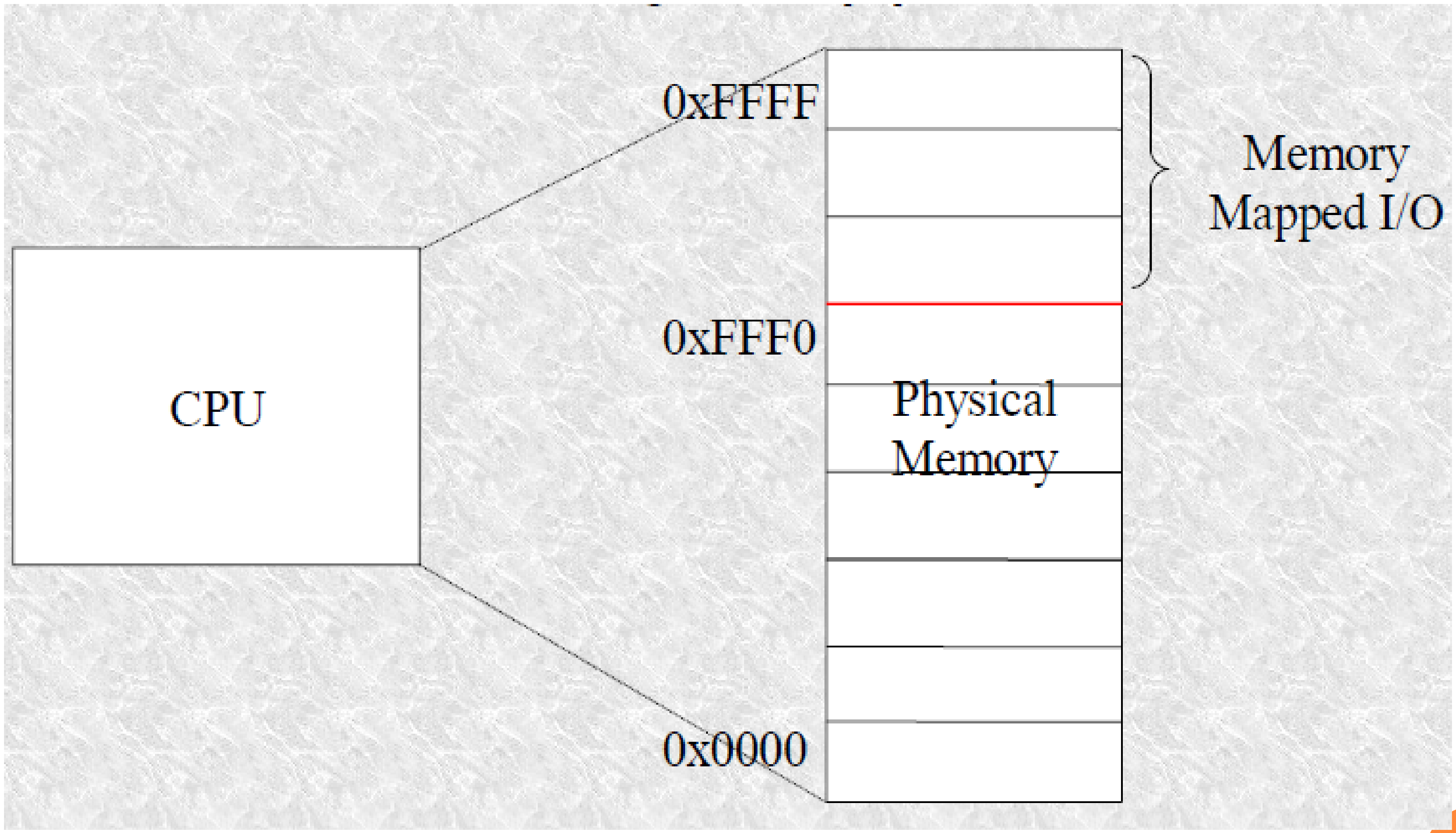
Communicating with Hardware & Interrupts

