Operating System Concepts and Implementations

Betriebssystemkonzepte und -implementierungen

III. Interrupts (Hardware)

October 27, 2026

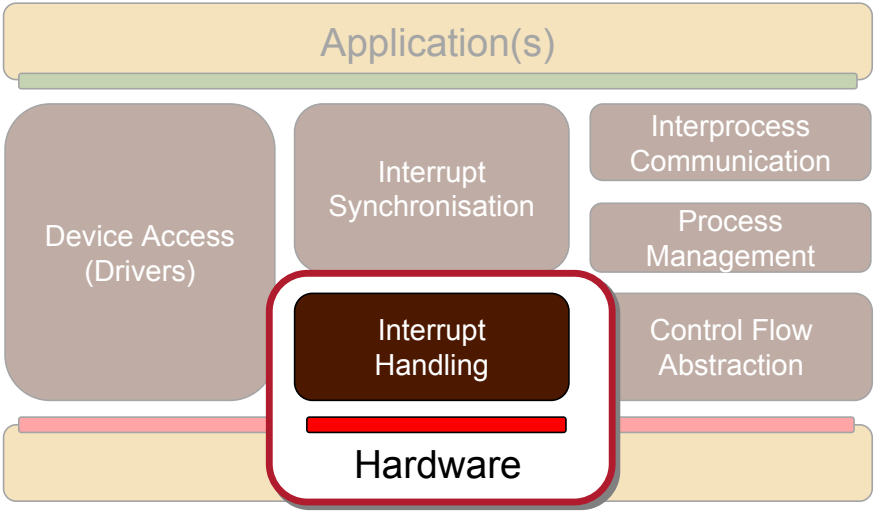
Benedict Herzog

Material based on *Chair of Computer Science 4 (System Software)*
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

RUHR
UNIVERSITÄT
BOCHUM

RUB

Overview: Today's Focus

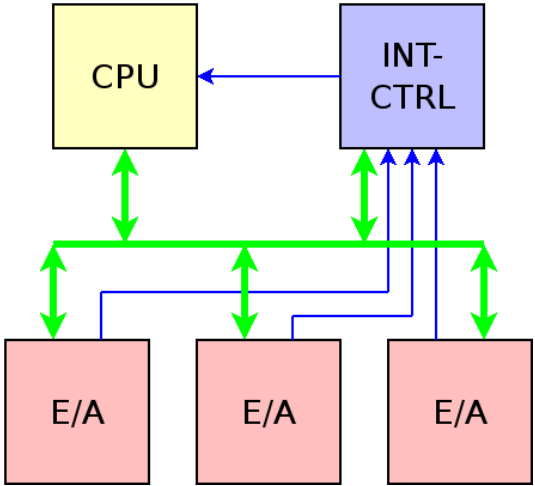


Purpose of Interrupts

A look back at the history of operating systems ...

- Overlapped Input/Output:
  - Input: Waste of processor cycles that could otherwise be used due to (often unpredictably long) *busy waiting*
  - Output: Independent operation of I/O devices (e.g. using *DMA*) reduces CPU load
- Time-sharing operation:
  - Timer interrupts allow the operating system to:
    - preempt processes
    - start activities at scheduled times

Interrupt Hardware



In the simplest case, the interrupt controller is an OR gate ...

Agenda

Context

Fundamentals

Hardware Architectures

Summary

Agenda

Context

Fundamentals

- Prioritisation
- IRQ Loss
- Interrupt Handler
- State Preservation
- Multiprocessor Systems
- Risks

Hardware Architectures

Summary

Prioritisation

- **Problem:**
  - multiple interrupt requests may be signalled simultaneously. Which one is more important?
  - while the CPU is responding to the highest-priority request, further requests may be signalled.

