

Windows Kernel Internals

Traps, Interrupts, Exceptions

David B. Probert, Ph.D.

Windows Kernel Development
Microsoft Corporation

What makes an OS interesting!

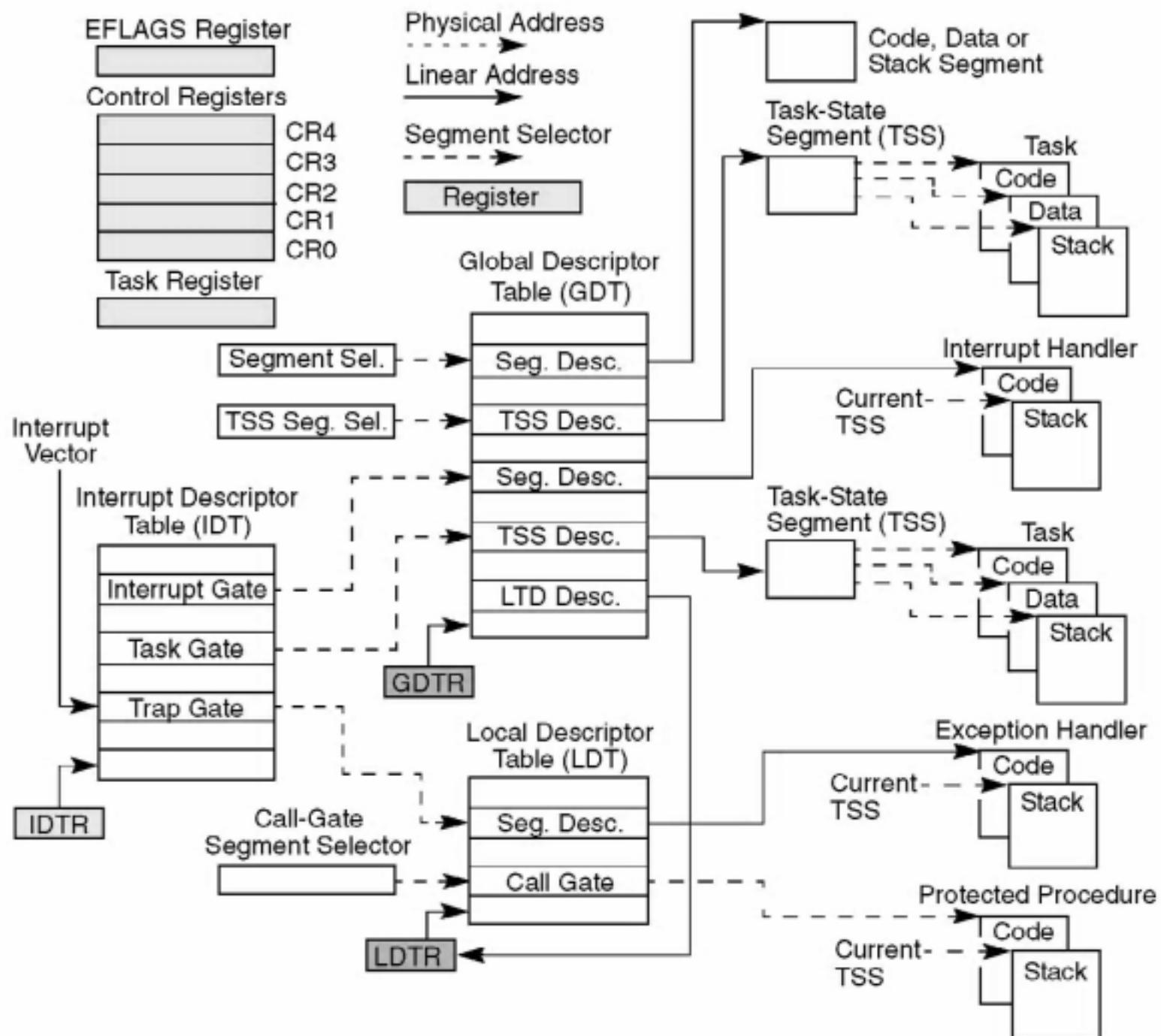
Fundamental abstractions in modern CPUs:

- normal processor execution

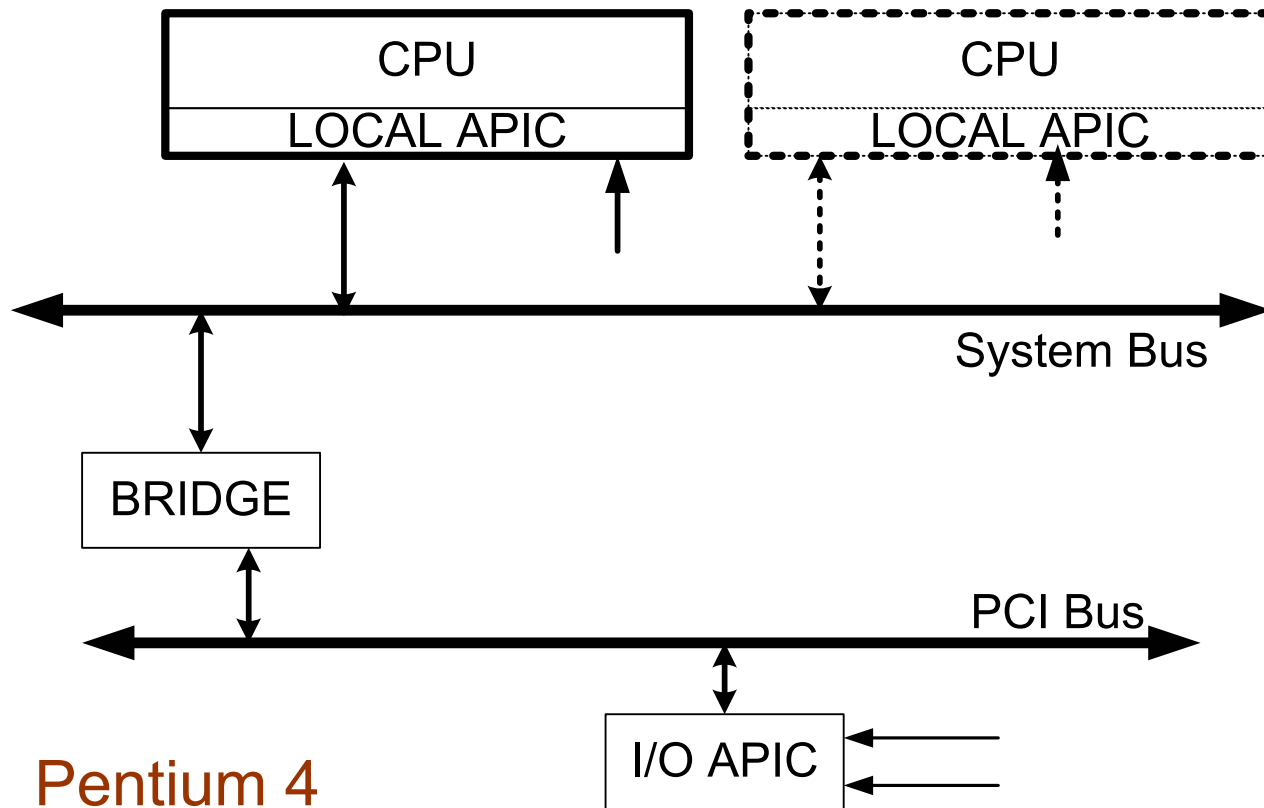
- virtual address protection/mapping

- interrupts to the above

Traps, hardware interrupts (devices, timers),
exceptions, faults, machine checks,
software interrupts



Intel's System Architecture



The local APIC

APIC: Advanced Programmable Interrupt Controller)

Local APIC built into modern Pentium processors

Receives interrupts from:

- processor interrupt pins

- external interrupt sources

 - hardwired devices

 - timers (including internal timer)