CDAC MUMBAI

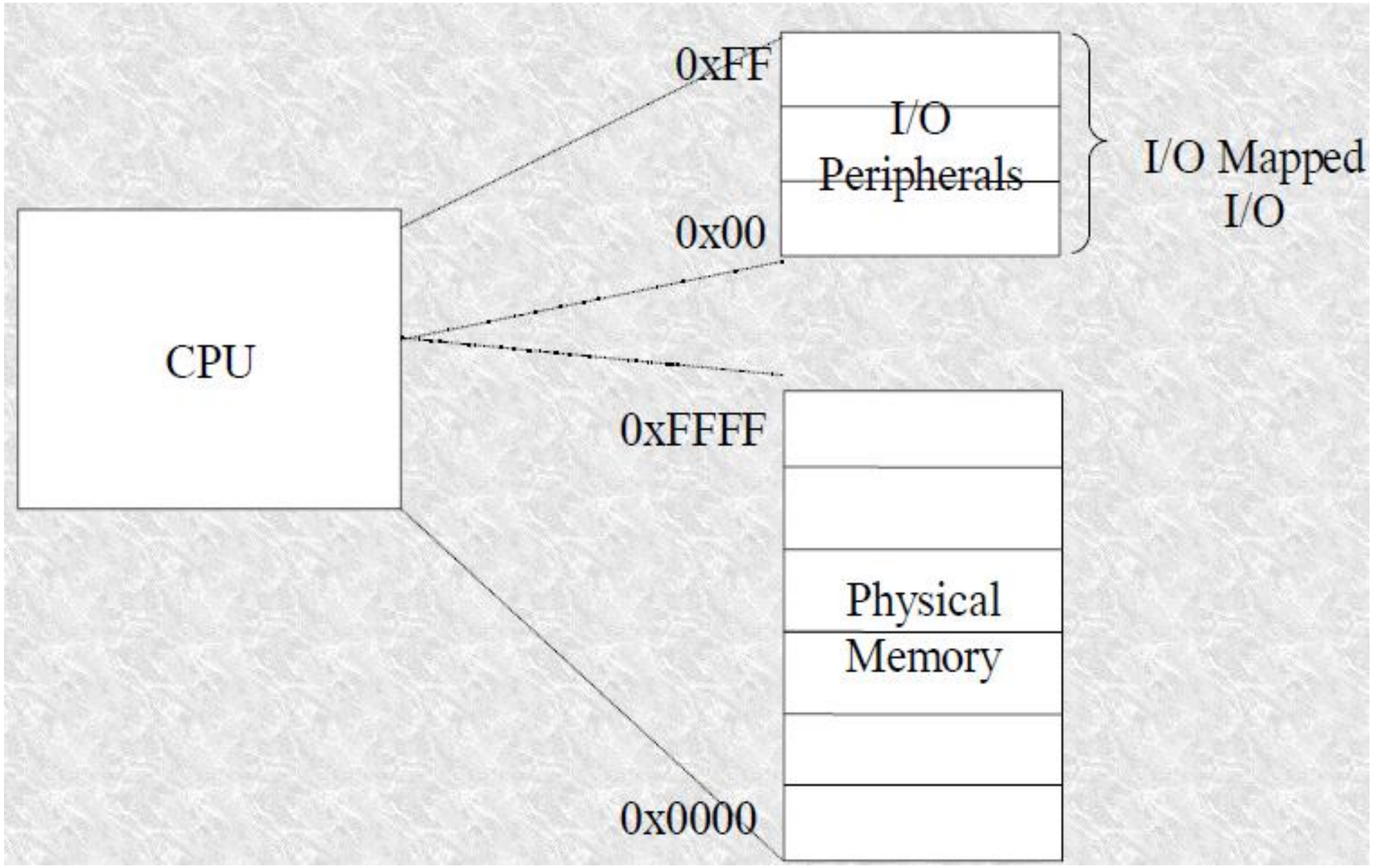
Agenda

- I/O Ports & I/O Mapping
- I/O Registers and Conventional Memory
- Using I/O Ports
- Parallel Port
- Using I/O Memory
- Installing the Interrupt Handler
- Auto Detecting the IRQ Number
- Implementing a Handler

