Interrupts

VEDA Solutions
Inspired by open Source

Interrupts

- ✓ An interrupt is an event that alters the sequence of instructions executed by a processor in corresponding to electrical signals generated by HW circuits both inside and outside CPU

- ✓ Interrupts: asynchronous interrupts
  Generated by HW devices (e.g., internal timers and I/O devices) at arbitrary times

- ✓ Exceptions: synchronous interrupts
  Produced by CPU control unit only after completion of an executing instruction
  E.g., divide-by-0, page faults

Exceptions

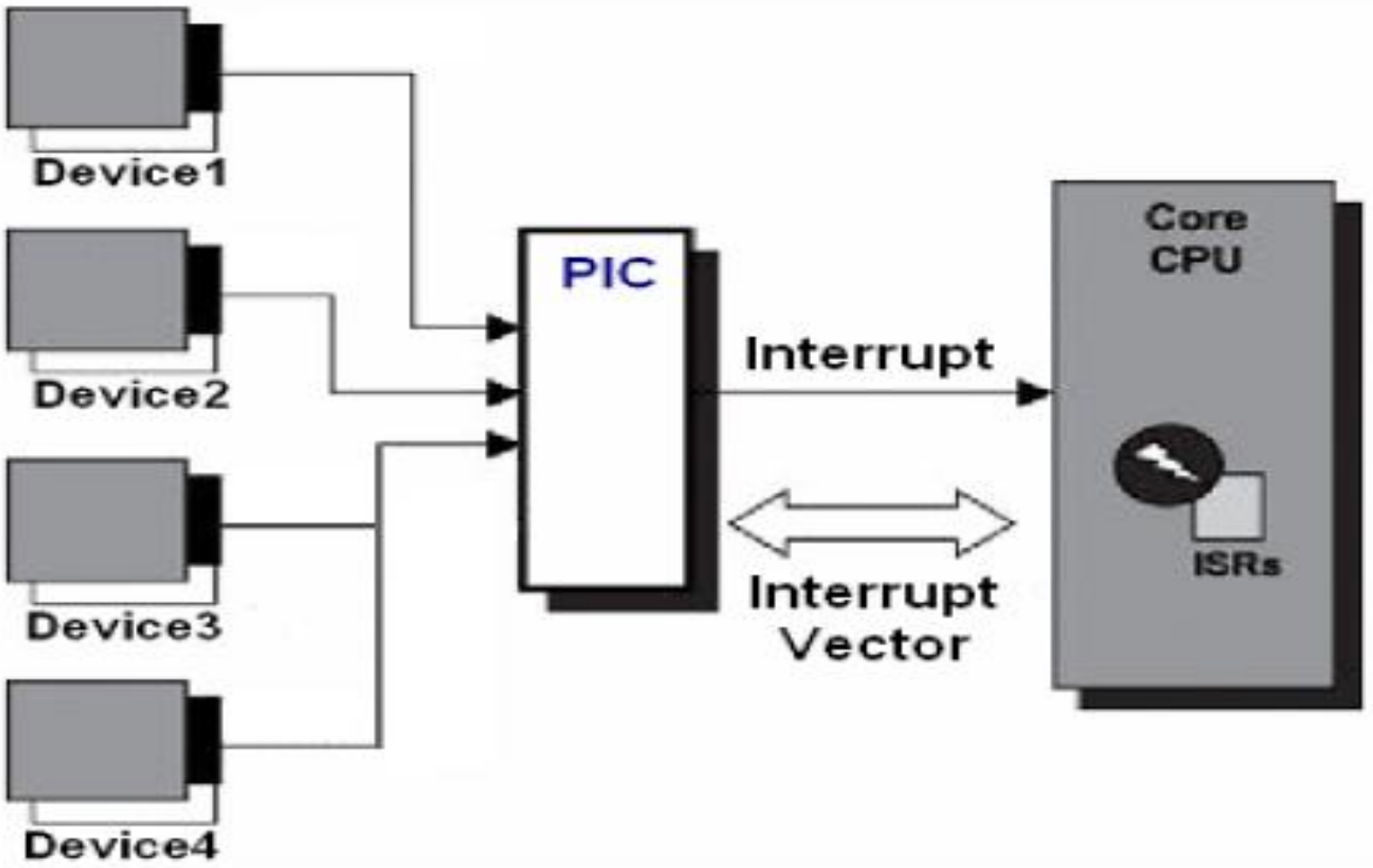
Processor-detected exceptions: when CPU detects anomalous condition while executing an instruction

- ✓ Faults: instructions causing faults
- ✓ Traps :Main usage: debugging purpose (e.g. reaching a breakpoint)
- ✓ Aborts: a serious error that may be unable to determine exact inst causing this error ➔ terminate affected process

Programmed exceptions: occur at the request of programmer

- ✓ Triggered by int, int3, into, bound instructions
- ✓ Handled by control unit as traps
- ✓ Often called SW interrupts
- ✓ Usage: to implement system calls and to notify a debugger of a specific event

