

# Procedure Calls, Interrupts, and Exceptions

---

This chapter describes the facilities in the Intel Architecture for executing calls to procedures or subroutines. It also describes how interrupts and exceptions are handled from the perspective of an application programmer.

## 28.1 Procedure Call Types

The processor supports procedure calls in two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

Both of these procedure call mechanisms use the procedure stack, commonly referred to simply as “the stack,” to save the state of the calling procedure, pass parameters to the called procedure, and store local variables for the currently executing procedure.

The processor’s facilities for handling interrupts and exceptions are similar to those used by the CALL and RET instructions.

## 28.2 Stack

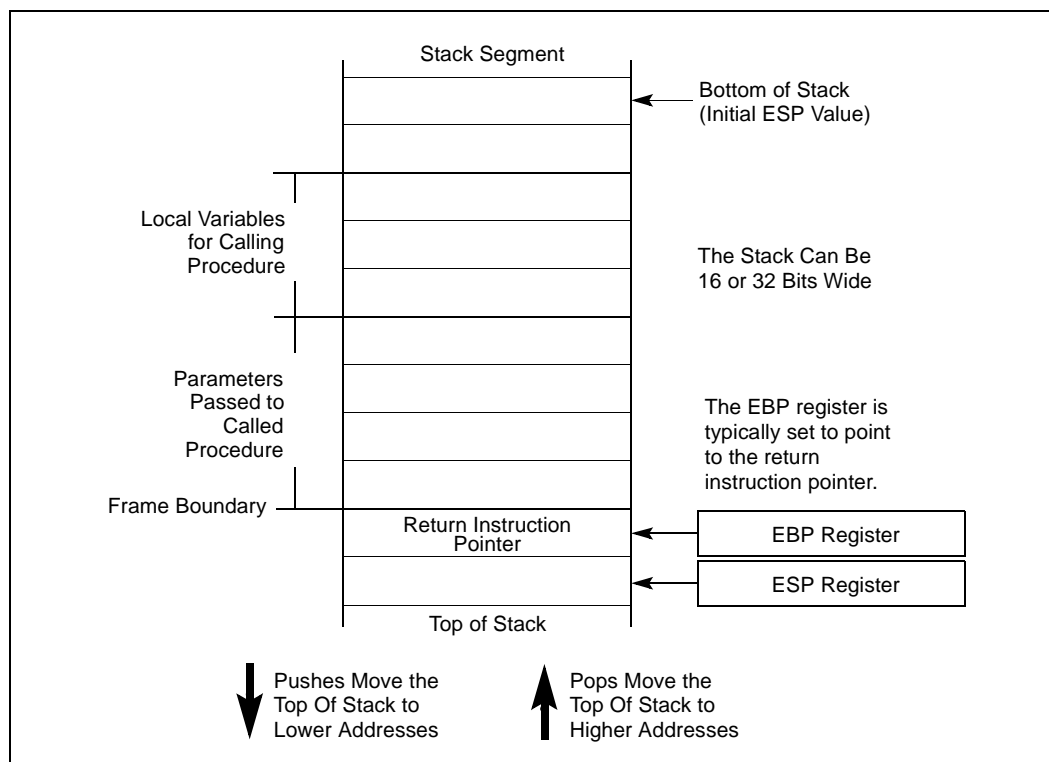
The stack (see Figure 28-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. (When using the flat memory model, the stack can be located anywhere in the linear address space for the program.) A stack can be up to 4 gigabytes long, the maximum size of a segment.

The next available memory location on the stack is called the top of stack. At any given time, the stack pointer (contained in the ESP register) gives the address (that is the offset from the base of the SS segment) of the top of the stack.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory. When a system sets up many stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

Figure 28-1. Stack Structure



The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

## 28.2.1 Setting Up a Stack

To set a stack and establish it as the current stack, the program or operating system/executive must do the following:

1. Establish a stack segment.
2. Load the segment selector for the stack segment into SS register using a MOV, POP, or LSS instruction.
3. Load the stack pointer for the stack into the ESP register using a MOV, POP, or LSS instruction. (The LSS instruction can be used to load the SS and ESP registers in one operation.)

See “Segment Descriptors” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for information on how to set up a segment descriptor and segment limits for a stack segment.

## 28.2.2 Stack Alignment

The stack pointer for a stack segment should be aligned on 16-bit (word) or 32-bit (double-word) boundaries, depending on the width of the stack segment. The D flag in the segment descriptor for the current code segment sets the stack-segment width (see “Segment Descriptors” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 3*). The PUSH and POP instructions use the D flag to determine how much to decrement or increment the stack pointer on a push or pop operation, respectively. When the stack width is 16 bits, the stack pointer is incremented or decremented in 16-bit increments; when the width is 32 bits, the stack pointer is incremented or decremented in 32-bit increments. If a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits.

The processor does not check stack pointer alignment. It is the responsibility of the programs, tasks, and system procedures running on the processor to maintain proper alignment of stack pointers. Misaligning a stack pointer can cause serious performance degradation and in some instances program failures.

