Lecture Topics

- Programmable interrupt controller (PIC)
  - Linux 8259A Initialization
- Linux abstraction of PIC
- General interrupt abstractions
  - Interrupt Chaining
  - Soft Interrupts

ECE391 EXAM 1

- **EXAM 1 – March 2 (Tuesday);**
  - UIUC students: 6:00pm to 8:00pm; Illinois time (or CST)
  - ZJUI students: 8:00pm to 10:00pm; China time (which is 6:00am to 8:00am Illinois time)
  - Detailed instructions will be provided soon

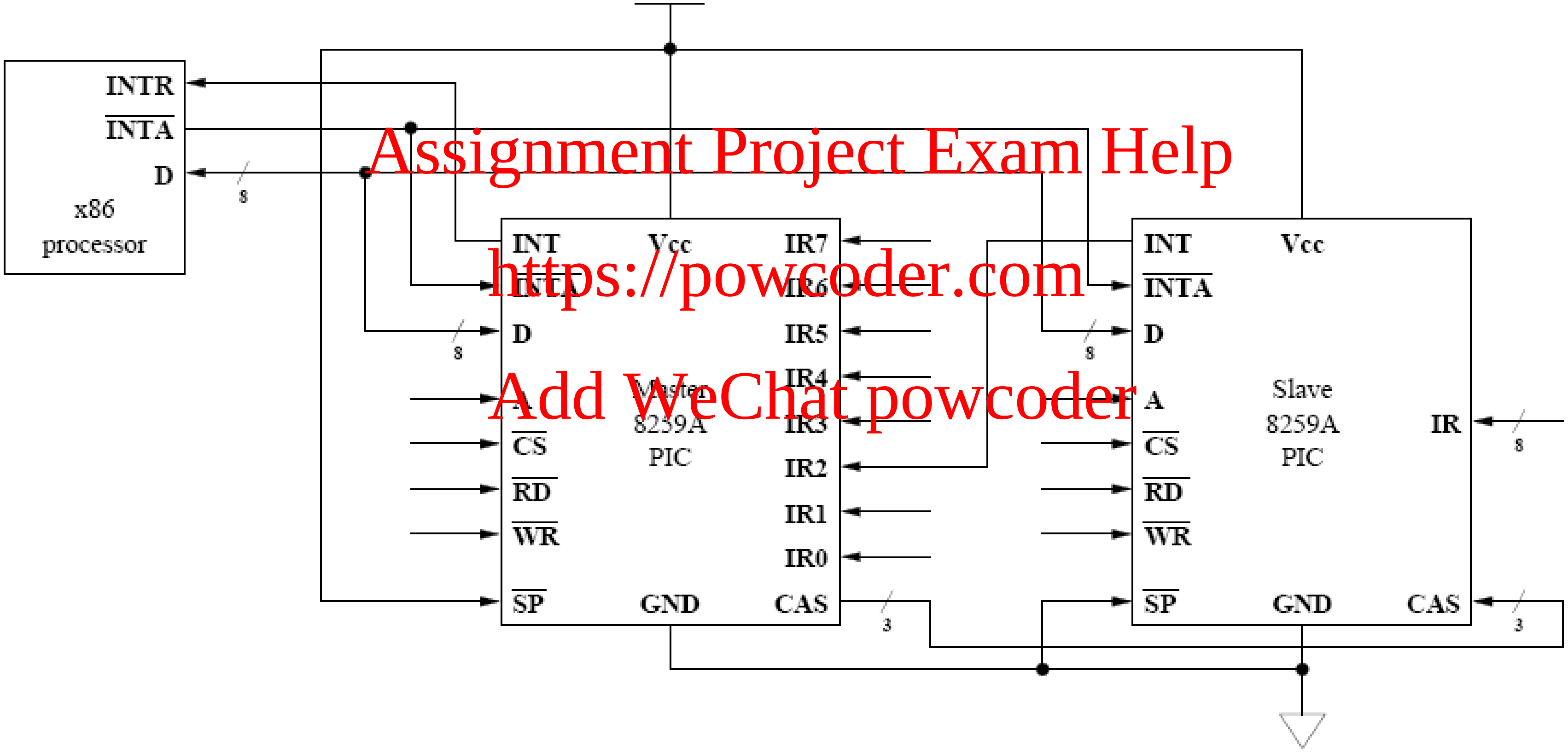
- **Conflict Exam**
  - Deadline to request conflict exam: Friday, February 26, 5:00pm (by email to: *kalbarcz@Illinois.edu*)
- **Exam 1 Synchronous Review Session in collaboration with HKN**
  - Saturday February 27; 8:00pm; (Illinois time)
  - Zoom link will be provided later this week

