up: Chapter 9 -- Exceptions and Interrupts
prev: 9.7 Error Code
next: 9.9 Exception Summary

9.8 Exception Conditions

The following sections describe each of the possible exception conditions in detail. Each description classifies the exception as a fault, trap, or abort. This classification provides information needed by systems programmers for restarting the procedure in which the exception occurred:

Faults
: The CS and EIP values saved when a fault is reported point to the instruction causing the fault.

Traps
: The CS and EIP values stored when the trap is reported point to the instruction dynamically after the instruction causing the trap. If a trap is detected during an instruction that alters program flow, the reported values of CS and EIP reflect the alteration of program flow. For example, if a trap is detected in a JMP instruction, the CS and EIP values pushed onto the stack point to the target of the JMP, not to the instruction after the JMP.

Aborts
: An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

9.8.1 Interrupt 0 -- Divide Error

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

9.8.2 Interrupt 1 -- Debug Exceptions

The processor triggers this interrupt for any of a number of conditions; whether the exception is a fault or a trap depends on the condition:

- Instruction address breakpoint fault.
- Data address breakpoint trap.
- General detect fault.
- Single-step trap.

- Task-switch breakpoint trap.

The processor does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception . Refer to Chapter 12 for more detailed information about debugging and the debug registers.

9.8.3 Interrupt 3 -- Breakpoint

The INT 3 instruction causes this trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in an executable segment with the breakpoint opcode. The operating system or a debugging subsystem can use a data-segment alias for an executable segment to place an INT 3 anywhere it is convenient to arrest normal execution so that some sort of special processing can be performed. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task.

The saved CS:EIP value points to the byte following the breakpoint. If a debugger replaces a planted breakpoint with a valid opcode, it must subtract one from the saved EIP value before returning . Refer also to Chapter 12 for more information on debugging.

9.8.4 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. When doing arithmetic on signed operands, careful programmers and compilers either test OF directly or use the INTO instruction.

9.8.5 Interrupt 5 -- Bounds Check

This fault occurs 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.

9.8.6 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.

9.8.7 Interrupt 7 -- Coprocessor Not Available

This exception occurs in either of two conditions:

- The processor encounters an ESC (escape) instruction, and the EM (emulate) bit ofCR0 (control register zero) is set.
- The processor encounters either the WAIT instruction or an ESC instruction, and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.

Refer to Chapter 11 for information about the coprocessor interface .

9.8.8 Interrupt 8 -- Double Fault

Normally, when the processor detects an exception while trying to invoke 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 80386 divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults. Table 9-3 shows this classification.

Table 9-4 shows which combinations of exceptions cause a double fault and which do not.

The processor always pushes an error code onto the stack of the double-fault handler; however, the error code is always zero. The faulting instruction may not be restarted. If any other exception occurs while attempting to invoke the double-fault handler, the processor shuts down.

Table 9-3. Double-Fault Detection Classes

Class	ID	Description
	1	Debug exceptions
	2	NMI
	3	Breakpoint

Benign Exceptions	4	Overflow
	5	Bounds check
	6	Invalid opcode
	7	Coprocessor not available
	16	Coprocessor error
	0	Divide error
	9	Coprocessor Segment Overrun
Contributory Exceptions	10	Invalid TSS
	11	Segment not present
	12	Stack exception
	13	General protection
Page Faults	14	Page fault

Table 9-4. Double-Fault Definition

		SECOND EXCEPTION		
		Benign Exception	Contributory Exception	Page Fault
FIRST EXCEPTION	Benign Exception	OK	OK	OK
	Contributory Exception	OK	DOUBLE	OK
	Page Fault	OK	DOUBLE	DOUBLE

9.8.9 Interrupt 9 -- Coprocessor Segment Overrun

This exception is raised in protected mode if the 80386 detects a page or segment violation while transferring the middle portion of a coprocessor operand to the NPX . This exception is avoidable. Refer to Chapter 11 for more information about the coprocessor interface.

9.8.10 Interrupt 10 -- Invalid TSS

Interrupt 10 occurs if during a task switch the new TSS is invalid. A TSS is considered invalid in the cases shown in Table 9-5. 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.