I/O Ports and I/O Memory-Memory Mapped I/O



I/O Ports and I/O Memory -I/O Mapped I/O



I/O Memory and Conventional Memory

- I/O memory refers to the physical registers of an I/O peripheral whereas conventional memory refers to the physical memory of the system.
- References to the I/O memory is a bit precarious as compared to RAM.
- The reason for this is the optimization that the CPU does on conventional memory accesses for faster processing.

I/O Memory and Conventional Memory

- Access to RAM is possible only from the CPU and the operations performed mainly include reading and writing back to it.
- In order to maintain a faster access rate, optimization can be done at both hardware and compiler levels. These optimizations include caching of frequently accessed memory locations, and reordering of data for a better storage reasons.
- This may not be much of a hindrance to the system execution for conventional memory access, but the addition of these optimizations to I/O memory is a potential cause for system inconsistencies.

I/O Memory and Conventional Memory

- The problem of hardware caching is sorted out by disabling the hardware cache when accessing I/O regions. This process is carried out by the Linux kernel itself for any access to the I/O memory

- The solution to software optimization and hardware reordering is to place a memory barrier between the operations that must be visible to the hardware in a particular order

I/O Memory and Conventional Memory

- The linux kernel provides 4 macros to cover all possible ordering needs
  - void barrier(void)
- The function tells the compiler to insert a memory barrier, which does not affect the hardware. The inclusion of this barrier restricts the compiler from accessing the memory with any optimization.
- The compiled code stores to memory all the values that are currently modified and resident in CPU registers and rereads them later when needed.

I/O Memory and Conventional Memory

- void rmb(void);
- void wmb(void);
- void mb(void);

- These functions insert hardware memory barriers in the compiled instruction flow. They are supersets of the barrier macro.
- An rmb guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read.
- wmb orders write and mb orders both read and write.

Using I/O Ports -Port Allocation

- Drivers communicate with many devices through the I/O ports. Linux introduces a set of functions that may be used to gain exclusive access to an I/O region and communicate data transfers to and from these ports.
- Before embarking into communication with the I/O ports, the driver must gain access to the required ports by calling the function
  - struct resource *request_region(unsigned long first, unsigned long n, const char *name);
    - The argument *first is the first address requested*
    - The argument *long specifies the length of the* addresses requested
    - The argument *name identifies the requested region by* this name in /proc/ioports

I/O Port Allocation

- When the I/O ports have been used as per requirement, they should be returned to the kernel so that other drivers may avail their existence. The I/O ports are returned by the function
  - void release_region(unsigned long start, unsigned long n);
- Before using a I/O region, a check on the availability has to be conducted to be certain that the requested region will be available for our use. A special function allows this check.
  - int check_region(unsigned long first, unsigned long n);
- This function returns a negative error code if the region is not available. This is a deprecated function and may not always prove to be true since it does not run atomic with the request_region function.

Manipulating I/O Ports

- On successfully being allotted the region, the actual communication to the ports may be carried out using the kernel provided functions.
- These functions are specific to the port sizes on the I/O devices and provide interfaces for 8-bit, 16-bit and 32-bit ports
  - unsigned inb (unsigned port);
  - unsigned inw (unsigned port);
  - unsigned inl (unsigned port);
- Get data from the port specified as an argument
  - void outb (unsigned char byte, unsigned port);
  - void outw (unsigned short word, unsigned port);
  - void outl (unsigned long word, unsigned port);
- Send data to the port specified in the argument

I/O Port Access from User Space

- The functions described before are restricted to kernel space, but the GNU C library defines user space variants for the same. The following conditions have to be applied for access of I/O ports from the user space
- The program must be compiled with the -O option to force the expansion of inline functions
- The *ioperm and iopl system calls should be used to get* permission to perform I/O operations on ports. *Ioperm* gets permission for individual ports, while *iopl gets* permission for the entire I/O space
- The programs must be run as root to gain access to the ports