IO interrupts



IRQ'S

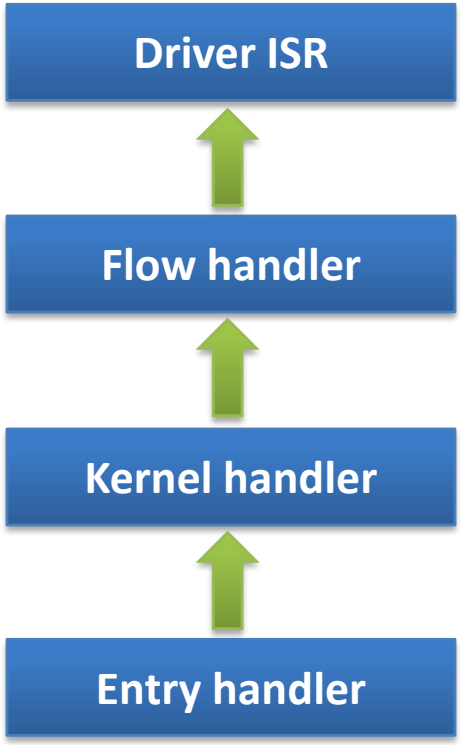
- ✓ Each HW device controller capable of issuing interrupts has an output line IRQ
- ✓ All existing IRQ lines are connected to the input pins of the Interrupt Controller
- ✓ Interrupt Controller (IC) executes

Monitoring IRQ lines, checking for raised signals
If a raised signal is detected on an IRQ line

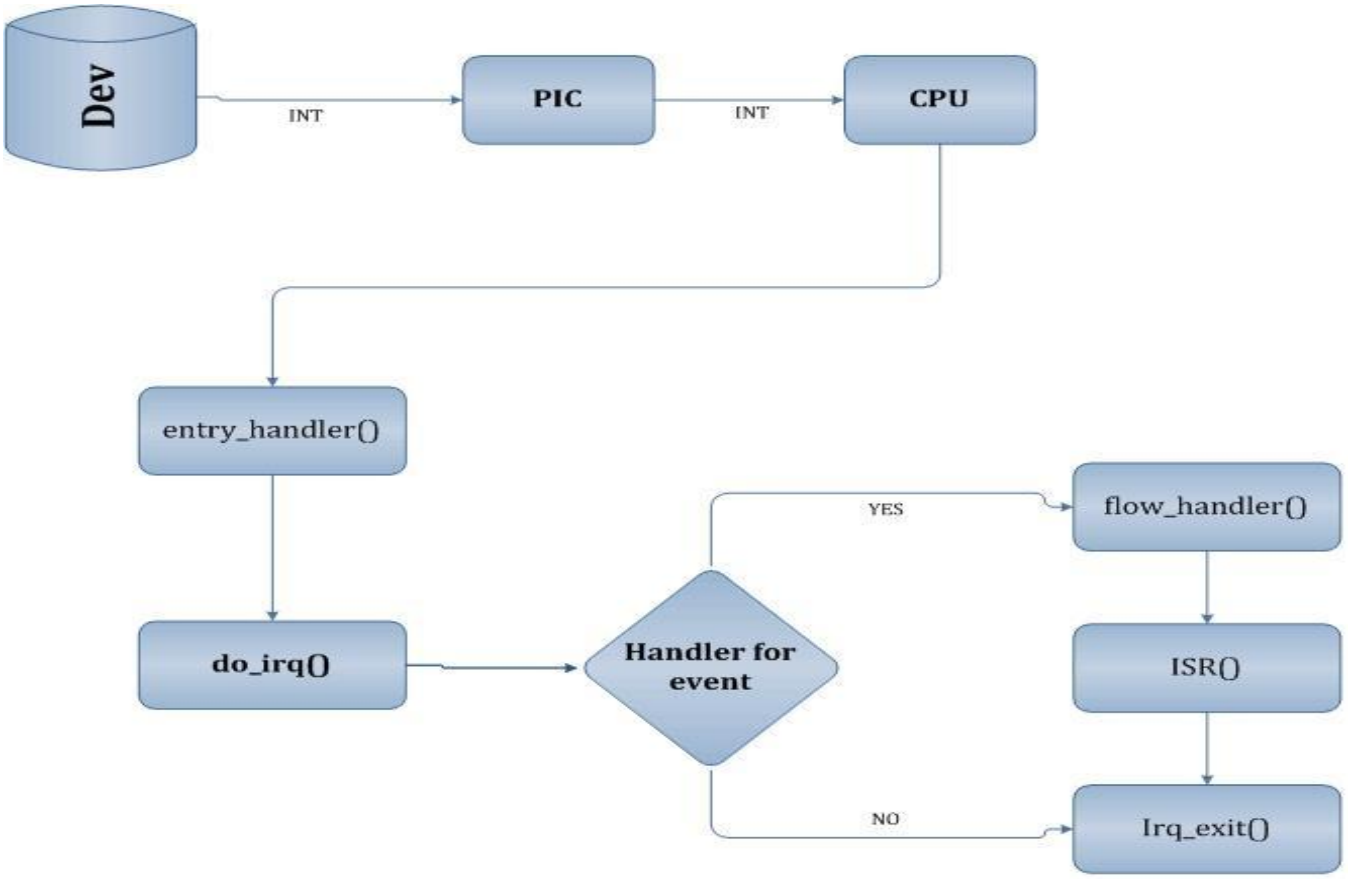
- ✓ Converts signal into a corresponding vector
- ✓ Stores vector in an IC I/O port, for CPU to read
- ✓ Sends a signal to CPU's INTR pin (i.e., issues an interrupt)
- ✓ CPU recognizes and writes to one of Programmable Interrupt Controller (PIC) I/O ports
- ✓ Clear INTR line

