

Embedded Systems Introduction



VIVEN
Embedded Academy

Definition

- **Embedded System:** Any device that includes a computer but is not itself a general-purpose computer.
- Hardware and software – part of some larger systems and expected to function without human intervention.
- Respond, monitor, control external environment using sensors and actuators.

Examples

- ♦ Personal digital assistant(PDA).
- ♦ Printer
- ♦ Cell Phone
- ♦ Auto mobiles (Micro controllers & Microprocessors)
- ♦ Television
- ♦ Household appliances.
- ♦ Digital Cameras.
- ♦ Surveillance Systems.

Examples

Product: Palm Vx Hand-held

- Microprocessor:
32-bit Motorola Dragonball EZ.

Examples

- Product: Motorola i1000plus iDEN Multi Service Digital Phone.
- Microprocessor:
Motorola 32-bit MCORE.

Application Examples

- ♦ Simple Control: Front panel of microwave oven, etc.
- ♦ Camera: Canon EOS 3 has 3 Microprocessor.
 - ♦ 32-bit RISC CPU runs auto-focus.
- ♦ Analog TV: channel selection,etc.
- ♦ Digital TV: Decompression, Descrambling etc.

Automotive Embedded Systems

- Today's high-end auto-mobile may have 100's of microprocessors:
 - 4-bit micro-controllers to check seat belt.
 - Microprocessors run dashboard devices.
 - 16/32-bit microprocessor control engine.
 - Automated breaking systems.

Characteristics of Embedded System

- Sophisticated functionality.
- Real-time operation (alway?).
- Low manufacturing cost.
- Application dependent processor
- Restricted Memory.
- Low power.
 - Power consumption is critical in battery-powered devices.
 - Excessive power consumption increases system cost even in wall-powered devices.

Manufacturing Cost

- Manufacturing cost has different components.
 - Non-recurring engineering cost for design and development.
 - Cost of production and marketing each unit.
- Best technology choice will depend on the number of units we plan to produce.

Real Time operation

- Must finish operations by deadlines.
 - Hard Real time: Missing deadlines causes failure.
 - Ex: Atomic reactor control. (Catastrophic)
 - Soft real time; Missing deadline results in degraded performance.
 - Ex: Playing video on mobile. We cannot decode a frame in time. Only disturb viewing experince.
- Many systems are **multi-rate**: must handle operations at widely varying rates.
 - Embedded devices receiving inputs from external worlds and inputs come at different rates to be handled.

Application depended requirements

- ♦ Fault-tolerance
 - ♦ Continue operation despite hardware or software faults.
 - ♦ Aircraft systems
 - ♦ Medical Systems
- ♦ Safe
 - ♦ Systems to avoid physical or economic damage to person or property.

More features.

- Dedicated Systems – Design considerations are different.
 - Predefined Functionality – accordingly hardware and software designed.
 - Programmability rarely used during lifetime of the system.
 - Real-time, fault-tolerant, safe.
 - Ex: Vending machines.(8-bit motorola microcontroller).
 - Web-enabled cashless vending machine.(through credit card, smart cards). Stock & Security can be monitored remotely.

More features.

- ♦ Product: NASA's Mars Sojourned Rover.
 - ♦ Microprocessor: 8-bit intel 80C85.
- ♦ Product: GPS receiver.
 - ♦ Microprocessor: 16-bit.(Receive input from satellites, send output and display).
- ♦ Product: MP3 Player.
 - ♦ Microprocessor: 32-bit RISC Microprocessor.
- ♦ Product: DVD Player
 - ♦ Microprocessor: 32-bit RISC Microprocessor.
- ♦ Product:L Sony Aibo ERS-110 Robotic Dog.
 - ♦ Microprocessor: 64-bit MIPS RISC.



Types of Embedded Systems.

- ♦ Similar to General Computing.
 - ♦ PDA, Video Games, Set-top boxes, automated teller machines.(Responds to user input).
- ♦ Control Systems
 - ♦ Feed-back control of real-time systems.
 - ♦ Vehicle engines, flight control, nuclear reactors.
- ♦ Signal Processing
 - ♦ Radar, Sonar, DVD players.
- ♦ Communication and Networking
 - ♦ Cellular Phones. Internet applicances.

Nature of system functions.

- ♦ Control laws.(Sensing).
- ♦ Sequencing logic.(task specific).
- ♦ Signal processing.(external sensing inputs).
- ♦ Application specific interfacing.
- ♦ Fault response (Graceful degradation. Catastrophic failure cannot happen. Battery low warning).

Implementing Embedded Systems.

-
- ♦ Hardware.
 - ♦ Processing element.
 - ♦ Peripherals
 - ♦ Input and output Devices.
 - ♦ Interfacing sensors and actuators.
 - ♦ Interfacing protocols
 - ♦ Memory
 - ♦ Bus
 - ♦ Software
 - ♦ System software.
 - ♦ Application.

FUNDAMENTALS

What is a Device

Any peripheral such as a graphics display , disk driver, terminal, or printer is a device.

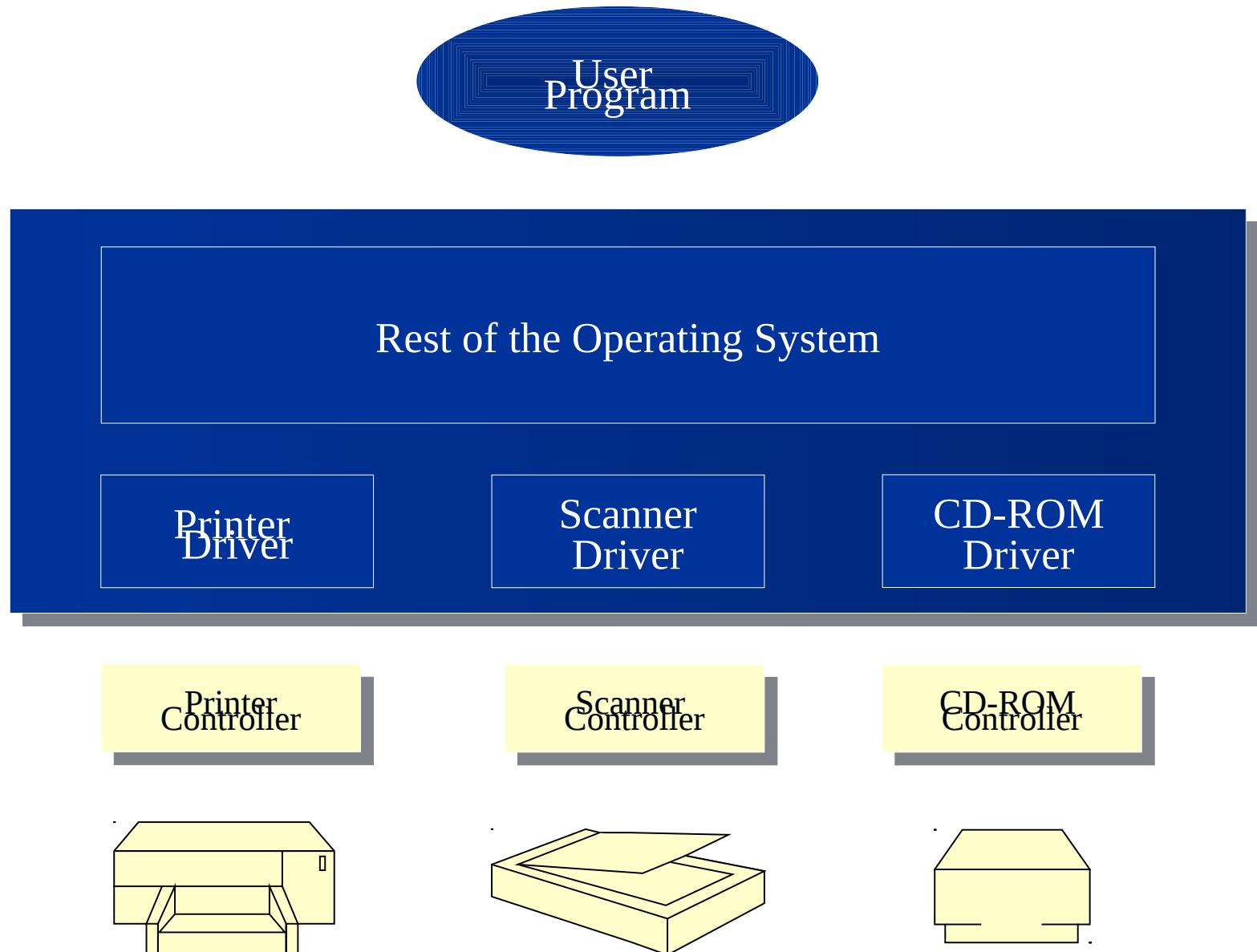
A device is usually considered to be piece of hardware that you can connect to your computer system and that you wish to manipulate by sending command and data

A device can be a software device such as random number device, virtual consoles, loop back devices etc.,

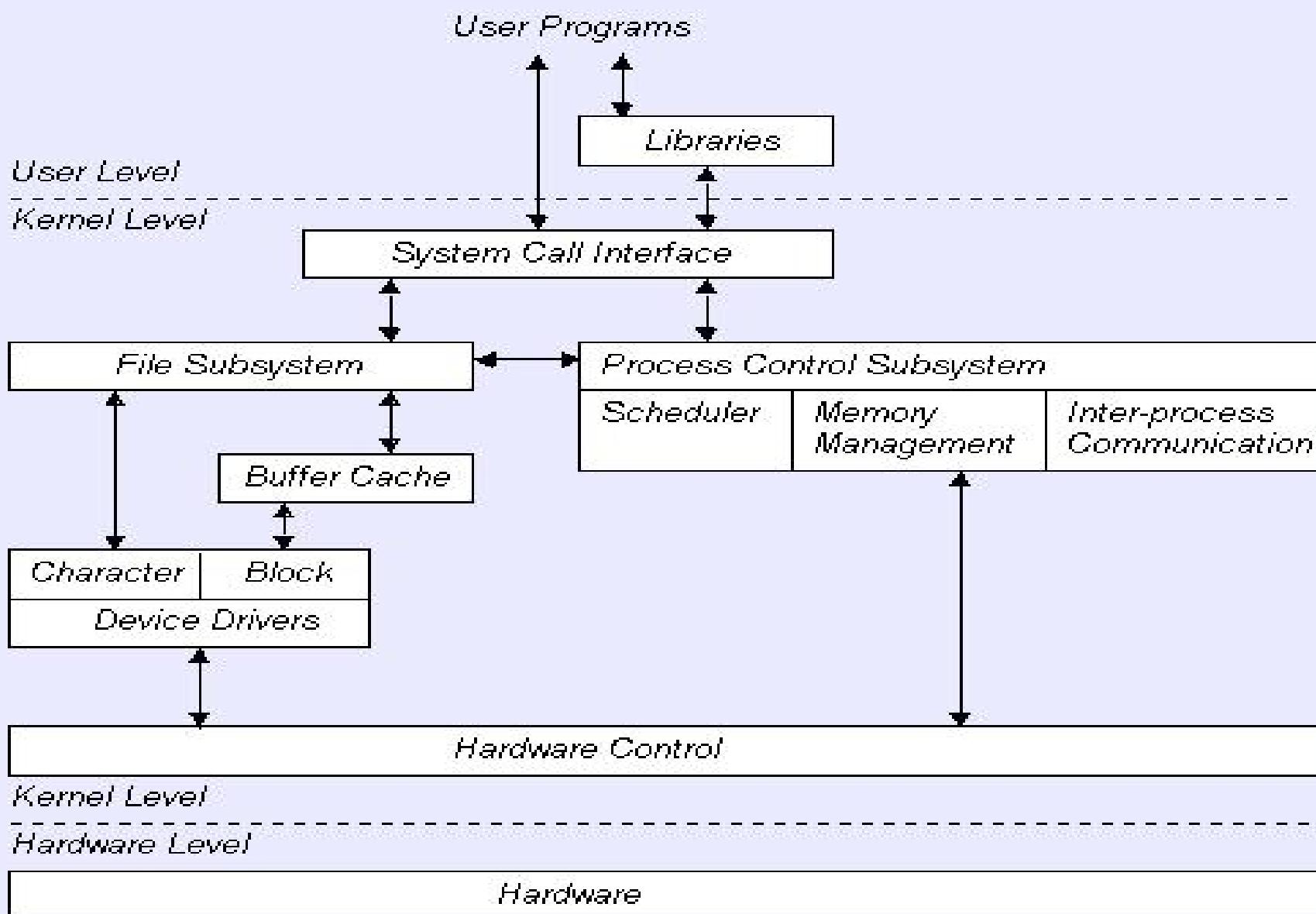
What is a Device Driver

- What's Device Driver?
 - “Black Boxes” respond to well defined internal API.
 - Hide completely the details of device works.
- The role
 - Mapping API calls to device specific hardware operations.
- Mechanism vs policy
 - What capabilities are to be provided(Mechanism)
 - How those capabilities are to be used(Policy)
 - Policy free drivers
 - Eg X Server vs Window Manager(KDE/GNOME).