ECE391 EXAM 1

- **Topics covered by EXAM 1**
  - Materia covered in lectures (Lecture1 – Lecture10)
    - x86 Assembly
    - C-Calling Convention
    - Synchronization
    - Interrupt control (using PIC)
  - Material covered in discussions
  - MP1

- **NO Lecture on Tuesday, March 2**

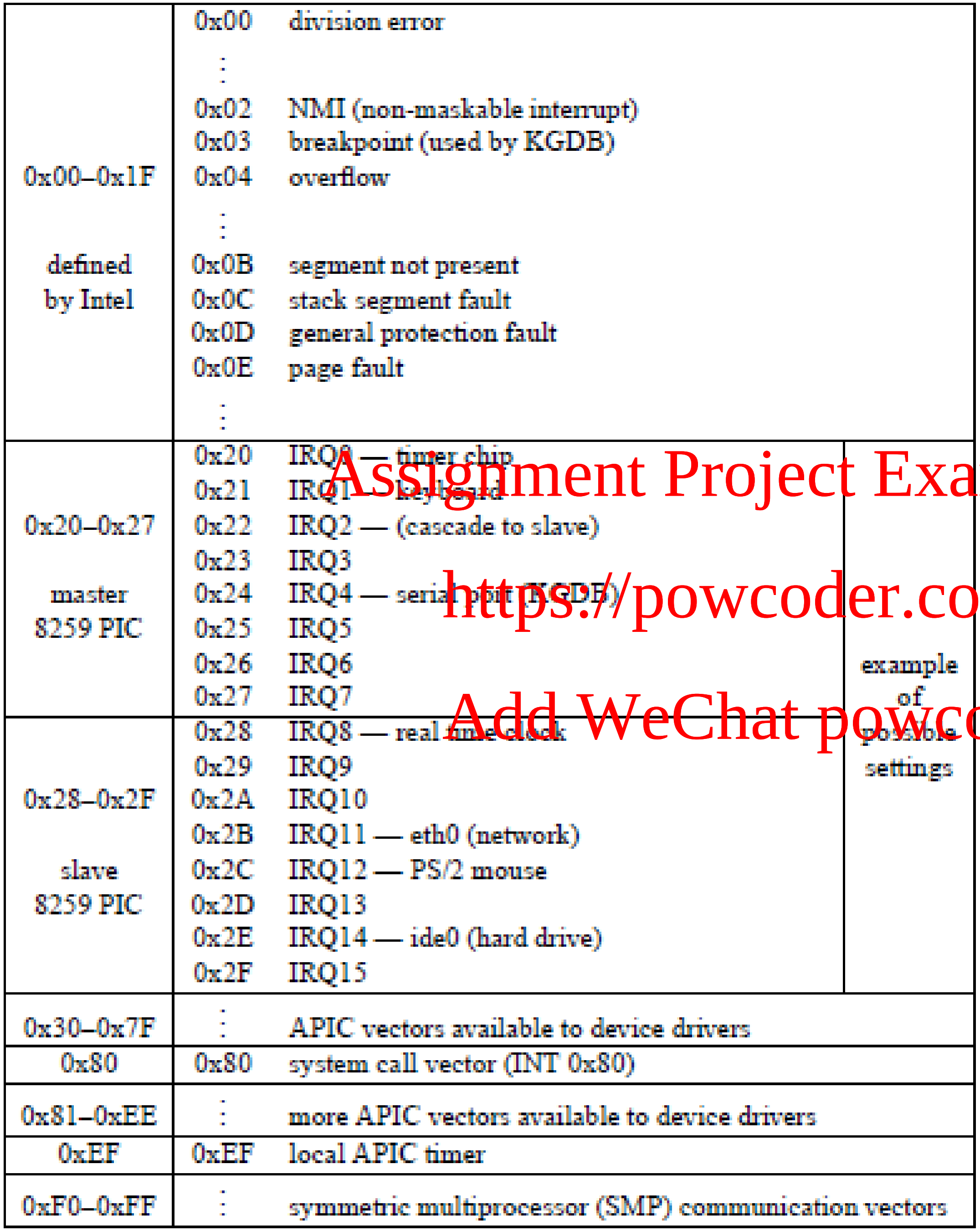
Cascade Configuration of PICs



PIC (cont.)