Go back to monitoring step

Linux interrupt Code



Interrupt Code Flow

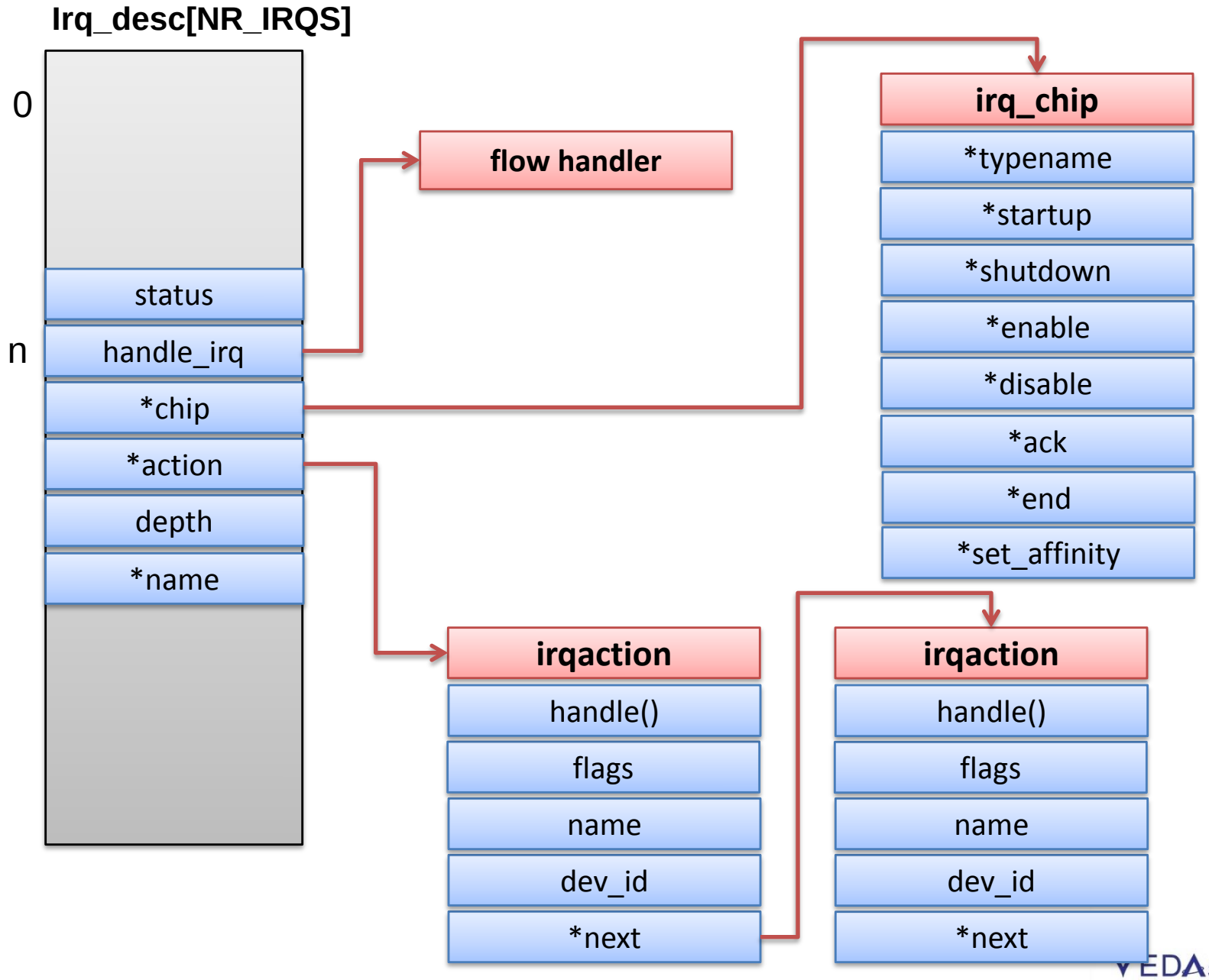


```
struct irq_desc {
      unsigned int status;                    /* IRQ line status */
      irq_flow_handler_t handle_irq;          /*   flow handler routine*/
      struct irq_chip  *chip;                 /*  chip operations */
      struct irqaction *action;               /* IRQ action ISR list */
      unsigned int depth;                     /* nested irq disables */
      const char *name                        /* name of flow control handler */
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];         // global variable


struct  irq_chip {
      const char * typename;
      unsigned int (*startup) (unsigned int irq);
      void (*shutdown) (unsigned int irq);
      void (*enable) (unsigned int irq);
      void (*disable) (unsigned int irq);
      void (*ack) (unsigned int irq);
      void (*end) (unsigned int irq);
      void (*set_affinity) (unsigned int irq, cpumask_t dest);
};
```