What is a Device Driver



Linux I/O Subsystem



Classes of devices and module

-
- Character devices
 - Can access as stream of bytes
 - `/dev/console`, `/dev/ttyS0`
 - Accessed by means of filesystem nodes `tty1`, `lp0`
 - Block devices
 - Something that can host a filesystem (e.g. disk)
 - Can access ONLY as multiples of a block (e.g. 1K)
 - Different with char device is transparent to user under Linux.
 - Network interfaces (`eth0`, `eth1`) – stream of packets
 - USB, Firewire, SCSI, I20

Character devices

Accessed as a stream of bytes (like a file)

- Such drivers at least implement ***open, close, read and write*** system calls
- Examples:
 - /dev/console (Text Console)
 - /dev/ttyS0 (Serial Ports)
 - /dev/lp0 (Line Printers)
- Character devices are accessed by means of file system nodes., such as /dev/tty1 and /dev/lp0. The only difference between difference between a char device and a regular file is that you can always move back and forth in the reuglar file, whereas most char devices are just data channels, which you can only access sequentially.

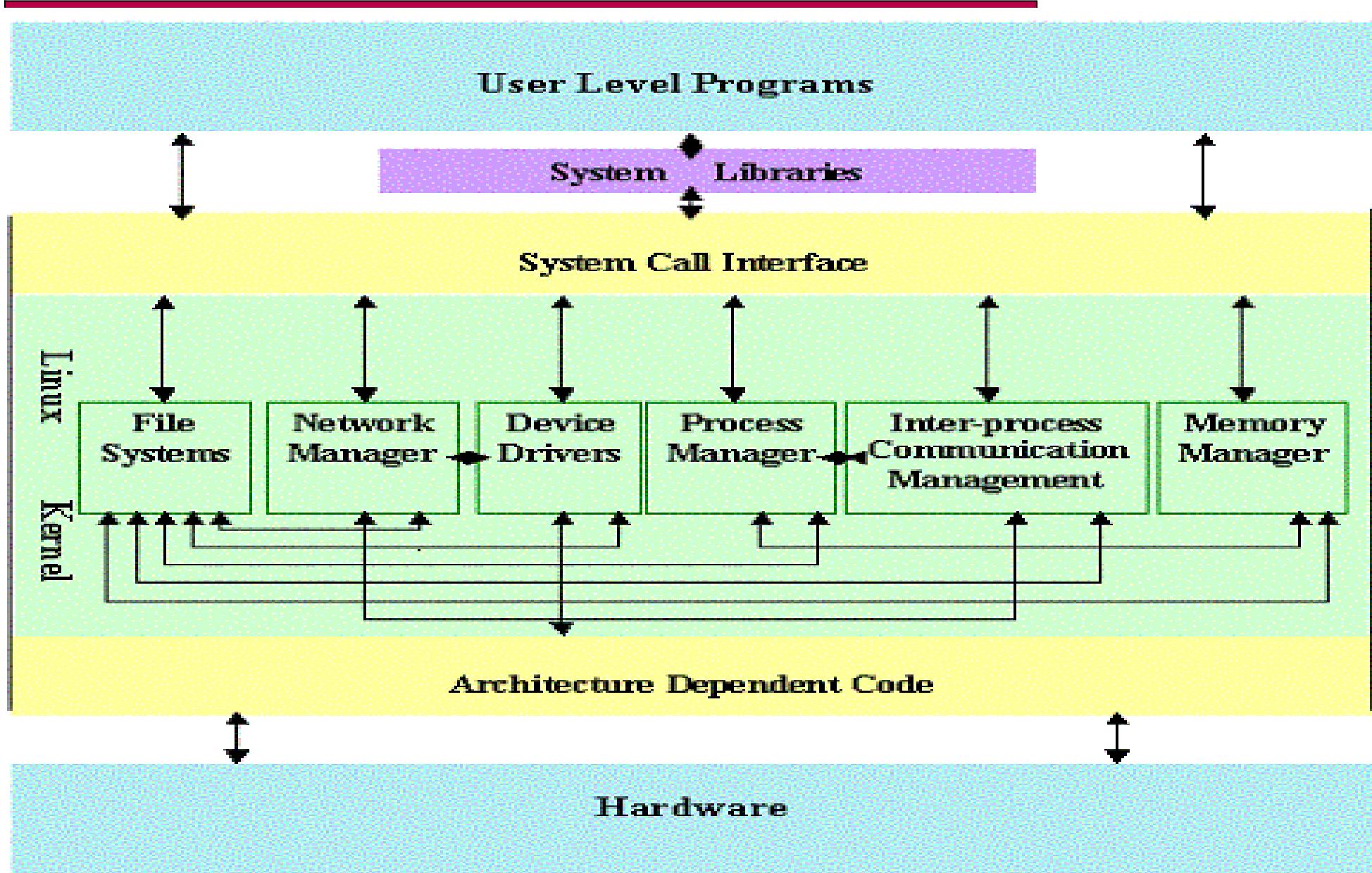
Block devices

- Block Devices also accessed by filesystem nodes in the /dev directory.
- Block Device can host a filesystem such as disk.
- Block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes(or a large power of 2) bytes in length.
- Linux allows the application to read and write a block device like a char device – it permits the transfer of any number of bytes at a time.
- Block and Char devices differ only in the way data is managed internally by the kernel,.

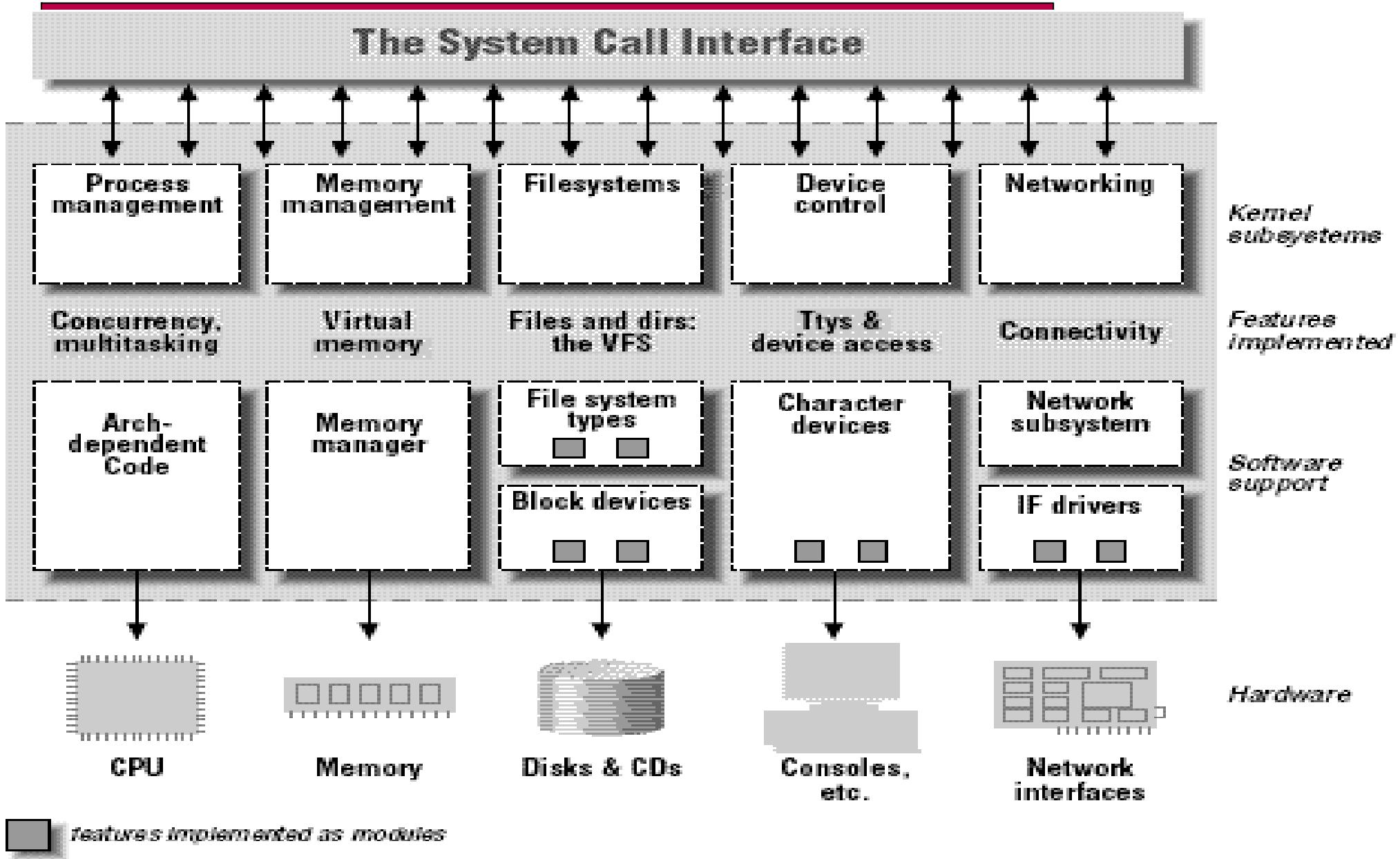
Network devices

- Any network interconnection is made through the interface(device), that is able to exchange data with other hosts.
- Transaction made using an interface.
- Interface can be hardware or software.
- Interface is incharge of sending and receiving data packets, driven by the network subsystem of the kernel.
- Communcation between kernel and network driver is completely different. Instead of read and write, the kernel calls functions related to packet transmission.
- Though UNIX will map it as eth0, it doesn't have any corresponding entry in the file system.
- A network driver knows nothing about individual connections. It only handles packets.

Linux Kernel Internals



Split view of kernel



Linux Device Drivers

- Distinct Black Boxes
- And they make a particular piece of H/W, respond to a well-defined internal programming interface
- They hide completely the details of how the device works
- A 'C' Program that controls a device.
- They stand between kernel and peripherals, translating requests for work into activities by hardware.

Device Drivers

- A device controller has:
 - Control registers
 - Status registers
 - Data buffers
- This structure changes from device to device
 - A mouse driver should know how far the mouse has moved and which buttons are pressed.
 - A disk driver should know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, ...
- Each I/O device need some device specific code for controlling it.
 - This code is called device driver.

I/O Software Layers



Applications Vs Device Driver

-
- Applications performs a single task from beginning to end, A module registers itself in order to serve future requests.
 - Application call functions resolves external references using appropriate lib functions during link stage, whereas modules are linked only to kernel, and only functions it can call are exported by kernel.

Locating Driver Entry Points – Switch Table

- The kernel maintains a set of data structures known as :
 - -block device switch table
 - -character device switch table
- These data structures are used to locate and invoke the entry points of a device driver.
- Each switch table is an array of structures. Each structure contains a number of function pointers.
- To decide which element of the switch table should be used, the system uses the “major device number” associated with the device special file that represents the device.