## 28.2.3 Address-Size Attributes for Stack Accesses

Instructions that use the stack implicitly (such as the PUSH and POP instructions) have two address-size attributes each of either 16 or 32 bits. This is because they always have the implicit address of the top of the stack, and they may also have an explicit memory address (for example, PUSH Array1[EBX]). The attribute of the explicit address is determined by the D flag of the current code segment and the presence or absence of the 67H address-size prefix, as usual.

The address-size attribute of the top of the stack determines whether SP or ESP is used for the stack access. Stack operations with an address-size attribute of 16 use the 16-bit SP stack pointer register and can use a maximum stack address of FFFFH; stack operations with a address-size attribute of 32 bits use the 32-bit ESP register and can use a maximum address of FFFFFFFFH. The default address-size attribute for data segments used as stacks is controlled by the B flag of the segment’s descriptor. When this flag is clear, the default address-size attribute is 16; when the flag is set, the address-size attribute is 32.

## 28.2.4 Procedure Linking Information

The processor provides two pointers for linking of procedures: the stack-frame base pointer and the return instruction pointer. When used in conjunction with a standard software procedure-call technique, these pointers permit reliable and coherent linking of procedures

### 28.2.4.1 Stack-Frame Base Pointer

The stack is typically divided into frames. Each stack frame can then contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack-frame base pointer (contained in the EBP register) identifies a fixed reference point within the stack frame for the called procedure. To use the stack-frame base pointer, the called procedure typically copies the contents of the ESP register into the EBP register prior to pushing any local variables on the stack. The stack-frame base pointer then permits easy access to data structures passed on the stack, to the return instruction pointer, and to local variables added to the stack by the called procedure.

Like the ESP register, the EBP register automatically points to an address in the current stack segment (that is, the segment specified by the current contents of the SS register).

### 28.2.4.2 Return Instruction Pointer

Prior to branching to the first instruction of the called procedure, the CALL instruction pushes the address in the EIP register onto the current stack. This address is then called the return-instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the RET instruction pops the return-instruction pointer from the stack back into the EIP register. Execution of the calling procedure then resumes.

The processor does not keep track of the location of the return-instruction pointer. It is thus up to the programmer to insure that stack pointer is pointing to the return-instruction pointer on the stack, prior to issuing a RET instruction. A common way to reset the stack pointer to the point to the return-instruction pointer is to move the contents of the EBP register into the ESP register. If the EBP register is loaded with the stack pointer immediately following a procedure call, it should point to the return instruction pointer on the stack.

The processor does not require that the return instruction pointer point back to the calling procedure. Prior to executing the RET instruction, the return instruction pointer can be manipulated in software to point to any address in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well defined code entry points.

## 28.3 Calling Procedures Using CALL and RET

The CALL instructions allows control transfers to procedures within the current code segment (**near call**) and in a different code segment (**far call**). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task. See “CALL—Call Procedure” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*, for a detailed description of the CALL instruction.

The RET instruction also allows near and far returns to match the near and far versions of the CALL instruction. In addition, the RET instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument ( $n$ ) to the RET instruction. See “RET—Return from Procedure” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*, for a detailed description of the RET instruction.

### 28.3.1 Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 28-2):

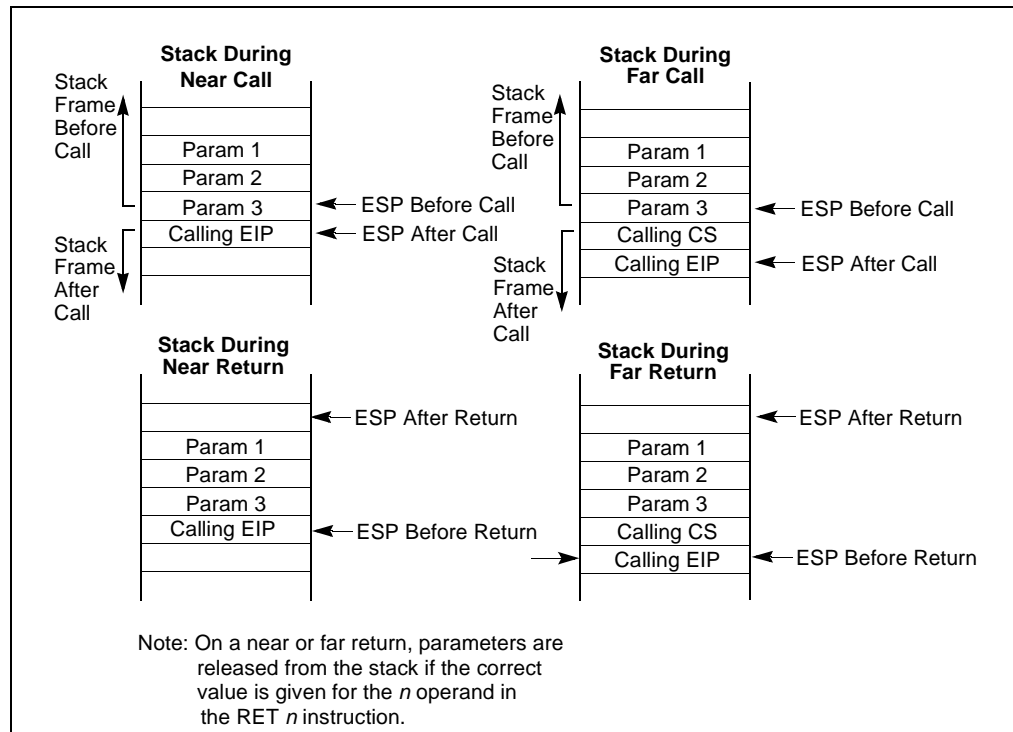
1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. (If the RET instruction has an optional  $n$  argument.) Increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack.