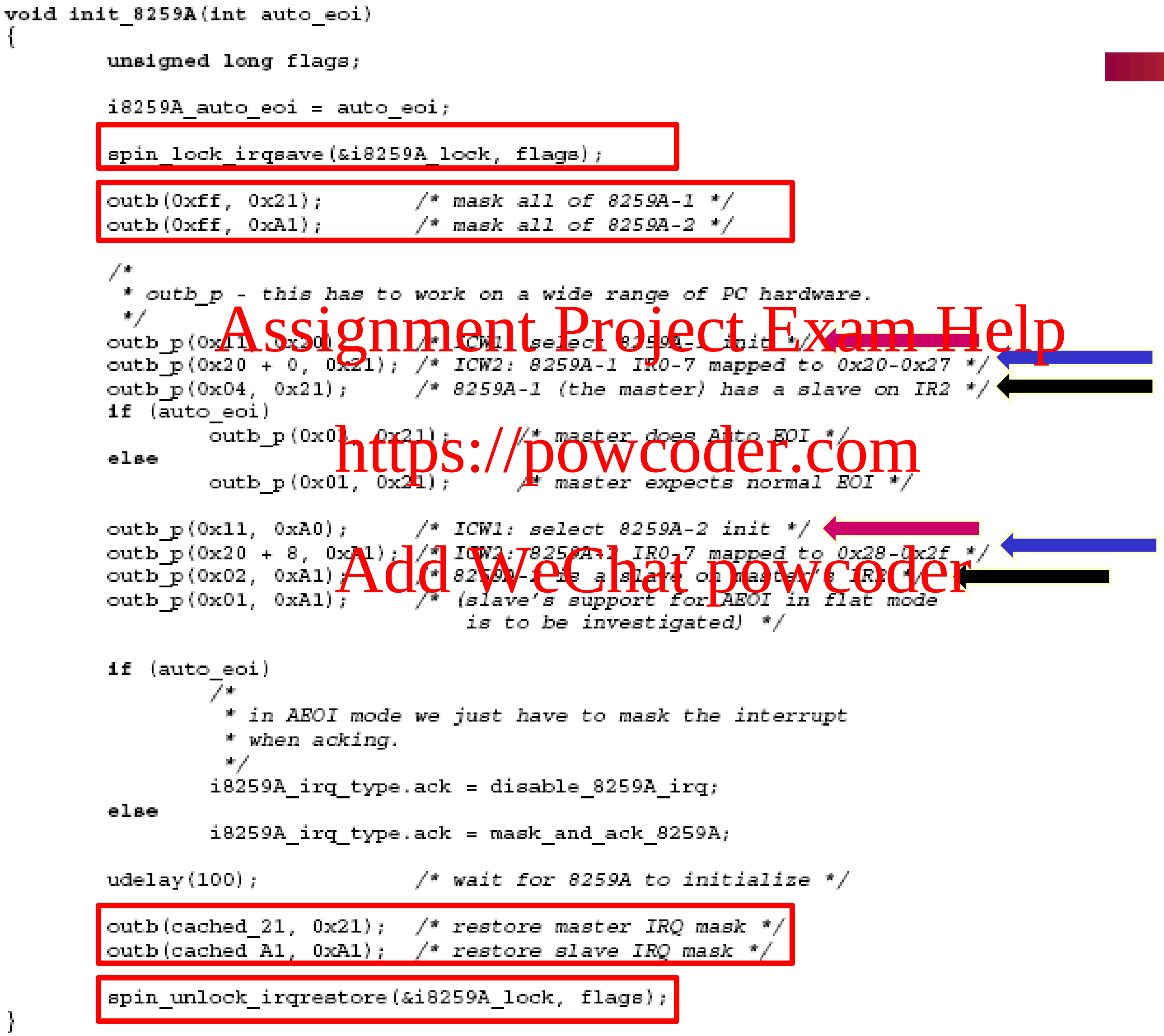
- In Linux (initialization code to be seen shortly)
  - master IR’s mapped to vector #’s 0x20 – 0x27
  - slave IR’s mapped to vector #’s 0x28 – 0x2F
  - remember the IDT?