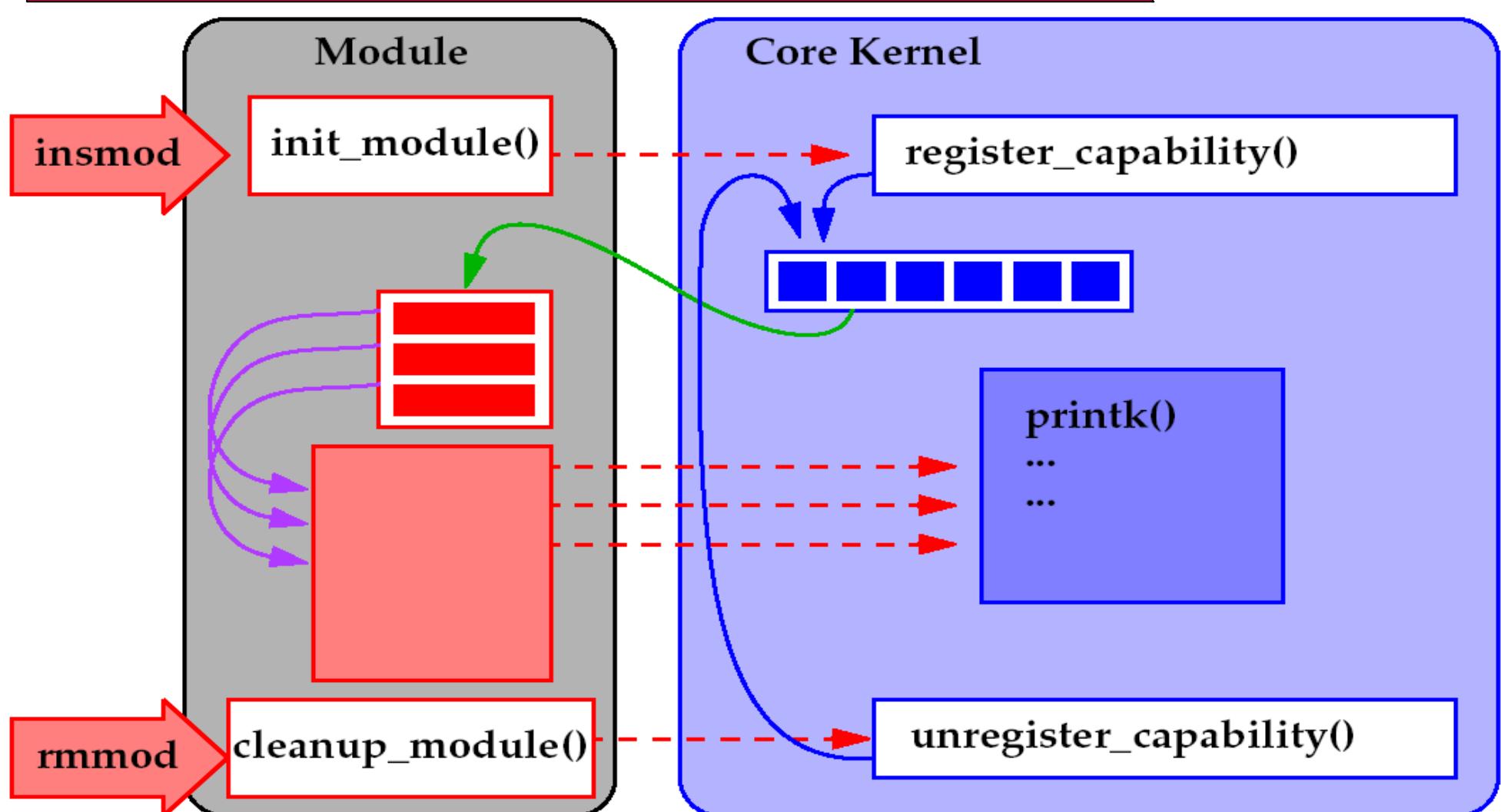
Locating Driver Entry Points – Switch Table

- Having decided which element of the switch table to use, the system decides which entry-point of the element should be used.
- This depends upon the system call used by the process. If the process used “open” system call, then the “open” entry point is used and so on.....

Hello World Module

- This module defines two functions, one to be invoked when the module is loaded into the kernel(hello_init). And one for when the module is removed (hello_exit).
- **MODULE_LICENSE** is used to tell the kernel that this module bears a free license. Without such a declaration the kernel complains when the module is loaded.
- The **printk** function is defined in the linux kernel and made available to modules, It behaves similar to the standard C library function printf. The string KERN_ALERT is the priority of the message.
- The kernel runs without the help of the C library.
- /linux/module.h contains many definitions of symbols and functions needed by loadable modules.
- /linux/init.h to specify your initialization and cleanup functions.

Linking a module to the kernel



 Functions
 Data

 Function call
 Function pointer

 Data pointer
 Assignment to data

Kernel Module Vs Applications

- Applications perform a single task from beginning to end whereas kernel modules register itself in order to serve future requests and its initialization function terminates immediately.(Event driven programming).
- The exit function of a module must carefully undo everything the init function built up, or the pieces remain in the system till it is booted.
- Ability to unload a module is a one of the good feature. Otherwise during development you need to go through the lengthy shut-down/reboot cycle each time.
- Applications can call functions it doesn't define; the linking stage resolves external references using the appropriate library of functions. A module is linked only to the kernel, and the only functions it can call are the ones exported by the kernel. There are no libraries to link.

Compiling Modules

- The build process for modules differs significantly from that used for user-space applications.
- The first thing is to ensure you have sufficiently current versions of the compiler, module utilities, and other necessary tools.
- Trying to build a kernel(and its modules) with the wrong tool versions can lead to difficult problems.
- You cannot build a loadable modules for a 2.6 kernel without this source tree on your file system
- Statements in makefile
 - Obj-m := hello.o (States that there is one module to be built from the object file hello.o. The resulting module is named hello.ko after being built from the object file.)
 - Obj-m := module.o
 - module-objs := file1.o file2.o (for two files)

- Make command required to build your module is
 - \$(MAKE) -C <kernel source directory path> M='pwd' modules
- This command starts by changing its directory to the one provided with the -C option which is kernel source directory. There it finds the kernel top level make file.
- The \$(MAKE) – invokes the kernel build system and the kernel makefiles take care of actually building the modules.
- The M= option causes that makefile to move back into your module source directory before trying to build the modules target. The modules are build in the path mention by the M= option.

- **Kbuild** includes functionality for building modules both within the kernel source tree and outside the kernel source tree. The latter is usually referred to as external or “out-of-tree” modules and is used both during development and for modules that are not planned to be included in the kernel tree.
- The author of the external module should supply a makefile that hides most of the complexity, so one only has to type 'make' to build the module.
- Kbuild offers functionality to build external modules, with the prerequisite that there is a pre-built kernel available with full source. A subset of the targets available when building the kernel is available when building an external module.

Loading and Unloading Modules

- **insmod** loads modules into the kernel.
 - `insmod hello.ko`
- It links any unresolved symbols in the module to the symbol table of the kernel.
- Insmod accepts a number of command line options and it can assign values to parameters in your module before linking it to kernel. If a module is correctly designed, it can be configured at load time. (Explained in Module Parameters)
- Insmod relies on a system call **sys_init_module** defined in `kernel/module.c`. This function allocates kernel memory to hold a module, it then copies the module text in that memory region, resolves the kernel references in the module via the kernel symbol table, and calls the module's initialization function.

Loading and Unloading Modules

- **Modprobe** like insmod loads a module into the kernel, It will look at eh module to be loaded to see whether it references any symbols that are not currently defined in the kernel.
- If any such references are found, modprobe looks for other modules in the current module search path that define the relevant symbols. When modprobe finds those modules it loads them into kernel as well.
- If you use insmod in this situation the command fails with 'unresolved symbol' message symbol message left in the system logfile.

Loading and Unloading Modules

- **rmmod** utility is used to remove the module.
 - rmmod hello.ko
- Rmmod removal fails if the kernel believes that the module is still in use or the kernel has been configured to disallow module removal.
- The **lsmod** module lists the modules currently loaded in the kernel. Lsmod works by reading the /proc/modules virtual file. Information on currently loaded module can also be found in the sysfs virutal filesystem under /sys/module.
- A look into the system log file (/var/log/messages) will reveal the specific problem that caused the module to fail to load.

- Module initialization registers any functionality offered by the module.
- The functionality can be a new driver or a software abstraction that can be accessed by an application.
- Initialization function should be declared static, since they are not meant to be visible outside the specific file. ldd
- Modules can register many different types of functionalities, including different kinds of devices, file systems and etc. For each functionality there is a specific kernel function that accomplishes the registration.
- The arguments passed to the kernel registration function are usually pointers to data structures which usually contains pointers to module functions, which is how the functions in the module body get called

Major and Minor Numbers

- Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree. They are located in the /dev directory.
- Special files for char drivers are identified by a “c” in the first column of the output of ls -l.

```
crw-rw---- 1 root root 4, 64 2011-01-10 12:15 ttyS0
```

```
crw-rw---- 1 root tty 4, 65 2011-01-10 12:15 ttyS1
```

```
crw-rw---- 1 root root 4, 66 2011-01-10 12:15 ttyS2
```

```
crw-rw---- 1 root root 4, 67 2011-01-10 12:15 ttyS3
```

- If you issue ls -l command, you'll see two numbers(separated by comma) in the device file entries before the data of the last modification. These numbers are the major and minor device numbers.

Major and Minor Numbers

- The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number;
- All device files with the same major number are controlled by the same driver. All the above major numbers are 4, because they're all controlled by the same driver. **The kernel uses the major number at open time to dispatch execution to the appropriate driver.**
- The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.
- When the system was installed, all of those device files were created by the mknod command. To create a new char device named `coffee' with major/minor number 12 and 2, simply do **mknod /dev/skull c 12 2**. You don't have to put your device into /dev directory.

NOR AND NAND

Nor and Nand

- **NOR** based flash has long erase and write times, but provides full address and data buses, allowing random access to any memory location. This makes it suitable replacement for older read-only memory(ROM) chips, which are used to store program code that rarely needs to be updated, such as a computers **BIOS** or the **firmware** of a set-top boxes. Its endurance is 10,000 to 1,000,000 erase cycles.
- **NAND** has reduced erase and write times, and requires less chip area per cell, thus allowing greater storage density and lower cost per bit than NOR Flash. It also has up-to ten times the endurance of NOR flash. However, the I/O interface of NAND flash does not provide random-access external address bus. Rather, data must be read block-wise basis, with typical block sizes of hundreds to thousands of bits. This made NAND flash unsuitable as a drop-in replacement for program ROM since most microprocessors and micro-controllers required byte-level random access.

Nor and Nand

- In this regard NAND flash is similar to any other secondary storage devices such as memory cards.
- **NOR Memories:** Reading from NOR Flash is similar to reading from random-access memory, provided the address and data bus are mapped correctly. Because of this, most microprocessors can use NOR flash memory as **execute in place**(XIP) memory, meaning that programs stored in NOR flash can be executed directly from the NOR flash without needing to be copied into RAM first. NOR flash may be programmed in a random-access manner similar to reading. Programming changes bits from a logical one to a zero. Bits that are already zero are left unchanged. Erasure must happen block at a time. And resets all the bits in the erased block back to one. Typical block sizes are 64,128 and 256KB. Some devices offer read-while write functionality so that code continues to execute while a program or erase operation is occurring in background.

Nor and Nand

- **NAND Memories:** These memories are accessed much like block devices such as hard disks or memory cards. Each block consists of a number of pages. The pages are typically 512, or 2,048 or 4,096 bytes in size. Associated with each page are a few bytes that can be used for storage of an error correcting code (ECC) checksum.
 - 32 pages of 512+16 bytes each for a block size of 16 KB
 - 64 pages of 2,048+64 bytes each for a block size of 128 KB
 - 64 pages of 4,096+128 bytes each for a block size of 256 KB
 - 128 pages of 4,096+128 bytes each for a block size of 512 KB.
- While reading and programming is performed on a page basis, erasure can only be performed on a block basis.
- When executing software from NAND memories, virtual memory strategies are often used: memory contents must first be paged or copied into memory mapped RAM and executed there(leading to a combination of NAND+RAM). A memory management unit(MMU) in the system is helpful, but this can also be accomplished using

Nor and Nand

With overlays. For this reason, some systems will use a combination of NOR and NAND memories, where a smaller NOR memory is used as software ROM and a larger NAND memory is partitioned with a file system for use as a non-volatile data storage area.

- Nand is best suited to systems requiring high capacity data storage. This type of flash architecture offers higher densities and larger capacities at lower cost with faster erase, sequential write, and sequential read speeds, sacrificing the random-access and execute in place of the NOR architecture.
- While reading and programming is performed on a page basis, erasure can only be performed on a block basis.
- When executing software from NAND memories, virtual memory strategies are often used: memory contents must first be paged or copied into memory mapped RAM and executed there(leading to a combination of NAND+RAM). A memory management unit(MMU) in the system is helpful.

RAM

RAM

- RAM is an **integrated circuit(IC)** made of millions of **transistors** and **capacitors**. A **transistor** and **capacitor** are paired to create a **memory cell**, which represents a **single bit** of data. i.e., internal memory consists of individual memory cell, which represents a single bit. So a capacitor holds the bit info which either contains an electrical charge (bit-value 1) or not(bit-value 0). The transistor acts like a switch that lets the control circuitry on the memory chip read the capacitor or change its state. A capacitor is like a small bucket that is able to store electrons To store a 1 n the memory cell, the bucket is filled with electrons. To store a 0, it is emptied. The problem with the capacitor's bucket is that it has a leak. In a matter of a few milliseconds a full bucket becomes empty. Therefore , for dynamic memory to work, either the CPU or memory controller has to come along and recharge all of the capacitors holding a 1 before they discharge. To do this, the memory controller reads the memory and then writes it right back.

Memory Cells & DRAM

- This refresh operation happens automatically thousands of times per second. This refresh is where the dynamic RAM gets its name. Dynamic RAM has to be dynamically refreshed all of the time or it forgets what it is holding. The downside of all of this refreshing is that it takes time and slows down the memory.
- **Memory Cells:** Memory cells are etched onto a silicon wafer in an array of columns (bit lines) and rows (word lines).
- DRAM works by sending a charge through the **appropriate column** (CAS) to activate the **transistor** at each bit in the column.
 - When writing, the row lines contains the state the capacitor should take on.
 - When reading, the sense amplifier determines the level of charge in the capacitor. If its more than 50 it reads as 0.
- Transistor is thus connected on two electrical lines, the **word line** and the **bit line**. During writing and reading process the CPU always activates the word line first, the transistor is permeable. RAM is

Memory Cells & DRAM

Transistor is thus connected on two electrical lines, the **word line** and the **bit line**. During writing and reading process the CPU always activates the word line first, the transistor is permeable. RAM is considered “random access” because you can access any **memory cell** directly if you know the row and column that intersect at that cell. While writing, the CPU transports information on the bit line.