Prioritisation

- **Problem:**
  - multiple interrupt requests may be signalled simultaneously. Which one is more important?
  - while the CPU is responding to the highest-priority request, further requests may be signalled.
- **Solution:** a **prioritisation mechanism ...**
  - **in software**: The CPU has only one IRQ (interrupt request) input and polls the devices in a specific order
  - **in hardware**: A prioritisation circuit assigns a priority to each device and forwards only the most urgent request for handling
  - **with fixed priorities**: Each device is statically assigned a priority
  - **with variable priorities**: Priorities can be changed dynamically or, for example, rotated cyclically

IRQ Loss

- **Problem:**
  - while interrupts are being handled or disabled, the CPU cannot handle new interrupts
  - the storage capacity for interrupt requests is finite
    - → typically, one bit per interrupt line

IRQ Loss

- **Problem:**
  - while interrupts are being handled or disabled, the CPU cannot handle new interrupts
  - the storage capacity for interrupt requests is finite
    - → typically, one bit per interrupt line
- **Solution:** in software
  - the interrupt handler should be kept as short as possible (in terms of execution time) to minimise the likelihood of losing interrupts
  - interrupts should not be disabled for longer than necessary
  - every device driver should assume that a **single** interrupt may indicate **more than one** completed I/O operation

Interrupt Handler Assignment

- **Problem:**
  - the software should be able to determine which device triggered the interrupt with as little overhead as possible
  - sequentially polling the devices not only takes time, but also changes the states of I/O buses and devices that are not involved

Interrupt Handler Assignment

- **Problem:**
  - the software should be able to determine which device triggered the interrupt with as little overhead as possible
  - sequentially polling the devices not only takes time, but also changes the states of I/O buses and devices that are not involved
- **Solution:**
  - each interrupt is assigned a number that is used as an index into a vector table
    - the vector number is not necessarily related to the interrupt priority
    - in practice, devices may unfortunately have to share a vector number (*interrupt sharing*)
  - the structure of the vector table varies depending on the processor architecture
    - it usually contains pointers to functions
    - less commonly, the entries themselves are instructions

State Preservation

- **Problem:**
  - after the interrupt handler has finished executing, it must be possible to return to the normal execution context
  - interrupt handling should be effectively unnoticeable (*transparency*)

State Preservation

- **Problem:**
  - after the interrupt handler has finished executing, it must be possible to return to the normal execution context
  - interrupt handling should be effectively unnoticeable (*transparency*)
- **Solution:**
  - state preservation by software
    - only the essential state is saved (e.g. return address and processor status word)
    - the state is restored using a special instruction (e.g. iret, rte, ...)
  - state preservation by hardware
    - since interrupts can occur at any time, the interrupt handler must also save and restore the required state

Nested Interrupt Handling

- **Problem:**
  - to respond quickly to high-priority events, interrupt handling itself must be interruptible
  - however, unlimited nesting depth must be avoided

Nested Interrupt Handling

- **Problem:**
  - to respond quickly to high-priority events, interrupt handling itself must be interruptible
  - however, unlimited nesting depth must be avoided
- **Solution:**
  - the CPU only allows interrupts with a higher priority than the interrupt currently being handled
  - the current priority is stored in the processor status word
  - the previous priority is saved on a stack

Interrupt Sequence

HW: I/O device signals an interrupt to the interrupt controller

HW: Interrupt controller signals an interrupt to the CPU

HW: CPU saves some registers (PC, `IE`, ...)

HW: CPU disables the interrupt input (`IE = 0`)

HW: CPU obtains the IRQ number from the interrupt controller

HW: CPU loads the vector from the interrupt vector table into the PC

SW: Interrupt handler saves additional registers

SW: Interrupt handler handles the interrupt

SW: Interrupt handler restores the registers

SW: Interrupt handler invokes `iret`

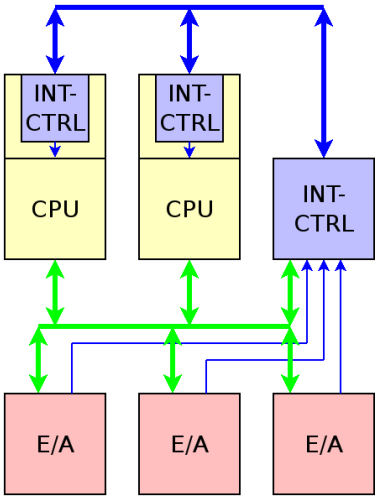
HW: CPU restores PC, `IE`, ...
Returns to the interrupted main program,
interrupt input is enabled again

Multiprocessor Systems

- **Problem:**
  - An interrupt can only be handled by one CPU. But which one?
  - There is another category of interrupts: inter-processor interrupts (IPIs).
- **Solution:** the hardware for interrupt handling in multiprocessor systems needs to be more complex. There are many design options ...
  - fixed assignment
  - random assignment
  - programmable assignment
  - assignment based on processor load

  ... and combinations of these approaches.