3. Resumes execution of the calling procedure.

Figure 28-2. Stack on Near and Far Calls



### 28.3.2 Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 28-2):

1. Pushes current value of the CS register on the stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

## 28.3.3 Parameter Passing

Parameters can be passed between procedures in any of three ways: through general-purpose registers, in an argument list, or on the stack.

### 28.3.3.1 Passing Parameters Through the General-Purpose Registers

The processor does not save the state of the general-purpose registers on procedure calls. A calling procedure can thus pass up to six parameter to the called procedure by copying the parameters into any of these registers (except the ESP and EBP registers) prior to executing the CALL instruction. The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

### 28.3.3.2 Passing Parameters on the Stack

To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure. Here, it is useful to use the stack-frame base pointer (in the EBP register) to make a frame boundary for easy access to the parameters.

The stack can also be used to pass parameters back from the called procedure to the calling procedure.

### 28.3.3.3 Passing Parameters in an Argument List

An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory. A pointer to the argument list can then be passed to the called procedure through a general-purpose register or the stack. Parameters can also be passed back to the calling procedure in this same manner.

## 28.3.4 Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments.

The PUSHA and POPA instruction facilitates saving and restoring the contents of the general-purpose registers. PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSH instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSH instruction (except the ESI value) from the stack to their respective registers.

If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former value before executing a return to the calling procedure.

If a calling procedure needs to maintain the state of the EFLAGS register it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

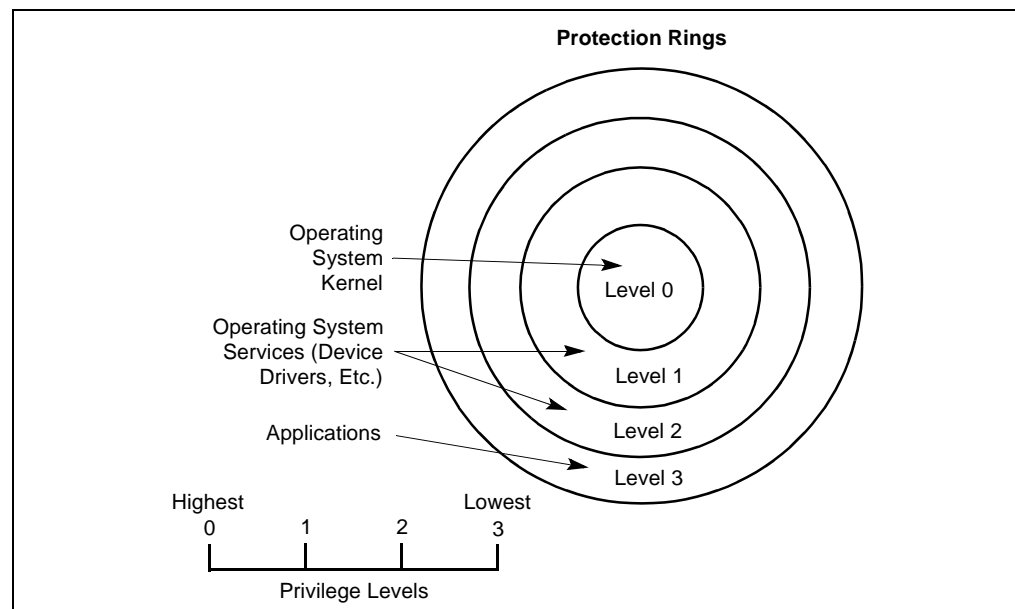
### 28.3.5 Calls to Other Privilege Levels

The Intel Architecture's protection mechanism recognizes four privilege levels, numbered from 0 to 3, where greater numbers mean lesser privileges. The primary reason to use these privilege levels is to improve the reliability of operating systems. For example, Figure 28-3 shows how privilege levels can be interpreted as rings of protection.

In this example, the highest privilege level 0 (at the center of the diagram) is used for segments that contain the most critical code modules in the system, usually the kernel of an operating system. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software.

Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called a **gate**. Attempts to access higher privilege segments without going through a protection gate and without having sufficient access rights causes a general-protection exception (#GP) to be generated.

Figure 28-3. Protection Rings



If an operating system or executive uses this multilevel protection mechanism, a call to a procedure that is in a more privileged protection level than the calling procedure is handled in a similar manner as a far call (see "Far CALL and RET Operation"). The differences are as follows:

- The segment selector provided in the CALL instruction references a special data structure called a **call gate descriptor**. Among other things, the call gate descriptor provides the following:
  - Access rights information.
  - The segment selector for the code segment of the called procedure.
  - An offset into the code segment (that is, the instruction pointer for the called procedure).
- The processor switches to a new stack to execute the called procedure. Each privilege level has its own stack. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more

privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and 0 stacks are stored in a system segment called the task state segment (TSS).

