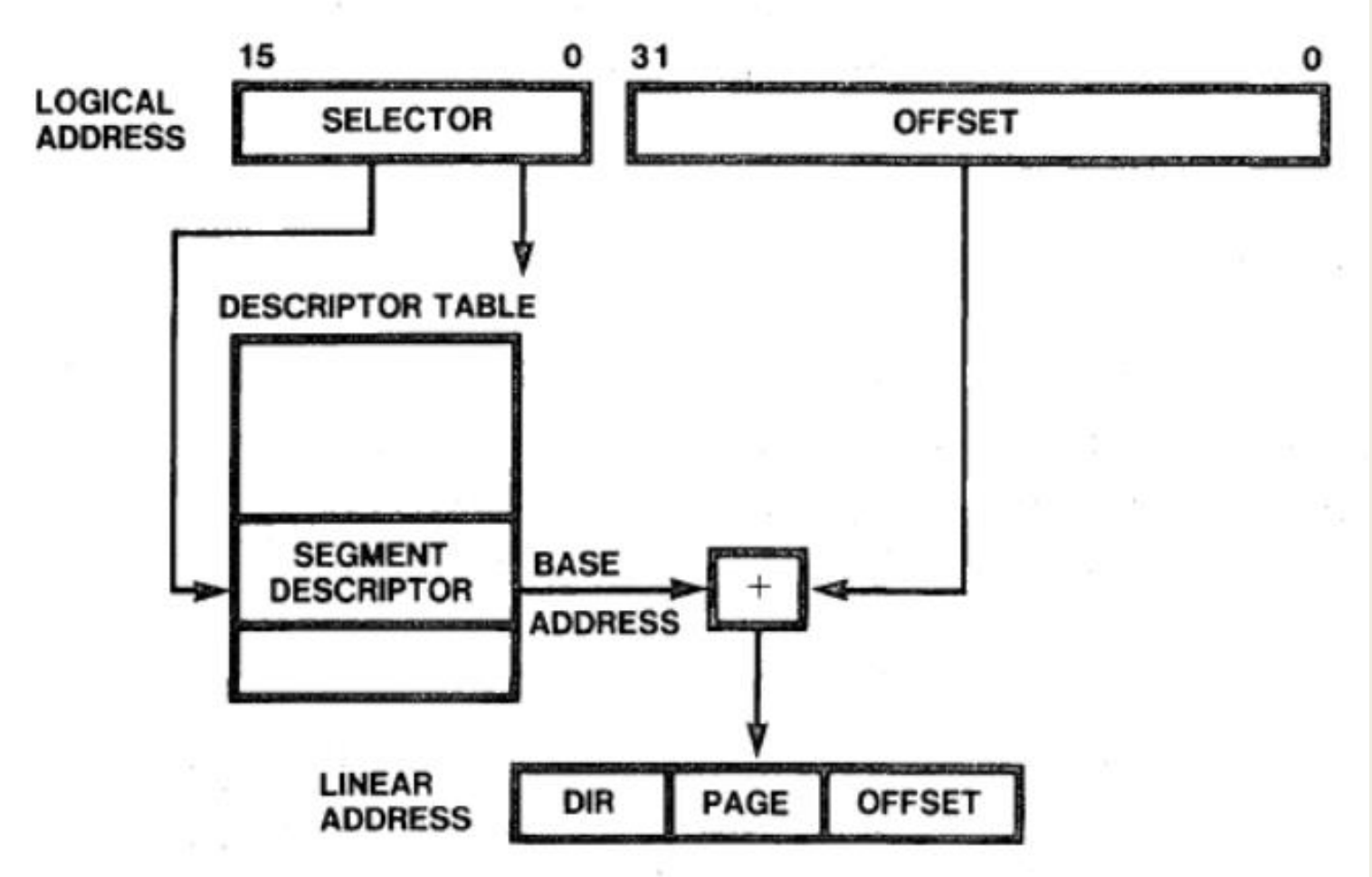
CONTENTS

- Segment Translation
- Page Translation
- Combining Segment and Page Translation.

- Memory management is a hardware mechanism which lets operating systems create simplified environments for running programs.
- For example, when several programs are running at the same time, they must each be given an independent address space.
- Memory management consists of segmentation and paging.
- Segmentation is used to give each program several independent, protected address spaces.
- Paging is used to support an environment where large address spaces are simulated using a small amount of RAM and some disk storage.
- Segmentation allows memory to be completely unstructured and simple, like the memory model of an 8-bit processor, or highly structured with address translation and protection. The memory management features apply to units called segments. Each segment is an independent, protected address space.
- Access to segments is controlled by data which describes its size, the privilege level required to access it, the kinds of memory references which can be made to it (instruction fetch, stack push or pop, read operation, write operation, etc.), and whether it is present in memory.

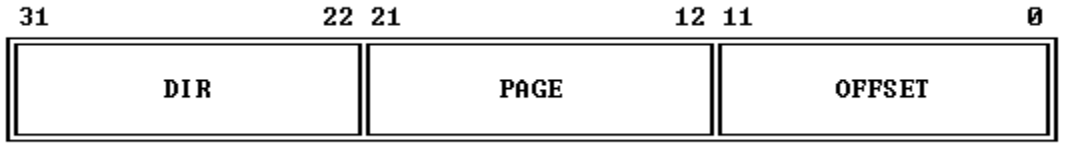
SEGMENT TRANSLATION

- A logical address consists of the 16-bit segment selector for its segment and a 32-bit offset into the segment.
- A logical address is translated into a linear address by adding the offset to the base address of the segment.
- The base address comes from the segment descriptor, a data structure in memory which provides the size and location of a segment, as well as access control information.
- The segment descriptor comes from one of two tables, the global descriptor table (GDT) or the local descriptor table (LDT).
- There is one GDT for all programs in the system, and one LDT for each separate program being run. If the operating system allows, different programs can share the same LDT. The system also may be set up with no LDTs; all programs may use the GDT.


15
0
31
0
LOGICAL ADDRESS
SELECTOR
OFFSET
DESCRIPTOR TABLE
SEGMENT DESCRIPTOR
BASE ADDRESS
+
LINEAR ADDRESS
DIR
PAGE
OFFSET

- A linear address refers indirectly to a physical address by specifying a page table, a page within that table, and an offset within that page. Figure 5-8 shows the format of a linear address.
- Figure below shows how the processor converts the DIR, PAGE, and OFFSET fields of a linear address into the physical address by consulting two levels of page tables.
- The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

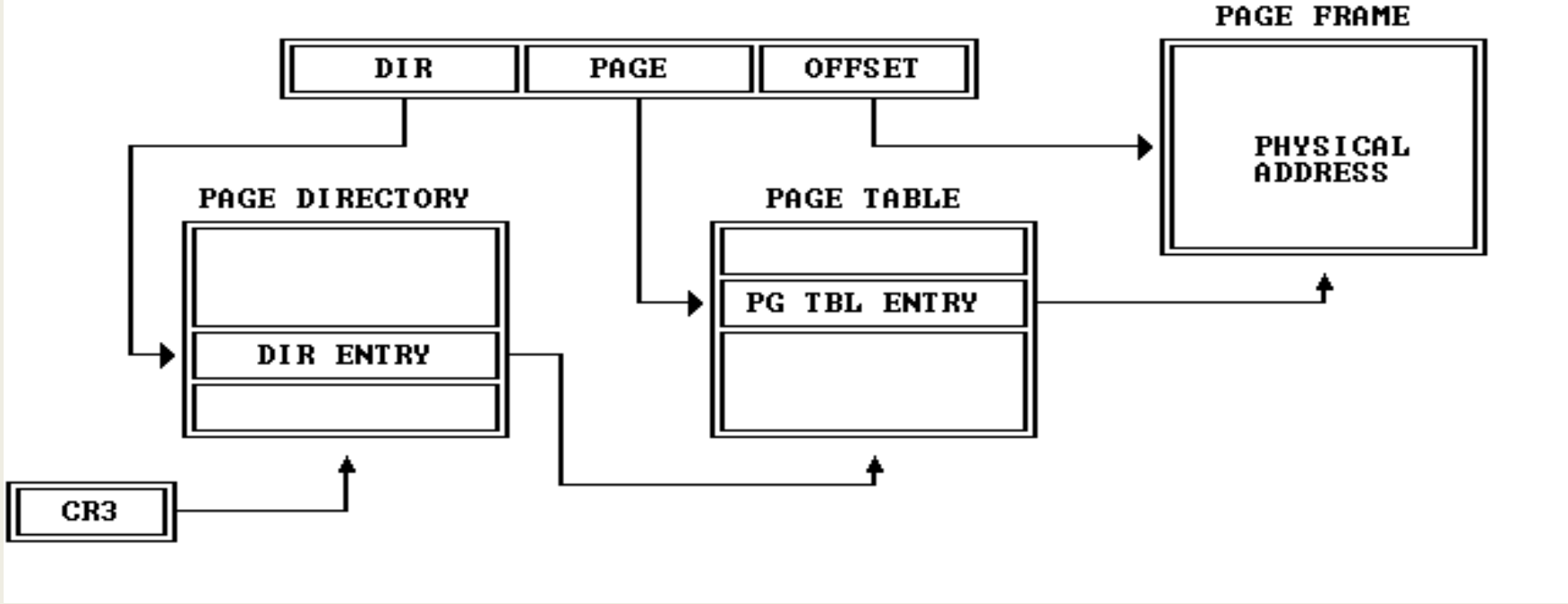
Figure 5-8. Format of a Linear Address



Page Translation

- When paging is used, a linear address is translated into its corresponding physical address.
- Paging is different from segmentation through its use of small, fixed-size pages. Unlike segments, which usually are the same size as the data structures they hold, on the 386 DX microprocessor, pages are always 4K bytes.
- If segmentation is the only form of address translation which is used, a data structure which is present in physical memory will have all of its parts in memory. If paging is used, a data structure may be partly in memory and partly in disk storage.

Figure 5-9. Page Translation



- The paging mechanism treats the 32-bit linear address as having three parts, two 10-bit indexes into the page tables and a 12-bit offset into the page addressed by the page tables. Because both the virtual pages in the linear address space and the physical pages of memory are aligned to 4K-byte page boundaries, there is no need to modify the low 12 bits of the address.

- Two levels of page tables are used. The top level page table is called the page directory. It maps the upper 10 bits of the linear address to the second level of page tables.

- The second level of page tables maps the middle 10 bits of the linear address to the base address of a page in physical memory (called a page frame address), or to an exception.

- The CR3 register holds the page frame address of the page directory. For this reason, it also is called the page directory base register or PDBR.