Interrupt Hardware (SMP)



Interrupt controllers are independent, configurable units ...

Risk: Spurious Interrupts

- **Problem:** a technical mechanism for interrupt handling carries the risk of spurious interrupt requests, e.g. due to ...
  - hardware faults
  - incorrectly programmed or configured devices
- **Solution:**
  - avoid hardware and software errors ☺
  - program the operating system defensively
    → anticipate spurious interrupts

Risk: Interrupt Storms

- **Problem:**
  - high-frequency interrupt requests can bring a computer to a standstill
  - they are either spurious interrupts or indicate that the system is unable to cope with the I/O load
  - this can easily be mistaken for *thrashing* (excessive memory swapping)
- **Solution:** by the operating system
  - detect interrupt storms
  - disable the device causing the interrupt storm

Agenda

Context

Fundamentals

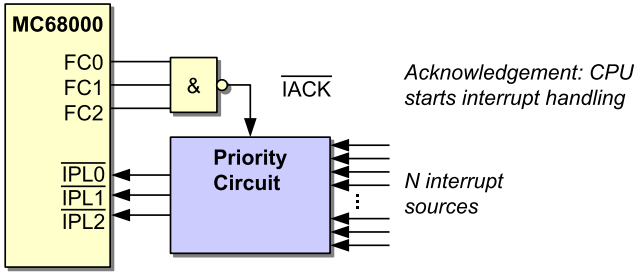
Hardware Architectures

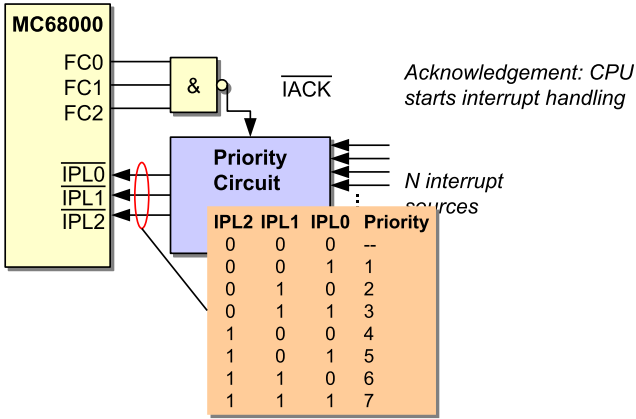
- Motorola/Freescale 68k
- Intel x86

Summary

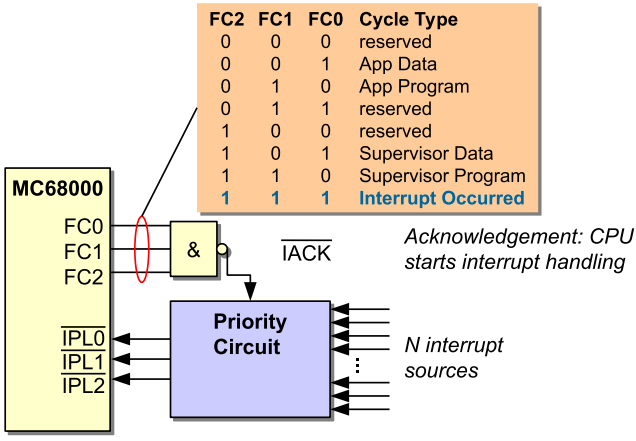
MC68000 Interrupts

MC68000 Interrupts



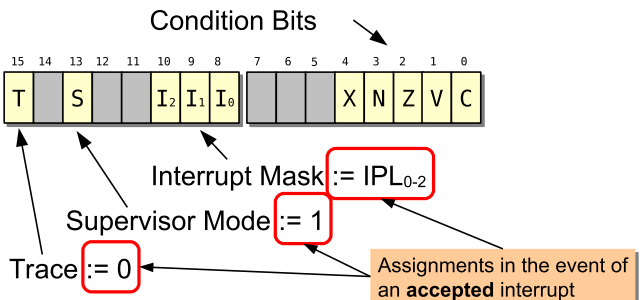

MC68000
FC0
FC1
FC2
&
IACK
Acknowledgement: CPU starts interrupt handling
Priority Circuit
IPL0
IPL1
IPL2
N interrupt sources
IPL2 IPL1 IPL0 Priority