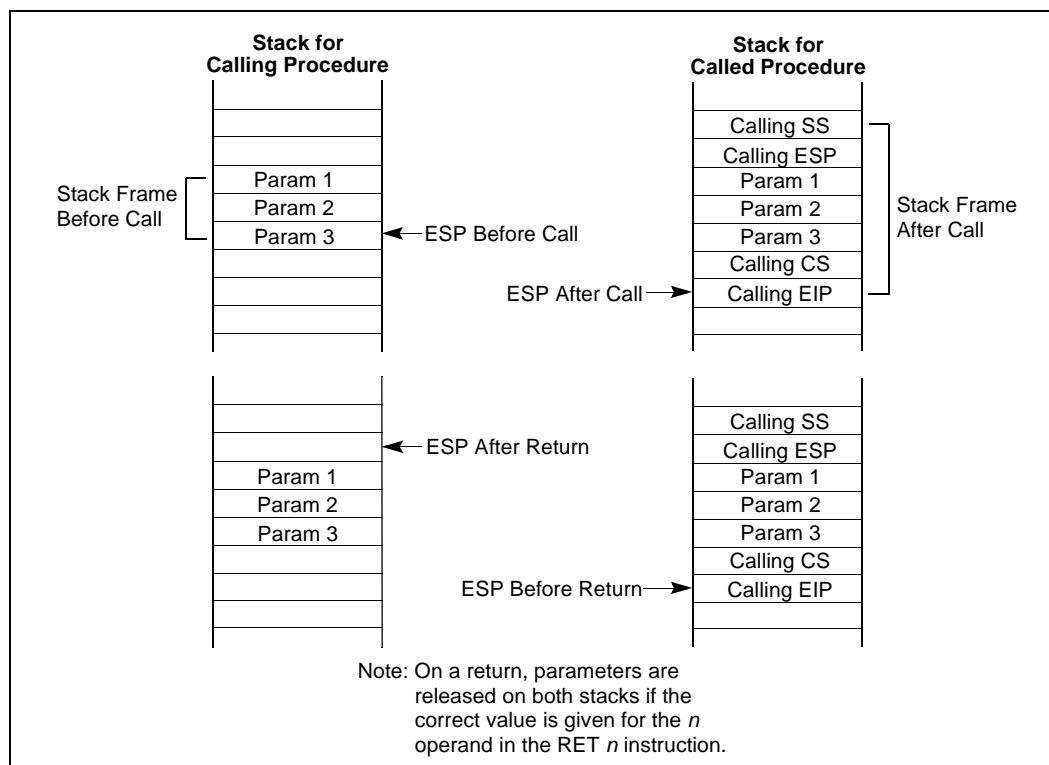
The use of a call gate and the TSS during a stack switch are transparent to the calling procedure, except when a general-protection exception is raised.

### 28.3.6 CALL and RET Operation Between Privilege Levels

When making a call to a more privileged protection level, the processor does the following (see Figure 28-2):

1. Performs an access rights check (privilege check).
2. Temporarily saves (internally) the current contents of the SS, ESP, CS, and EIP registers.

**Figure 28-4. Stack Switch on a Call to a Different Privilege Level**



1. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
2. Pushes the temporarily saved SS and ESP values for the calling procedure's stack onto the new stack.
3. Copies the parameters from the calling procedure's stack to the new stack. (A value in the call gate descriptor determines how many parameters to copy to the new stack.)
4. Pushes the temporarily saved CS and EIP values for the calling procedure to the new stack.
5. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.



6. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
3. (If the RET instruction has an optional  $n$  argument.) Increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET  $n$  instruction must be used to release the parameters from both stacks. Here, the  $n$  operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by  $n$  for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. (If the RET instruction has an optional  $n$  argument.) Increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

See Chapter 4, *Protection*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on calls to privileged levels and the call gate descriptor.

## 28.4 Interrupts and Exceptions

The processor provides two mechanisms for interrupting program execution: interrupts and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The Intel architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.

The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive through assembly-language calls. The remainder of this section gives a brief overview of the processor's interrupt and exception handling mechanism. See Chapter 5, *Interrupt and Exception Handling* in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of this mechanism.

The Intel Architecture defines 16 predefined interrupts and exceptions and 224 user defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a **vector**. Table 28-1 lists the interrupts and exceptions with entries

in the IDT and their respective vector numbers. Vectors 0 through 8, 10 through 14, and 16 through 18 are the predefined interrupts and exceptions, and vectors 32 through 255 are the user-defined interrupts, called **maskable interrupts**.

Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See “Exception and Interrupt Vectors” in Chapter 5 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for more information about the interrupts and exceptions that the Intel Architecture supports.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

### 28.4.1 Call and Return Operation for Interrupt or Exception Handling Procedures

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see “CALL and RET Operation Between Privilege Levels”). Here, the interrupt vector references one of two kinds of gates: an **interrupt gate** or a **trap gate**. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information.
- The segment selector for the code segment that contains the handler procedure.
- An offset into the code segment to the first instruction of the handler procedure.

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler’s privilege level.

**Table 28-1. Exceptions and Interrupts (Sheet 1 of 2)**

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. <sup>1</sup>
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.