Page Tables

- A page table is an array of 32-bit entries. A page table is itself a page, and contains 4096 bytes of memory or, at most, 1K 32-bit entries.
- Two levels of tables are used to address a page of memory. The top level is called the page directory. It addresses up to 1K page tables in the second level. A page table in the second level addresses up to 1K pages in physical memory.
- The physical address of the current page directory is stored in the CR3 register, also called the **page directory base register (PDBR).**

Page-Table Entries

- **PAGE FRAME ADDRESS**: The page frame address is the base address of a page.
- **PRESENT BIT:** The Present bit indicates whether the page frame address in a page table entry maps to a page in physical memory. When set, the page is in memory. When the Present bit is clear, the page is not in memory.
- **ACCESSED AND DIRTY BITS:** These bits provide data about page usage in both levels of page tables. The Accessed bit is used to report read or write access to a page or second-level page table. The Dirty bit is used to report write access to a page.
- **READ/WRITE AND USER/SUPERVISOR BITS:** The Read/Write and User/Supervisor bits are used for protection checks applied to pages, which the processor performs at the same time as address translation.

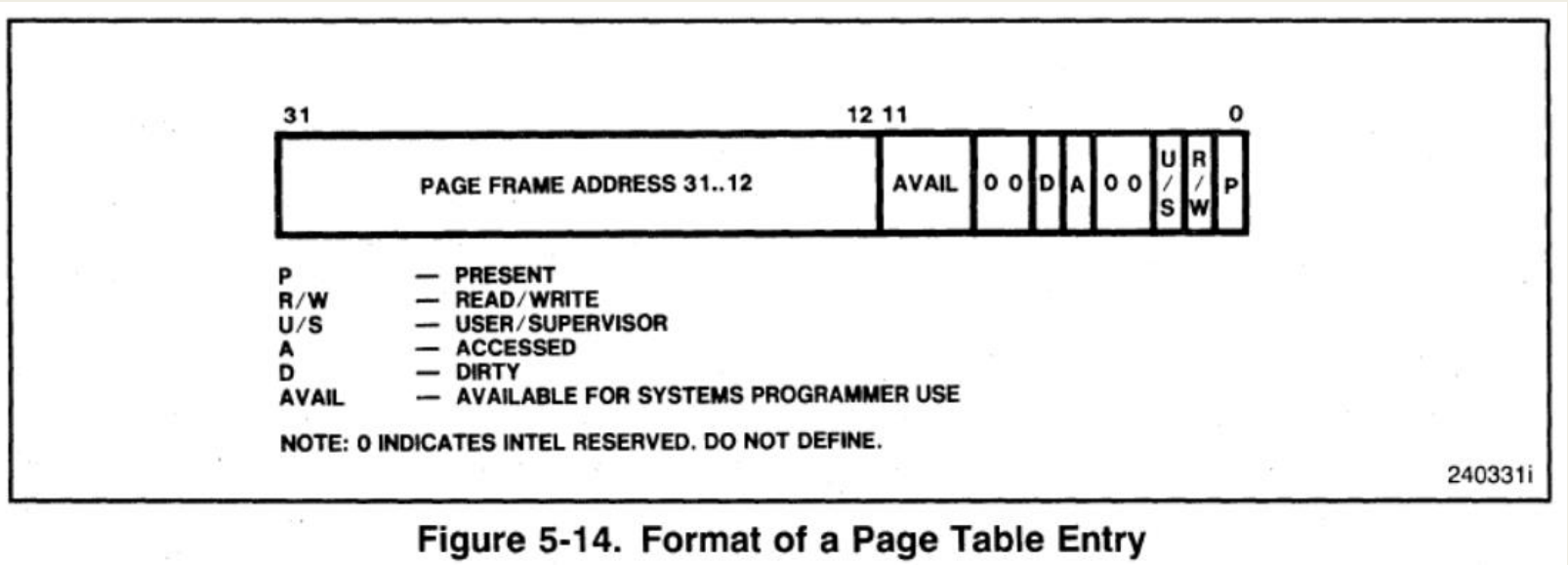


Figure 5-14. Format of a Page Table Entry

COMBINING SEGMENT AND PAGE TRANSLATION

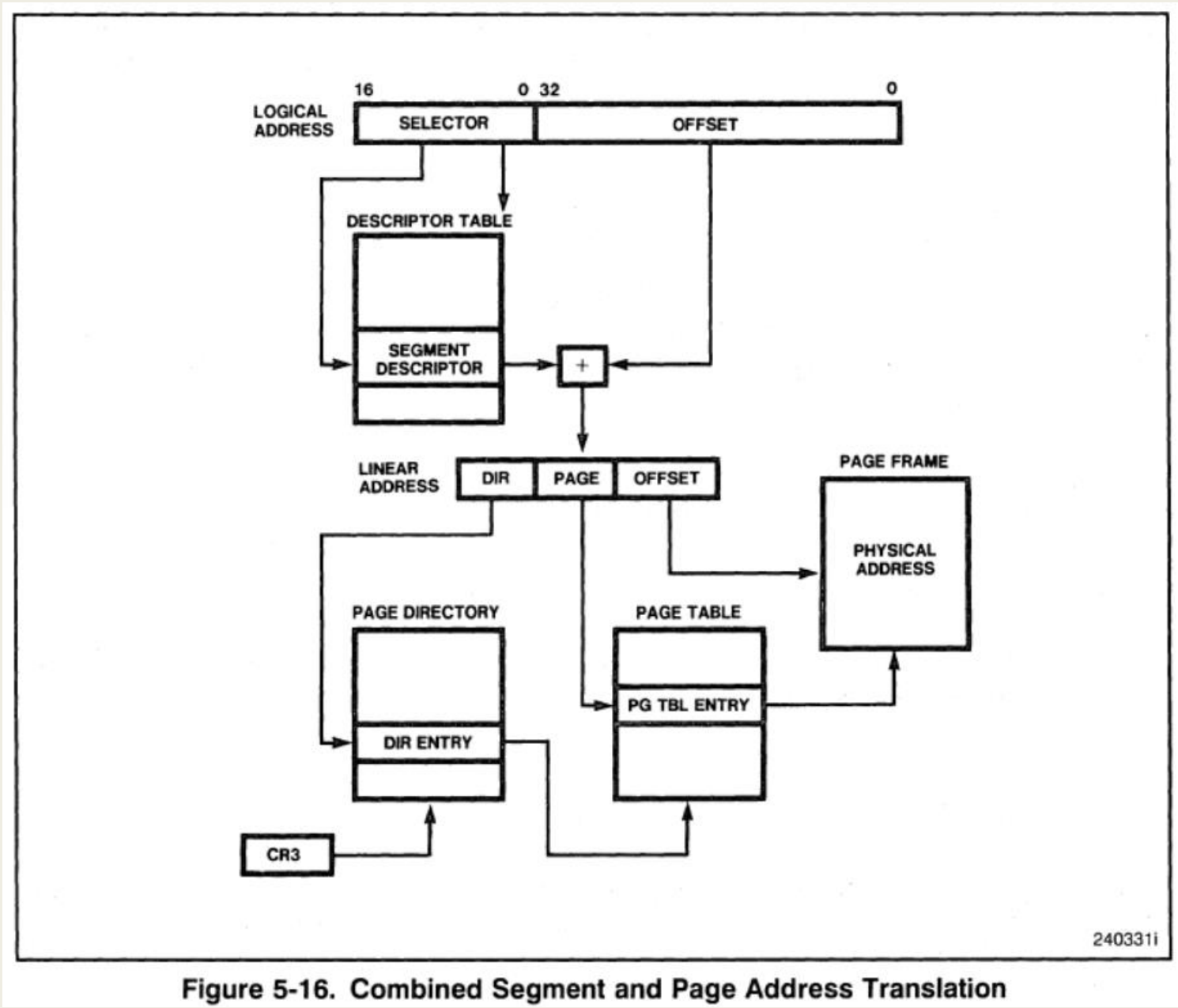


Figure 5-16. Combined Segment and Page Address Translation

PROTECTION

CONTENTS

- Need of Protection
- Overview of 80386DX Protection Mechanisms
- Segment Level Protection
- Page Level Protection
- Combining Segment and Page Level Protection

Need of Protection

- Protection is necessary for reliable multitasking.
- Protection can be used to prevent tasks from interfering with each other. For example, protection can keep one task from overwriting the instructions or data of another task.
- During program development, the protection mechanism can give a clearer picture of program bugs.
- When a program makes an unexpected reference to the wrong memory space, the protection mechanism can block the event and report its occurrence.
- In end-user systems, the protection mechanism can guard against the possibility of software failures caused by undetected program bugs.
- If a program fails, its effects can be confined to a limited domain. The operating system can be protected against damage, so diagnostic information can be recorded and automatic recovery may be attempted.

SEGMENT-LEVEL PROTECTION

- All five aspects of protection apply to segment translation:
  - *Type checking*
  - *Limit checking*
  - *Restriction of addressable domain*
  - *Restriction of procedure entry points*
  - *Restriction of instruction set*
- The segment is the unit of protection, and segment descriptors store protection parameters.
- Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access.
- Segment registers hold the protection parameters of the currently addressable segments.

Type Checking

- Type checking can be used to detect programming errors which would attempt to use segments in ways not intended by the programmer.

