

Learn today. Design tomorrow.



Boston



Device Drivers Demystified ESC-408

Doug Abbott, Principal Consultant



www.intellimetrix.us





➤ **Introduction**

- Why device drivers?
- What's a device driver?

➤ **Abstract model of device driver**

- "OS agnostic"

➤ **What makes drivers seem complicated?**

- Independently loaded program
- Protected modes
- **Interrupts!!!**



- **The Linux driver model**
 - Kernel loadable modules
 - Character device drivers
 - User space/Kernel space communication
 - Interrupt handling
 - Blocking I/O

Why device drivers?



- **It's good programming practice!!!**
 - Cleanly separate hardware dependent code from hardware independent code
 - Hides “messy” programming details from application programmer

- **I/O access may require higher privilege level**

What's a "device driver" anyway?



- **A Set of Linkable Functions**
- **An Independently Loadable Program**
- **A Separate Task or Set of Tasks**
- **A Well-defined driver API**
- **All of the Above??**



- **int read (void *buffer, int length, int *actual);**
- **int write (void *buffer, int length, int *actual);**

Multiple device types



➤ **read_serial ();** **write_serial ();**
➤ **read_AD ();** **write_AD();**
➤ **read_display ();** **write_display ();**

Multiple implementations of a device type



- **read_serial_8250 () write_serial_8250 ()**
- **read_serial_68328 () write_serial_68328 ()**
- **read_serial_atmel () write_serial_atmel ()**



- `int read (device_t *dev, void *buffer, int length, int *actual);`
- `int write (device_t *dev, void *buffer, int length, int *actual);`

Fill in the dev structure

`dev->read = read_serial_8250;`

`dev->write = write_serial_8250;`

Connecting to a device



```
device_t dev;

int device_init ()
{
    /* fill in the dev struct */
    .
    .
    device_register ("serial", &dev);

    return 0;           // success
}
```



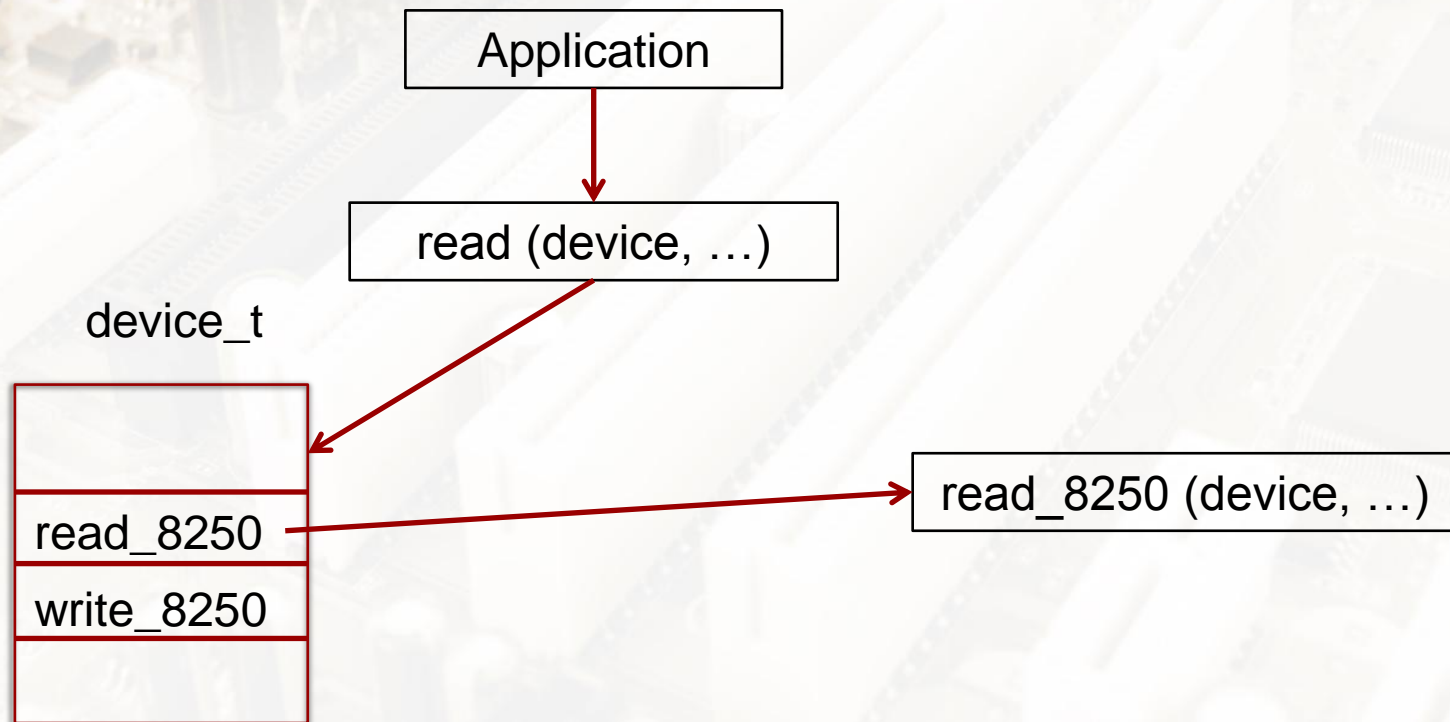
```
device_t serial_dev;  
char command[80];
```

```
·  
·  
·
```

```
status = connect ("serial", &serial_dev);
```

```
·  
·
```

```
status = read (&serial_dev, command, sizeof  
              (command), &actual);
```





```
int control (device_t *dev, int command, ...);
```

```
if (control (&serial_dev, SET_BAUD_RATE, 115200))  
{  
    /* Report error */  
}
```