Table 28-1. Exceptions and Interrupts (Sheet 2 of 2)

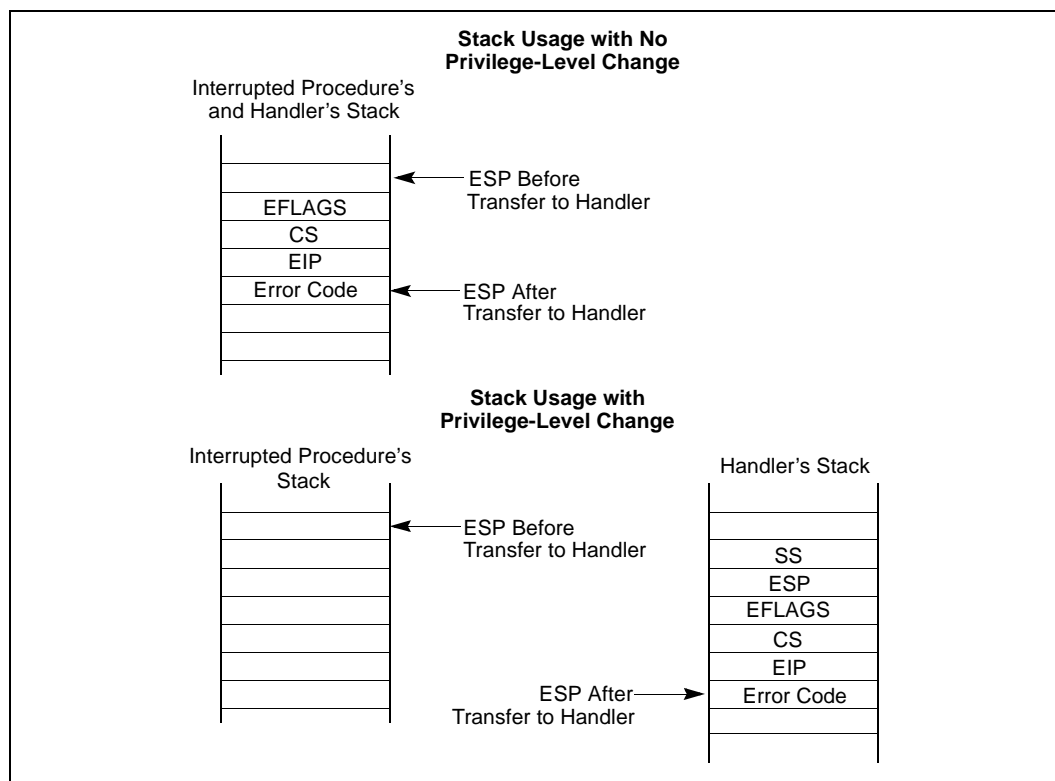
Vector No.	Mnemonic	Description	Source
9		CoProcessor Segment Overrun (reserved)	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		(Intel reserved. Do not use.)	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19-31		(Intel reserved. Do not use.)	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

1. The UD2 instruction was introduced in the Pentium® Pro processor.
2. Intel Architecture processors after the Intel386™ processor do not generate this exception.
3. This exception was introduced in the Intel486™ processor.
4. This exception was introduced in the Pentium processor and enhanced in the Pentium Pro processor.

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 28-4):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure at the new privilege level.

Figure 28-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines



If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure:

When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.

2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

## 28.4.2 Calls to Interrupt or Exception Handler Tasks

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks.

The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of the processor's mechanism for handling interrupts and exceptions through handler tasks.

## 28.4.3 Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector number as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures.

The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction.

See Chapter 15, *8086 Emulation*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on handling interrupts and exceptions in real-address mode.

## 28.4.4 INT *n*, INTO, INT 3, and BOUND Instructions

The INT *n*, INTO, INT 3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction uses an interrupt vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector number of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT 3 instruction explicitly calls the breakpoint exception (#BP) handler.

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT *n* instruction with an argument of 5 (the vector number of the bounds-check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

## 28.5 Procedure Calls for Block-Structured Languages

The Intel Architecture supports an alternate method of performing procedure calls with the ENTER (enter procedure) and LEAVE (leave procedure) instructions. These instructions automatically create and release, respectively, stack frames for called procedures. The stack frames have predefined spaces for local variables and the necessary pointers to allow coherent returns from called procedures. They also allow scope rules to be implemented so that procedures can access their own local variables and some number of other variables located in other stack frames.

The ENTER and LEAVE instructions offer two benefits:

- They provide machine-language support for implementing block-structured languages, such as C and Pascal.
- They simplify procedure entry and exit in compiler-generated code.

### 28.5.1 ENTER Instruction

The ENTER instruction creates a stack frame compatible with the scope rules typically used in block-structured languages. In block-structured languages, the scope of a procedure is the set of variables to which it has access. The rules for scope vary among languages. They may be based on the nesting of procedures, the division of the program into separately compiled files, or some other modularization scheme.

The ENTER instruction has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage for the procedure being called. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables. The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in a hierarchy of procedure calls. The lexical level is unrelated to either the protection privilege level or to the I/O privilege level of the currently running program or task.