This fault can occur either in the context of the original task or in the context of the new task. Until the processor has completely verified the presence of the new TSS, the exception occurs in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete; i.e., TR is updated and, if the switch is due to a CALL or interrupt, the backlink of the new TSS is set to the old TSS. Any errors discovered by the processor after this point are handled in the context of the new task.

To insure a proper TSS to process it, the handler for exception 10 must be a task invoked via a task gate.

Table 9-5. Conditions That Invalidate the TSS

Error Code	Condition
TSS id + EXT	The limit in the TSS descriptor is less than 103
LTD id + EXT	Invalid LDT selector or LDT not present
SS id + EXT	Stack segment selector is outside table limit
SS id + EXT	Stack segment is not a writable segment
SS id + EXT	Stack segment DPL does not match new CPL
SS id + EXT	Stack segment selector RPL < > CPL
CS id + EXT	Code segment selector is outside table limit
CS id + EXT	Code segment selector does not refer to code segment
CS id + EXT	DPL of non-conforming code segment < > new CPL
CS id + EXT	DPL of conforming code segment > new CPL
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS segment selector is outside table limits
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS is not readable segment

9.8.11 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.

This fault is restartable. If the exception handler makes the segment present and returns, the interrupted program will resume execution.

If a not-present exception occurs during a task switch, not all the steps of the task switch are complete. During a task switch, the processor first loads all the

segment registers, then checks their contents for validity. If a not-present exception is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory. The not-present handler should not rely on being able to use the values found in CS, SS, DS, ES, FS, and GS without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that make diagnosis more difficult. There are three ways to handle this case:

1. Handle the not-present fault with a task. The task switch back to the interrupted task will cause the processor to check the registers as it loads them from the TSS.
2. PUSH and POP all segment registers. Each POP causes the processor to check the new contents of the segment register.
3. Scrutinize the contents of each segment-register image in the TSS, simulating the test that the processor makes when it loads a segment register.

This exception pushes an error code onto the stack. The EXT bit of the error code is set if an event external to the program caused an interrupt that subsequently referenced a not-present segment. The I-bit is set if the error code refers to an IDT entry, e.g., an INT instruction referencing a not-present gate.

An operating system typically uses the "segment not present" exception to implement virtual memory at the segment level. A not-present indication in a gate descriptor, however, usually does not indicate that a segment is not present (because gates do not necessarily correspond to segments). Not-present gates may be used by an operating system to trigger exceptions of special significance to the operating system.

9.8.12 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.

An instruction that causes this fault is restartable in all cases. The return pointer pushed onto the exception handler's stack points to the instruction that needs to be restarted. This instruction is usually the one that caused the exception; however, in the case of a stack exception due to loading of a not-present stack-segment descriptor during a task switch, the indicated instruction is the first instruction of the new task.

When a stack fault occurs during a task switch, the segment registers may not be usable for referencing memory. During a task switch, the selector values are loaded before the descriptors are checked. If a stack fault is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory. The stack fault handler should not rely on being able to use the values found in CS, SS, DS, ES, FS, and GS without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that make diagnosis more difficult.

9.8.13 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):

1. Exceeding segment limit when using CS, DS, ES, FS, or GS
2. Exceeding segment limit when referencing a descriptor table
3. Transferring control to a segment that is not executable
4. Writing into a read-only data segment or into a code segment
5. Reading from an execute-only segment
6. 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
7. Loading SS, DS, ES, FS, or GS with the descriptor of a system segment
8. Loading DS, ES, FS, or GS with the descriptor of an executable segment that is not also readable
9. Loading SS with the descriptor of an executable segment
10. Accessing memory via DS, ES, FS, or GS when the segment register

contains a null selector
11. Switching to a busy task
12. Violating privilege rules
13. Loading CR0 with PG=1 and PE=0.
14. Interrupt or exception via trap or interrupt gate from V86 mode to privilege level other than zero.
15. Exceeding the instruction length limit of 15 bytes (this can occur only if redundant prefixes are placed before an instruction)

The general protection exception is a fault. In response to a general protection exception, the processor pushes an error code onto the exception handler's stack. If loading a descriptor causes the exception, the error code contains a selector to the descriptor; otherwise, the error code is null. The source of the selector in an error code may be any of the following:

1. An operand of the instruction.
2. A selector from a gate that is the operand of the instruction.
3. A selector from a TSS involved in a task switch.