String Operations and Pausing I/O

- The kernel also supports string operations that render its services in allowing a string of 'n' bytes to be transferred between the driver and the I/O port
  - void insb (unsigned port, void *addr, unsigned long count);
- Similarly..insw, insl
- Copies count bytes of information from the port to addr
  - void outsb(unsigned port, void *addr, unsigned long count);
- Write count data pointed by addr to the port
- The kernel provides mechanisms of synchronization between a high end processor and a relatively slower I/O by allowing a pause functionality in data transfer. This feature may be accessed by ending the function names previously discussed with an '_p', such as inb_p, outb_p....

INTERRUPTS

- Devices have to deal with numerous functions such as spinning disks, moving tape, wires etc.., Majority of these actions are controlled by an Operating System that sits on a high end processor.
- Since it is undesirable to have a processor wait for each of these events through mechanisms like polling, the devices themselves contend for the processor as and when their needs warrant. This is achieved through interrupts.
- An interrupt is simply a signal that the hardware can send when it wants the processor's attention.
- Linux handles interrupts in much the same way as signals of the user space
- The driver needs to only register a handler for its device's interrupts, and handle them properly whenever they arrive

Interrupts and Exceptions

- Interrupts and exceptions are much similar in the way they function, but for some small differences between the two.
- The most evident difference between an interrupt and an exception is that the former acts asynchronously, whereas the latter occurs in synchronization with the system clock.
- The kernel infrastructure to handle both interrupts and exceptions is similar.
- Exceptions in most cases refer to programming errors (for example a divide by 0) or abnormal conditions that must be added by the kernel (such as a page fault).
- Exceptions are not necessarily a negative phenomenon. Even system calls that are issued by the user space applications are treated as exceptions in the kernel

Interrupt Handler

- Different devices can be associated with unique interrupts by means of a unique value associated with each interrupt. These interrupt values are often called Interrupt Request Lines or simply as IRQ Numbers.
- Every device which generates an interrupt must be associated with an handler for that interrupt. The handler is part of the device driver that drives the device.
- The handler implements a specific role of interacting with the device for data transfer between the application and the device.
- Each handler has to register itself with the kernel whenever an operation is to be performed on the device. Registering a handler is a notification by the driver to the kernel, claiming authority for access to the device through a requested IRQ number.
- On completion of access to the device, the handler may be unregistered and the irq number freed for allowing access to other applications.

Installing an Interrupt Handler

- Registering an interrupt handler to an irq number and notifying the kernel is done by
  - int request_irq(unsigned int irq, irqreturn_t (*handler) (int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);
- The value returned from *request_irq to the is either 0* to indicate success or a negative error code, as usual.
- It is not uncommon for the function to return –EBUSY to signal that another driver is already using the requested interrupt line.
- The handler is freed by calling
  - void free_irq (unsigned int irq, void *dev_id);

Arguments to request_irq

- unsigned int irq

- The interrupt number being requestd

- irqreturn_t (*handler)(int, void *, struct pt_regs *)

- Pointer to the interrupt handler that is involved

- const char *dev_name

- Just a simple name to the irq being alloted.

- void *dev_id

- Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and may also be used by the driver to point to its own private data area.

- If the interrupt is not shared, it may be set to NULL, but in general points to the device structure.

Arguments to request_irq

- unsigned long flags
- There are three flags that may be used as a bit mask of options

  - SA_INTERRUPT
- This bit indicates a “fast” handler.

  - SA_SHIRQ
- This bit indicates that the interrupt can be shared between devices.

  - SA_SAMPLE_RANDOM
- Adds to the kernel entropy pool to increase the randomness.