- `int connect (char *name, device_t *dev);`
- `int read (device_t *dev, void *buffer, int len, int *transferred);`
- `int write (device_t *dev, void *buffer, int len, int *transferred);`
- `int control (device_t *dev, int command, ...);`
- `int disconnect (device_t *dev);`



- **What if...**
 - No data available (yet) on read?
 - Buffer full on write?
- **Task must “sleep” until condition is satisfied**
- **Allows another task to run – multitasking**
 - **Reentrancy!**

Blocking methods



- `int get_sem (semaphore_t *sem);`
- `int put_sem (semaphore_t *sem);`
 - often called from ISR

```
int read (device_t *dev, void *buffer, int len, int *actual)
{
    prepare to read;

    if (!readable)
        get_sem (&wait);
    do read;
    *actual = bytes read;
    return success;
}
```

Learn today. Design tomorrow.



Boston

Interrupt Service Routine



```
void interrupt_service (void)
{
    determine interrupt source;
    clear interrupt flag;
    put_sem (&wait);
}
```



- **A device driver is:**
 - Abstraction mechanism
 - Set of functions called by application

- **Device-independent functions call device-dependent functions**

- **Usually an independently loaded program file**



- **Everything is a file**
 - Peripherals have nodes in file system
 - Device nodes go in /dev directory

- **Well, not quite**
 - Network devices don't have file system nodes

- **Device nodes have *major* and *minor* numbers**
 - Major number links to device driver



➤ **Character**

- Stream of bytes

➤ **Block**

- Mass storage
- Requires file system

➤ **Network**

- Different interface to kernel
- Responds to packets coming from outside

➤ **Pipe**

The /dev directory



# ls -l /dev	Major Number	Minor Number	
brw-rw----	1 root disk 3,	0 May 5 1998 hda	
brw-rw----	1 root disk 3,	1 May 5 1998 hda1	
brw-rw----	1 root disk 3,	10 May 5 1998 hda10	
lrwxrwxrwx psaux	1 root root	5 Feb 23 10:00 mouse ->	
.			
crw-rw-r--	1 root root 10,	1 Feb 23 17:14 psaux	
crw-----	1 root root 4,	0 May 5 1998 tty0	
crw-----	1 root tty 4,	1 Feb 23 20:06 tty1	
crw-rw----	1 root uucp 19,	0 Apr 17 1999 ttyC0	

Links

Owner

Group



- `int open (const char *path, int oflags);`
- `size_t read (int file_des, void *data, size_t len);`
- `size_t write (int file_des, const void *data, size_t len);`
- `int close (int file_des);`
- `int ioctl (int file_des, int cmd, ...);`

Problems

- Data not buffered
- Kernel calls are inefficient

Standard I/O library, stdio



- `FILE *fopen (const char *filename, const char *mode);`
- `size_t fread (void *buff, size_t size, size_t n, FILE *stream);`
- `size_t fwrite (const void *buff, size_t size, size_t n, FILE *stream);`
- `int fclose (FILE *stream);`
- `getc`, `putc`, and `gets` families
- Formatted I/O: `printf`, etc.