9.8.14 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.

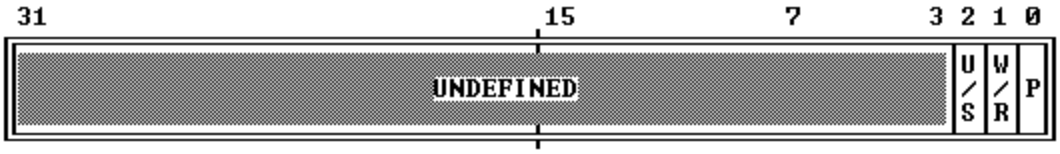
The processor makes available to the page fault handler two items of information that aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 9-8). The error code tells the exception handler three things:
  1. Whether the exception was due to a not present page or to an access rights violation.
  2. Whether the processor was executing at user or supervisor level at the time of the exception.
  3. 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.

Figure 9-8. Page-Fault Error Code Format

Field	Value	Description
U/S	0	The access causing the fault originated when the processor was executing in supervisor mode.
	1	The access causing the fault originated when the processor was executing in user mode.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
P	0	The fault was caused by a not-present page.
	1	The fault was caused by a page-level protection violation.



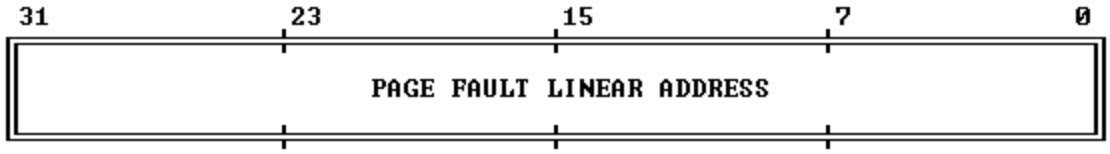
9.8.14.1 Page Fault During Task Switch

The processor may access any of four segments during a task switch:

1. Writes the state of the original task in the TSS of that task.
2. Reads the GDT to locate the TSS descriptor of the new task.
3. Reads the TSS of the new task to check the types of segment descriptors from the TSS.
4. May read the LDT of the new task in order to verify the segment registers stored in the new TSS.

A page fault can result from accessing any of these segments. In the latter two cases the exception occurs in the context of the new task. The instruction pointer refers to the next instruction of the new task, not to the instruction that caused the task switch. If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be invoked via a task gate.

Figure 9-9. CR2 Format



9.8.14.2 Page Fault with Inconsistent Stack Pointer

Special care should be taken to ensure that a page fault does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for earlier processors in the 8086 family often uses a pair of instructions to change to a new stack; for example:

```
MOV SS, AX
MOV SP, StackTop
```

With the 80386, because the second instruction accesses memory, it is possible to get a page fault after SS has been changed but before SP has received the corresponding change. At this point, the two parts of the stack pointer SS:SP (or, for 32-bit programs, SS:ESP) are inconsistent.

The processor does not use the inconsistent stack pointer if the handling of the page fault causes a stack switch to a well defined stack (i.e., the handler is a task or a more privileged procedure). However, if the page fault handler is invoked by a trap or interrupt gate and the page fault occurs at the same privilege level as the page fault handler, the processor will attempt to use the stack indicated by the current (invalid) stack pointer.

In systems that implement paging and that handle page faults within the faulting task (with trap or interrupt gates), software that executes at the same privilege level as the page fault handler should initialize a new stack by using the new LSS instruction rather than an instruction pair shown above. When the page fault handler executes at privilege level zero (the normal case), the scope of the problem is limited to privilege-level zero code, typically the kernel of the operating system.

9.8.15 Interrupt 16 -- Coprocessor Error

The 80386 reports this exception when it detects a signal from the 80287 or 80387 on the 80386's ERROR# input pin. The 80386 tests this pin only at the beginning of certain ESC instructions and when it encounters a WAIT instruction while the EM bit of the MSW is zero (no emulation). Refer to Chapter 11 for more information on the coprocessor interface.

---

up: Chapter 9 -- Exceptions and Interrupts
prev: 9.7 Error Code
next: 9.9 Exception Summary