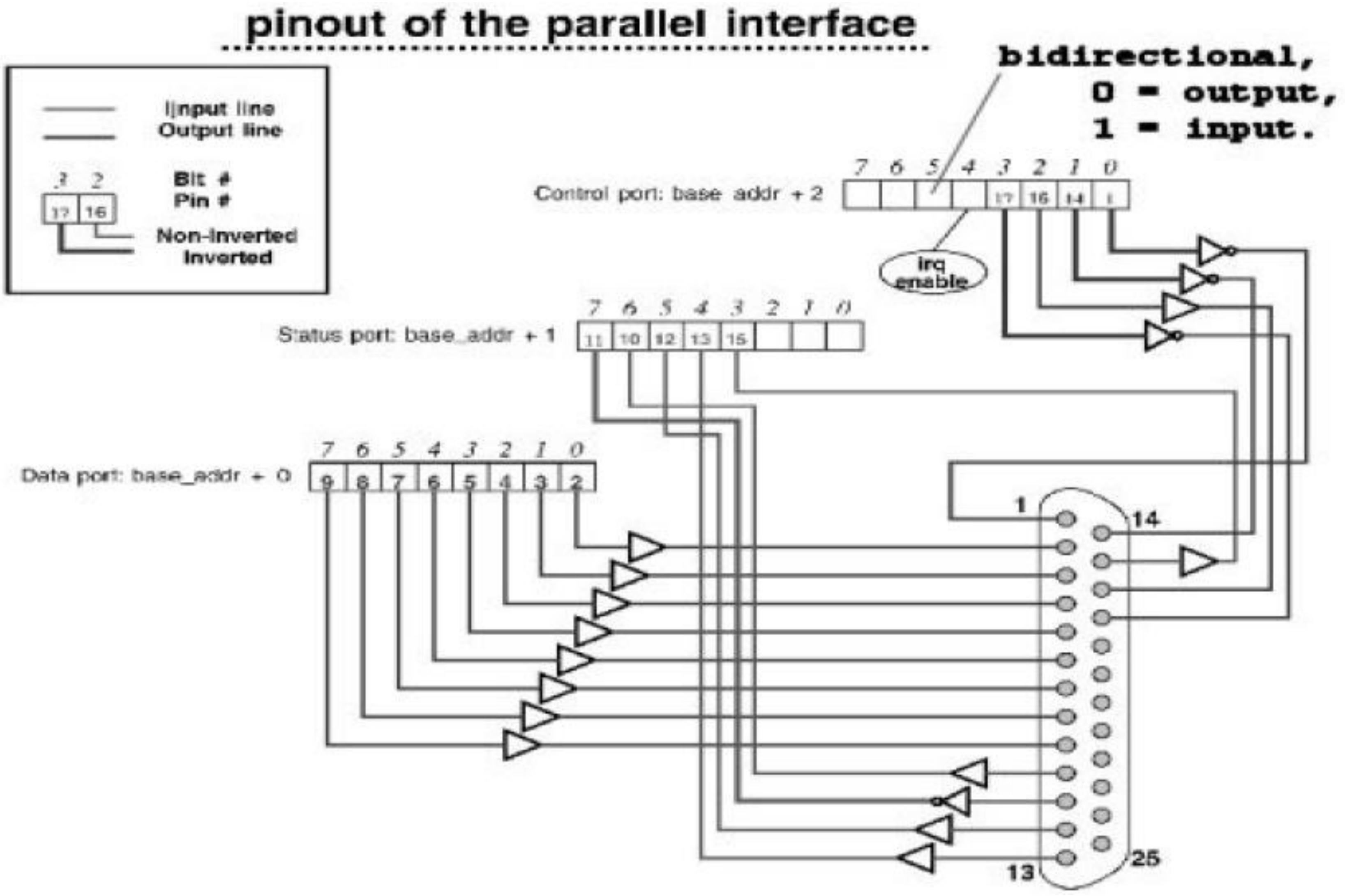
Where to install an Interrupt Handler

- There are two places where an interrupt handler can be installed.