Thereby the charge of the capacitor aligns with the potential of the bit line, which corresponds to the value 1 or 0. The capacitor de-allocates its charge on the bit line during the writing process whereby its potential rises or falls-depending on whether the capacitor is loaded or not. The CPU then interprets this as 1 or 0. And as the line is discharged during reading, a fresh writing process follows after each reading process, which recovers the cell contents(the write back).

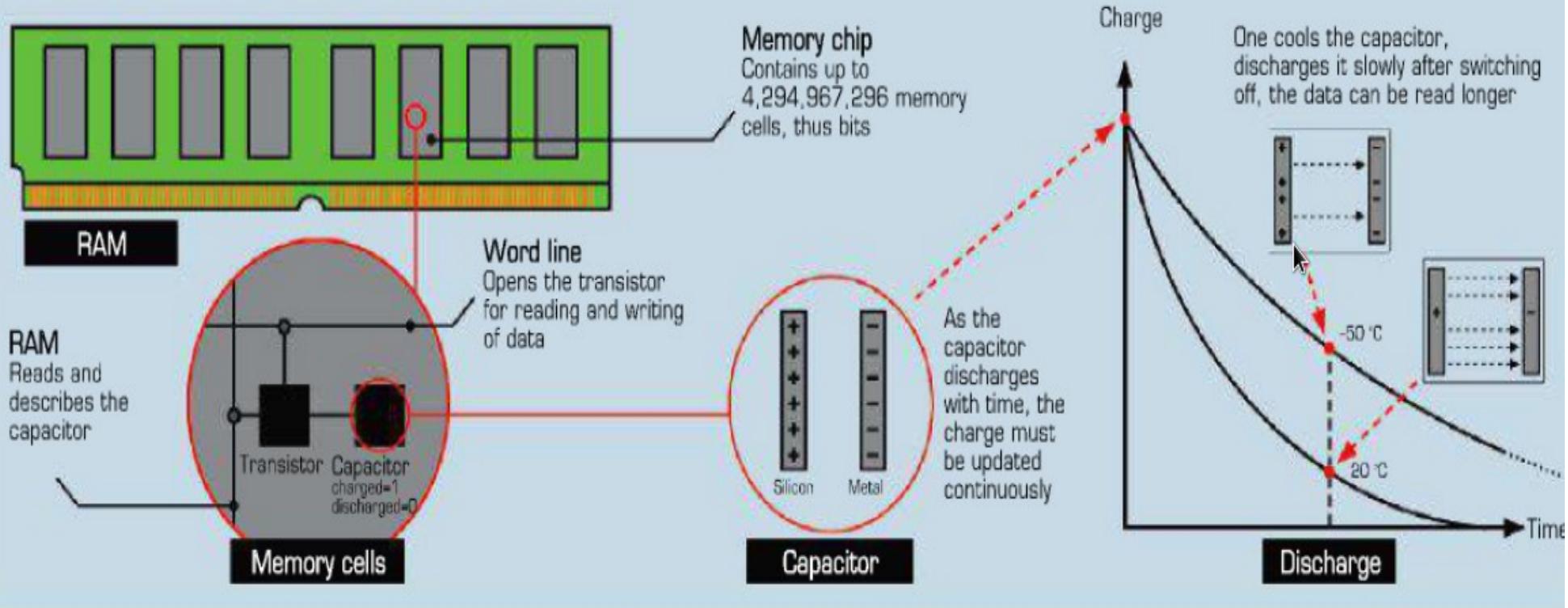
A loaded capacitor saves only around 100000 electrons in one of the

RAM

The capacitor disk at that point . This minor charge quantity can be discharged very fast by current leakage in the surrounding material of the chip, if one cuts the current supply. Thus the internal memory is updated in current memory chips even 15 microseconds-thus several thousand times per second – to prevent this. (refresh). The memory is not empty in milliseconds as some cells discharge slower depending on the design. Most of the bits survive at room temperature till two seconds But if one cools down the memory chips with a coolant spray to -50 C then charges remain longer in the capacitor. The resistance is increased in the semiconductor material of the memory chips at this temperature. That prevents the fast discharging of charge by leakage current. In our field trial the compute awoke again even from the standby mode after we had removed a cooled DDR RAM and had inserted again after few seconds.

Configuration and discharge of the internal memory

The RAM saves data (bits) as charges in capacitors. These survive, if one reloads the capacitor permanently. However if one cools the RAM, then the charge survives even without electricity.



Setting up Cross Compile Environment

Decompressing ARM Tool Chain

- A main step to build a development environment is to set up a cross compile environment. The following steps will introduce how to build a compile environment that can compile arm-linux kernels, drivers and applications.
- From updated **Linux-2.6.29** we use “**arm-linux-gcc-4.3.2**” as our cross compiler.
- **Step1:** Copy the compressed file “**arm-linux-gcc-4.3.2.tgz**” into a /tmp directory. Enter into the /tmp directory and execute the following command.

```
#cd \tmp  
#tar -xvzf arm-linux-gcc-4.3.2.tgz -C /
```

These commands will install “**arm-linux-gcc**” in the “**/usr/local/arm/4.3.2**”. **Note:Enter the commands as root.**

- You can see lot of previously used tools which are compatible to ARM in **/usr/local/arm/4.3.2/bin**.

Decompressing ARM Tool Chain

tar xvzf file-1.0.tar.gz - for uncompress a gzip tar file (.tgz or .tar.gz)

tar xvjf file-1.0.tar.bz2 - for uncompress a bzip2 tar file (.tbz or .tar.bz2)

tar xvf file-1.0.tar - for uncompressed tar file (.tar)

* x = eXtract, this indicated an extraction (c = create to create)

* v = verbose (optional) the files with relative locations will be displayed.

* z = gzip-ped; j = bzip2-zipped

* f = from/to file ... (what is next after the f is the archive file)

Decompressing ARM Tool Chain(Cont...)

- **Step2:** Run the command below to add the compiler's path to system variables:

#vi /root/.bashrc or gedit /root/.bashrc

- This is to edit the “/root/.bashrc” file. Append line “export PATH=\$PATH:/usr/local/arm/4.3.2/bin”, save and exit the file.
- Logout and Login the system again(no need to reboot the system, just go to “start”->“logout”). The configurations will be effective and you can check by using env command.
- Type “arm-linux-gcc -v”, you can see the below output.

```

File Edit View Search Terminal Help
[root@localhost vamsi]# arm-linux-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabi
Configured with: /scratch/julian/lite-respin/linux/src/gcc-4.3/configure --build=i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi --enable-threads --disable-libmudflap --disable-libssp --disable-libstdcxx-pch --with-gnu-as --with-gnu-ld --enable-languages=c,c++ --enable-shared --enable-symvers=gnu --enable-cxa_atexit --with-pkgversion='Sourcery G++ Lite 2008q3-72' --with-bugurl=https://support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery --with-sysroot=/opt/codesourcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/scratch/julian/lite-respin/linux/install/arm-none-linux-gnueabi/libc --with-gmp=/scratch/julian/lite-respin/linux/obj/host-libs-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-mpfr=/scratch/julian/lite-respin/linux/obj/host-libs-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --disable-libgomp --enable-poison-system-directories --with-build-time-tools=/scratch/julian/lite-respin/linux/install/arm-none-linux-gnueabi/bin --with-build-time-tools=/scratch/julian/lite-respin/linux/install/arm-none-linux-gnueabi/bin
Thread model: posix
gcc version 4.3.2 (Sourcery G++ Lite 2008q3-72)
[root@localhost vamsi]#

```

Building & Running up Linux Applications

Building linux Applications (Hello World Program)

- The Hello World source program **hello.c** can be found in **/examples/hello** directory.
- **Compiling Hello World Program:**
 - First enter into the /examples/hello.
 - You can compile the program and get the executable file by using two ways as listed below.
 - Compile the program using compiler
#arm-linux-gcc main.c -o hello
 - Using make file
#make

Downloading and running “Hello World Program”

- There are four ways to download a compiled executable to a target board.
 - Copy it to storage media(USB Drive).
 - Download via FTP.
 - Download via a serial port.
 - Run via NFS

Copy it to a storage media (USB Drive)

- Copy the compiled program to a **USB Drive** and **mount** the **USB drive** to the **target board** and copy the program from the **USB Drive** to the **/bin** directory of the board.
- After plugging the **USB Drive** to the **target boards** USB Host port, the driver will be automatically mounted on the “**/udisk**” directory. Execute the following commands to run the program.

```
#cd /udisk
```

USB Drive is mounted to udisk automatically.

```
./hello
```

To run the program

- **Note:** If the USB Drive is to be unplugged, you need to go back to the root directory to run “umount /udisk” for next auto mount.

Download via FTP (via LAN Cable) Chit..

- Execute the below commands for target boards command prompt.

```
vamsi@localhost:/home/vamsi
File Edit View Search Terminal Help
[ root@FriendlyARM / ]# ls
bin          home        lost+found   root        tmp         www
dev          lib          opt          sbin        usr
etc          linuxrc      proc         sys         var
[ root@FriendlyARM / ]# cd home
[ root@FriendlyARM /home]# ls
plg
[ root@FriendlyARM /home]# cd plg
[ root@FriendlyARM plg]# ls
hello  led
[ root@FriendlyARM plg]# ./hello
hello, FriendlyARM!
[ root@FriendlyARM plg]# 
```

Download via a serial port

- Please refer to the user's manual on how to transfer files via a serial port. Follow the way to transfer the compiled program to a target board. Remember to add the executable mode to the file.

```
#chmod +x hello
```

Run via NFS(Network File System)

- Its quite popular to run programs via **NFS** in Linux. This way saves you time for program download. In this example, the executable is very small, so downloading does not take too much time. However large files could take longer time to download. In case of larger file this method presents is advantage.
 - Please refer to user's manual on how to configure the NFS service and then run the commands below in the host machine.
- #mount -t nfs -o nolock 192.168.1.123:/opt/FriendlyARM/micro2440/root_nfs /mnt**
- If mounting succeeds, copy the hello program to the “/mnt” directory, this automatically transfers the file to **target board's** directory “**/opt/FriendlyARM/micro2440/root_nfs**” .
 - Go to the above mentioned directory on **target board** and run the commands below on the target board.

```
#cd /mnt  
#./hello
```

Decompressing ARM Tool Chain

- A main step to build a development environment is to set up a cross compile environment. The following steps will introduce how to build a compile environment that can compile arm-linux kernels, drivers and applications.
- From updated **Linux-2.6.29** we use “**arm-linux-gcc-4.3.2**” as our cross compiler.
- **Step1:** Copy the compressed file “**arm-linux-gcc-4.3.2.tgz**” into a /tmp directory. Enter into the /tmp directory and execute the following command.

```
#cd \tmp  
#tar -xvzf arm-linux-gcc-4.3.2.tgz -C /
```

These commands will install “**arm-linux-gcc**” in the “**/usr/local/arm/4.3.2**”. **Note: Enter the commands as root.**

- You can see lot of previously used tools which are compatible to ARM in **/usr/local/arm/4.3.2/bin**.

Configuring and Compiling a BootLoader

Bootloaders

- **Boot-loader** means to download and boot systems. It is similar to BIOS in a PC. Most chip vendors provides embedded systems with this program which are very mature and stable. Usually it is unnecessary for one to develop from scratch.
- Boot-loader is responsible for processor, board-specific initializations, loading a kernel and an optional initial ramdisk into memory and passing control to the kernel.
- In addition, a boot-loader might be in charge of providing BIOS service, performing POST, supporting firmware downloads to the target, and passing memory layout and configuration information to the kernel.
- Some boot-loaders support a debug monitor to load and debug stand-alone code on to the target device.

Bootloaders(Contd....)

- **Boot-loader** architecture depends on the processor family, the chipset present on the hardware platform, the boot device, and the operating system running on the device.
- The **boot-loader** needs a mechanism to be transferred from host development pc to the target's boot device. This is called **boot-strapping**.
- **Boot-strapping** is straightforward in PC where **BIOS** flash is programmed using an external burner if it is corrupted or updated after booting into OS if its healthy.
- Embedded devices however do not have a generic method for **boot-strapping**.
- Types of Boot-loader on Friendly ARM Boards
 - Vivi
 - Supervivi
 - U-boot

Popular Bootloaders

- **Vivi:** Supported by samsung, it was originally created by mizi, it is open source, and should be compiled with “arm-linux-gcc”, but it no longer gets official support now. It is best suited for Samsung's S3C24xx ARM chips. It supports file download via FTP, system boot via NFS and common applications.
- **Supervivi:** Developed and maintained by **Friendly Arm**, It is based on vivi. Currently Friendly Arm doesn't provide its source code. It keep vivi's functions and integrates extra features such as support for **CRAMFS, YAFFS, USB Download and System Booting** for Linux, Wince, uCos and VxWorks. It also enables downloading to and running applications in RAM. In addition, it has a unique feature: system backup and restore and is ready for mass production. It is very strong and popular bootloader for 2440/2410 sytems.