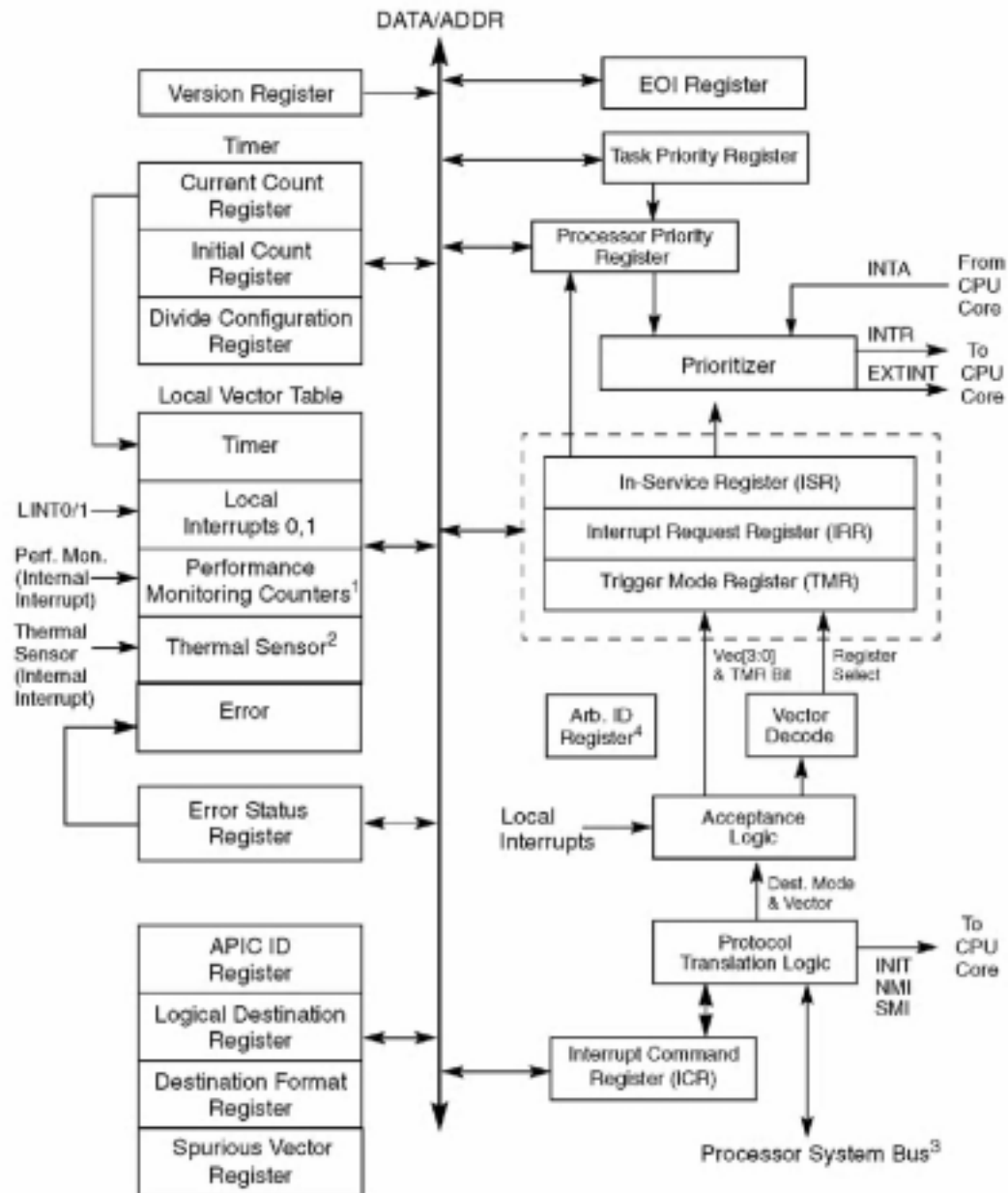
 - Perf monitors

 - Thermal monitors

 - Internal errors

- and/OR an external I/O APIC

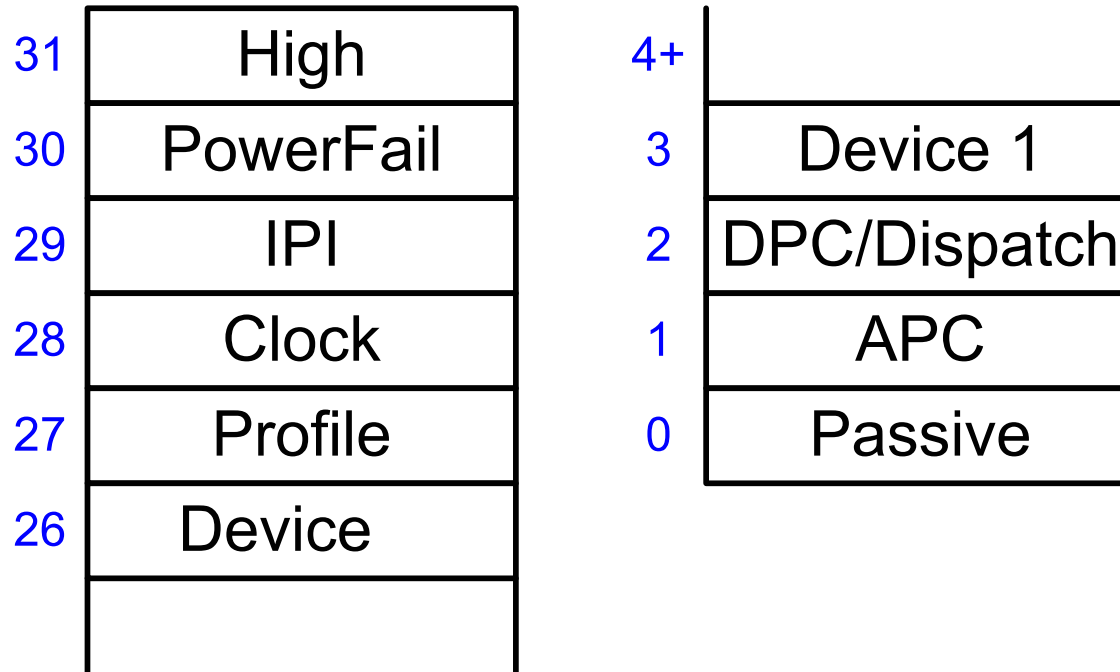
Sends IPIs in MP systems



1. Introduced in P6 family processors.
2. Introduced in the Pentium 4 and Intel Xeon processors.
3. Three-wire APIC bus in P6 family and Pentium processors.
4. Not implemented in Pentium 4 and Intel Xeon processors.

Figure 8-4. Local APIC Structure

NT Interrupt levels



Software Interrupt Delivery