  - During the driver initialization at init_module

  - In the open method of the driver
- Installing a handler in the init_module does not prove to be a very efficient idea, especially if the driver does not share interrupts. Because the number of interrupt lines are limited, it cannot be wasted and has to be used sparingly.
- If the driver selects the interrupt as shared, the handler may be installed in the init_module.
- If implemented in the open call, it allows the use of the interrupt line only when the application requests for it.
- The disadvantage of this technique is that a per device open count has to be maintained to know when interrupts can be disabled.

Parallel Port



AUTO DETECTING THE IRQ NUMBER

- One of the most challenging problems for a driver at initialization can be how to determine which IRQ line is going to be used by the device
- Sometimes, auto detection depends on the knowledge that some devices feature a default behaviour that rarely changes. In such cases, the IRQ number may be defaulted according to the standard. For example, Parallel Port is defaulted to 0x378 with an IRQ of 7.
- Some devices are more advance and simply announce which interrupt they are going to use. In this case, the driver simply retrieves this information by reading a status byte from one of the device's I/O ports. This mechanism of getting the IRQ number from the device directly is called probing for the IRQ number. Eg…PCI devices

Kernel Assisted Probing

- The Linux kernel offers a low-level facility for probing the interrupt number. It works only for non shared interrupts.
- Most hardware that are capable of working in a shared interrupt mode provide better ways of finding the configured interrupt number.
- The kernel provides two functions to achieve the same

  - unsigned long probe_irq_on (void)
    - It returns a mask of unassigned interrupts. The driver must preserve the returned bit mask and pass it to the other function.

  - int probe_irq_off (unsigned long)
    - Returns the irq number that the device uses. If no interrupts occurred, 0 is returned. If more than one interrupt occurred, a negative value is returned

Kernel Assisted Probing

- mask = probe_irq_on();
  - outb_p (0x10, short_base+2); /enable interrupts
  - outb_p (0x00, short_base); /clear the bit
  - outb_p (0xFF, short_base); /Set the bit
  - outb_p (0x00, short_base+2); //disable the interrupt

  udelay(5);

  irq_number = probe_irq_off(mask);

Do it Yourself Probing

Guess????????

DO IT YOURSELF

Fast & Slow Handlers

- Historically, Linux differentiated between interrupt handlers that were fast versus slow. Fast Handlers were assumed to execute quickly, but potentially very often, so the behaviour of the interrupt handling was modified to enable them to execute as quickly as possible
- Today, there is only one difference between: Fast interrupt handlers run with all interrupts disabled on the local processor. This enables fast handlers to complete very quickly without being interrupted by other interrupts
- By default (without this flag), all interrupts are enabled except the interrupt lines of any running handlers which are masked out on all processors

Implementing an Interrupt Handler

- The typical declaration for an interrupt handler:
  - static irqreturn_t intr_handler (int irq, void *dev_id, struct pt_regs *regs);
  - The return value of an interrupt handler is the special type irqreturn_t.
  - An interrupt handler can return two special values, IRQ_NONE or IRQ_HANDLED.
  - IRQ_NONE is returned when the handler detects an interrupt for which its device was not the originator.
  - IRQ_HANDLED is returned if the interrupt handler was correctly invoked

Shared Handlers

- A shared handler is registered and executed much like a non-shared handler. There are three main differences:
  - The SA_SHIRQ flag must be set in the *flags argument to request_irq();*
  - The dev_id must be unique to each registered handler. A pointer to any per-device structure is sufficient. NULL cannot be passed to this field
  - The interrupt handler must be capable of detecting whether its device generated an interrupt. This requires both hardware support and associated logic in the handler

Enabling and Disabling Interrupts

- Single Interrupts

  - Sometimes the driver may need to disable interrupt delivery for a specific line. This is achieved by using one of the functions

  - void disable_irq(int irq);
  - void disable_irq_nosync(int irq);
  - void enable_irq(int irq);
- Disable all Interrupts
  - void local_irq_save(unsigned long flags);
  - void local irq_disable(void);
- Re-enables the interrupts
  - void local_irq_restore(unsigned long flags);
  - void local_irq_enable(void);