Table 6-1. System Segment and Gate Types

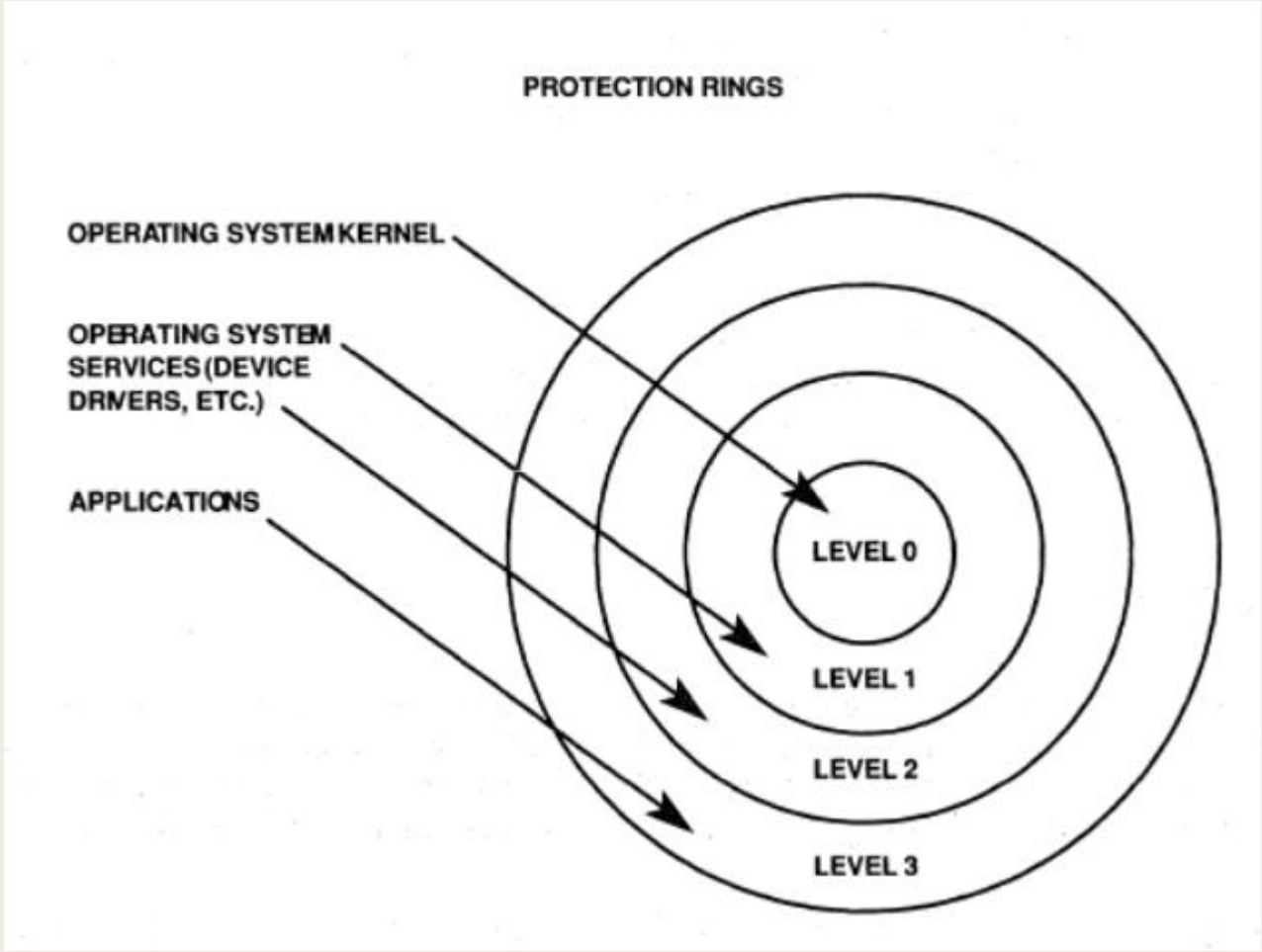
Type	Description
0	reserved
1	Available 80286 TSS
2	LDT
3	Busy 80286 TSS
4	Call Gate
5	Task Gate
6	80286 Interrupt Gate
7	80286 Trap Gate
8	reserved
9	Available 386™ DX TSS
10	reserved
11	Busy 386 DX TSS
12	386 DX Call Gate
13	reserved
14	386 DX Interrupt Gate
15	386 DX Task Gate

Limit Checking

- The Limit field of a segment descriptor prevents programs from addressing outside the segment. The effective value of the limit depends on the setting of the G bit (Granularity bit).
- For data segments, the limit also depends on the E bit (Expansion-Direction bit).
- The E bit is a designation for one bit of the Type field, when referring to data segment descriptors.
- Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier.
- Without limit checking, these errors could overwrite critical memory in another module.

Privilege Levels

- The protection mechanism recognizes four privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges.
- Privilege levels can be used to improve the reliability of operating systems. By giving the operating system the highest privilege level, it is protected from damage by bugs in other programs.



PAGE-LEVEL PROTECTION

- When the flat model for memory segmentation has been used, page-level protection prevents programs from interfering with each other.
- Each memory reference is checked to verify that it satisfies the protection checks.
- There are two page-level protection checks:
- 1. Restriction of addressable domain
- 2. Type checking

RESTRICTING ACCESS TO DATA

- To address operands in memory, a segment selector for a data segment must be loaded into a data-segment register (the OS, ES, FS, GS, or SS registers). The processor checks the segment's privilege levels.
- The three privilege levels which are checked are:

  1. The CPL (current privilege level) of the program. This is held in the two least significant bit positions of the CS register.

  2. The DPL (descriptor privilege level) of the segment descriptor of the segment containing the operand.

  3. The RPL (requestor's privilege level) of the selector used to specify the segment containing the operand.

MULTITASKING

CONTENTS

- Task State Segment
- TSS Descriptor
- Task Register
- Task Gate Descriptor
- Task Switching
- Task Linking
- Task Address Space.

Introduction

- A task is a program which is running, or waiting to run while another program is running.
- The registers and data structures which support multitasking are:
  - *Task state segment*
  - *Task state segment descriptor*
  - *Task register*
  - *Task gate descriptor*

Task State Segment

- The processor state information needed to restore a task is saved in a type of segment, called a task state segment or TSS.
- The fields of a TSS are divided into two main categories:

1. Dynamic fields the processor updates with each task switch. These fields store:

- The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI).
- The segment registers (ES, CS, SS, DS, FS, and OS).
- The flags register (EFLAGS).
- The instruction pointer (EIP).
- The selector for the TSS of the previous task (updated only when a return is expected).

2. Static fields the processor reads, but does not change. These fields are set up when a task is created. These fields store:

- The selector for the task's LDT.
- The logical address of the stacks for privilege levels 0, 1, and 2.
- The T-bit (debug trap bit) which, when set, causes the processor to raise a debug exception when a task switch occurs.
- The base address for the I/O permission bit map. If present, this map is stored in the TSS at higher addresses. The base address points to the beginning of the map.

TSS DESCRIPTOR

- The task state segment, like all other segments, is defined by a descriptor.

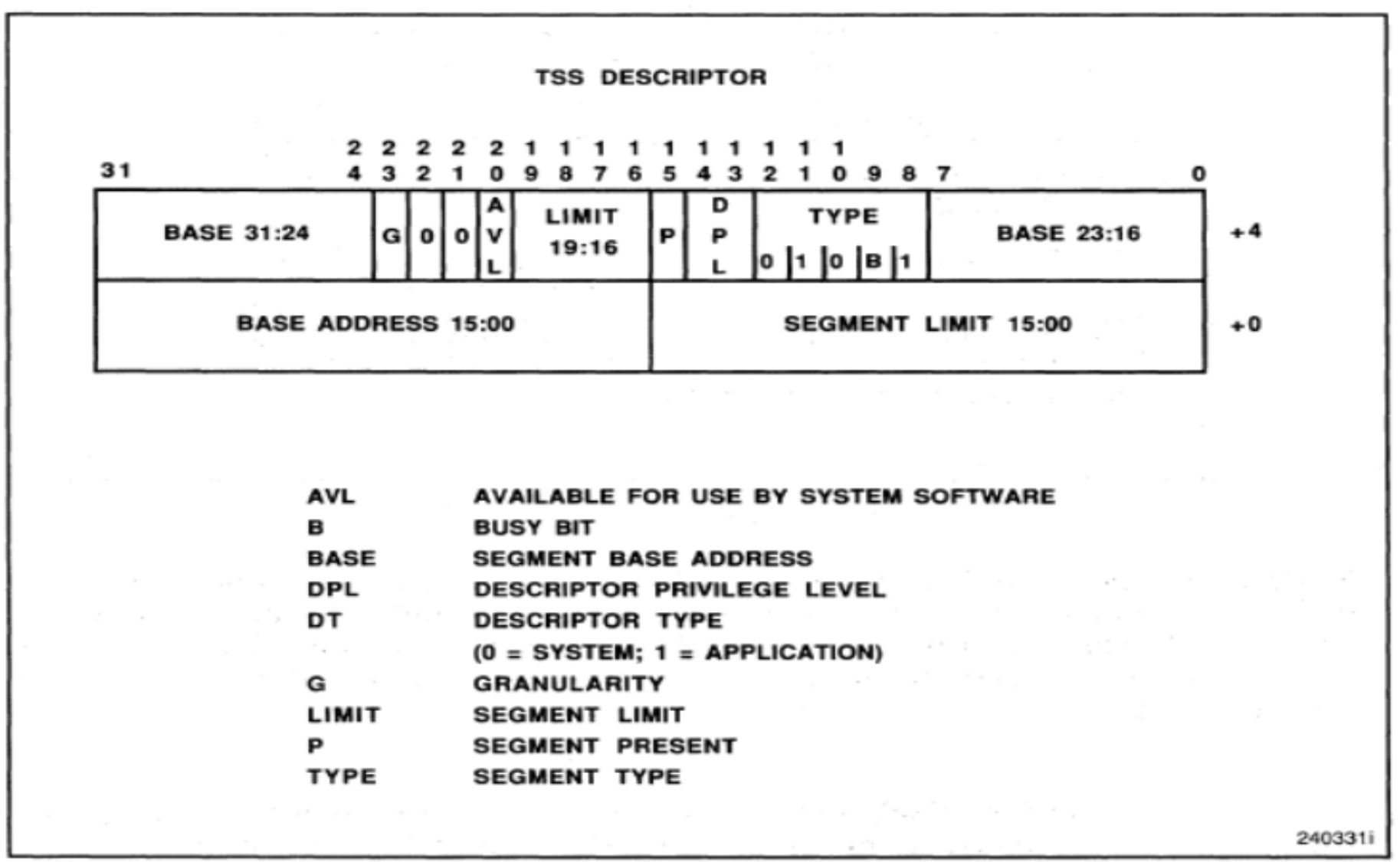


Figure 7-2. TSS Descriptor

- The Busy bit in the Type field indicates whether the task is busy. A busy task is currently running or waiting to run.
- A Type field with a value of 9 indicates an inactive task; a value of 11 (decimal) indicates a busy task.
- The Base, Limit, and DPL fields and the Granularity bit and Present bit have functions similar to their use in data-segment descriptors.
- The Limit field must have a value equal to or greater than 67H, one byte less than the minimum size of a task state.

Task Register

- The task register (TR) is used to find the current TSS.
- The task register has both a "visible" part (Le., a part which can be read and changed by software) and an "invisible" part (i.e., a part maintained by the processor and inaccessible to software).
- The selector in the visible portion indexes to a TSS descriptor in the GDT.
- The processor uses the invisible portion of the TR register to retain the base and limit values from the TSS descriptor. Keeping 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 and STR instructions are used to modify and read the visible portion of the task register.
  - ***LTR (Load task register)*** *loads the visible portion of the task register with the operand, which must index to a TSS descriptor in the GDT.*
  - ***STR (Store task register)*** *stores the visible portion of the task register in a general register or memory.*

Task Gate Descriptor

- A task gate descriptor provides an indirect, protected reference to a task.