Problems

- Non-deterministic
- No control function (`ioctl`)

Installable kernel modules are...



- **A way to “extend” the kernel**
- **Dynamically loaded and unloaded**
- **Executed at Privilege Level 0**
- **Useful for:**
 - Device Drivers
 - File systems



➤ **Shell Commands**

- `insmod <module name> [<param>...]`
- `rmmod <module name>`

➤ **Required Functions**

- module initialization
- module cleanup



Install command

```
insmod my_module.ko my_int = 1 my_string = "Hello"
```

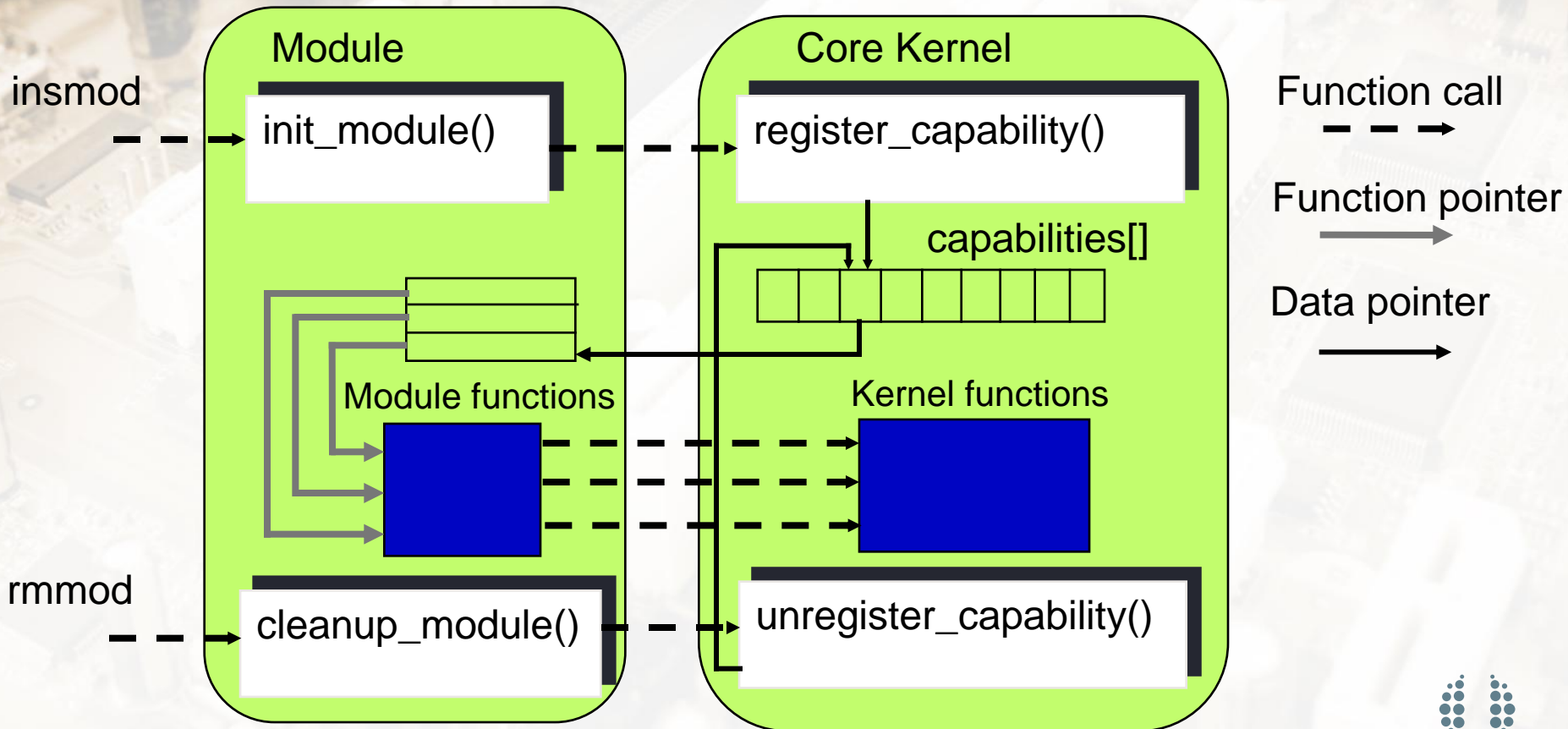
In my_module.c

```
int my_int;  
char *my_string;
```

```
module_param (my_string, charp, S_IRUGO);  
module_param (my_int, int, S_IRUGO);
```

```
static int __init my_init (void)  
{  
    register_capability();  
    return 0;  
}  
module_init (my_init);
```