0 0 0 --
0 0 1 1
0 1 0 2
0 1 1 3
1 0 0 4
1 0 1 5
1 1 0 6
1 1 1 7

MC68000 Interrupts



MC68000 Status Register (SR)

- Contains the current interrupt mask (among other things)
  - When an interrupt occurs, the CPU checks whether $\mathrm{IPL}_{0\text{-}2} > \mathrm{I}_{0\text{-}2}$.
    - $\rightarrow$ If not, the request is not (yet) accepted.
  - An interrupt with $\mathrm{IPL}_{0\text{-}2} = 7$ is always handled (NMI)



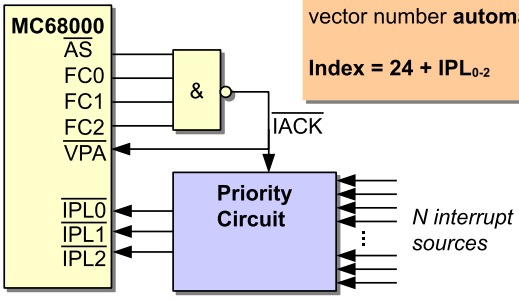
MC68000 Vector Table

Index	Address	Description
0	0x000	Reset: Initial Supervisor Stack Pointer
1	0x004	Reset: Initial PC
2	0x008	Bus Error
3	0x00c	Address Error
4	0x010	Illegal Instruction
5	0x014	Division by Zero
...		
24	0x060	Spurious Interrupt
25	0x064	**Autovectored Interrupt, Level 1**
26	0x068	**Autovectored Interrupt, Level 2**
...		
30	0x078	**Autovectored Interrupt, Level 6**
31	0x07c	**Autovectored Interrupt, Level 7 (NMI)**
32-47	0x080	TRAP Instruction Vectors
48-63	0x0c0	reserved
64-255	0x100	**User Interrupt Vectors**

Autovectored Interrupts

External circuitry signals via **VPA** that the CPU should calculate the vector number **automatically**:

Index = 24 + $IPL_{0\text{-}2}$



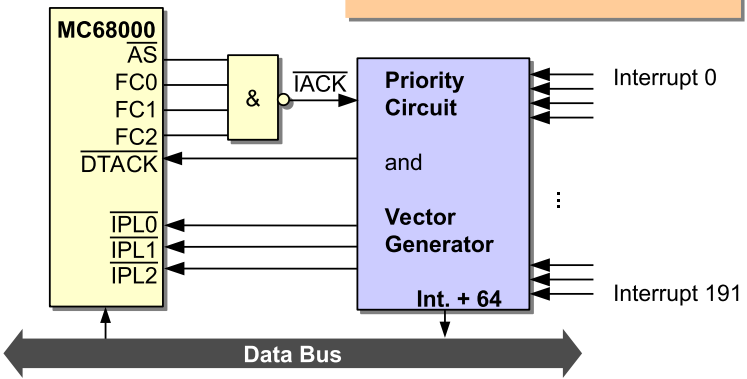
Problem

Only 6 vectors are available for devices.
With more devices, *sharing* is unavoidable.