Popular Bootloaders(Contd...)

• **U-Boot:** An open source bootloader dedicated to linux systems. It should be compiled by **arm-linux-gcc**, and has strong network functions.(Without network, U-Boot almost loses all its advantages. In 2440/2410 systems, network is facilitated is thorough extension of network chips. This will definitely add to its cost). It supports download and system start-up via NFS. Now U-Boot is very popular and being in great move, but does not support booting from Nand Flash in 2440/2410 systems.

Booting Sequence

- We can select the booting mode by toggling the **S2** switch.
- When toggling the **S2** switch to the “**Nor Flash**” side the system will be booted from the Nor Flash.
- When toggling the **S2** switch to the “**Nand Flash**” side the system will be booted from the Nand Flash.
- Board is by default shipped with the S2 switch toggled to the **Nand flash** side and it will boot from the Nand Flash.
- Both the Nor and Nand flash have been installed with the same **BIOS**(which supports both types of Flash. The only difference is that the system will have different start-up interfaces).
- When you boot from Nand Flash the linux kernel image loads and you can see the linux interface.

Make menuconfig

- “**Make menuconfig**” is used to configure **Kernel & Bootloader**.
- This is where you pick and choose the pieces that form part of the operating system. You may specify whether each desired component is to be statically or dynamically linked to the kernel.
- Menuconfig** is a text interface to the kernel configuration menu. The configuration information that you choose is saved in a file named **.config** in the root of your source tree.
- If you don't want to weave the configuration from scratch, use the file `arch/your-arch/defconfig`(or `arch/your-arch/configs/your-machine_defconfig` if there are several supported platforms for your architecture) as the starting point.
 - Ex: If you are compiling the kernel for the 32-bit x86 architecture, do this:
`bash> cp arch/x86/configs/i386_defconfig .config`

Configuring and Compiling V

- **Note:** To compile vivi, install arm-linux-gcc-2.95.3 cross compiler.
- Download the cross compiler from the below link
<http://ftp.armlinux.org.uk/pub/armlinux/toolchain/>
- Click the link **cross-2.95.3.tar.bz** and the download starts.
- Follow the below steps to install **cross-2.95.3.tar.bz** cross compiler.
- Rename **cross-2.95.3.tar.bz** to **cross-2.95.3.tar** and use the below command to decompress the tar file.
“**tar xf cross-2.95.3.tar**” decompresses to a folder named “**2.95.3**”

How to install:

```
Cd /usr/local
```

```
mkdir arm
```

```
cd arm
```

```
copy the folder “2.95.3” to /usr/local/arm
```

Configuring and Compiling VI

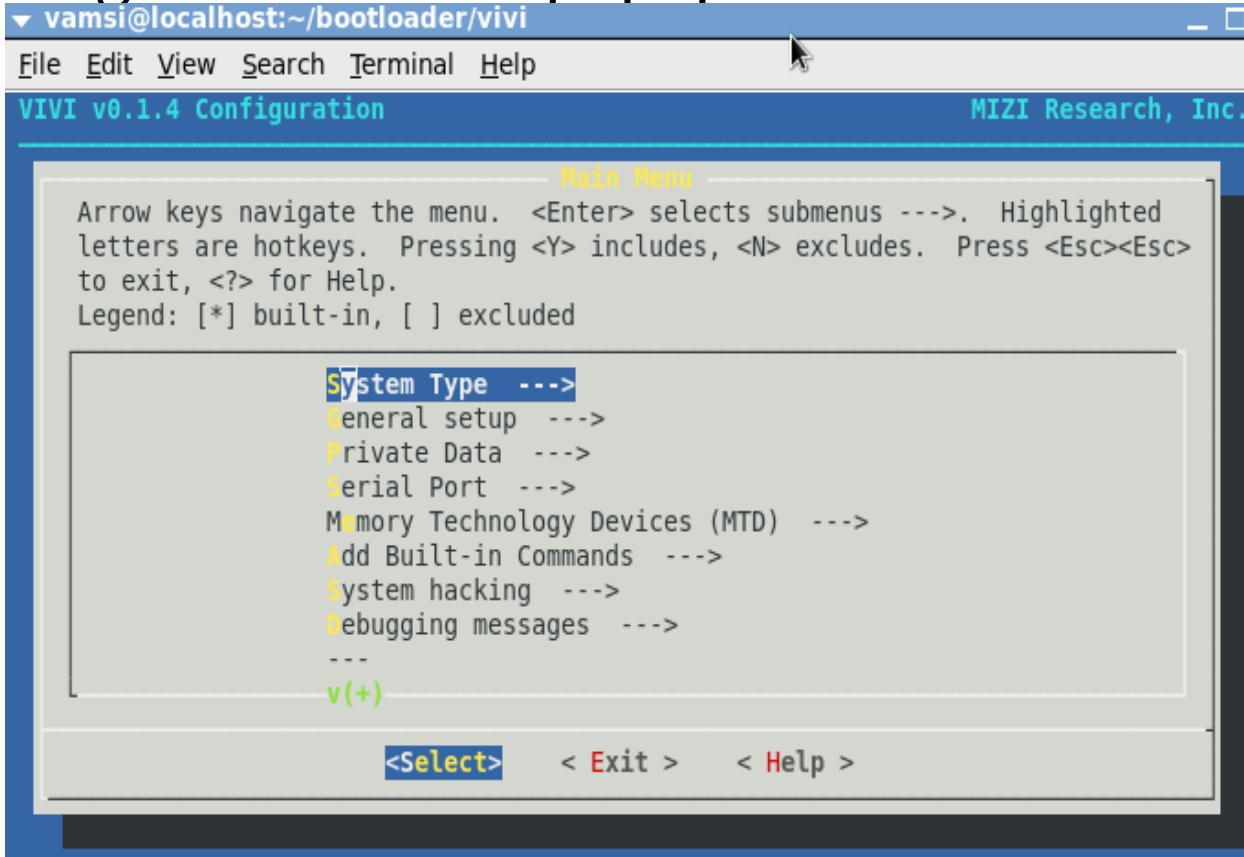
- After uncompressed and installing a arm-linux-gcc-2.95.3 compiler follow the below steps to compile vivi.

Cd /root/bootloader/vivi (goto to vivi source code folder)

make clean

make menuconfig

The following window will popup



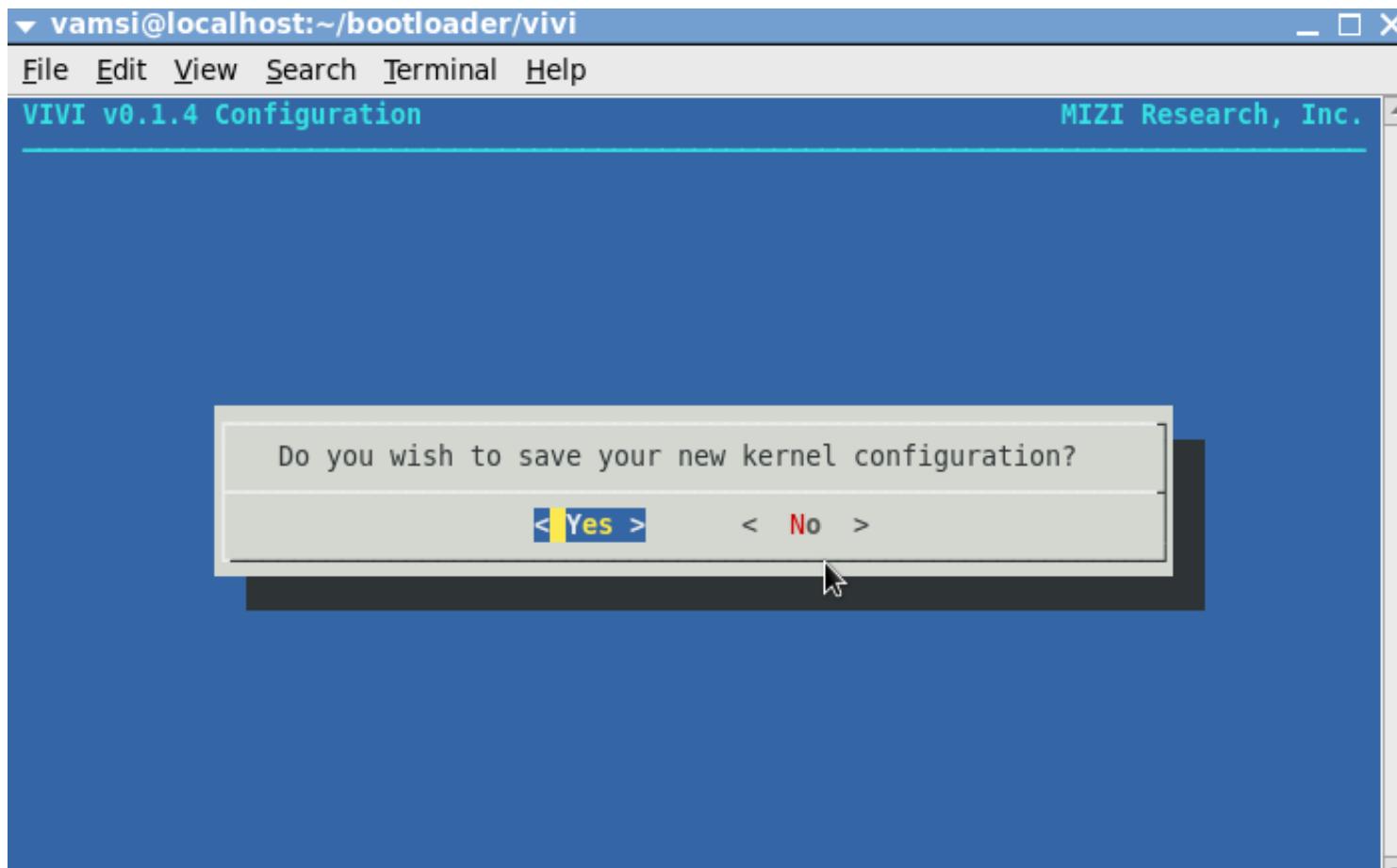
Configuring and Compiling VI

Just follow the default options and “Exit”

- Choose “Yes” and enter, run “make” to compile:

#make

- A vivi will be generated in the current directory. Download it to a target board’s Nand Flash.



Configuring and Compiling U-Boot

```
cd uboot  
git clone git://repo.or.cz/u-boot-openmoko/mini2440.git  
export CROSS_COMPILE=arm-none-linux-gnueabi-  
make mini2440_config  
make
```

```
cp config_mini2440_n35 .config
```

```
#make menuconfig
```

```
#make zImage
```

Configuring and Compiling a Kernel

Configuring the Kernel

- Configuring the kernel is required before building it. Because the kernel offers myriad features and supports a varied basket of hardware there is a lot to configure.
- Configuration in the kernel is controlled by configuration options, which are prefixed by CONFIG in the form of CONFIG_FEATURE. For example, symmetrical multiprocessing(SMP) is controlled by the configuration option CONFIG_SMP. If this option is set, SMP is enabled; if unset SMP is disabled. The configure options are used both to decide which files to build and to manipulate code via preprocessor directives.
- Configuration options that control the build are either **booleans** and **tristates**.
 - A **Boolean option** is either **yes** or **no**. Kernel features, such as **CONFIG_PREEMPT**, are usually Boolean s
 - A **tristate option** is one of **yes**, **no** or **module**. The module setting represents a configuration option that is set but is to be compiled as module(separate dynamically loaded object).A **yes** option means to compile the code into the **main kernel image** and not as module.

Configuration Methods(1)

- The Linux Kernel Build System(**kbuild**) includes support for variety of multiple tools to facilitate configuration. These utilities divide various configuration options into categories such as “**Processor Type and Features**”. You can move through the categories, view the kernel options and change their values.

\$make config : Provides a command line interface. This utility goes through each option, one by one, and asks the user to interactively select **yes, no**(for tristate) **modules**. If a **.config** configuration file is already present, it uses that file to set the default values of the options it asks you to set. This utility takes a longer time.

\$make menuconfig : This utility is based on **ncurses-based graphical libraries**. If a **.config** file is present it uses it to set default values.

\$make gconfig : gtk+ based graphic utility.

• All the above tools generate a **.config** file in the root of the kernel source tree containing configuration options.

Configuration Methods(2)

- To view the kernel configuration menu, type the appropriate command at the command line with the proper parameters. For example to cross compile the Linux Kernel for use on embedded ARM system, you might use the following command line

\$make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig

(Note that the cross compiler prefix ends with hyphen which form's **arm-linux-gcc** and there is a space between that and the **menuconfig** itself.