The ENTER instruction in the following example, allocates 2K bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure.

```
ENTER 2048,3
```

The lexical nesting level determines the number of stack frame pointers to copy into the new stack frame from the preceding frame. A stack frame pointer is a doubleword used to access the variables of a procedure. The set of stack frame pointers used by a procedure to access the variables of other procedures is called the display. The first doubleword in the display is a pointer to the previous stack frame. This pointer is used by a LEAVE instruction to undo the effect of an ENTER instruction by discarding the current stack frame.

After the ENTER instruction creates the display for a procedure, it allocates the dynamic local variables for the procedure by decrementing the contents of the ESP register by the number of bytes specified in the first parameter. This new value in the ESP register serves as the initial top-of-stack for all PUSH and POP operations within the procedure.

To allow a procedure to address its display, the ENTER instruction leaves the EBP register pointing to the first doubleword in the display. Because stacks grow down, this is actually the doubleword with the highest address in the display. Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

The following pseudo code shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.

```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0
  THEN
    DO (LEVEL - 1) times
      EBP ← EBP - 4;
      PUSH Pointer(EBP); (* doubleword pointed to by EBP *)
    OD;
  PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP - STORAGE;
```

The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

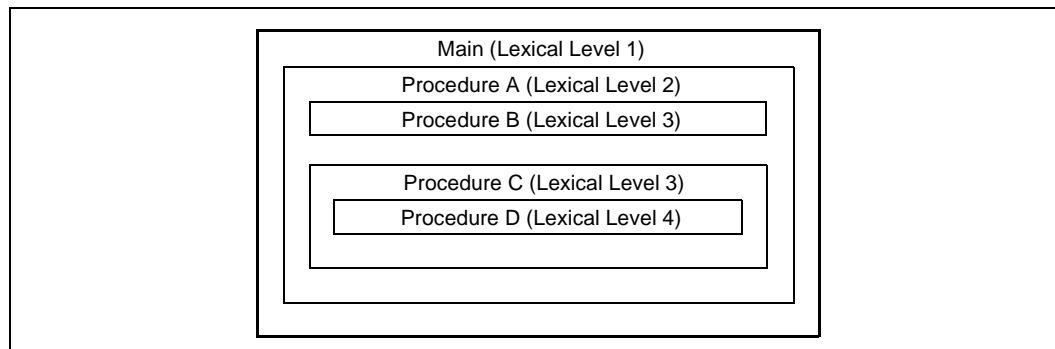
A procedure which calls another procedure at a lower lexical level gives the called procedure access to the variables of the caller. The ENTER instruction provides this access by placing a pointer to the calling procedure's stack frame in the display.

A procedure which calls another procedure at the same lexical level should not give access to its variables. In this case, the ENTER instruction copies only that part of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure's stack frame.

The ENTER instruction treats a re-entrant procedure as a call to a procedure at the same lexical level. In this case, each succeeding iteration of the re-entrant procedure can address only its own variables and the variables of the procedures within which it is nested. A re-entrant procedure always can address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, the ENTER instruction makes certain that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 28-5).

**Figure 28-6. Nested Procedures**



Block-structured languages can use the lexical levels defined by ENTER to control access to the variables of nested procedures. In Figure 28-5, for example, if procedure A calls procedure B which, in turn, calls procedure C, then procedure C will have access to the variables of the MAIN procedure and procedure A, but not those of procedure B because they are at the same lexical level. The following definition describes the access to variables for the nested procedures in Figure 28-5.

1. MAIN has variables at fixed locations.
2. Procedure A can access only the variables of MAIN.
3. Procedure B can access only the variables of procedure A and MAIN. Procedure B cannot access the variables of procedure C or procedure D.
4. Procedure C can access only the variables of procedure A and MAIN. procedure C cannot access the variables of procedure B or procedure D.
5. Procedure D can access the variables of procedure C, procedure A, and MAIN. Procedure D cannot access the variables of procedure B.

In Figure 28-7, an ENTER instruction at the beginning of the MAIN procedure creates three doublewords of dynamic storage for MAIN, but copies no pointers from other stack frames. The first doubleword in the display holds a copy of the last value in the EBP register before the ENTER instruction was executed. The second doubleword holds a copy of the contents of the EBP register following the ENTER instruction. After the instruction is executed, the EBP register points to the first doubleword pushed on the stack, and the ESP register points to the last doubleword in the stack frame.



When MAIN calls procedure A, the ENTER instruction creates a new display (see Figure 28-9). The first doubleword is the last value held in MAIN's EBP register. The second doubleword is a pointer to MAIN's stack frame which is copied from the second doubleword in MAIN's display. This happens to be another copy of the last value held in MAIN's EBP register. Procedure A can access variables in MAIN because MAIN is at level 1. Therefore the base address for the dynamic storage used in MAIN is the current address in the EBP register, plus four bytes to account for the saved contents of MAIN's EBP register. All dynamic variables for MAIN are at fixed, positive offsets from this value.