Linking a module to the kernel





➤ **Modules are kernel version specific**

➤ Kernel APIs change over time

➤ **Build in kernel source tree**

➤ Need kernel header files

➤ **Makefile**

```
obj-m := module.o
```

```
all:
```

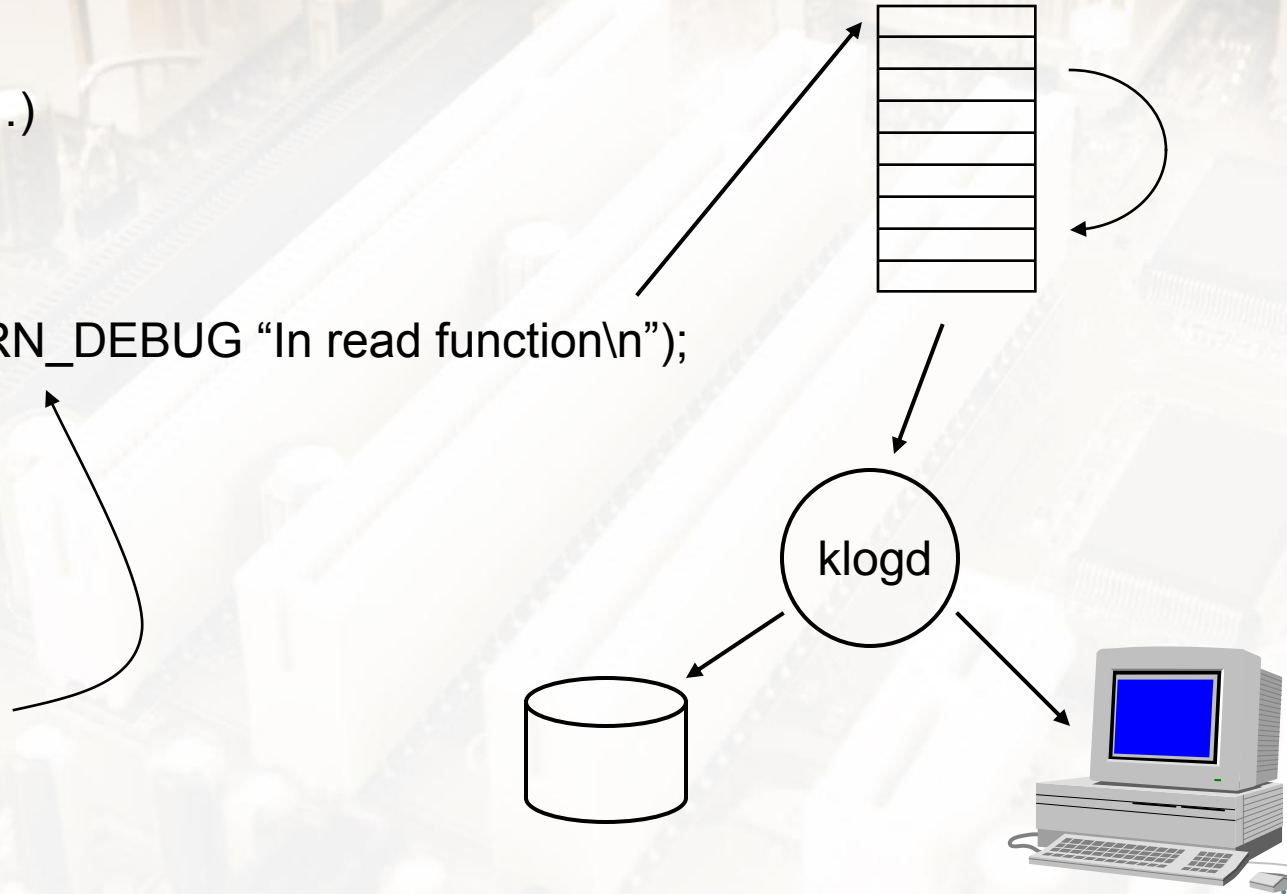
```
make -C /lib/module/$(shell uname -r)/build  
M=$(shell pwd) modules
```