Software interrupts delivered by writing ICR in APIC

```
xor    ecx, ecx
mov    cl, _HalpIRQLtoTPR[eax]    ; get IDTEntry for IRQL
or     ecx, (DELIVER_FIXED OR ICR_SELF)
mov    dword ptr APIC[LU_INT_CMD_LOW], ecx
```

```
_HalpIRQLtoTPR label byte
        db    ZERO_VECTOR          ; IRQL 0
        db    APC_VECTOR           ; IRQL 1
        db    DPC_VECTOR           ; IRQL 2
```

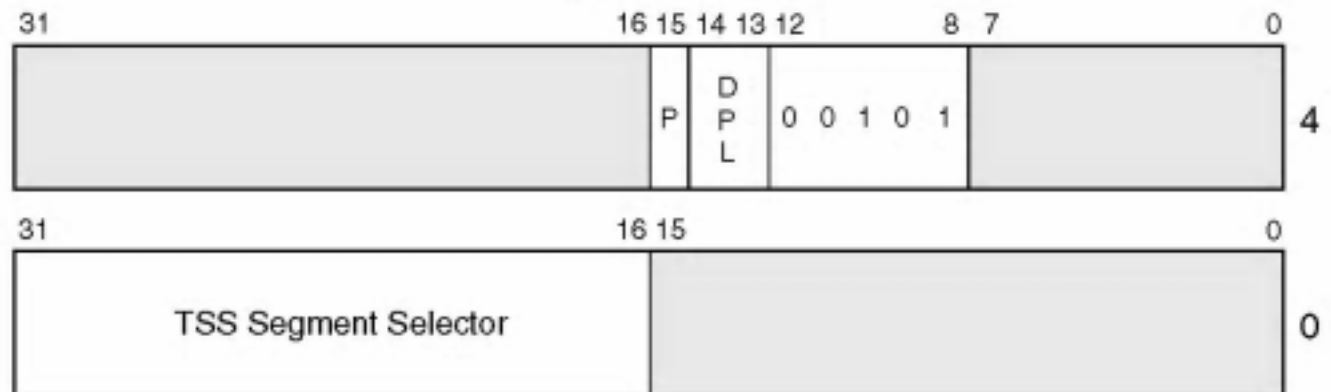
```
#define APC_VECTOR          0x3D    // IRQL 01 APC
#define DPC_VECTOR          0x41    // IRQL 02 DPC
```