Non-Autovectored Interrupts

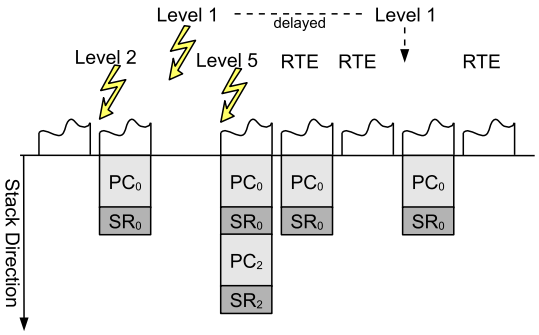
External circuitry signals via **DTACK** that the CPU should read the vector number **from the data bus.**

Index = 64 + D_{0-7}



MC68000 State Preservation

- The previous contents of the SR and the PC are saved on the supervisor stack when an interrupt occurs
- The RTE (*return from exception*) instruction restores state



MC68000 Summary

- 6 priority levels for hardware interrupts + NMI
  - Interrupt levels 1–6, NMI at level 7
  - Masking via status register $I_{0\text{-}2}$
- Only higher-priority interrupts and the NMI can interrupt an interrupt handler currently being executed
  - The status register is adjusted automatically
- Automatic state preservation on the supervisor stack;
  nested interrupt handling is possible
- The vector number is generated either …
  - autovectored: index = priority + 24
  - non-autovectored (by external hardware): index = 64 … 255
- No multiprocessor support

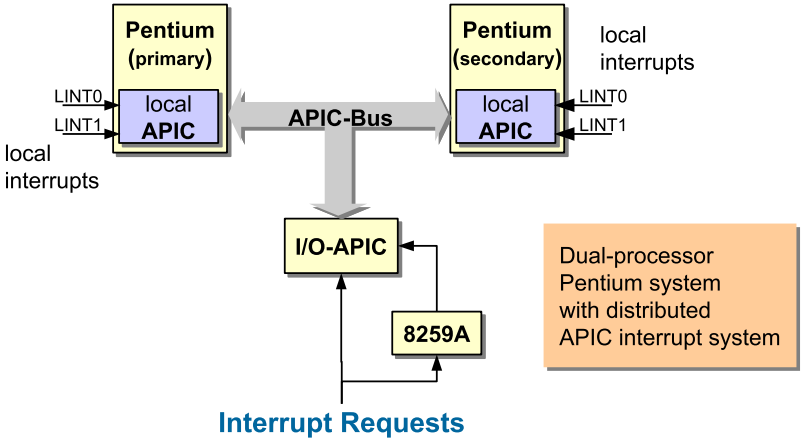
Interrupts in x86 CPUs

Interrupts in x86 CPUs

- Up to and including the i486, x86 CPUs had only one IRQ and one NMI line
- External hardware was responsible for prioritisation and vector number generation
  - using a chip called the **PIC 8259A**
    - 8 interrupt lines
    - 15 lines when cascading two PICs
    - no multiprocessor support
- Modern x86 CPUs contain the much more powerful **Advanced Programmable Interrupt Controller (APIC)**
  - required for **multiprocessor systems**
  - now also active in all uniprocessor systems
    - of course, the PIC 8259A interface is still available for compatibility ☺

APIC Architecture

- An APIC interrupt system consists of a local APIC on each CPU and an I/O APIC



I/O APIC

- Nowadays typically integrated into the southbridge of PC chipsets
- Usually 24 interrupt lines
  - Cyclic polling (round-robin prioritisation)
- For each input, there is a 64-bit entry in the **Interrupt Redirection Table**
  - Describes the interrupt signal
  - Used to generate the APIC bus message

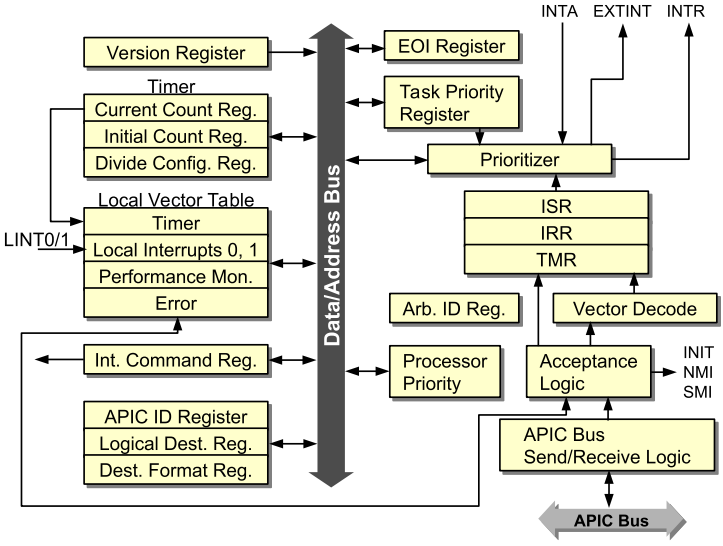
Structure (Bits) of an *Interrupt Redirection Table entry*