printk()



```
driver_read(...)  
{  
    .  
    .  
    printk (KERN_DEBUG "In read function\n");  
    .  
    .  
}
```

loglevel



Learn today. Design tomorrow.



Boston

Module example





- **GPL = Gnu General Public License**
- **Applications in User Space need not be GPL**
- **Kernel is GPL**
 - Any code built into the kernel is GPL
- **Kernel modules need not be GPL**
 - Some developers disagree



```
struct file_operations {
    struct module *owner;
    loff_t (*llseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t *);
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);
    int (*readdir)(struct file *, void *, filedir_t);
    unsigned int (*poll)(struct file *, struct poll_table_struct *);
    int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap)(struct file *, struct vm_area_struct *);
    int (*open)(struct inode *, struct file *);
    int (*flush)(struct file *);
    int (*release)(struct inode *, struct file *);
```

File operations table II



```
int (*fsync)(struct file *, struct dentry *, int);
int (*aio_fsync)(struct kiocb *, int);
int (*fasync)(int, struct file *, int);
int (*lock)(struct file *, int, struct file_lock *);
ssize_t (*readv)(struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev)(struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
ssize_t (*sendpage)(struct file *, struct page *, int, size_t, loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                   unsigned long, unsigned long);
int (*check_flags)(int);
int (*dir_notify)(struct file *, unsigned long);
}
```



➤ **Allocate one or more device numbers**

➤ `dev_t` is 32-bit number

➤ `int register_chrdev_region (dev_t first, unsigned int count, char *name);` or...

➤ `int alloc_chrdev_region (dev_t *dev, unsigned int firstminor, unsigned int count, char *name,);`

➤ **Create nodes in /dev for device numbers**

➤ **Fill in file operations table**

➤ **Register driver**

Allocating device numbers



- ➔ **dev_t** - a 32-bit number in kernel 2.6. Combines major and minor numbers
- ➔ **MKDEV(int major, int minor)** - creates a **dev_t**
- ➔ **MAJOR(dev_t dev)** - Returns the major part
- ➔ **MINOR(dev_t dev)** - Returns the minor part



In `init_module()`

```
#include <linux/cdev.h>
```

```
struct cdev *my_cdev = cdev_alloc ( );  
my_cdev->ops = &my_fops;
```

or

```
void cdev_init (struct cdev *my_cdev, struct file_operations *my_fops);
```

followed by

```
my_cdev.owner = THIS_MODULE;  
int cdev_add (struct cdev *my_cdev, dev_t num, unsigned int count);
```



➤ **Creates device nodes for you**

- Uses major device number 10
- Can dynamically allocate minors

```
struct miscdevice {  
    int minor;  
    const char *name;  
    const struct file_operations *fops;  
    .  
    .  
}  
  
int misc_register (struct miscdevice *);
```



- `int open (struct inode *inodep, struct file *filep);`
 - Returns 0 on success.
- `int release (struct inode *inodep, struct file *filep);`
 - Equivalent to `close ()`. Returns 0 on success.
- `ssize_t read (struct file *filep, char __user *buff, size_t count, loff_t *f_pos);`
 - Positive return = bytes transferred. Negative is error
- `ssize_t write (struct file *filep, char __user *buff, size_t count, loff_t *f_pos);`