Interrupt Descriptor Table



0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 master 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to slave)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F slave 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Linux 8259A Initialization



```
void init_8259A(int auto_eoi)
{
        unsigned long flags;

        i8259A_auto_eoi = auto_eoi;

        spin_lock_irqsave(&i8259A_lock, flags);

        outb(0xff, 0x21);       /* mask all of 8259A-1 */
        outb(0xff, 0xA1);       /* mask all of 8259A-2 */

        /*
         * outb_p - this has to work on a wide range of PC hardware.
         */
        outb_p(0x11, 0x20);     /* ICW1: select 8259A-1 init */
        outb_p(0x20 + 0, 0x21); /* ICW2: 8259A-1 IR0-7 mapped to 0x20-0x27 */
        outb_p(0x04, 0x21);     /* 8259A-1 (the master) has a slave on IR2 */
        if (auto_eoi)
                outb_p(0x03, 0x21);     /* master does Auto EOI */
        else
                outb_p(0x01, 0x21);     /* master expects normal EOI */

        outb_p(0x11, 0xA0);     /* ICW1: select 8259A-2 init */
        outb_p(0x20 + 8, 0xA1); /* ICW2: 8259A-2 IR0-7 mapped to 0x28-0x2f */
        outb_p(0x02, 0xA1);     /* 8259A-2 is a slave on master's IR2 */
        outb_p(0x01, 0xA1);     /* (slave's support for AEOI in flat mode
                                    is to be investigated) */

        if (auto_eoi)
                /*
                 * in AEOI mode we just have to mask the interrupt
                 * when acking.
                 */
                i8259A_irq_type.ack = disable_8259A_irq;
        else
                i8259A_irq_type.ack = mask_and_ack_8259A;

        udelay(100);            /* wait for 8259A to initialize */

        outb(cached_21, 0x21);  /* restore master IRQ mask */
        outb(cached_A1, 0xA1);  /* restore slave IRQ mask */

        spin_unlock_irqrestore(&i8259A_lock, flags);
}
```

Comments on Linux' 8259A Initialization Code

1. What is the auto_eoi parameter? always = 0

2. Four initialization control words to set up the master 8259A

3. Four initialization control words to set up the slave 8259A

	port(A=?)	info contained in Initialization Control Word
ICW1	0	start init, edge-triggered inputs, cascade mode, 4 ICWs
ICW2	1	high bits of vector #
ICW3	1	master: bit vector of slaves; slave: input pin on master
ICW4	1	ISA=x86, normal/auto EOI

Comments on Linux' 8259A Initialization Code (cont.)

5. What does the "_p" mean on the "outb" macros?
   - add PAUSE instruction after OUTB; "REP NOP" prior to P4
   - delay needed for old devices that cannot handle processor's output rate

6. Critical section spans the whole function; why?
   - avoid other 8259A interactions during initialization sequence
   - (device protocol requires that four words be sent in order)

7. Why use _irqsave for critical section?
   - this code called from other interrupt initialization routines
   - which may or may not have cleared IF on processor

Linux Abstraction of PICs

- Uses a jump table
  - same as vector table (array of function pointers)

- Table is *hw_irq_controller* structure (or struct irq_chip)
  - each vector # associated with a table
  - table used to interact with appropriate PIC (e.g., 8259A, or Advanced PIC)

human-readable name
startup function
shutdown function
enable function
disable function
mask function
mask_ack function
unmask function
(+ several others…)

Linux Abstraction of PICs

- *hw_irq_controller* structure definition
  - IRQs are #’d 0-15 (correspond to vector # - 0x20)

```
const char* name;
unsigned int (*startup)(unsigned int irq);
void (*shutdown)(unsigned int irq);
void (*enable)...
void (*disable)...
void (*ack)...
void (*end)...
/* we’ll ignore the others... */
```