IRQ Descriptors



IRQ Status Listing

```
/*
 * IRQ line status.
 */

#define IRQ_INPROGRESS   1       /* IRQ handler active - do not enter! */
#define IRQ_DISABLED     2       /* IRQ disabled - do not enter! */
#define IRQ_PENDING      4       /* IRQ pending - replay on enable */
#define IRQ_REPLAY       8       /* IRQ has been replayed but not acked yet */
#define IRQ_AUTODETECT   16      /* IRQ is being autodetected */
#define IRQ_WAITING      32      /* IRQ not yet seen - for autodetection */
#define IRQ_LEVEL        64      /* IRQ level triggered */
#define IRQ_MASKED       128     /* IRQ masked - shouldn't be seen again */
#define IRQ_PER_CPU      256     /* IRQ is per CPU */
```

Registering Interrupt Service Routine

- ✓ Drivers can register an IH and enable a given interrupt line via

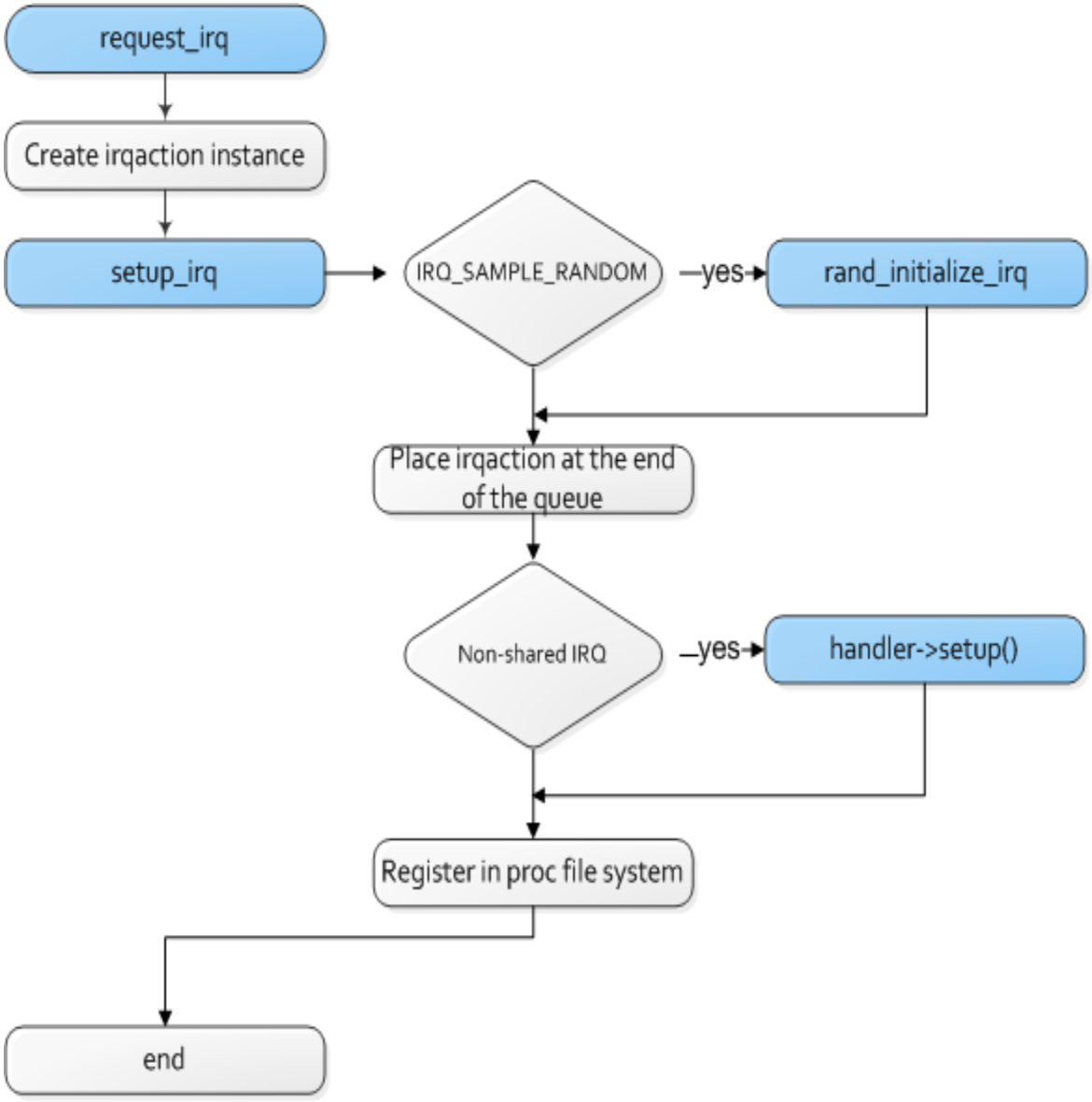
```
int int request_irq(unsigned int irq,
                    irqreturn_t handler,
                    unsigned long irqflags,
                    const char * devname,
                    void *dev_id);
```