Figure 28-7. Stack Frame after Entering the MAIN Procedure

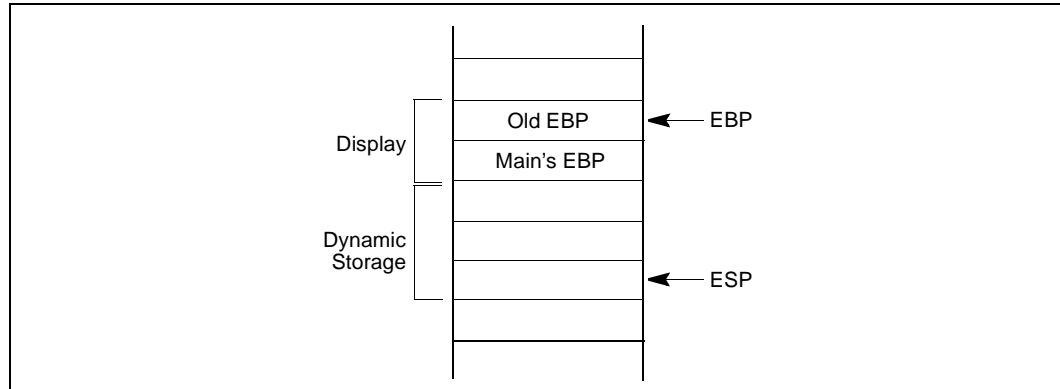
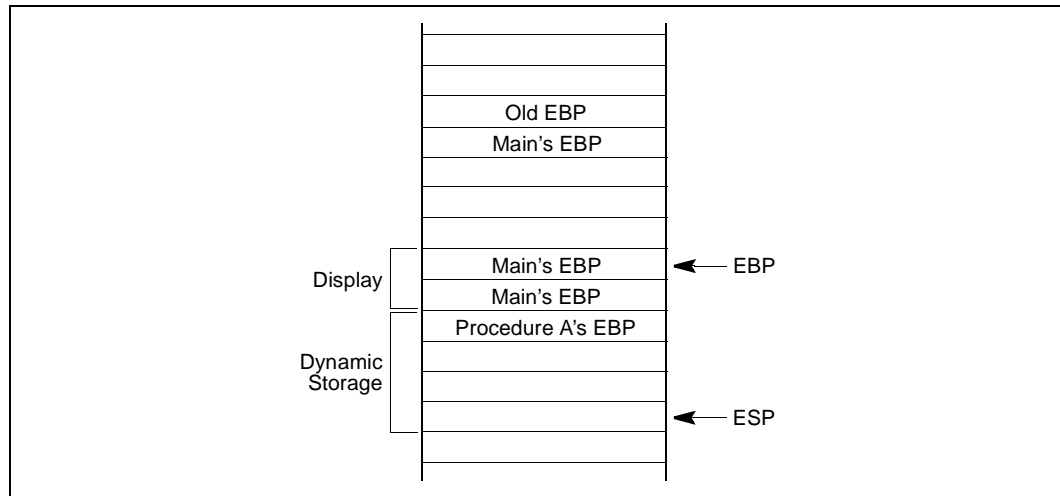
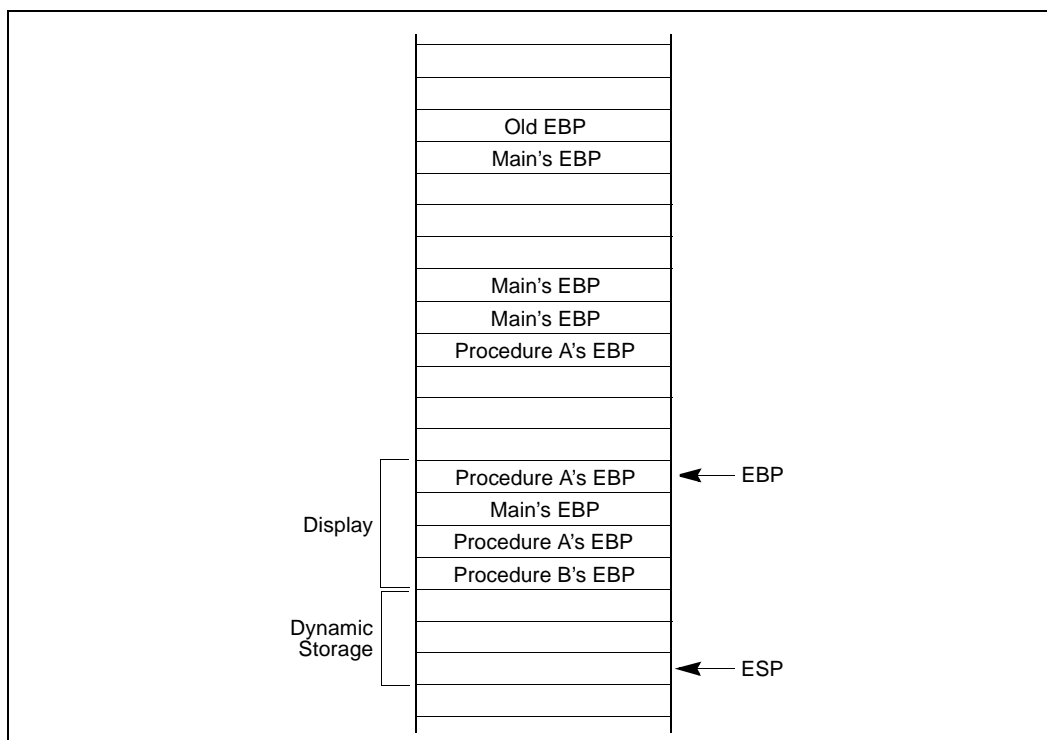


Figure 28-8. Stack Frame after Entering Procedure A



When procedure A calls procedure B, the ENTER instruction creates a new display (see Figure 28-8). The first doubleword holds a copy of the last value in procedure A's EBP register. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. Procedure B can access variables in procedure A and MAIN by using the stack frame pointers in its display.