➤ **file structure**

- Created before call to `open ()`
- Passed to all driver functions
- May have multiple file structures simultaneously
- Represented in user space by file descriptor

➤ **inode structure**

- One for each file and device entry in the file system
- Describes the file system entry
- Passed to `open()`, `release()`, and `ioctl()`



```
struct file {  
    struct list_head  f_list;  
    struct dentry     *f_dentry;  
    struct vsfmount   *f_vsfmount;  
    struct file_operations *f_op;  
    .  
    .  
    void *private_data;  
    .  
}
```

Learn today. Design tomorrow.



Boston

Driver example





- **Files in /proc don't really exist**
 - Data generated in real time when file is read
- **Window from user space into kernel space**
 - Many utilities get their info from /proc files
 - Numbered subdirectories represent processes
- **Create /proc files as driver debugging tools**



- **System resources include:**
 - I/O ports
 - Device numbers
 - Interrupts

- **Must *request* or *register* resources**
 - Resources are allocated exclusively
 - Registration function includes *name* argument that becomes an entry in a /proc file



```
#include <linux/interrupt.h>
```

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

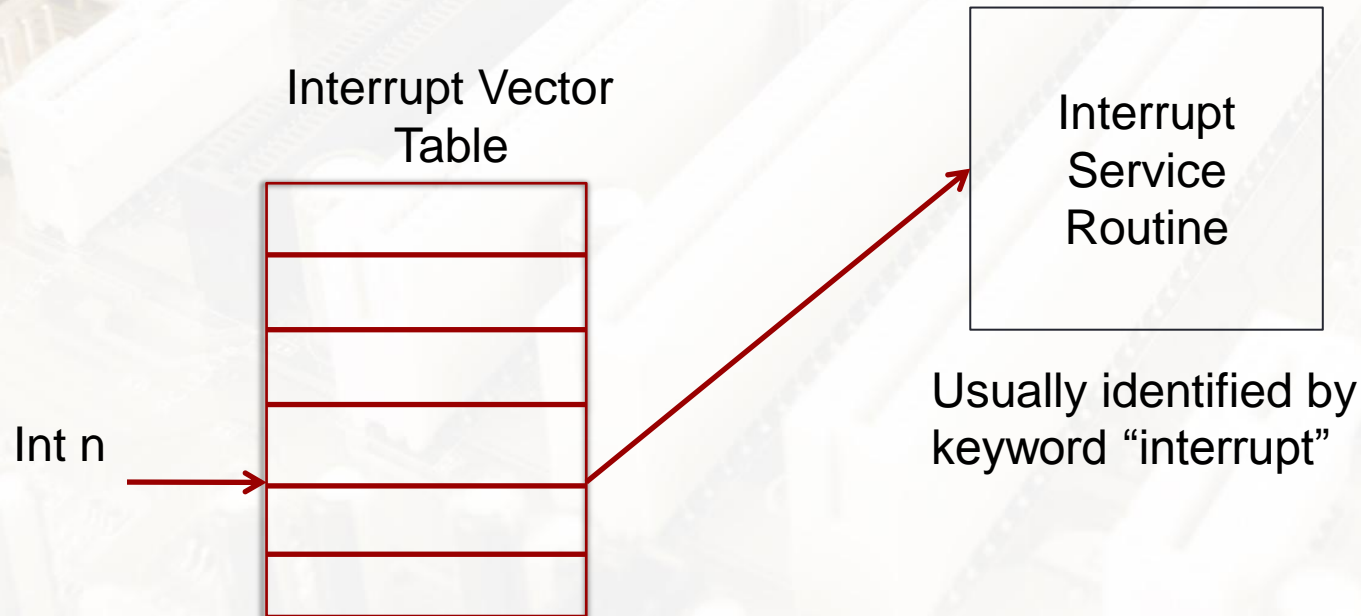
```
void free_irq (unsigned int irq, void *dev_id);
```