  - irq: the interrupt line # to allocate
  - handler: pointer to actual ISR
  - irqflags: Attributes
  - devname: an ASCII text representation such as “keyboard”
  - dev_id: is used as an unique cookie when this line is shared
    - A common practice is to pass driver’s device structure

irqflags Options

- IRQF_DISABLED
  - The given IH is a fast IH: it runs with all interrupts disabled on local processor
  - By default (w/o this flag), all interrupts are enabled except the interrupt lines of any running handlers
- IFQF_PROBE_SHARED
  - Verify current irq is set to shared
- IFQF_SHARED
  - The interrupt line cab be shared among multiple ISRs
  - See include/linux/interrupt.h

request_irq()


request_irq
Create irqaction instance
setup_irq
IRQ_SAMPLE_RANDOM
yes
rand_initialize_irq
Place irqaction at the end of the queue
Non-shared IRQ
yes
handler->setup()
Register in proc file system
end

request_irq Usage

- To request an interrupt line and install a handler

```
if (request_irq(irqn, my_interrupt, IRQF_SHARED, “my-device”, dev)) {
    printk(KERN_ERR “my_device: cannot register IRQ %d\n”, irqn);
    return –EIO;
}
```

  - If return 0 ➔ handler was successfully installed

- To free an interrupt line, call

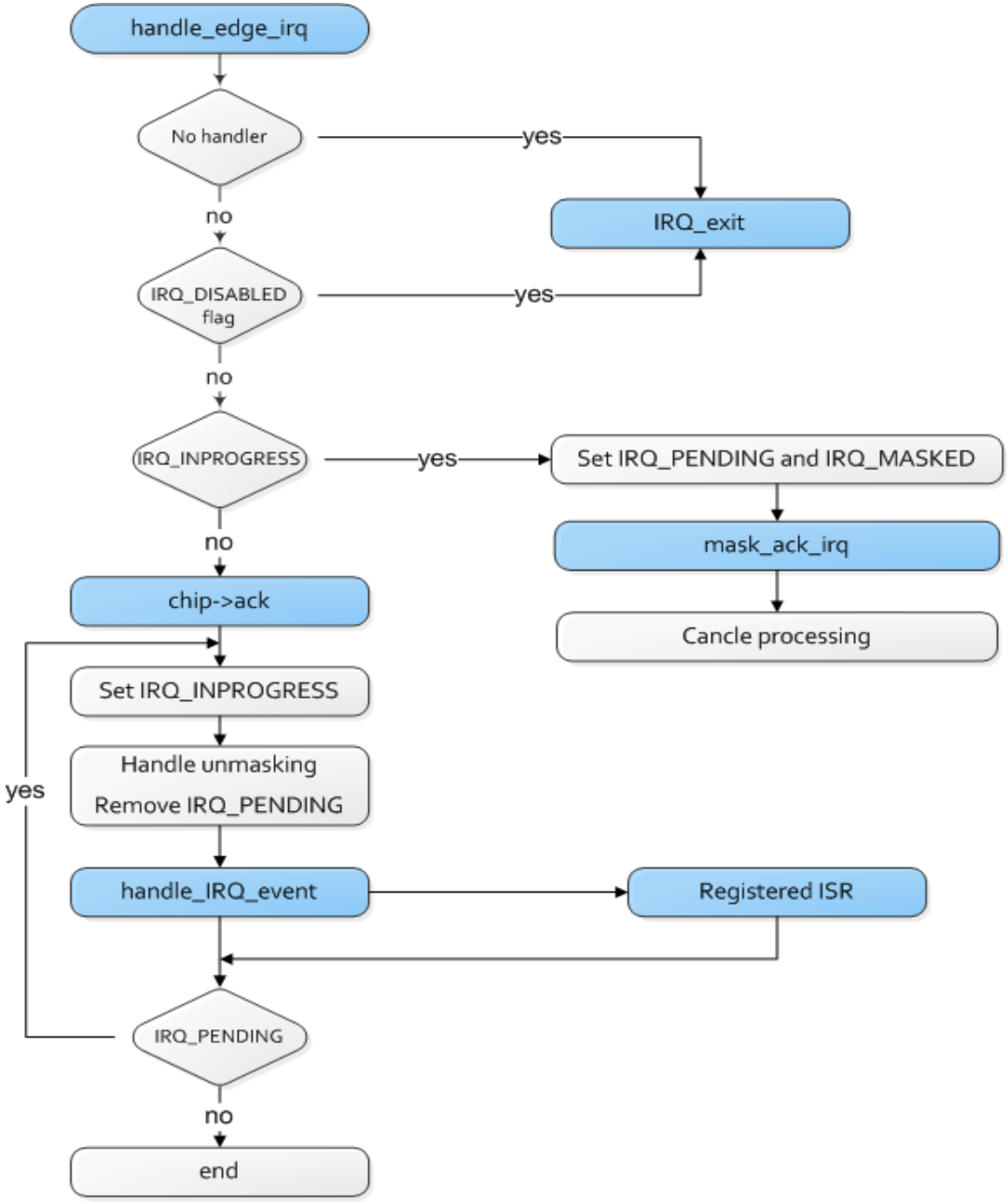
```
void free_irq(unsigned int irq, void *dev_id);
```