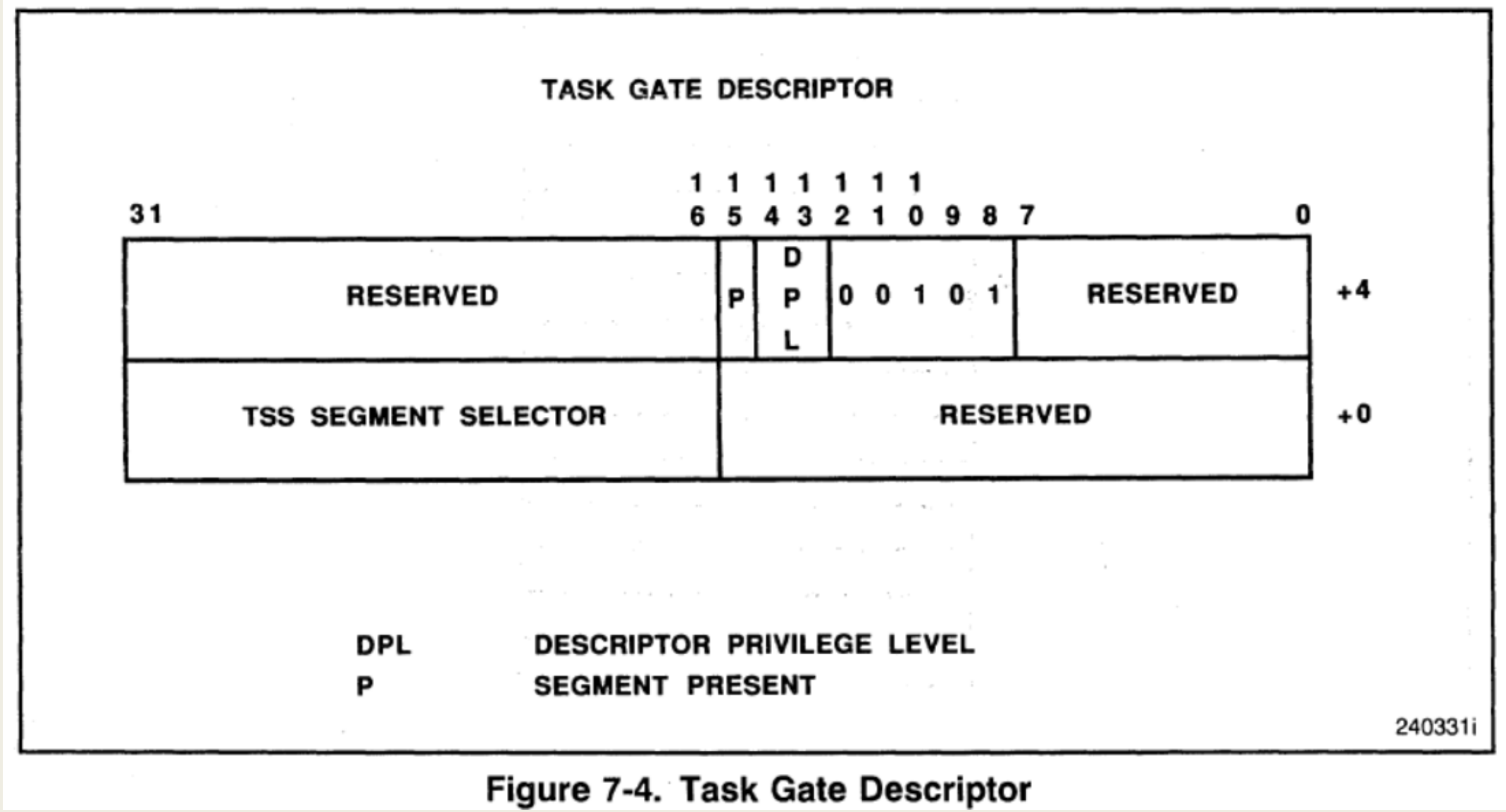


Figure 7-4. Task Gate Descriptor

- The Selector field of a task gate indexes to a TSS descriptor. The RPL in this selector is not used.
- The DPL of a task gate controls access to the descriptor for a task switch.
- A procedure with access to a task gate can cause a task switch, as can a procedure with access to a TSS descriptor.
- Both task gates and TSS descriptors are provided to satisfy three needs:
  - *The need for a task to have only one Busy bit.*
  - *The need to provide selective access to tasks.*
  - *The need for an interrupt or exception to cause a task switch.*

Task Switching

- The 386 DX microprocessor transfers execution to another task in any of four cases:
  1. *The current task executes a JMP or CALL to a TSS descriptor.*
  2. *The current task executes a JMP or CALL to a task gate.*
  3. *An interrupt or exception indexes to a task gate in the IDT.*
  4. *The current task executes an IRET when the NT flag is set.*
- To cause a task switch, a JMP or CALL instruction can transfer execution to either a TSS descriptor or a task gate.
- An exception or interrupt causes a task switch when it indexes to a task gate in the IDT.
- A task switch has following steps:

1. Check that the current task is allowed to switch to the new task.
2. Check that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
3. Save the state of the current task.
4. Load the TR register with the selector to the new task's TSS descriptor, set the new task's Busy bit, and set the TS bit in the CRO register.
5. Load the new task's state from its TSS and continue execution.

TASK LINKING

- The Link field of the TSS and the NT flag 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 Link field of the current task's TSS holds the TSS selector for the higher-level task, if there is one.

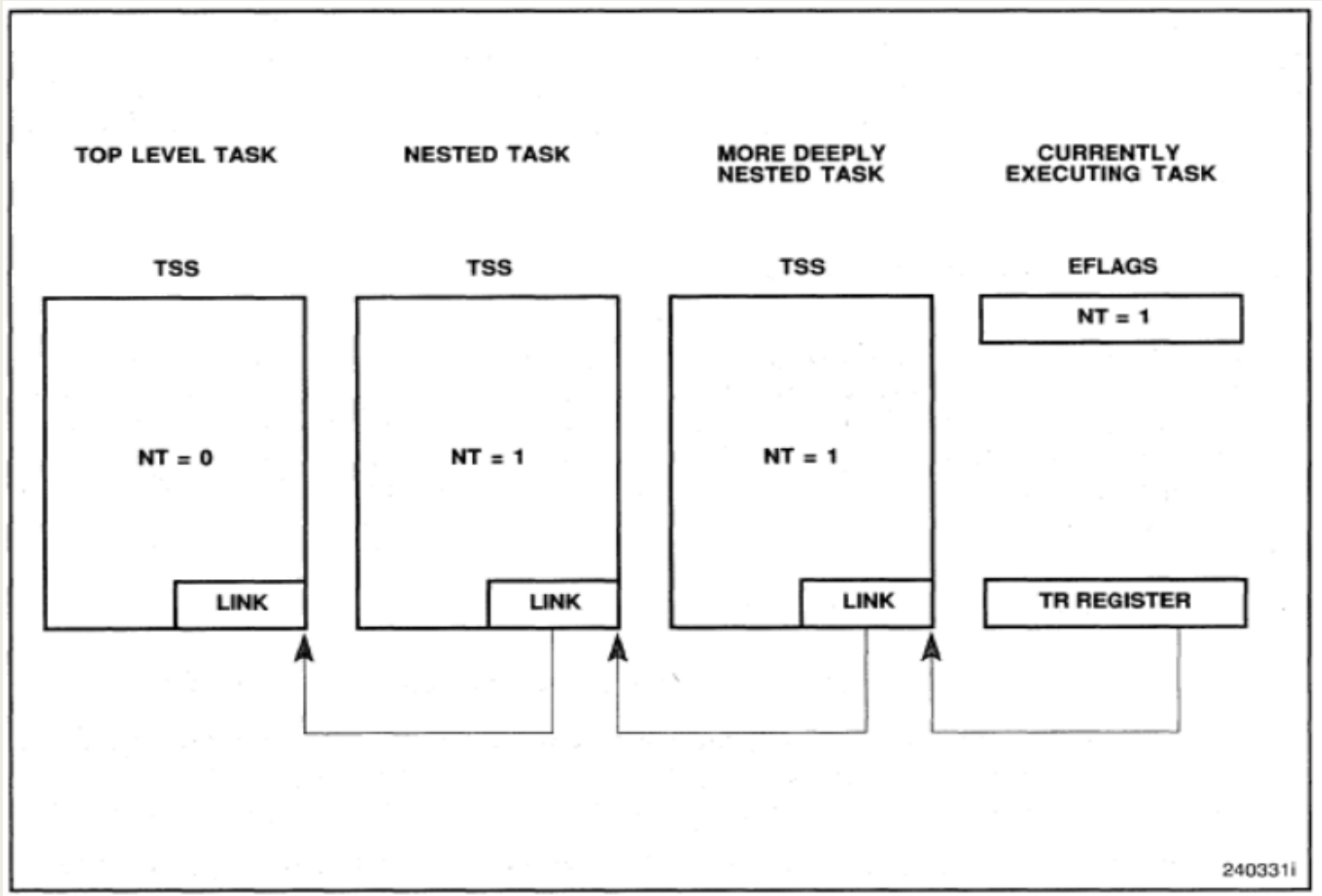


Figure 7-6. Nested Tasks

- When an interrupt, exception, jump, or call causes a task switch, the 386 DX microprocessor copies the segment selector for the current task state segment into the TSS for the new task and sets the NT flag.
- The NT flag indicates the Link field of the TSS has been loaded with a saved TSS selector.
- The new task releases control by executing an IRET instruction. When an IRET instruction is executed, the NT flag is checked.
- If it is set, the processor does a task switch to the previous task.

Busy Bit Prevents Loops

- The Busy bit of the TSS descriptor prevents re-entrant task switching.
- The processor manages the Busy bit as follows:

1. *When switching to a task, the processor sets the Busy bit of the new task.*

2. *When switching from a task, the processor clears the Busy bit of the old task if that task is not to be placed in the chain (i.e., the instruction causing the task switch is a JMP or IRET instruction). If the task is placed in the chain, its Busy bit remains set.*

3. *When switching to a task, the processor generates a general-protection exception if the Busy bit of the new task already is set.*

Modifying Task Linkages

- Modification of the chain of suspended tasks may be needed to resume an interrupted task before the task which interrupted it.
- A reliable way to do this is:

1. *Disable interrupts.*

2. *First change the Link field in the TSS of the interrupting task, then clear the Busy bit in the TSS descriptor of the task being removed from the chain.*

3. *Re-enable interrupts.*

TASK ADDRESS SPACE

Task Linear-to-Physical Space Mapping

- The choices for arranging the linear-to-physical mappings of tasks fall into two general classes:
- 1. One linear-to-physical mapping 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 mapping is obtained by using one page directory for all tasks. The linear space may exceed the available physical space if demand-paged virtual memory is supported.
- 2. Independent linear-to-physical mappings for each task. This form of mapping comes from using a different page directory for each task. Because the PDBR (page directory base register) is loaded from the TSS with each task switch, each task may have a different page directory.