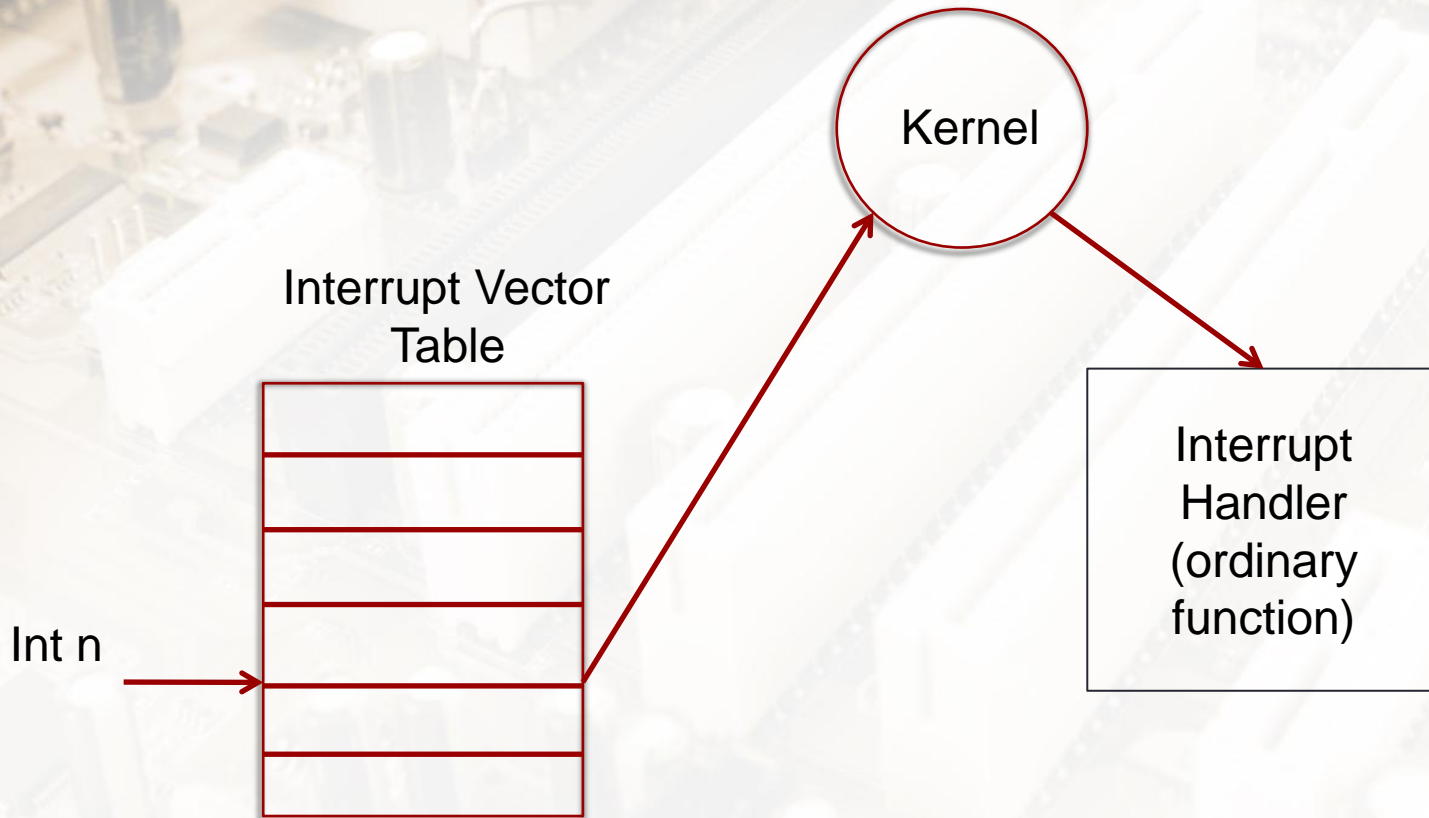
- ➔ **IRQF_DISABLED – run ISR with interrupts disabled**
- ➔ **IRQF_SHARED – interrupt is shared. Multiple registrations OK.**
- ➔ **IRQF_SAMPLE_RANDOM – interrupt can contribute to “entropy pool.”**

