**How Linux kernel handles interrupt**

At any time one CPU in a Linux system can be:

* serving a hardware interrupt in kernel mode
* serving a softirq, tasklet or bottom half in kernel mode
* running a process in user mode.

Hardware devices capable of issuing IRQs are connected to *Interrupt Controller*. Interrupt Descriptor Table (IDT) associates each exception or interrupt vector with a gate descriptor for the handler used to service the associated exception or interrupt . IDT need not contain more than 256 descriptors since there are only that amount of interrupt or exception vectors. The IDT must be properly initialized before the kernel enables interrupts. The idtr register allows the IDT to be located anywhere in the memory. Linux uses interrupt gates to handle interrupts and trap gates to handle mexceptions. Linux does not use task gates.

All interrupt handlers preform the same five basic actions:

1. Save IRQ value and the registers to the kernel mode stack.

2. Send ack to the PIC thus allowing it to issue next interrupt.

3. Execute ISRs associated with all the devices that share same IRQ.

4. Execute active softirqs

5. Terminate by jumping to the ret from intr().

**Data structures**

An irq desc array includes NR IRQS irq desc t type descriptors.

typedef struct {

unsigned int status; /\* IRQ status \*/

hw\_irq\_controller \*handler;

struct irqaction \*action; /\* IRQ action list \*/

unsigned int depth; /\* nested irq disables \*/

spinlock\_t lock;

} \_\_\_\_cacheline\_aligned irq\_desc\_t;

The **hw interrupt type** descriptor includes group of pointers to the low-level I/O routines that interact with a specific interrupt controller.

struct hw\_interrupt\_type {

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,

unsigned long mask);

};

typedef struct hw\_interrupt\_type hw\_irq\_controller;

The irqaction descriptors can be chained in case of shared IRQs used

struct irqaction {

void (\*handler)(int, void \*, struct pt\_regs \*);

unsigned long flags; //discussed later

unsigned long mask;

const char \*name;

void \*dev\_id;

struct irqaction \*next;

};

**The asm\_do\_IRQ() Function**

The asm\_do\_ IRQ() function handles all normal device IRQs (IPIs have their own specific handlers). It first gets lock to the specific IRQ descriptor, so that the first CPU getting the lock takes care of that specific IRQ. Next thing is to acknowledge the IRQ to PIC as fast as possible:

desc->handler->ack(irq);

After that the status of the IRQ descriptor is updated (”we want to handle this specific IRQ”) and after that IRQ descriptor is unlocked. Next thing is to call the IRQ handler:

handle\_IRQ\_event(irq, &regs, action);

following IRQ descriptor locking, IRQ descriptor status updating,IRQ descriptor unlocking and calling:

desc->handler->end(irq);

to deal with interrupts which got disabled while the handler was running and finally check and execute possible softirqs:

if (softirq\_pending(cpu))

do\_softirq();

return 1;

Interrupt service routines (ISRs) implementing device specific function are all called with same parameters, which

are: irq, dev id and regs. The first parameter allows service of multiple IRQs inside one ISR, the second parameter

allows service of several devices of same type and the last parameter allows the access to the execution context of the interrupted kernel control path. In practice these parameters are seldom used inside ISRs.

**Returning from Interrupts**

There are three entry points ret from **intr(), ret from sys call() and ret from exception()** which are used when returning from interrupts, exceptions and system calls.

**Installing an Interrupt handler**

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 that it handles signals in user space. For the most part, a driver need only register a handler for its device’s interrupts, and handle them properly when they arrive.

A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers,

as we will see. The following functions, declared in *<linux/interrupt.h>*, implement the interrupt registration interface:

*int request\_irq(unsigned int irq, irqreturn\_t (\*handler)(int, void \*, struct pt\_regs \*), unsigned long flags, const char \*dev\_name, void \*dev\_id);*

*void free\_irq(unsigned int irq, void \*dev\_id);*

The value returned from *request\_irq* to the requesting function is either 0 to indicate success or a negative error code, as usual. It’s not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

*unsigned int irq*

The interrupt number being requested.

*irqreturn\_t (\*handler)(int, void \*, struct pt\_regs \*)*

The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

unsigned long flags

As you might expect, a bit mask of options (described later) related to interrupt management.

*const char \*dev\_name*

The string passed to *request\_irq* is used in */proc/interrupts* to show the owner of the interrupt.

*void \*dev\_id*

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, dev\_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure.

The bits that can be set in flags are as follows:

SA\_INTERRUPT

When set, this indicates a “fast” interrupt handler. Fast handlers are executed with interrupts disabled on the current processor

SA\_SHIRQ

This bit signals that the interrupt can be shared between devices.

SA\_SAMPLE\_RANDOM

This bit indicates that the generated interrupts can contribute to the entropy pool used by */dev/random* and */dev/urandom*. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption. Such random numbers are extracted from an entropy pool that is contributed by various random events. If your device generates interrupts at truly random times, you should set this flag.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module’s initialization function might sound like a good idea, it often isn’t, especially if your device does not share interrupts. Because the number of interrupt lines is limited,

you don’t want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used. Requesting the interrupt at device open, on the other hand, allows some

sharing of resources.

**Enabling/Disabling Interrupts**

There are times when a device driver must block the delivery of interrupts for a (hopefully short) period of time. Often, interrupts must be blocked while holding a spinlock to avoid deadlocking the system. There are ways of disabling interrupts that do not involve spinlocks.

void disable\_irq(int irq);

void disable\_irq\_nosync(int irq);

void enable\_irq(int irq);

Calling any of these functions may update the maskfor the specified irq in the programmable interrupt controller (PIC), thus disabling or enabling the specified IRQ across all processors.