- This presents a graphical configuration menu from which available options can be selected. Many features and drivers are available as modules, and it is possible to choose whether to build features into the kernel or as modules at this stage. Once the kernel has been configured, you can quit the kernel configuration menu via the **Escape key** or the **Exit menu** item. The kernel configuration system will ask whether to save the new configuration. Choosing **Yes** saves the new configuration into a new **.config** file.

Compiling the Kernel(1)

- Compiling the kernel involves a number of steps. These includes building the kernel image and building the kernel modules. Each step uses a different **make** command. However, you could also carry out all these steps using a single command.
- **Building the kernel:** Building the kernel requires little more than a simple call to GNU make. Depending upon you choose architecture, you might also need to specify what kind of image will be produced. For example in the case of ARM platform, you could use the following command to create a compressed image.

```
#make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

- The **zImage** target instructs the Makefile to build a kernel image that is compressed using the **gzip algorithm**. The **vmlinux** target instructs the **Makefile** to build only the uncompressed image. Note that this image is generated even when a compressed image is requested.
- If you need further more information look into **Documentation/i386/boot.txt** included in kernel sources.

Compiling the Kernel(2)

- **Building the Modules:** With the kernel image properly built, you can now build the kernel modules:

```
$make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

- The duration of this stage depends largely on the number of kernel options you choose to build as modules instead of having been linked in as part of the main kernel image.

- With both the kernel image and the kernel modules now built, it is time to install them onto the target system. Before you do so, note that if you needed to clean up the kernel's sources and return them to their initial state prior to any configuration, dependency building or compilation, you could use the following command:

```
$make ARCH=arm CROSS_COMPILE=arm-linux- distclean
```

Compiling the Kernel(3)

- For storing kernel related configuration you will require four files;
 - The uncompressed kernel image(vmlinux).
 - The kernel symbol map(System.map).
 - The configuration file(.config).
- All these three are found in the kernel source's root directory.
- The fourth file compressed kernel image file is found in the ***arch/your_arch/boot***. Where **your_arch** is the name of your target's architecture, and is called zImage(size limit only 512kb) or bImage(no size limit).
- Have a look at the ***arch/your_arch/Makefile*** for a full description of all the Makefile boot image targets for your architecture.

Installing Kernel & Modules

- The kernel you just generated and its modules will have to be copied to your target to be used. The actual copying of kernel and its modules is covered in later sections.
- Installing Modules: The kernel **Makefile** includes the **modules_install** target for installing the kernel modules. By default, the modules are installed in the **/lib/modules** directory. This appropriate for most **desktop** and **enterprise Linux environment** but doesn't work so well when you are using **cross-development environment**.
- In case of cross development environment its not mandatory to install into your host /lib/modules. Instead you need to use an alternative location.
- To install the Linux kernel modules in an alternate directory, use this command:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- INSTALL_MOD_PATH  
=${PRJROOT}/images/modules-2.6.20 \ modules_install
```

- Once it is done copying the modules, the kernel build system will try to build the module dependencies needed for the module utilities during runtime. Look for **depmod**

Integrating Hello Module into Kernel

The source code of “**Hello Module**” is located in the linux-2.6.29/drivers/char directory, the file name is “**mini2440_hello_module.c**” and the source file is presented below.

```
#include <linux/kernel.h>
#include <linux/module.h>
static int __init mini2440_hello_module_init(void)
{
    printk("Hello, Mini2440 module is installed !\n");
    return 0;
}
static void __exit mini2440_hello_module_cleanup(void)
{
    printk("Good-bye, Mini2440 module was removed!\n");
}
module_init(mini2440_hello_module_init);
module_exit(mini2440_hello_module_cleanup);
MODULE_LICENSE("GPL");
```

Integrating Hello Module into Kernel

- ♦ Embed the file into the kernel and compile it.
- ♦ This step shows how to link this source file into the kernel. In the 2.6 kernel you need to add a driver's source code into the kernel tree and configure it.
- ♦ **Step1:** Edit the Kconfig file in “linux-2.6.29/drivers/char/” directory and check the driver option to make it listed in the pop up window after executing “**make menuconfig**”. Open the Kconfig file and add the rectangle marked sections in the next slide. Save and exit.
- ♦ Go to the linux-2.6.29 directory and run “make menuconfig”. Select “Device Driver”->”Character devices” and you will find the newly added item. Press the space key, it will be marked “<M>” which means this item will be compiled as a module. Press it a second time it will turn to “<*>” which means it will be compiled into the kernel.
- ♦ **Step 2:** The previous step describes how to set an item's compile option in the kernel configuration, but the **mini2440_hello_module.c** still cannot be compiled into the kernel.

```
vamsi@localhost:/home/vamsi/FRIENDLYARM/linux-2.6.29/drivers/c
File Edit View Search Terminal Help
/dev/kmem device is rarely used, but can be used for cer
kind of kernel debugging operations.
When in doubt, say "N".

config LEDS_MINI2440
    tristate "LED Support for Mini2440/QQ2440 GPIO LEDs"
    depends on ARCH_S3C2410
    help
        This option enables support for LEDs connected to GPIO lines
        on Mini2440/QQ2440 boards.

config MINI2440_HELLO_MODULE
    tristate "Mini2440/QQ2440 module sample"
    depends on ARCH_S3C2410
    default m if MACH_FRIENDLY_ARM_MINI2440
    help
        Mini2440/QQ2440 module sample.

config MINI2440_BUTTONS
    tristate "Buttons driver for FriendlyARM Mini2440/QQ2440 development bo
rds"
    depends on MACH_FRIENDLY_ARM_MINI2440
    default y if MACH_FRIENDLY_ARM_MINI2440
search hit BOTTOM, continuing at TOP
```

```
vamsi@localhost:/home/vamsi/FRIENDLYARM/linux-2.6.29
File Edit View Search Terminal Help
.config - Linux Kernel v2.6.29.4 Configuration
```

Character devices

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < >

-*- Virtual terminal	*
[] Support for binding and unbinding console drivers	
[] /dev/kmem virtual device support	
<*> LED Support for Mini2440/QQ2440 GPIO LEDs	
<M> Mini2440/QQ2440 module sample	
<*> Buttons driver for FriendlyARM Mini2440/QQ2440 development bo	
[] QQ2440 buttons driver	
<*> Buzzer driver for FriendlyARM Mini2440/QQ2440 development bo	
[*] ADC driver for FriendlyARM Mini2440/QQ2440 development boards	
[] Non-standard serial port support	
v(+)	

<Select> < Exit > < Help >

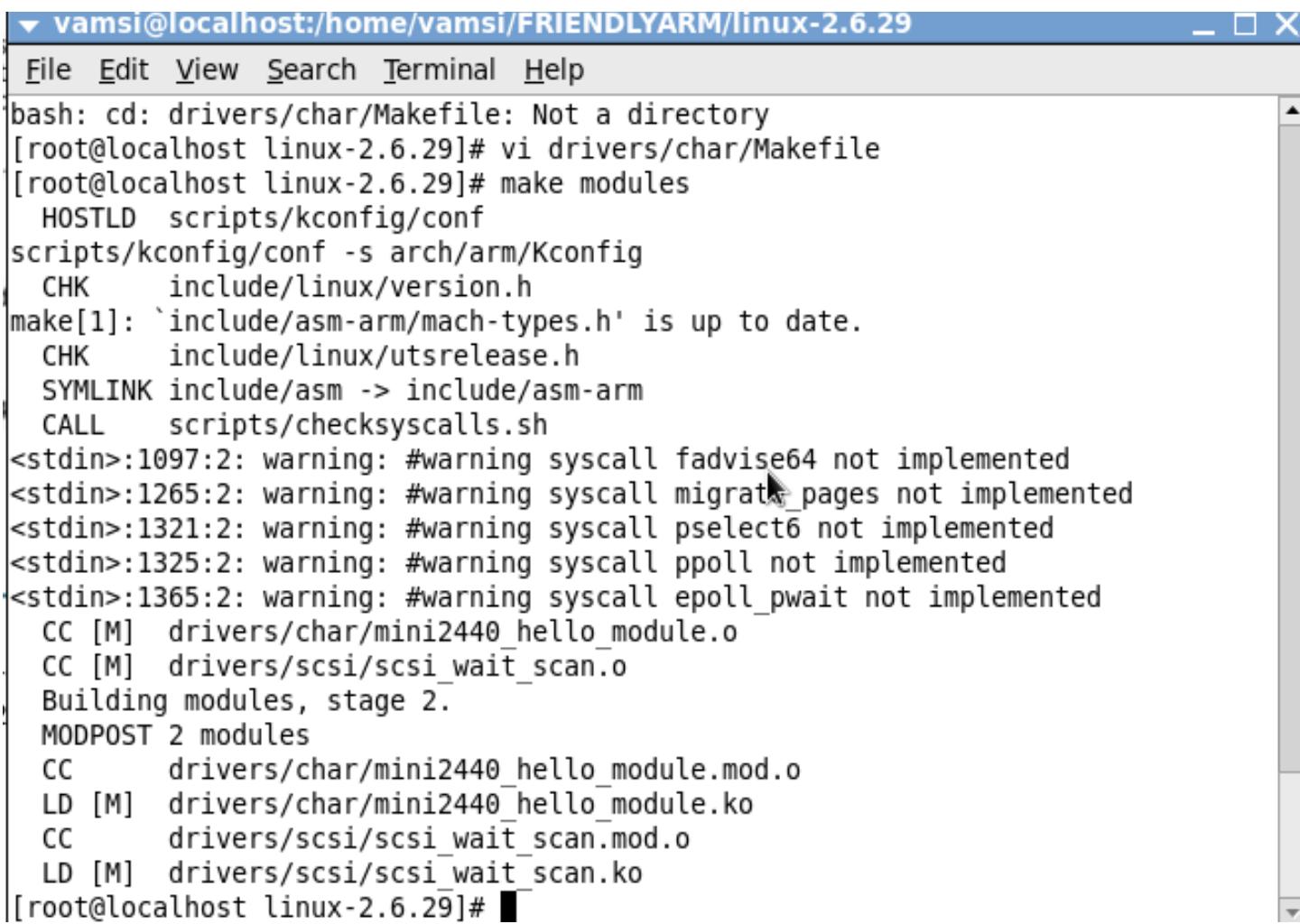
Integrating Hello Module into Kernel

- Link the source code to its kernel configuration. Edit the “**linux-2.6.29/drivers/char/Makefile**” as shown below, save and exit.

```
vamsi@localhost:/home/vamsi/FRIENDLYARM/linux-2.6.29
File Edit View Search Terminal Help
obj-$(CONFIG_IPMI_HANDLER)      += ipmi/
obj-$(CONFIG_HANGCHECK_TIMER)   += hangcheck-timer.o
obj-$(CONFIG_TCG_TPM)           += tpm/
obj-$(CONFIG_PS3_FLASH)         += ps3flash.o
obj-$(CONFIG_JS_RTC)           += js rtc.o
js-rtc-y = rtc.o
obj-$(CONFIG_LEDS_MINI2440)     += mini2440 leds.o
obj-$(CONFIG_MINI2440_HELLO_MODULE) += mini2440 hello module.o
obj-$(CONFIG_MINI2440_BUTTONS)   += mini2440 buttons.o
obj-$(CONFIG_MINI2440_BUZZER)    += mini2440 pwm.o
obj-$(CONFIG_MINI2440_ADC)       += mini2440 adc.o
# Files generated that shall be removed upon make clean
clean-files := consolemap_deftbl.c defkeymap.c
quiet_cmd_conmk = CONMK    $@
cmd_conmk = scripts/conmakehash $< > $@
$(obj)/consolemap_deftbl.c: $(src)/$(FONTPMAPFILE)
```

Integrating Hello Module into Kernel

- Step 3: Go back to the root directory of the linux-2.6.29 source code, execute “**make modules**” and a kernel moduel “mini2440_hello_module.ko” will be generated:
- The module compilation has been completed.



A screenshot of a terminal window titled "vamsi@localhost:/home/vamsi/FRIENDLYARM/linux-2.6.29". The window contains the following text output from the "make modules" command:

```
File Edit View Search Terminal Help
bash: cd: drivers/char/Makefile: Not a directory
[root@localhost linux-2.6.29]# vi drivers/char/Makefile
[root@localhost linux-2.6.29]# make modules
HOSTLD scripts/kconfig/conf
scripts/kconfig/conf -s arch/arm/Kconfig
CHK include/linux/version.h
make[1]: `include/asm-arm/mach-types.h' is up to date.
CHK include/linux/utsrelease.h
SYMLINK include/asm -> include/asm-arm
CALL scripts/checksyscalls.sh
<stdin>:1097:2: warning: #warning syscall fadvise64 not implemented
<stdin>:1265:2: warning: #warning syscall migrate_pages not implemented
<stdin>:1321:2: warning: #warning syscall pselect6 not implemented
<stdin>:1325:2: warning: #warning syscall ppoll not implemented
<stdin>:1365:2: warning: #warning syscall epoll_pwait not implemented
CC [M] drivers/char/mini2440_hello_module.o
CC [M] drivers/scsi/scsi_wait_scan.o
Building modules, stage 2.
MODPOST 2 modules
CC drivers/char/mini2440_hello_module.mod.o
LD [M] drivers/char/mini2440_hello_module.ko
CC drivers/scsi/scsi_wait_scan.mod.o
LD [M] drivers/scsi/scsi_wait_scan.ko
[root@localhost linux-2.6.29]#
```

Integrating Hello Module into Kernel

- Now its time to download the program to target board and run it.
- Here we use usb tool to download the “mini2440_hello_module.ko” to the target board and run:

#insmod mini2440_hello_module.ko

- The module will be loaded:
- Try the command below, the module will be unloaded.

#rmmmod mini2440_hello_module.ko

- Note: To rmmod the module correctly, you need to put the module in the “lib/modules/2.6.29-FriendlyARM” directory.

LED Driver

- **Driver's code location:** linux-2.6.29/drivers/char/mini2440_leds.c
- **Device name:** /dev/leds
- **Test program location:** micro2440/examples/led/led.c
- To write an appropriate driver for a device, you need to learn its hardware, e.g., registers, physical addresses, interrupts and so on. The LED Driver described in this section is simple example. It uses the following hardware resources.
- This MICRO2440 Board has 4 LED's

LED	IO Register	CPU PIN
LED1	GPB5	K2
LED2	GPB6	L5
LED3	GPB7	K7
LED4	GPB8	K5

- To operate an IO, you need to configure its register by calling related functions or macros e.g. **s3c2410_gpio_cfgpin**. This function is defined in linux-2.6.29/arch/arm/mach-s3c2410/include/mach/hardware.h and implemented in linux-2.6.29/arch/arm/plat-s3c24xx/gpio.c

LED Driver

- Most of the popular embedded systems have already defined **s3c2410_gpio_cfgpin** kind of functions so there is no need for users to create new ones unless they use a CPU whose architecture is not supported by Linux.
- The LED Driver code details how “**s3c2410_gpio_cfgpin**” gets called. To make “**s3c2410_gpio_cfgpin**” work, in addition you also need to call other helper functions such as **misc_register**, creating a device function structure with “**file_operations**”, loading and exiting a module with “**module_init**” and “**module_exit**”. The last two are used in the “**Hello Module**” example.
- Some of the functions shown below are not used quite often. You will get more familiar with them as you become more proficient in Linux Driver Development.

Miscellaneous char drivers

- **misc_register** is used to register a small device needing a single entry point with the **LED** driver.
- The linux kernel exports **misc_register** interface to allow modules to register their own small drivers.
- In UNIX, linux and similar other operatins ysystems every device sis identified by two numbers: a “major” number and a “minor” number. These numbers can be seen by invoking **ls -l /dev**.
- Every device driver register its major number with the kernel and is completely responsible for managing its minor numbers. Use of any device with that major number will fall on the same device driver, regardless of the minor number. As a result, every driver needs to register a major number, even if it only deals with a single device.
- Since the kernel keeps a static table of device drivers, unnecessary allocation of major numbers is rather wasteful of RAM. The linux kernel therefore, offers a simplified interface for simple drivers- those that will register a single entry point.

Miscellaneous char drivers (Contd...)

- The misc driver exports two functions for user modules to register and unregister their own minor number.

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

```
Int misc_register(struct miscdevice *misc);
```

```
Int misc_deregister(struct miscdevice *misc);
```

- Each user module can use the register function to create its own entry point for a minor number, and **deregister** to release resources at unload time. The **miscdevice.h** also declares **struct miscdevice** in the following way.

```
struct miscdevice {  
    int minor;  
    const char *name;  
    struct file_operations *fops;  
    struct miscdevice *next, *prev;  
};
```

Miscellaneous char drivers (Contd...)

- The five fields have the following meaning:
- **Minor** is the minor number being registered. Every **misc device** must feature a different **minor number**, because such a number is the only link between the file in **/dev** and the **driver**
- **Name** is the name for this device, meant for human consumption: users will find the name in the **/proc/misc** file.
- **Fops** is a pointer to the file operations which must be used to act on the device.
- **Next** and **prev** are used to manage a circular-linked list of registered drivers. Fill the first three fields with sensible values.
- The real question with the misc device driver is “what is a sensible value for the minor field?” Assignment of minor number is performed in two ways: either you can use an “officially assigned” number, or you can resort to dynamic assignment. In the latter case, your driver asks for free minor number, and the kernel returns one.

Miscellaneous char drivers (Contd...)

- The typical code sequence for assigning a dynamic minor number is as follows:

```
static struct miscdevice my_dev;  
int init_module(void)  
{  
    int retval;  
    my_dev.minor = MISC_DYNAMIC_MINOR;  
    my_dev.name = "my";  
    my_dev.fops = &my_fops;  
    retval = misc_register(&my_dev);  
    if (retval) return retval;  
    printk("my: got minor %i\n",my_dev.minor);  
    return 0;  
}
```

Miscellaneous char drivers (Contd...)

- Needless to say, a real module will perform some other task within **init_module**. After successful registration, the new misc device will appear in /proc/misc. This informative file reports which misc drivers are available and their minor numbers. After loading my, the file will include the following line:

63 my

- This shows that 63 is the minor number returned. If you want to create entry point in /dev for your misc module, you can use a script. The script takes care of creating the device node and giving it the desired permission and ownership.
- You might choose to find an unused minor number and hard-code it in your driver. This would save invoking a script to load the module, but the practice is strongly discouraged.
- If the same minor number is registered twice, only the first one will be accessible from user space. Although seemingly unfair, this can't be considered a kernel bug, as no data structure is corrupted.

Miscellaneous char drivers (Contd...)

- If you wish to register a safe minor number, you should used dynamic allocations.
- The file **Documentation/devices.txt** in the kernel source tree lists all the official numbers, including all the minor numbers for the misc drivers.
- Originally, the misc driver was designed as a wrapper for all the **“busmouse”** drivers- the kernel drivers for every non-serial pointer device.

How operations are dispatched.

- Every time a process interacts with a device driver, the implementation of the system call gives control to the correct driver by means of the **file_operation** structure.
- This structure is carried around by **struct file_operations**: every open file descriptor is associated to one such structure, and **file.fop** points to its own **file_operations** structure. At open time, the kernel allocates a new file structure to describe the object being opened, and initializes its operations structure according to what the file is. Sockets, FIFO's, disk files and devices get their own, different, operations. When a device is opened, its operations are looked up according to the major device number by referencing an array. The open method within the driver is then called.
- The open method of the misc driver is able to dispatch operations to the actual low-level driver by modifying **file.f_op**; the assigned value is the one in **my_dev.f_op**.

Character Driver

Character Driver Basics

- Char drivers can capture raw data from several types of devices: printers, mice, watchdogs, tapes, memory, RTCs and so on. They are however, not suitable for managing data residing on block devices capable of random access such as hard disks, floppies or compact discs.
- To access a char device, a system user invokes a suitable application program. The application is responsible for talking to the device, but to do that, it needs to find the identity of suitable driver. The contact details of the driver are exported to user space via the **/dev** directory.

```
bash> ls -l /dev
total 0
crw----- 1 root root 5, 1 Jul 16 10:02 console
...
Lwxrwxrwx 1 root root 3 Oct 6 10:02 cdrom->hdc
...
brw-rw---- 1 root disk 3, 0 Oct 6 2010 hda
brw-rw---- 1 root disk 3, 1 Oct 6 2010 hda1
...
crw----- 1 root tty 4, 2 Oct 6 10:20 tty1
crw----- 1 root tty 4, 2 Oct 6 10:02 tty2
```

Character Driver Basics

- The first character in each line of the ls output denotes the driver type: **c** signifies a **char driver**, **b** stands for a **block driver**, and **l** denotes a **symbolic link**. The number in the fifth column are called **major numbers**, and those in the sixth column are **minor numbers**. A **major number** broadly identifies the **driver**, whereas a **minor number** pinpoints the exact device serviced by the driver.
 - Ex: The IDE block storage driver /dev/hda owns a major number of 3 and is in charge of handling the hard disk on system, but when you further specify a minor number of 1 (/dev/hda1), that narrows down to the first disk partition.

Peek inside a Character Driver

- From the **code-flow** perspective, char drivers have the following.
 - An initialization (or **init()**) routine that is responsible for initializing the device and tying the driver to the rest of the kernel via registration functions.
 - A set of entry points (or methods) such as **open()**, **read()**, **ioctl()**, **llseek()**, and **write()**, which directly correspond to **I/O system calls** invoked by user applications over the associated /dev node.
 - **Interrupt routines, bottom halves, timer handlers, helper kernel threads**, and other support infrastructure. These are largely transparent to user applications.

Peek inside a Character Driver

- From the **data-flow** perspective, char drivers have the following.
 - A **per-device** structure. This is the information repository around which the driver revolves.
 - **Struct cdev**, a kernel abstraction for character drivers. This structure is usually embedded inside the per-device structure referred previously.
 - **Struct file_operations**, which contains the addresses of all driver entry points.
 - **Struct file**, which contains information about the associated **/dev** node.
- Each driver method has a system call counterpart that application use, we will look at the system calls and the matching driver methods in tandem.

Driver Initialization

- The driver **init()** method is the bedrock of the registration mechanism. Its responsible for the following:
 - Requesting allocation of device major numbers.
 - Allocating memory for the per-device structure.
 - Connecting the **entry points**(**open()**, **read()**, and so on) with the char driver's **cdev abstraction**.
 - Associating the device major number with the driver's **cdev**.
 - Creating nodes under **/dev** and **/sys**.
 - Initializing the hardware.
- First the **init()** method invokes **alloc_chrdev_region()** to dynamically request an **unused major number**. *dev_number* contains the allocated major number if the call is successful. The second and third arguments to **alloc_chrdev_region()** specify the **start minor number** and the number of supported **minor devices**. The last argument is the device name used to identify in **/proc/devices**.

Driver Initialization (Contd...)