Two approaches to Interrupt handling

The “direct” approach



“Indirect” approach Linux does this





```
#include <linux/wait.h>
```

Create a wait queue

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

or

```
wait_queue_head_t my_queue;  
init_waitqueue_head (&my_queue);
```

Sleep on a queue

```
wait_event (queue, condition)  
wait_event_interruptible (queue, condition)  
wait_event_timeout (queue, condition, timeout)  
wait_event_interruptible_timeout (queue, condition, timeout)
```

Wake up a process on a queue

```
wake_up (&queue);  
wake_up_interruptible (&queue);
```

Putting a process to “sleep”



```
DECLARE_WAIT_QUEUE_HEAD(my_wait);
int flag = 0;
int buff_count;

ssize_t sleepy_write (struct file *file, char *buff, int count, loff_t *offset)
{
    copy_from_user (buffer, buff, buff_count = count);
    flag = 1;
    wake_up_interruptible (&my_wait);
    return count;
}

ssize_t sleepy_read (struct file *file, char *buff, int count, loff_t *offset)
{
    wait_event_interruptible (my_wait, flag != 0);
    copy_to_user (buffer, buff, buff_count);
    return buff_count;
}
```



```
ssize_t sleepy_read (struct file *file, char *buff, int count, loff_t
    *offset)
{
    while (flag == 0)
        wait_event_interruptible (my_wait, flag != 0);

    copy_to_user (buffer, buff, buff_count);
    return buff_count;
}
```



- **Where are I/O registers located?**
 - Memory space or separate I/O space
- **I/O operations may have “side effects”**
 - Compiler optimizations can get in the way
 - Caching
 - Reordering
- **I/O space must be allocated**



Single item transfer

unsigned inb (unsigned port);
void outb (unsigned char byte, unsigned port);
unsigned inw (unsigned port);
void outw (unsigned short word, unsigned port);
unsigned inl (unsigned port);
void outl (unsigned longword, unsigned port);

“String” transfer

void ins x (unsigned port, void *addr, unsigned long count);
void outs x (unsigned port, void *addr, unsigned long count);

Note: x is “b”, “w”, or “l”.



```
void *ioremap (unsigned long phys_addr, unsigned long size);  
void iounmap (void *address);
```

```
unsigned int ioreadn (void *address);  
void iowriten (un value, void * address);
```

```
unsigned int ioreadn_rep (void *address, void *buf,  
    unsigned long count);  
void iowriten_rep (void * address , const void *buf,  
    unsigned long count);
```

Note: **n** = 8, 16, or 32



```
struct resource *request_region (unsigned long first,  
                                unsigned long n, const char *name);  
void release_region (unsigned long start, unsigned long n);  
  
struct resource *request_mem_region (unsigned long first,  
                                     unsigned long n, const char *name);  
void release_mem_region (unsigned long start,  
                        unsigned long n);
```



Compiler memory barrier

```
#include <linux/kernel.h>  
void barrier (void);
```

Hardware barriers

```
#include <asm/system.h>  
void rmb (void);  
void wmb (void);  
void mb (void);
```

Doing it in user space



- `fd = open (“/dev/mem”, O_RDWR);`
- `vaddr = mmap (0, size, access, shared, fd, phys_addr);`



- **Everything is a file – almost**
 - Peripherals are “files” in the /dev directory
 - Except network devices
 - Four device classes

- **Kernel loadable modules**
 - Extend the kernel
 - Dynamically loadable at run time
 - Kernel version specific

- **File Operations Table**
 - Links driver to rest of system



➤ **Tools**

- `printk()` – writes to `/var/log/messages`
- `/proc` file system – window from user space into kernel space

➤ **Resources**

- Device numbers, I/O ports, interrupts
- Must be allocated or registered exclusively

➤ **Accessing hardware**

- I/O port space vs. memory space
- “Side effects” – compiler optimizations
- Interrupts

Key takeaway



- Fundamentally, a device driver is just a set of functions that are called on behalf of, and in the context of, a user space process.
- Complexities arise due to:
 - Driver is independently loaded program
 - User space to kernel space transition
- Questions, revised slides/sources
 - www.intellimetrix.us/downloads
 - doug@intellimetrix.us