IDT table

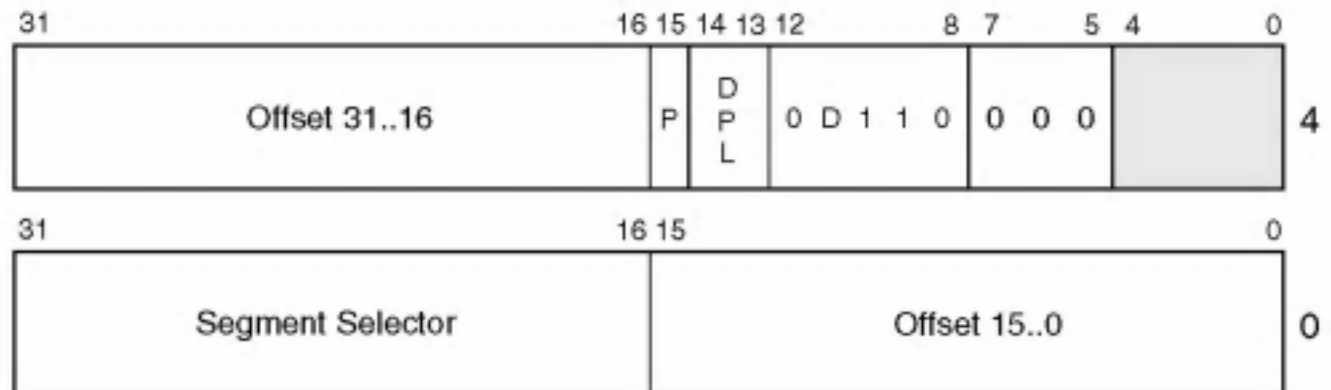
_IDT label byte

IDTEntry _KiTrap00 ;	0: Divide Error	IDTEntry _KiTrap0A ;	A: Invalid TSS
IDTEntry _KiTrap01 ;	1: DEBUG TRAP	IDTEntry _KiTrap0B ;	B: no Segment
IDTEntry _KiTrap02 ;	2: NMI/NPX Error	IDTEntry _KiTrap0C ;	C: Stack Fault
IDTEntry _KiTrap03 ;	3: Breakpoint	IDTEntry _KiTrap0D ;	D: GenProt
IDTEntry _KiTrap04 ;	4: INTO	IDTEntry _KiTrap0E ;	E: Page Fault
IDTEntry _KiTrap05 ;	5: PrintScreen	IDTEntry _KiTrap0F ;	F: Reserved
IDTEntry _KiTrap06 ;	6: Invalid Opcode	IDTEntry _KiTrap10 ;	10: 486 coproc
IDTEntry _KiTrap07 ;	7: no NPX	IDTEntry _KiTrap11 ;	11: 486 align
IDTEntry _KiTrap08 ;	8: DoubleFault	IDTEntry _KiTrap0F ;	12: Reserved
IDTEntry _KiTrap09 ;	9: NPX SegOvrn	IDTEntry _KiTrap0F ;	13: XMMI
...		IDTEntry _KiTrap0F ;	14: Reserved