*disable\_irq* not only disables the given interrupt but also waits for a currently executing interrupt handler, if any, to complete.

In the 2.6 kernel, it is possible to turn off all interrupt handling on the current processor with either of the following two functions (which are defined in *<asm/system.h>*):

*void local\_irq\_save(unsigned long flags);*

*void local\_irq\_disable(void);*

A call to *local\_irq\_save* disables interrupt delivery on the current processor after saving the current interrupt state into flags. Note that flags is passed directly, not by pointer. *local\_irq\_disable* shuts off local interrupt delivery without saving the state; you should use this version only if you know that interrupts have not already been

disabled elsewhere. Turning interrupts back on is accomplished with:

*void local\_irq\_restore(unsigned long flags);*

*void local\_irq\_enable(void);*

The first version restores that state which was stored into flags by *local\_irq\_save*, while *local\_irq\_enable* enables interrupts unconditionally. Unlike *disable\_irq*, *local\_irq\_disable* does not keep track of multiple calls. If more than one function in the call chain might need to disable interrupts, *local\_irq\_save* should be used.

(Note: There are 32 interrupts in arm.In arch/arm/kernel/entry-armv.S,a function asm\_do\_IRQ is being called.For each IRQ,there will be a structure “struct irq\_desc”…in the file before that do\_IRQ is called it will save the current value i.e

irq\_save\_user\_regs

bl do\_irq

irq\_restore\_user\_regs

(Difference between Exception/Interrupt-**Interrupts** are hardware generated but **exceptions** are software generated i.e through instructions. **Interrupts** are handled by the processor after finishing the current instruction. If it finds a signal on its interrupt pin, it will look up the address of the interrupt handler in the interrupt table and pass that routine control. After returning from the interrupt handler routine, it will resume program execution at the

instruction after the interrupted instruction. **Exception**s on the other hand are divided into three kinds. These are *Faults, Traps and Aborts*. *Faults* are detected and serviced by the processor before the faulting instructions. *Traps* are serviced after the instruction causing the trap. User defined interrupts go into this category and can be said to be traps; this includes the MS-DOS INT 21h software interrupt, for example. Aborts are used only to signal severe system problems, when operation is no longer possible.)

**Top/Bottom Halves**

One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other,

leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called *top half* is the routine that actually responds to the interrupt—the one you register with *request\_irq*. The *bottom half* is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that’s why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

**Softirq**

Softirqs are the basic bottom half mechanism and have strong locking requirements. They are used only by a few performance-sensitive subsystems such as the networking layer, SCSI layer, and kernel timers.

**Tasklets**

Tasklets must be declared with the DECLARE\_TASKLET macro:

*DECLARE\_TASKLET(name, function, data);//we can use tasklet\_init also*

*name* is the name to be given to the tasklet, *function* is the function that is called to execute the tasklet (it takes one unsigned long argument and returns void), and *data* is an unsigned long value to be passed to the *tasklet* function.

The function *tasklet\_schedule* is used to schedule a tasklet for running.

*tasklet\_schedule(&short\_tasklet);*

You will find a structure “struct tasklet\_struct” in /include/linux/interrupt.h.

**Workqueue**

Steps to implement workqueue

* Create a work queue (or a workqueue\_struct) with one or more associated kernel threads. To create a kernel thread to service a ***workqueue\_struct***, use ***create\_singlethread\_workqueue().*** To create one worker thread per CPU in the system, use the ***create\_workqueue()***variant
* Create a work element (or a work\_struct). A ***work\_struct*** is initialized using ***INIT\_WORK(),*** which populates it with the address and argument of your work function.
* Submit the work element to the work queue. A ***work\_struct*** can be submitted to a dedicated queue using ***queue\_work(),*** or to the default kernel worker thread using ***schedule\_work().***

In external uart through i2c

struct xrm1172\_uart\_port {

struct uart\_port port;

struct workqueue\_struct \*workqueue; ///include/linux/workqueue.h

struct work\_struct work;

…….

};

In startup()

struct xrm1172\_uart\_port \*s = container\_of(port, struct xrm1172\_uart\_port ,port );

s->workqueue = create\_freezeable\_workqueue("XRM7112");

if(!s->workqueue)

{

printk("SPA Errr in Workqueue XRM7112 \n");

return -EBUSY;

}

INIT\_WORK(&s->work, xrm1172\_work);

In the ISR,call

xrm1172\_do\_work(chip);

where

struct xrm1172\_uart\_port \*chip = dev\_id;

static void xrm1172\_do\_work(struct xrm1172\_uart\_port \*s)

{

if (!work\_pending(&s->work))

queue\_work(s->workqueue,&s->work);

}

static void xrm1172\_work(struct work\_struct \*w)

{

….

Handle receive characters or Interrupt

….

}

**Difference between the above three**

**Softirqs Tasklets Work Queues**

**Execution**

**Context** Deferred work runs in deferred work runs in Deferred work runs in process

interrupt context interrupt context. Context.

**Reentrancy** Can run simultaneously Cannot run simultaneously Can run simultaneously

on different CPUs. on different CPUs. Different on different CPUs

CPUs can run different

tasklets

**Sleep**

**Semantics** Cannot go to sleep Cannot go to sleep May go to sleep

**Preemption** Cannot be preempted Cannot be preempted/ May be preempted/

/scheduled. scheduled. scheduled.

**When to use** If deferred work will not If deferred work will not If deferred work

go to sleep and if you go to sleep may go to sleep

have crucial scalability

or speed requirements.