  - If line is not shared, it removes handler and disables the line
  - Otherwise, the line is only disabled at removal of last handler
  - dev_id is used to uniquely identify an interrupt handler

Interrupt Flow types

- ✓ Edge-triggered interrupts are signalled by a change in the interrupt line - from low voltage to high, from high to low, or both.
- ✓ These interrupts do not necessarily have to be masked while being processed
- ✓ kernel must track "pending" interrupts, and the interrupt handler must loop until all interrupts have been dealt with.

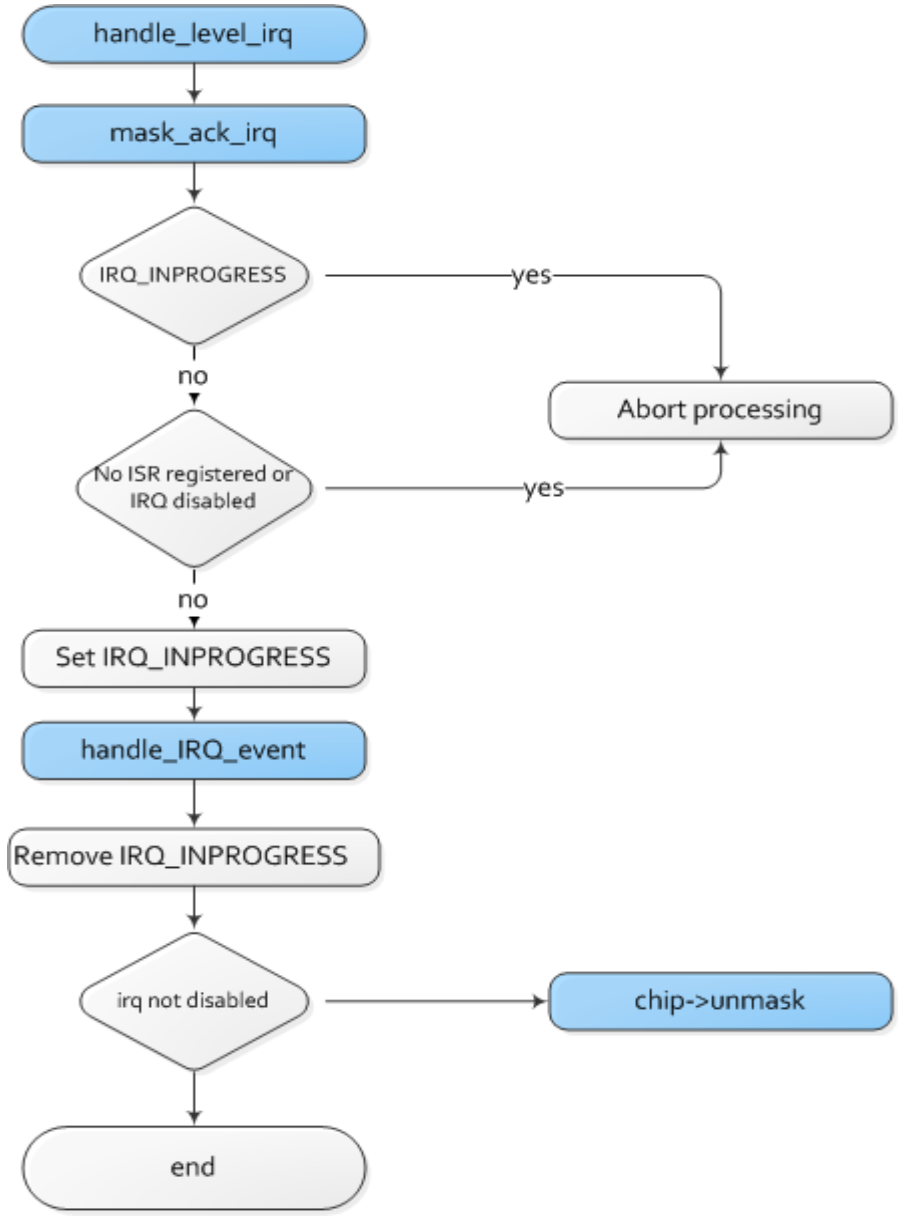
Edge handler



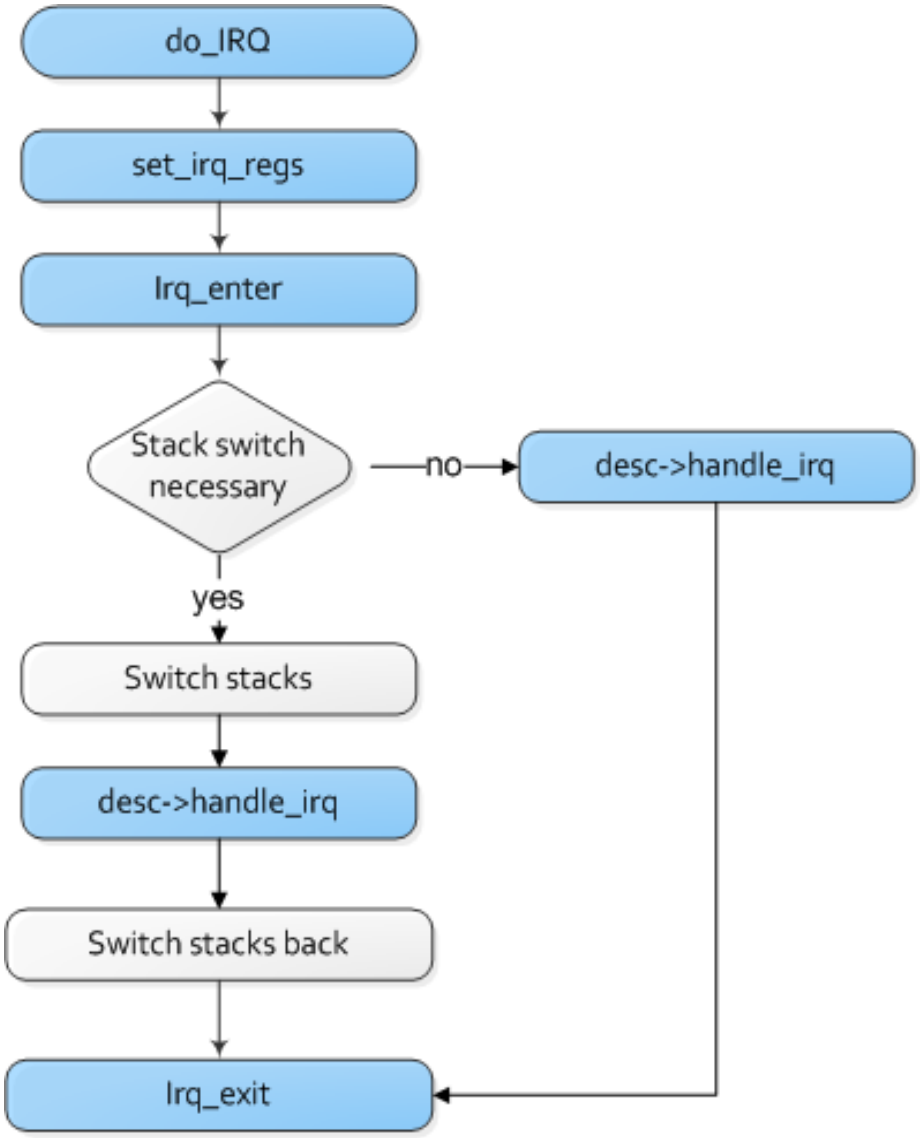
Interrupt Flow types

- ✓ A level-triggered interrupt is a class of interrupts where the presence of an interrupt is indicated by a high level (1), or low level (0), of irq line.
- ✓ A device wishing to signal an interrupt drives line to its active level, and then holds it at that level until serviced.
- ✓ Level-triggered interrupts are active as long as the device asserts its IRQ line.
- ✓ These interrupts must be masked while being processed