63:56	**Destination Field** depending on Bit 11:	– R/W. 8-bit destination address. CPU APIC ID (*physical mode*) or CPU group (*logical mode*)
55:17	<reserved>	
16	**Interrupt Mask**	– R/W. Mask this interrupt
15	**Trigger Mode**	– R/W. *Edge* (0) or *level (1) trigger*
14	**Remote IRR**	– RO. Type of received acknowledgement
13	**Interrupt Pin Polarity**	– R/W. Signal polarity (high/low is active)
12	**Delivery Status**	– RO. Interrupt message in flight?
11	**Destination Mode**	– R/W. *Physical mode (0)* or *logical mode (1)*
10:8	**Delivery Mode**	– R/W. Destination APIC effect
	000 – *Fixed*:	Deliver to all destination CPUs
	001 – *Lowest Priority*:	Deliver to CPU with lowest current priority
	010 – SMI:	*System Management Interrupt*
	100 – NMI:	*Non Maskable Interrupt*
	101 – INIT:	Initialise destination CPUs (reset)
	111 – ExtINT:	Answer to PIC 8259A
7:0	**Interrupt Vector**	– R/W. **8-bit vector number (16 – 254)**

Local APICs

- Receive interrupt requests from the APIC bus
- Perform selection and prioritisation
- Can directly handle two local interrupts (`lint0`/`lint1`)
- Contain additional functional units
  - Built-in timer and performance counter
  - Command register
    - For sending own APIC messages
    - In particular, inter-processor interrupts (IPIs)
- Programmable via 32-bit registers (starting at `0xfee00000`)
  - Memory-mapped (without external bus cycles)
  - Each CPU programmes its own Local APIC

Local APICs — Registers



APIC Architecture — Summary

- Flexible distribution of interrupts among CPUs in an x86 multiprocessor system
  - Fixed, to groups, or to the CPU with the lowest priority
  - If multiple IRQs are pending, they are prioritised according to their vector numbers
- Vector numbers 16–254 can be freely assigned
  - Should (in principle) be sufficient to avoid interrupt sharing
- Local APIC expects an explicit EOI (*end of interrupt*)
  - Must be issued by software!
- With the APIC, x86 also supports priority levels in principle
  - System software must act accordingly
    (re-enable interrupts and, if necessary, use the Task Priority Register)

IRQ Sharing

- In practice, 24 IRQ lines proved to be insufficient
- … especially 4/8 lines for PCI devices

PIRQ Line	#A	#B	#C	#D	#E	#F	#G	#H
AGP-Slot	shared							
PCI 1						shared		
PCI 2							used	
PCI 3					used			
PCI 4								shared
PCI 5						shared		
PCI 6			shared					
1. USB 1.1	shared							
2. USB 1.1				used				
3. USB 1.1			shared					
USB 2.0								shared
AC-97 Sound						shared		

- **Message-Signalled Interrupts** (MSIs) finally resolved this.

Agenda

Context
Fundamentals
Hardware Architectures
Summary

Summary and Outlook

- Interrupt-handling hardware implements …
  - Prioritisation
  - Assignment and execution of an interrupt handler
  - State preservation and nested interrupt handling
- Modern interrupt-handling hardware can …
  - Freely assign interrupt vectors
  - Avoid interrupt vector sharing
  - Flexibly distribute interrupts among CPUs in a multiprocessor system
- The operating system must …
  - Take problems such as *spurious interrupts* and *interrupt storms* into account
  - Propagate the event from the interrupt handler to higher layers and ultimately to the application process