Task logical Address Space

- To share data, tasks must also have a common logical-to-linear space mapping; i.e., they also must have access to descriptors which point into a shared linear address space.
- There are three ways to create shared logical-to-physical address-space mappings:

  1. Through the segment descriptors in the GDT. All tasks have access to the descriptors in the GDT. If those descriptors point into a linear-address space which is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.

  2. Through shared LDTs. Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT for use in address translation. Segment descriptors in the LDT addressing linear space mapped to overlapping physical space provide shared physical memory.

  3. Through segment descriptors in the LDTs which map to the same linear address space. If the linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share space. Such descriptors are commonly called "aliases."

INPUT/OUTPUT

CONTENTS

- I/O Addressing,
- I/O Instructions

Introduction

- Input/output is accomplished through I/O ports, which are registers connected to peripheral devices.
- An I/O port can be an input port, an output port, or a bidirectional port.
- Some I/O ports are used for carrying data, such as the transmit and receive registers of a serial interface.
- Other I/O ports are used to control peripheral devices, such as the control registers of a disk controller.

I/O ADDRESSING

- The 386 DX microprocessor allows I/O ports to be addressed in either of two ways:
  - *Through a separate I/O address space accessed using I/O instructions.*
  - *Through memory-mapped I/O, where I/O ports appear in the address space of physical memory.*
- The 386 DX microprocessor provides a separate I/O address space, distinct from the address space for physical memory, where I/O ports can be placed.
- The I/O address space consists of 21 (64K) individually addressable 8-bit ports; any two consecutive 8-bit ports can be treated as a 16-bit port, and any four consecutive ports can be a 32-bit port.
- A program can specify the address of a port in two ways. With an immediate byte constant, the program can specify:
  - *256 8-bit ports numbered 0 through 255.*
  - *128 16-bit ports numbered 0, 2, 4, … , 252, 254.*
  - *64 32-bit ports numbered 0, 4, 8, … , 248, 252.*
- Using a value in the DX register, the program can specify:
  - *8-bit ports numbered 0 through 65535.*
  - *16-bit ports numbered 0, 2, 4, … , 65532, 65534.*
  - *32-bit ports numbered 0, 4, 8, … , 65528, 65532.*

- The 386 DX microprocessor can transfer 8, 16, or 32 bits to a device in the I/O space.
- Like words in memory, 16-bit ports should be aligned to even addresses so that all 16 bits can be transferred in a single bus cycle. Like doublewords in memory, 32-bit ports should be aligned to addresses which are multiples of four.
- The processor supports data transfers to unaligned ports, but there is a performance penalty because an extra bus cycle must be used.
- The IN and OUT instructions move data between a register and a port in the I/O address space. The instructions INS and OUTS move strings of data between the memory address space and ports in the I/O address space.

Memory-Mapped I/O

- I/O devices may be placed in the address space for physical memory. This is called memory-mapped I/O.
- As long as the devices respond like memory components, they can be used with memory-mapped I/O.Memory-mapped I/O provides additional programming flexibility.
- Any instruction which references memory may be used to access an I/O port located in the memory space.
- For example, the MOV instruction can transfer data between any register and a port. The AND, OR, and TEST instructions may be used to manipulate bits in the control and status registers of peripheral devices (see Figure 8-1).
- Memory-mapped I/O can use the full instruction set and the full complement of addressing modes to address I/O ports.

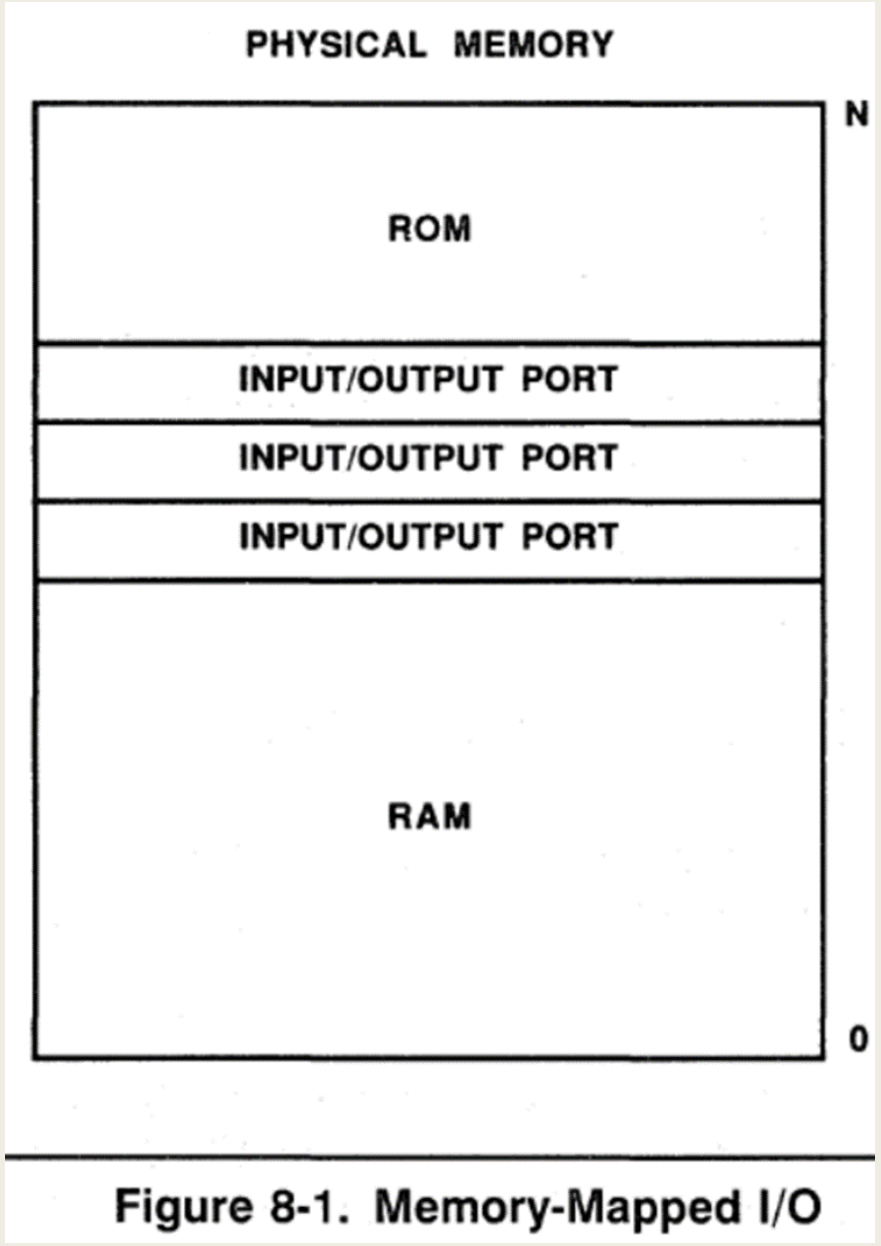


Figure 8-1. Memory-Mapped I/O

I/O INSTRUCTIONS

- The I/O instructions of the 386 DX microprocessor provide access to the processor's I/O ports for the transfer of data. These instructions have the address of a port in the I/O address space as an operand.
- There are two kinds of I/O instructions:

  1. Those which transfer a single item (byte, word, or doubleword) to or from a register.

  2. Those which transfer strings of items (strings of bytes, words, or doublewords) located in memory. These are known as "string I/O instructions" or "block I/O instructions. “

Register I/O Instructions

- The I/O instructions IN and OUT move data between I/O ports and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register.
- The IN and OUT instructions address I/O ports either directly, with the address of one of 256 port addresses coded in the instruction, or indirectly using an address in the DX register to select one of 64K port addresses.
- **IN (Input from Port)** transfers a byte, word, or doubleword from an input port to the AL, AX, or EAX registers.
  - *A byte IN instruction transfers 8 bits from the selected port to the AL register. A word IN instruction transfers 16 bits from the port to the AX register. A doubleword IN instruction transfers 32 bits from the port to the EAX register.*
- **OUT (Output from Port)** transfers a byte, word, or doubleword from the AL, AX, or EAX registers to an output port.
  - *A byte OUT instruction transfers 8 bits from the AL register to the selected port. A word OUT instruction transfers 16 bits from the AX register to the port. A doubleword OUT instruction transfers 32 bits from the EAX register to the port.*