8259A’s human-readable name is “XT-PIC”; see /proc/interrupts

PIC Functions in Jump Table: Explanation

- Initially, all 8259A interrupts are masked out using mask on 8259A

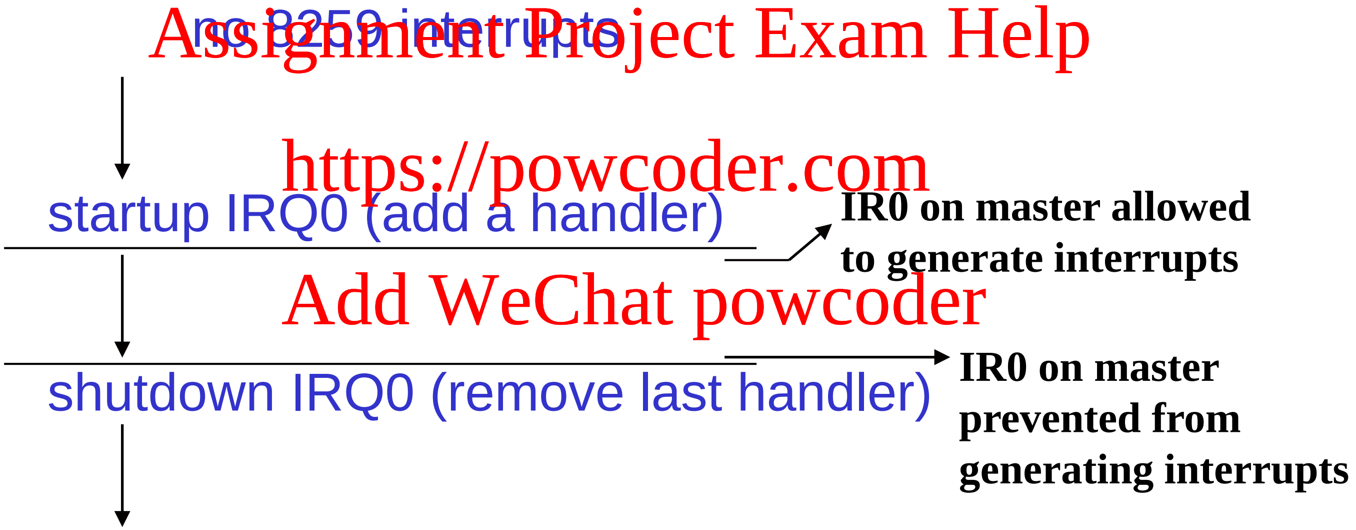
- *startup* and *shutdown* functions

  - startup is called when first handler is installed for an interrupt

  - shutdown is called after last handler is removed for an interrupt
  - both functions change the corresponding mask bit in 8259A implementaion

PIC Functions in Jump Table: Explanation (cont.)

- Example



no 8259 interrupts

startup IRQ0 (add a handler)

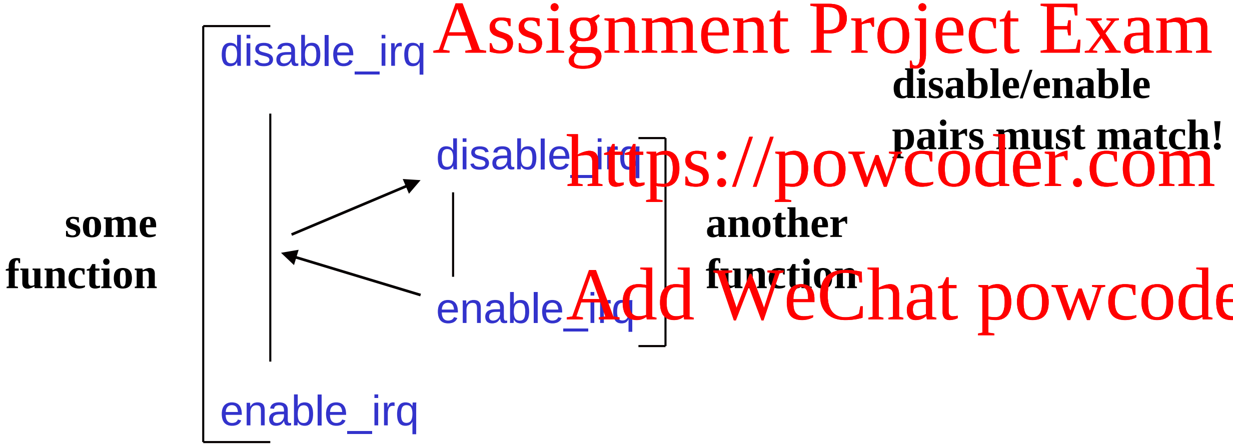
IR0 on master allowed to generate interrupts

shutdown IRQ0 (remove last handler)

IR0 on master prevented from generating interrupts

PIC Functions in Jump Table (cont.)

- disable/enable functions
  - used to support nestable interrupt masking (disable_irq, enable_irq)



  - on 8259
    - first *disable_irq* calls jump table disable, which masks interrupt on PIC
    - last *enable_irq* calls jump table enable, which unmasks interrupt on PIC

PIC Functions in Jump Table (cont.)