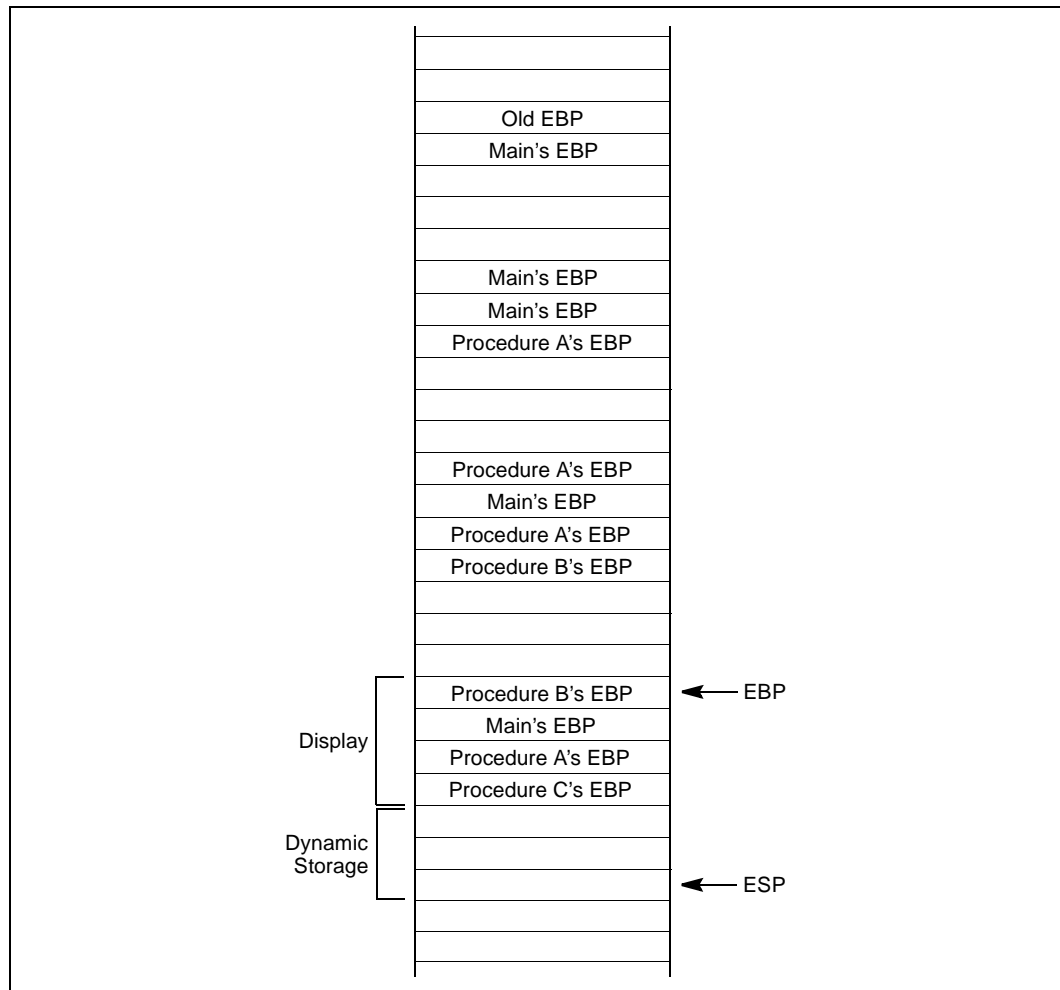
Figure 28-9. Stack Frame after Entering Procedure B



When procedure B calls procedure C, the ENTER instruction creates a new display for procedure C (see Figure 28-9). The first doubleword holds a copy of the last value in procedure B's EBP register. This is used by the LEAVE instruction to restore procedure B's stack frame. The second and third doublewords are copies of the two stack frame pointers in procedure A's display. If procedure C were at the next deeper lexical level from procedure B, a fourth doubleword would be copied, which would be the stack frame pointer to procedure B's local variables.

Note that procedure B and procedure C are at the same level, so procedure C is not intended to access procedure B's variables. This does not mean that procedure C is completely isolated from procedure B; procedure C is called by procedure B, so the pointer to the returning stack frame is a pointer to procedure B's stack frame. In addition, procedure B can pass parameters to procedure C either on the stack or through variables global to both procedures (that is, variables in the scope of both procedures).

Figure 28-10. Stack Frame after Entering Procedure C



## 28.5.2 LEAVE Instruction

The LEAVE instruction, which does not have any operands, reverses the action of the previous ENTER instruction. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then it restores the old value of the EBP register from the stack. This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.