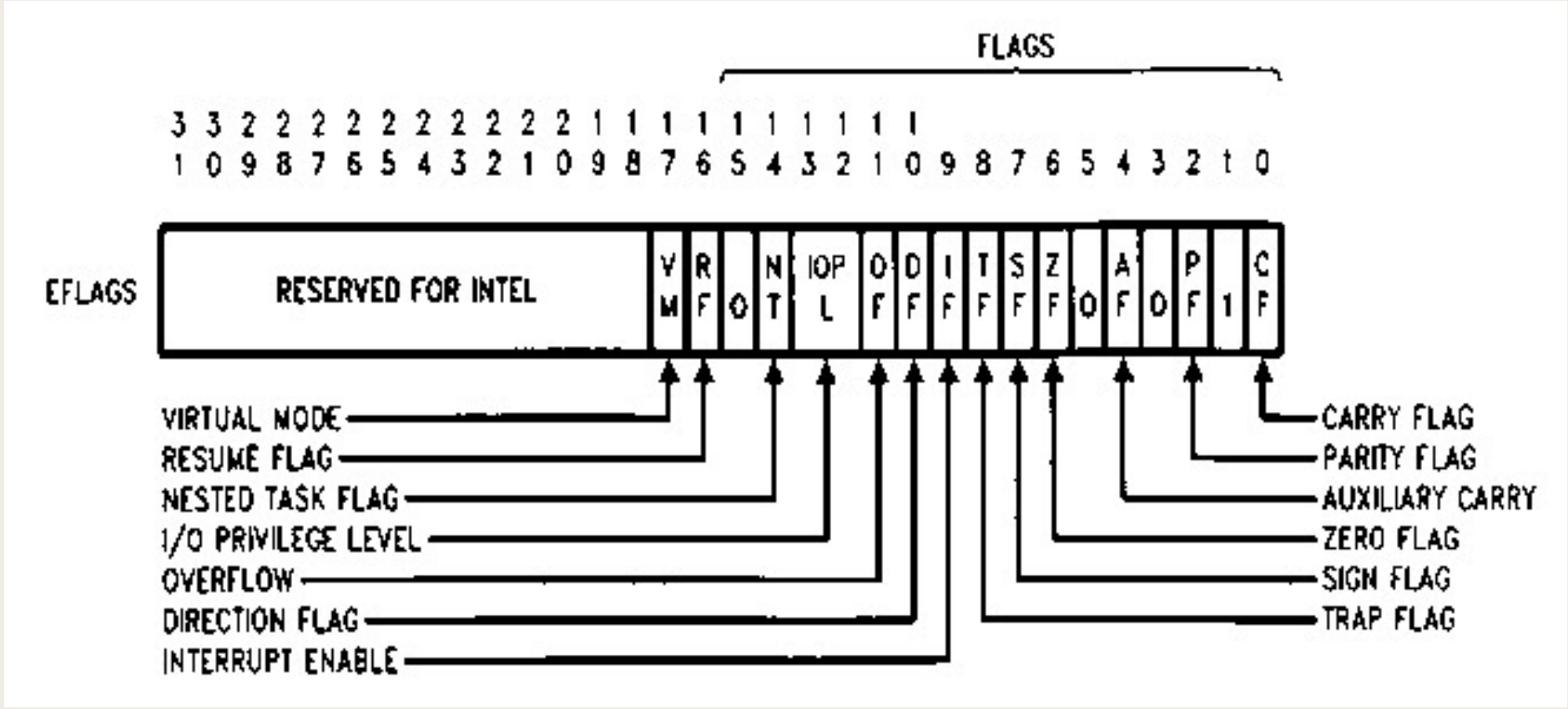
Block I/O Instructions

- The INS and OUTS instructions move blocks of data between I/O ports and memory.
- Block I/O instructions use an address in the DX register to address a port in the I/O address space. These instructions use the DX register to specify:
  - *8-bit ports numbered 0 through 65535.*
  - *16-bit ports numbered 0, 2, 4, … , 65532, 65534.*
  - *32-bit ports numbered 0, 4, 8, … , 65528, 65532.*
- Block I/O instructions use either the SI or DI register to address memory. For each transfer, the SI or DI register is incremented or decremented, as specified by the DF flag.
- **INS (Input String from Port)** transfers a byte, word, or doubleword string element from an input port to memory.
  - ***The INSB instruction*** *transfers a byte from the selected port to the memory location addressed by the ES and EDI registers.*
  - ***The INSW instruction*** *transfers a word.*
  - ***The INSD instruction*** *transfers a doubleword. A segment override prefix cannot be used to specify an alternate destination segment.*
  - *Combined with a REP prefix, an INS instruction makes repeated read cycles to the port, and puts the data into consecutive locations in memory.*

- OUTS (Output String from Port) transfers a byte, word, or doubleword string element from memory to an output port.
  - ***The OUTSB instruction*** *transfers a byte from the memory location addressed by the ES and EDI registers to the selected port.*
  - ***The OUTSW instruction*** *transfers a word.*
  - ***The OUTSD instruction*** *transfers a doubleword.*
  - *Combined with a REP prefix, an OUTS instruction reads consecutive locations in memory, and writes the data to an output port.*

PROTECTION AND I/O

- The I/O architecture has two protection mechanisms:
    1. The IOPL field in the EFLAGS register controls access to the I/O instructions.
    2. The I/O permission bit map of a TSS segment controls access to individual ports in the I/O address space.
- **I/O Privilege Level:** In systems where I/O protection is used, access to I/O instructions is controlled by the IOPL field in the EFLAGS register.
- This permits the operating system to adjust the privilege level needed to perform I/O.
- In a typical protection ring model, privilege levels 0 and 1 have access to the I/O instructions. This lets the operating system and the device drivers perform I/O, but keeps applications and less privileged device drivers from accessing the I/O address space.


FLAGS
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
EFLAGS
RESERVED FOR INTEL
VM
RF
0
NT
IOPL
OF
DF
IF
TF
SF
ZF
0
AF
0
PF
1
CF
VIRTUAL MODE
RESUME FLAG
NESTED TASK FLAG
I/O PRIVILEGE LEVEL
OVERFLOW
DIRECTION FLAG
INTERRUPT ENABLE
CARRY FLAG
PARITY FLAG
AUXILIARY CARRY
ZERO FLAG
SIGN FLAG
TRAP FLAG

- The following instructions can be executed only if CPL <= IOPL:
  - *IN*
  - *INS*
  - *OUT*
  - *OUTS*
  - *CLI*
  - *STI*
- These instructions are called "sensitive" instructions, because they are sensitive to the IOPL field.

I/O Permission Bit Map

- The 386 DX microprocessor can generate exceptions for references to specific I/O addresses. These addresses are specified in the I/O permission bit map in the TSS.
- The size of the map and its location in the TSS are variable. The processor finds the I/O permission bit map with the I/O map base address in the TSS. The base address is a 16-bit offset into the TSS. This is an offset to the beginning of the bit map. The limit of the TSS is the limit on the size of the I/O permission bit map.

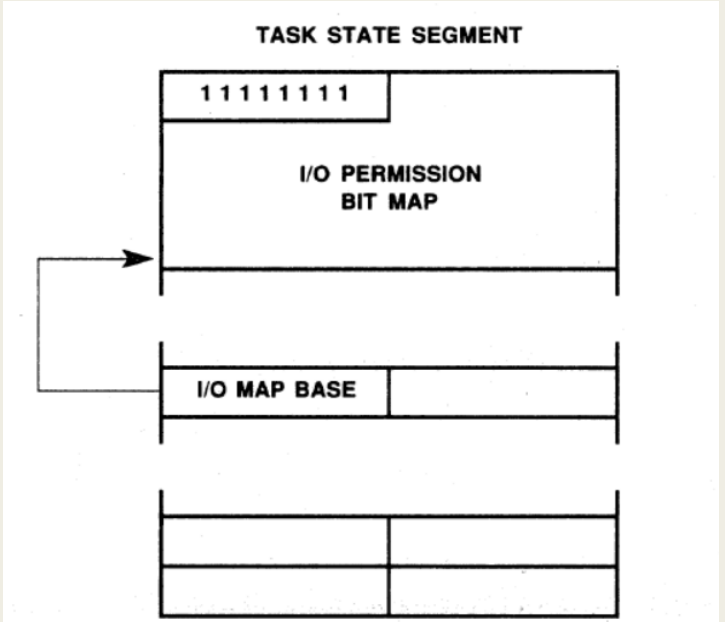


Figure 8-2. I/O Permission Bit Map

EXCEPTIONS AND INTERRUPTS

CONTENTS

- Identifying Interrupts
- Enabling and Disabling Interrupts
- Priority among Simultaneous Interrupts and Exceptions

- Exceptions and interrupts are forced transfers of execution to a task or a procedure. The task or procedure is called a handler.
- Interrupts occur at random times during the execution of a program, in response to signals from hardware.
- Exceptions occur when instructions are executed which provoke exceptions. Usually, the servicing of interrupts and exceptions is performed in a manner transparent to application programs.
- Interrupts are used to handle events external to the processor, such as requests to service peripheral devices.
- Exceptions handle conditions detected by the processor in the course of executing instructions, such division by 0.

There two sources for interrupts and two sources for exceptions:

1. **Interrupts**
   - *Maskable interrupts, which are received on the INTR input of the 386-DX microprocessor. Maskable interrupts do not occur unless the interrupt-enable flag (IF) is set.*
   - *Nonmaskable interrupts, which are received on the NMI (Non-Maskable Interrupt) input of the processor. The processor does not provide a mechanism to prevent nonmaskable interrupts.*
2. **Exceptions**
   - *Processor-detected exceptions. These are further classified as faults, traps, and aborts.*
   - *Programmed exceptions. The INTO, INT 3, INT n, and BOUND instructions may trigger exceptions. These instructions often are called "software interrupts," but the processor handles them as exceptions.*

EXCEPTION AND INTERRUPT VECTORS

- The processor associates an identifying number with each different type of interrupt or exception. This number is called a vector.
- The NMI interrupt and the exceptions are assigned vectors in the range 0 through 31.

Table 9-1. Exception and Interrupt Vectors

Vector Number	Description
0	Divide Error
1	Debug Exception
2	NMI Interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Coprocessor Not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Fault
13	General Protection
14	Page Fault
15	(Intel® reserved. Do not use.)
16	Coprocessor Error
17-31	(Intel reserved. Do not use.)
32-255	Maskable Interrupts

- Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether restart of the instruction which caused the exception is supported.
  - *Faults*
  - *Traps*
  - *Aborts*
- **Faults** are exceptions that are reported "before" the instruction causing the exception. Faults are either detected before the instruction begins to execute, or during execution of the instruction. If detected during the instruction, the fault is reported with the machine restored to a state that permits the instruction to be restarted.
- **A trap** is an exception which is reported at the instruction boundary immediately after the instruction in which the exception was detected.
- **An abort** is an exception which does not always report the location of the instruction causing the exception and does not allow restart of the program which caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

ENABLING AND DISABLING INTERRUPTS

- Certain conditions and flag settings cause the processor to inhibit certain kinds of interrupts and exceptions.
- **NMI Masks Further NMIs**

While an NMI interrupt handler is executing, the processor disables additional calls to the procedure or task which handles the interrupt until the next IRET instruction is executed. This prevents stacking up calls to the interrupt handler.

- **IF Masks INTR**

The IF flag can turn off servicing of interrupts received on the INTR pin of the processor. When the IF flag is clear, INTR interrupts are ignored; when the IF flag is set, INTR interrupts are serviced. The STI and CLI instructions set and clear the IF flag.

PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS

- If more than one exception or interrupt is pending at an instruction boundary, the processor services them in a predictable order.
- The processor first services a pending exception or interrupt from the class which has the highest priority, transferring execution to the first instruction of the handler.
- Lower priority exceptions are discarded; lower priority interrupts are held pending.
- Discarded exceptions are re-issued when the interrupt handler returns execution to the point of interruption.