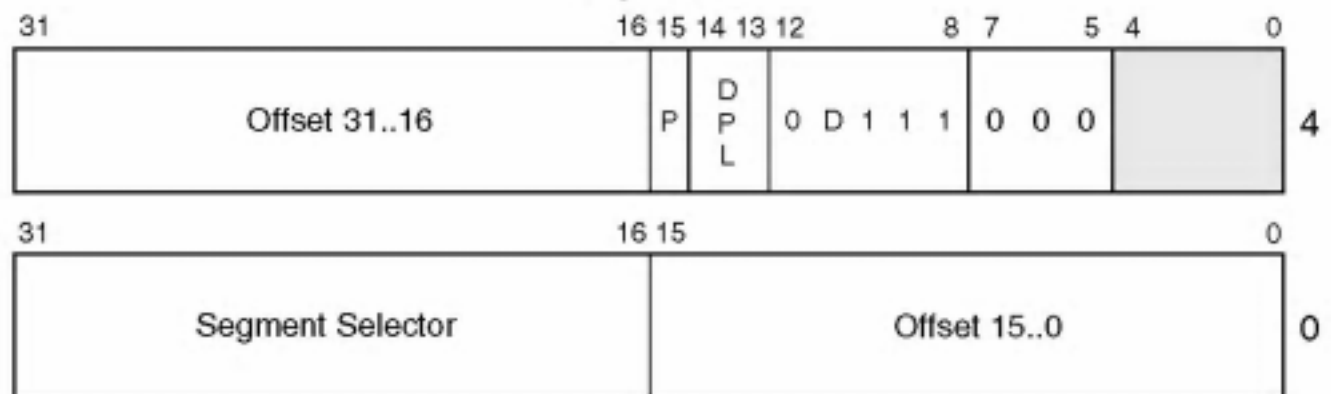
Task Gate



Interrupt Gate



Trap Gate



Entry of Interrupt Descriptor Table (KIDTENTRY)

```
typedef struct _KIDTENTRY {  
    USHORT Offset;  
    USHORT Selector;  
    USHORT Access;  
    USHORT ExtendedOffset;  
} KIDTENTRY;
```

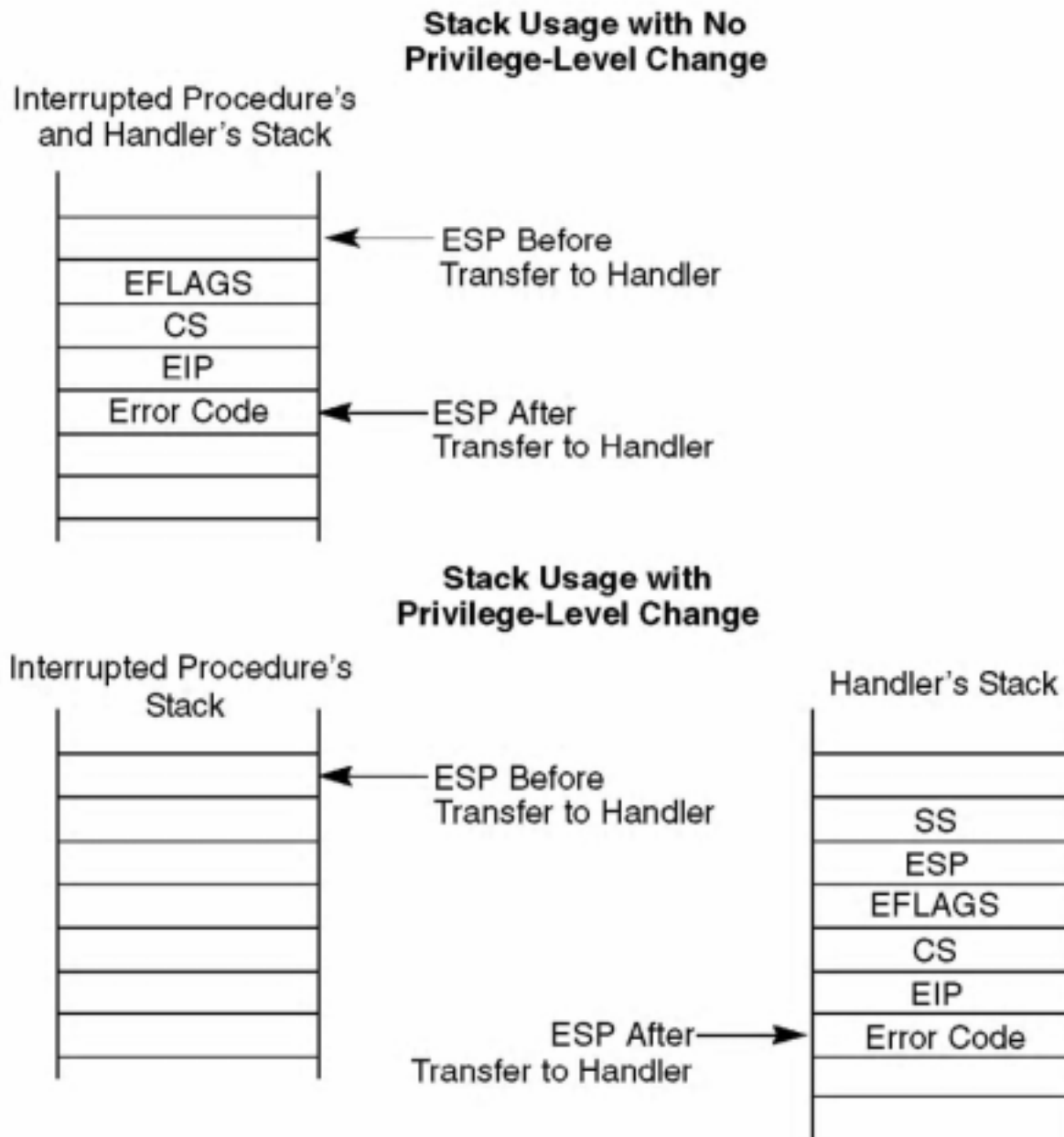


Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

KiTrapxx - trap entry points

Entry points are for internally generated exceptions not external interrupts, or user software interrupts