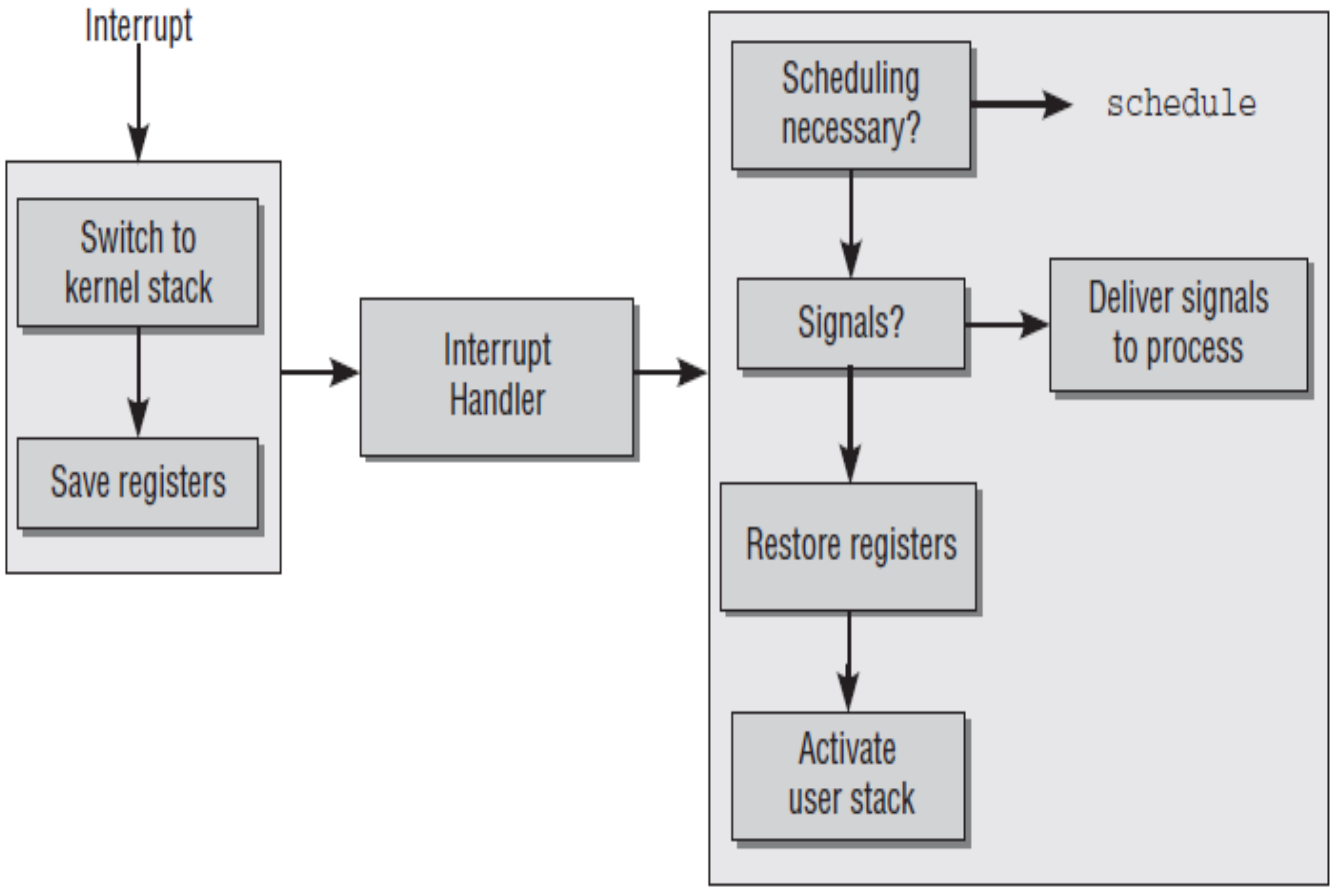
handle_level_irq



Irq stacks


do_IRQ
set_irq_regs
Irq_enter
Stack switch necessary
no
desc->handle_irq
yes
Switch stacks
desc->handle_irq
Switch stacks back
Irq_exit

Handling an Interrupt



Control Interfaces

- ✓ Purpose: to allow disabling the interrupt system for current CPU or mask out an interrupt line for entire machine
- ✓ Disable/enable interrupts locally for current processor:
  - local_irq_disable();
  - local_irq_enable();
  - local_irq_save(flags); // save and disable
  - local_irq_restore(flags); // restore and enable

Control Interfaces (2)

- ✓ Disable only a specific interrupt line for entire system
  - disable_irq(unsigned int irq);
    - Wait until any currently executing handler completes
  - disable_irq_nosync(unsigned int irq);
    - Will not wait
  - enable_irq(unsigned int irq);
    - If disable_irq() is called twice, only the 2nd enable_irq() will actually enable the interrupt line
  - synchronize_irq(unsigned int irq);
    - Wait for a specific IH to exit, if executing, before returning
- ✓ Status checking
  - irqs_disable()
    - returns nonzero if interrupt system on local CPU is disabled, or 0 otherwise
  - in_interrupt()
    - return nonzero if kernel is in interrupt context (including in IH or BH)
    - return zero if kernel is in process context
  - in_irq()
    - return nonzero if kernel is executing an interrupt handler