INTERRUPT DESCRIPTOR TABLE

- The interrupt descriptor table (IDT) associates each exception or interrupt vector with a descriptor for the procedure or task which services the associated event.
- Like the GDT and LDTs, the IDT is an array of 8-byte descriptors.
- The IDT may reside anywhere in physical memory.
- As shown in figure, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.
- The LIDT and SIDT instructions load and store the contents of the IDTR register. Both instructions have one operand, which is the address of six bytes in memory.
- **LIDT (Load IDT register)** loads the IDTR register with the base address and limit held in the memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another.
- **SIDT (Store IDT register)** copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

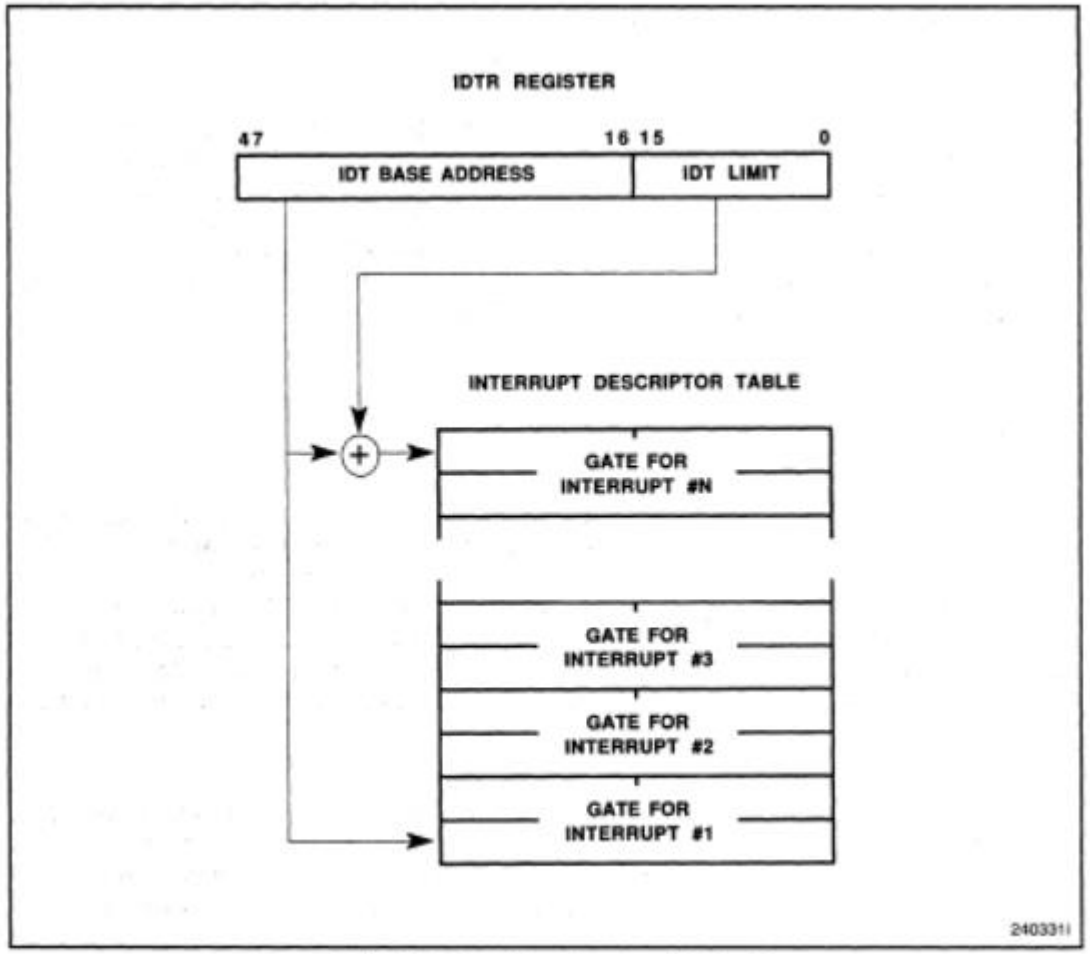
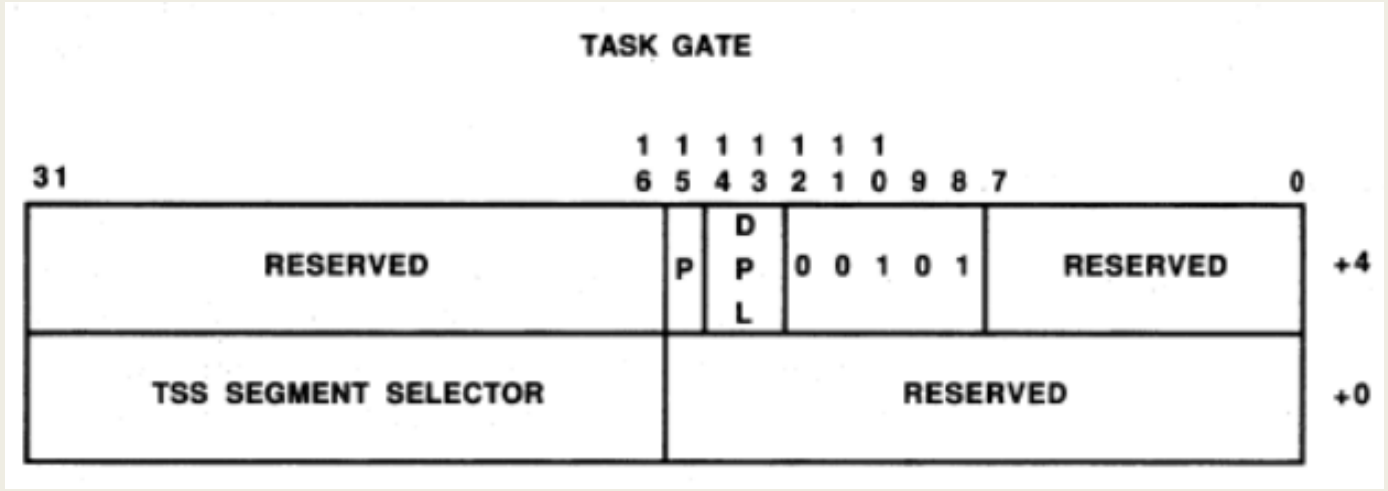
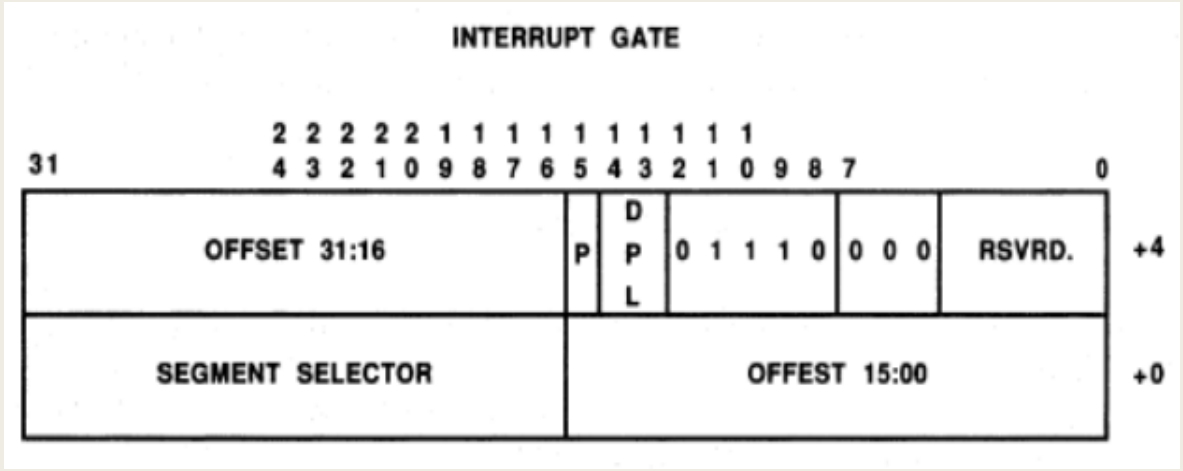


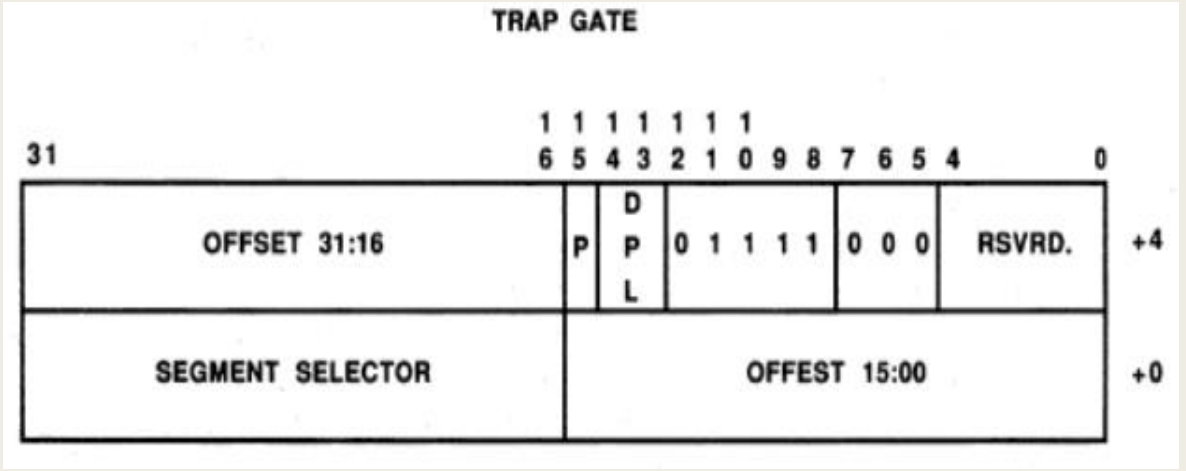
Figure 9-1. IDTR Register Locates IDT in Memory

IDT DESCRIPTORS

- The IDT may contain any of three kinds of descriptors:
  - *Task gates*
  - *Interrupt gates*
  - *Trap gates*







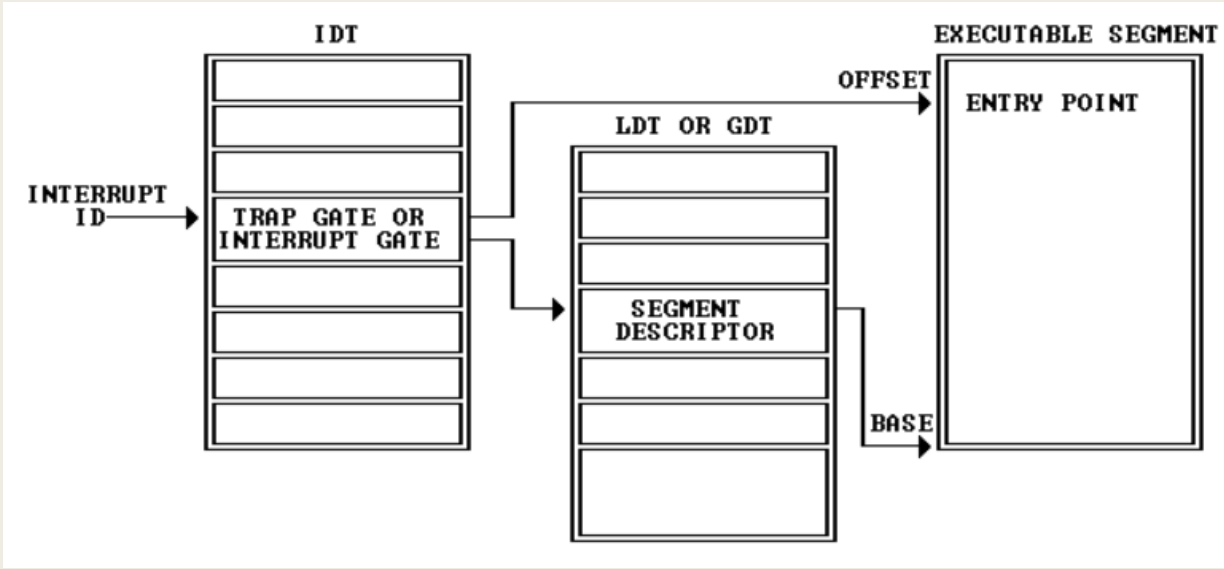
DPL	DESCRIPTOR PRIVILEGE LEVEL
OFFSET	OFFSET TO PROCEDURE ENTRY POINT
P	SEGMENT PRESENT BIT
RESERVED	DO NOT USE
SELECTOR	SEGMENT SELECTOR FOR DESTINATION CODE SEGMENT

INTERRUPT TASKS AND INTERRUPT PROCEDURES

- An exception or interrupt can "call" an interrupt handler as either a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector to index to a descriptor in the IDT. If the processor indexes to an interrupt gate or trap gate, it calls the handler in a manner similar to a CALL to a call gate. If the processor finds a task gate, it causes a task switch in a manner similar to a CALL to a task gate.