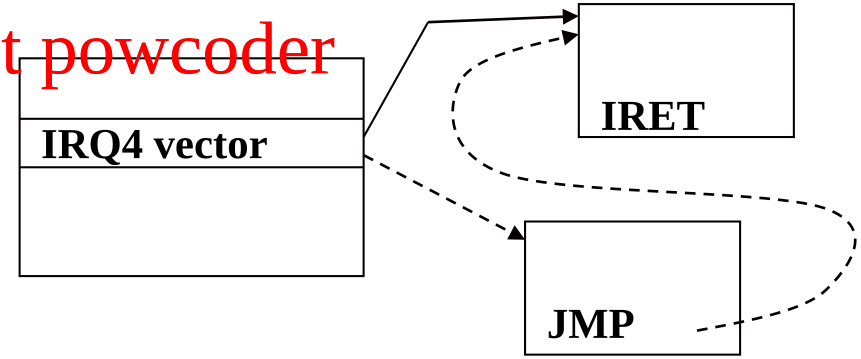
- ack function
  - called at start of interrupt handling to ack receipt of the interrupt
  - on 8259 (mask and ack) , masks interrupt on PIC, then sends EOI to PIC
- end function
  - called at end of interrupt handling
  - on 8259, enables interrupt (unmasks it) on PIC

General Interrupt Abstractions: Interrupt Chaining

- Hardware view: 1 interrupt → 1 handler
- Problems
  - may have > 15 devices
  - > 1 software routines may want to act in response to device
  - examples:
    - hotkeys for various functions
    - move mouse to lower-right corner to start screen-saver

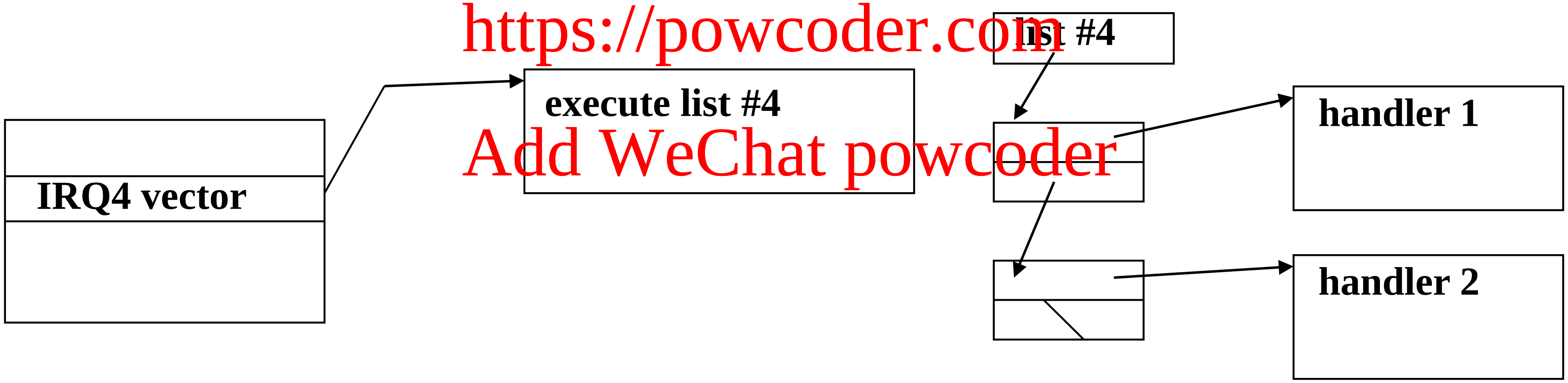
General Interrupt Abstractions: Interrupt Chaining (cont.)

- One approach
  - used by terminate and stay resident (TSR) programs in DOS
  - form linked list (chain) of handlers using JMP instructions
  - not very clean
    - no way to remove self
    - unless you're first in list
  - to be fair
    - TSR program not designed for removal



General Interrupt Abstractions
Interrupt Chaining (cont.)

- Solution
  - interrupt chaining with linked list data structure
  - (not list embedded into code!)



General Interrupt Abstractions: Interrupt Chaining (cont.)

- Drawbacks of chaining
  - for > 1 device
    - must query devices to see if they raised interrupt
    - not always possible
  - for 1 device
    - must avoid stealing data/confusing device
    - example
      - by sending two characters to serial port
      - in response to interrupt declaring port ready for one char.

General Interrupt Abstractions: Soft Interrupts (cont.)

- Recall: why support interrupts?
  - slow device gets timely attention from fast processor
  - processor gets device responses without repeatedly asking for them
- A useful concept in software
  - example: network encryption/decryption
  - packet arrives, given to decrypter
  - when decrypter (software program) is done
    - want to interrupt program
    - to transfer data from packet
  - but has no access to INTR pin