On entry the stack looks like:

```
[ss]  
[esp]  
eflags  
cs  
eip  
ss:sp-> [error]
```

CPU saves previous SS:ESP, eflags, and CS:EIP on the new stack if there was a privilege transition

Some exceptions save an error code, others do not

ENTER_TRAP

Macro Description: Build frame and set registers needed by trap or exception.

Save:

- Non-volatile regs,
- FS,
- ExceptionList,
- PreviousMode,
- Volatile regs
- Seg Regs from V86 mode
- DS, ES, GS

Don't Save:

- Floating point state

Set:

- Direction,
- DS, ES

Don't Set:

- PreviousMode,
- ExceptionList

Intel exception lexicon

Faults - correctable, faulting instruction re-executed

Traps - correctable, trapping instruction generally skipped

Aborts - unrecoverable, cause

CommonDispatchException()

CommonDispatchException (

ExceptCode - Exception code to put into exception record

ExceptAddress - Instruction at which HW exception

NumParms, Parameters 1, 2, 3

)

Allocates exception record on stack

Sets up exception record using specified parameters

Sets up arguments and calls `_KiDispatchException()`

KiDispatchException()

KiDispatchException (

IN PEXCEPTION_RECORD ExceptionRecord,

IN PKEXCEPTION_FRAME ExceptionFrame,

IN PKTRAP_FRAME TrapFrame,

IN KPROCESSOR_MODE PreviousMode,

IN BOOLEAN FirstChance

)

Move machine state from trap and exception frames to a context frame

Select method of handling the exception based on previous mode

Kernel-mode: try KD, try [RtlDispatchException\(\)](#), otherwise bugcheck

User-mode: try DebugPort, else copy exception to user stack, set

TrapFrame->Eip = (ULONG)[KeUserExceptionDispatcher](#)

and return

PspLookupKernelUserEntryPoints()

```
// Lookup the user mode "trampoline" code for exception dispatching
PspLookupSystemDllEntryPoint
    ("KiUserExceptionDispatcher", &KeUserExceptionDispatcher)
// Lookup the user mode "trampoline" code for APC dispatching
PspLookupSystemDllEntryPoint
    ("KiUserApcDispatcher", &KeUserApcDispatcher)
// Lookup the user mode "trampoline" code for callback dispatching
PspLookupSystemDllEntryPoint
    ("KiUserCallbackDispatcher", &KeUserCallbackDispatcher)
// Lookup the user mode "trampoline" code for callback dispatching
PspLookupSystemDllEntryPoint ("KiRaiseUserExceptionDispatcher",
    &KeRaiseUserExceptionDispatcher)
```

KeUserExceptionDispatcher

`ntdll:KiUserExceptionDispatcher()`

```
// Entered on return from kernel mode to dispatch user mode exception
// If a frame based handler handles the exception
//   then the execution is continued
//   else last chance processing is performed
```

basically this just wraps **RtlDispatchException()**

RtlDispatchException()

`RtlDispatchException(ExceptionRecord, ContextRecord)`

// attempts to dispatch an exception to a call frame based handler

// searches backwards through the stack based call frames

// search begins with the frame specified in the context record

// search ends when handler found, stack is invalid, or end of call chain

```
for (RegistrationPointer = RtlpGetRegistrationHead();
```

```
    RegistrationPointer != EXCEPTION_CHAIN_END;
```

```
    RegistrationPointer = RegistrationPointer->Next)
```

```
{
```

```
    check for valid record (#if ntos: check DPC stack too)
```

```
    switch RtlpExecuteHandlerForException()
```

```
    case ExceptionContinueExecution: return TRUE
```

```
    case ExceptionContinueSearch: continue
```

```
    case ExceptionNestedException: ...
```

```
    default: return FALSE
```

```
}
```

Discussion