- **Interrupt Procedures**

An interrupt gate or trap gate indirectly references a procedure which runs in the context of the currently executing task, as shown in Figure. The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception or interrupt handling procedure.



Stack of Interrupt Procedure

- A transfer to an exception or interrupt handling procedure uses the stack to store the processor state.
- An interrupt pushes the contents of the EFLAGS register onto the stack before pushing the address of the interrupted instruction.
- Certain types of exceptions also push an error code on the stack.
- An exception handler can use the error code to help diagnose the exception.

RETURNING FROM AN INTERRUPT PROCEDURE

- An interrupt procedure differs from a normal procedure in the method of leaving the procedure.
- The IRET instruction is used to exit from an interrupt procedure.
- The IRET instruction is similar to the RET instruction except that it increments the contents of the EIP register by an extra four bytes and 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 CPL ≤ IOPL.

FLAG USAGE BY INTERRUPT PROCEDURE

- Interrupts that vector through either interrupt gates or trap gates cause TF (the trap flag) to be reset after the current value of TF is saved on the stack as part of EFLAGS. By this action the processor prevents debugging activity that uses single-stepping from affecting interrupt response. A subsequent IRET instruction restores TF to the value in the EFLAGS image on the stack.
- The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF.

Error Code

- With exceptions related to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task).
- The error code resembles a segment selector; however instead of an RPL field, the error code contains two one-bit fields:
    1. The processor sets the EXT bit if an event external to the program caused the exception.
    2. The processor sets the IDT bit if the index portion of the error code refers to a gate descriptor in the IDT.
- If the IDT bit is not set, the TI bit indicates whether the error code refers to the GDT (TI bit clear) or to the LDT (TI bit set).
- The remaining 14 bits are the upper bits of the selector for the segment.
- The error code is pushed on the stack as a doubleword. This is done to keep the stack aligned on addresses which are multiples of four. The upper half of the doubleword is reserved.

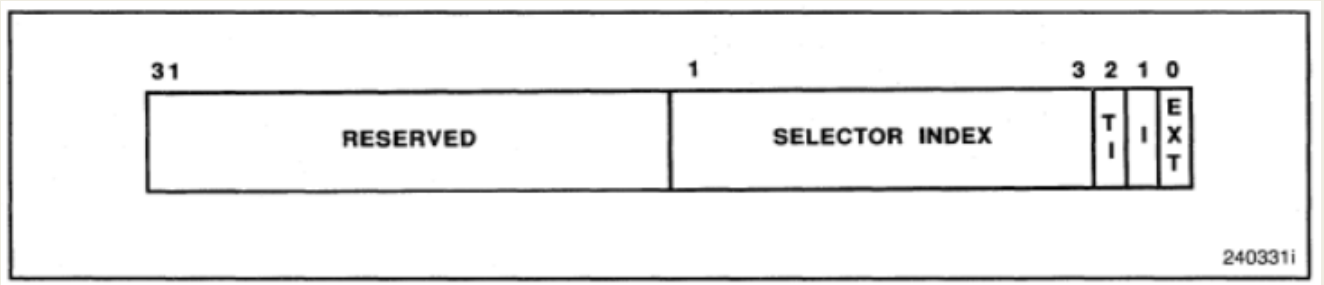


Figure 9-6. Error Code

Exception Conditions

- **Interrupt 0-Divide Error**

  The divide-error fault occurs during a DIV or an IDIV instruction when the divisor is 0.

- **Interrupt 1 - Debug Exceptions**

  The processor generates a debug exception for a number of conditions; whether the exception is a fault or a trap depends on the condition, as shown below:

  - *Instruction address breakpoint fault*
  - *Data address breakpoint trap*
  - *General detect fault*
  - *Single-step trap*
  - *Task-switch breakpoint trap*

- **Interrupt 3 – Breakpoint**

  The INT 3 instruction generates a breakpoint trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task.

- **Interrupt 4 -- Overflow**

  This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor cannot determine which is intended and therefore does not cause overflow exceptions automatically. Instead it merely sets OF when the results, if interpreted as signed numbers, would be out of range.

- **Interrupt 5 - Bounds Check**

  The bounds-check fault is generated when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A program can use the BOUND instruction to check a signed array index against signed limits defined in a block of memory.

- **Interrupt 6 -- Invalid Opcode**

  This fault occurs when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e., prefetching an invalid opcode does not cause this exception.) No error code is pushed on the stack. The exception can be handled within the same task.

  This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP referencing a register operand, or an LES instruction with a register source operand.

- **Interrupt 7-Coprocessor Not Available**
- The coprocessor-not-available fault is generated by either of two conditions:
- • The processor executes an ESC instruction, and the EM bit of the CRO register is set.
- • The processor executes a WAIT instruction or an ESC instruction, and both the MP bit and the TS bit of the CRO register are set.
- **Interrupt 8 - Double Fault**
- When the processor detects an exception while trying to call the handler for a prior exception, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception instead.
- To determine when two faults are to be signalled as a double fault, the 386 DX microprocessor divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults.

- Khgjngfhjng

Table 9-3. Interrupt and Exception Classes

Class	Vector Number	Description
Benign Exceptions and Interrupts	1	Debug Exceptions
	2	NMI Interrupt
	3	Breakpoint
	4	Overflow
	5	Bounds Check
	6	Invalid Opcode
	7	Coprocessor Not Available
	16	Coprocessor Error
Contributory Exceptions	0	Divide Error
	9	Coprocessor Segment Overrun
	10	Invalid TSS
	11	Segment Not Present
	12	Stack Fault
	13	General Protection
Page Faults	14	Page Fault

- **Interrupt 9-Coprocessor Segment Overrun**

  The coprocessor-segment overrun abort is generated if the middle portion of a coprocessor operand is protected or not-present. This exception can be avoided.

- **Interrupt 10 – Invalid TSS**

  Interrupt 10 occurs if during a task switch the new TSS is invalid. An error code is pushed onto the stack to help identify the cause of the fault. The EXT bit indicates whether the exception was caused by a condition outside the control of the program; e.g., an external interrupt via a task gate triggered a switch to an invalid TSS.

- **Interrupt 11 – Segment Not Present**

  Exception 11 occurs when the processor detects that the present bit of a descriptor is zero. The processor can trigger this fault in any of these cases:

  - *While attempting to load the CS, DS, ES, FS, or GS registers; loading the SS register, however, causes a stack fault.*
  - *While attempting loading the LDT register with an LLDT instruction; loading the LDT register during a task switch operation, however, causes the "invalid TSS" exception.*
  - *While attempting to use a gate descriptor that is marked not-present.*

- **Interrupt 12 – Stack Exception**

  A stack fault occurs in either of two general conditions:

  - *As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER, and LEAVE, as well as other memory references that implicitly use SS (for example, MOV AX, [BP+6]). ENTER causes this exception when the stack is too small for the indicated local-variable space.*
  - *When attempting to load the SS register with a descriptor that is marked not-present but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, an LSS instruction, or a MOV or POP instruction to SS.*

When the processor detects a stack exception, it pushes an error code onto the stack of the exception handler. If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment in question (the exception handler can test the present bit in the descriptor to determine which exception occurred); otherwise the error code is zero.

- **Interrupt 13 – General Protection Exception**

All protection violations that do not cause another exception cause a general protection exception. This includes (but is not limited to):

- *Exceeding segment limit when using CS, DS, ES, FS, or GS*
- *Exceeding segment limit when referencing a descriptor table*
- *Transferring control to a segment that is not executable*
- *Writing into a read-only data segment or into a code segment*
- *Reading from an execute-only segment*
- *Loading the SS register with a read-only descriptor (unless the selector comes from the TSS during a task switch, in which case a TSS exception occurs*
- *Loading SS, DS, ES, FS, or GS with the descriptor of a system segment*
- *Loading DS, ES, FS, or GS with the descriptor of an executable segment that is not also readable*
- *Loading SS with the descriptor of an executable segment*
- *Accessing memory via DS, ES, FS, or GS when the segment register contains a null selector*
- *Switching to a busy task*
- *Violating privilege rules*
- *Loading CR0 with PG=1 and PE=0.*
- *Interrupt or exception via trap or interrupt gate from V86 mode to privilege level other than zero.*
- *Exceeding the instruction length limit of 15 bytes (this can occur only if redundant prefixes are placed before an instruction)*

- **Interrupt 14 – Page Fault**
- This exception occurs when paging is enabled (PG=1) and the processor detects one of the following conditions while translating a linear address to a physical address:
  - *The page-directory or page-table entry needed for the address translation has zero in its present bit.*
  - *The current procedure does not have sufficient privilege to access the indicated page.*
- The processor makes available to the page fault handler two items of information that aid in diagnosing the exception and recovering from it:
- An error code on the stack. The error code for a page fault has a format different from that for other exceptions. The error code tells the exception handler three things:
  - *Whether the exception was due to a not present page or to an access rights violation.*
  - *Whether the processor was executing at user or supervisor level at the time of the exception.*
  - *Whether the memory access that caused the exception was a read or write.*
- CR2 (control register two). The processor stores in CR2 the linear address used in the access that caused the exception (see Figure 9-9). The exception handler can use this address to locate the corresponding page directory and page table entries. If another page fault can occur during execution of the page fault handler, the handler should push CR2 onto the stack.

- **Interrupt 16-Coprocessor Error**

A coprocessor-error fault is generated when the processor detects a signal from the 387™ DX numerics coprocessor on the ERROR# pin. If the EM bit of the CRO register is clear (no emulation), the processor tests this pin at the beginning of certain ESC instructions or when it executes a WAIT instruction.