```
>cat /proc/devices | grep tty
```

4 **tty**

4 **ttyS**

5 **/dev/tty**

188 **ttyUSB**

- 4 is the **dynamically** allocated **major number** for the **serial device**. During pre-2.6 days, dynamic device node allocation was not supported, so character drivers made call to **register_chrdev()** to **statically** request specific **major numbers**.
- **ser_dev** is the per-device data structure referred to earlier. **ser_fops** is the **file_operations** structure that contains the address of driver entry points. **ser_fops** also has a field called **owner** that is set to **THIS_MODULE**, the address of the driver module in question.
- The kernel uses an abstraction called **cdev** to internally represent char devices. Char drivers usually embed their **cdev** inside their per-device structure.

Driver Initialization (Contd...)

- **serial_init()** associates the file operations **ser_ops** with the cdev, and **cdev_add()** connects the major/minor numbers allocated by **alloc_chrdev_region()** to cdev.
- **class_create()** populates a **sysfs** entry for the device, and **class_device_create()** results in the generation of two **uevent**. **udevd** listens to uevents and generates device nodes after consulting its rules database. Add the following ot the udev rules directory (/etc/udev/rules.d)to produce devicenodes corresponding to the serial device(/dev/ttyS0 and /dev/ttyS1) on receiving the respective uevents.
- Device drivers that need to operate on a range of I/O addresses stake claim to the addresses via a call to **request_region()**. The regulatory mechanism ensures that request by others for the same region fail until the occupant releases it via a call to **release_region()**.
request_region() is commonly invoked by I/O bus drivers such as PCI and ISA to mark ownership of on-card memory in the processors address space.

Drivers Open and Release

- The kernel calls the drivers **open()** method when an application opens the corresponding device node. You can trigger execution of **ser_open** by doing this.

bash>cat /dev/ttys0

- The kernel calls the **release()** method when an application closes an open device. So when cat closes the file descriptor attached to /dev/ttys0 after reading the contents of serial device, the kernel invokes **ser_release()**.
- Lets take a closer look at **ser_open()**. The first is the extraction of **ser_dev**. The **inode** passed as an argument to **ser_open()** contains the address of the **cdev** structure allocated during initialization. **Cdev** is embedded inside **ser_dev**. To elicit the address of the container structure **ser_dev**, **ser_open()** uses the kernel helper function, **container_of()**.

Drivers Open and Release (Contd..)

- The other notable operation in **ser_open()** is the usage of the **private_data** field that is part of **struct file**, the second argument. You can use this field(file->private_data) as a place-holder to conveniently correlate information from inside other drivers methods.
- **try_module_get()/module_put()** include /linux/module.h
 - These manipulate the module usage count, to protect against removal(a module also can't be removed if another module uses one of its exported symbols). Before calling into module code, you should call try_module_get() on that module. If it fails, the module is being removed and you should act as if it was'nt there. Otherwise you can safely enter the module and call module_put() when you are finished. Mod

SCULL DEVICE

Allocating and Freeing Device Numbers

- One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with.
*int register_chrdev_region(dev_t first, unsigned int count, char *name)*
- **First** – Beginning device number of the range you would like to allocate. First contains two parts one is for start minor number and major number. The minor number portion of first is often 0, but there is no requirement to that effect.
- **Count** – Contiguous device numbers you are requesting.
- **Name** – Name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.
- Kernel can allocate a major number for you using the below function
*int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);*
- On successful completion, **dev** holds the **valid major number**.

The inode Structure

- The **inode** structure is used by the kernel internally to represent files. Therefore, it is different from the **file structure** that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single **inode** structure.
- The inode structure contains a great deal of info about a file. But only two fields are relevant for driver code.
- **dev_t i_rdev:** Represents device files, this field contains actual device numbers.
- **Struct cdev *i_cdev:** Kernel's internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device.

Char Device Registration(1)

- The kernel uses structure of type **struct cdev** to represent char devices internally. Before the kernel invokes your device's operations, you must allocate and register one or more of these structures. To do so your code should include <linux/cdev.h>, where the structure and its associated helper functions are defined.
- There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone **cdev** structure at runtime, you may do so with codes such as:

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

- However you will want to embed the **cdev** structure within a device specific structure of your own, that is what scull does. In that case you should initialize the structure that you have already allocated with
void cdev_init(struct cdev *cdev, struct file_operations *fops)
- Either way, there is one other **struct cdev** field that you need to initialize – owner field should be set to THIS_MODULE.

Char Device Registration(2)

- Once the **cdev** structure is set-up, the final step is to tell the kernel about it with a call to:
int cdev_add(struct cdev *dev, dev_t num, unsinged int count);
- Here, **dev** is the **cdev structure**, **num** is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device. Often the count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.
- As soon as **cdev_add** returns, your device is “**live**” and its operations can be called by the kernel. You should not call **cdev_add** until your driver is completely ready.
- To remove a char device from the system call:

void cdev_del(struct cdev *dev)

Interrupts & Exceptions

- An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events corresponds to electrical signals generated by hardware circuits which are inside and outside the CPU.
- **Interrupts** are often divided into **synchronous** and **asynchronous** interrupts.
- **Synchronous interrupts(Exceptions)** are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction. Exceptions on the other hand are caused by **programming errors** or by **anomalous conditions** that must be handled by the kernel. In the first case , the kernel handles the exception by delivering to the current process one of the **signals** familiar to **Unix** programmer. In the second case the kernel performs all the steps needed to recover from anomalous condition, such as a page fault or a request(via an int instruction) for a kernel service.

Interrupts & Exceptions

- **Asynchronous interrupts(Interrupts)** are generated by other hardware devices at arbitrary times with respect to the CPU clock signals. Interrupts are issued by either interval timer or I/O devices. For instance, the arrival of a keystroke from a user sets off an interrupt.

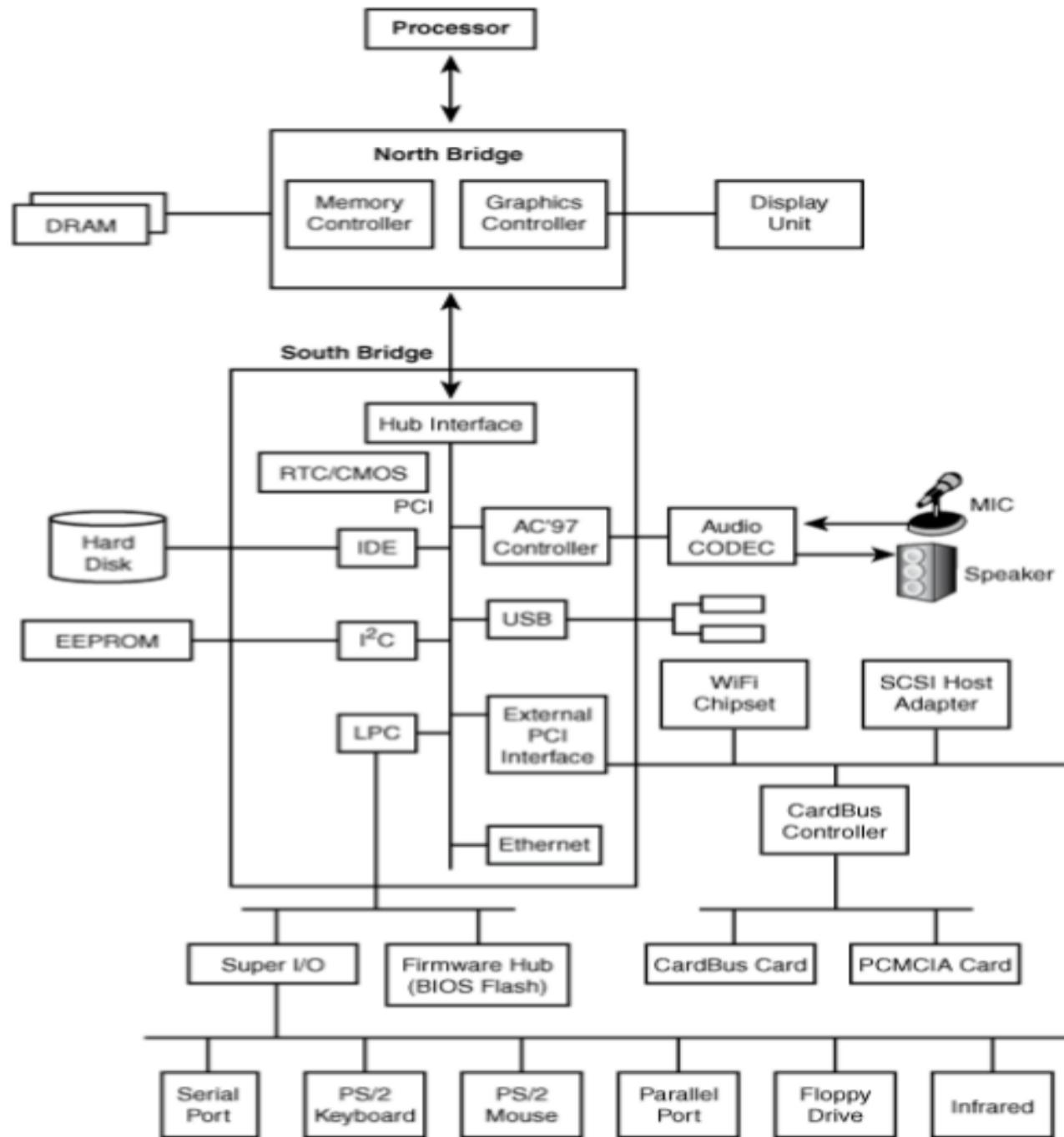


Devices & Drivers

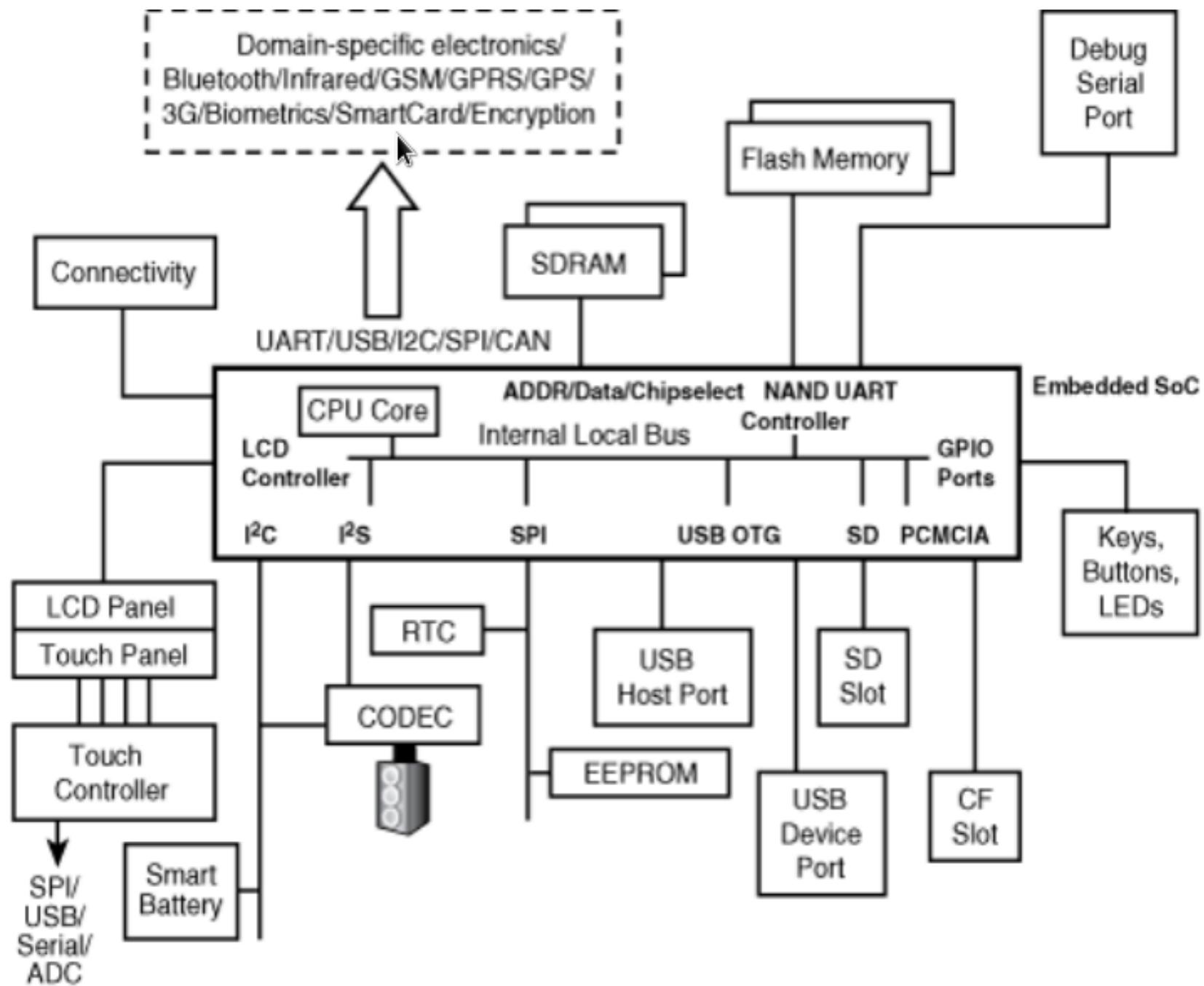
Devices & Drivers

- User **applications** cannot directly communicate with hardware because that entails possessing privileges such as executing special instructions and handling interrupts. **Device drivers** assume the burden of interacting with hardware and export interfaces that applications and rest of the kernel can use to access devices.
- **Applications** operate on devices via nodes in the **/dev** directory and access device information using nodes in the **/sys** directory.
- Next figure shows the hardware block diagram of typical PC-compatible system. The system supports diverse devices and interface technologies such as memory, video, audio, USB, PCI, WiFi, PCMCIA, I2C, IDE, Ethernet, serial port, keyboard, mouse, floppy driver, parallel port and infrared etc. The memory and graphics controller are part of a **North Bridge** chip-set in the PC architecture, whereas peripheral buses are sourced out of a **South Bridge**.

Hardware block diagram of a PC-compatible system



Hardware block diagram of a PC-compatible system



Serial Driver

Drivers Open and Release (Contd..)

- The other notable operation in **ser_open()** is the usage of the **private_data** field that is part of **struct file**, the second argument. You can use this field(file->private_data) as a place-holder to conveniently correlate information from inside other drivers methods.

Proc File System

Proc file system

- In linux there is an additional mechanism for the kernel and kernel modules to send information to processes - the /proc file system.
- Originally designed to allow easy access to information about processes(hence the name). It is now used by every bit of the kernel which has something interesting to report, such as /proc/modules which has the list of modules and /proc/meminfo which has memory usage statistics.
- The method to use the proc file system is very similar to the one used with device drivers you create a structure with all the information needed for the /proc file, including pointers to any handler function(in our case two functions, called when somebody attempts to read or write to the /proc file). Then, **init_module** registers the structure with the kernel and **cleanup_module** un-register it.

Blocking I/O Wait Queues

Blocking I/O

- How does a driver respond if it cannot immediately satisfy the request? A call to read may come when no data is available, but more is expected in the future. Or a process can attempt to write, but your device is not ready to accept the data, because your output buffer is full.
- The calling process usually does not care about such issues: the programmer simply expects to call ***read*** or ***write*** and have the call return after the necessary work has been done. So in such cases, your driver should block the process putting it to sleep until the request can proceed. Process cannot sleep unless it is assured that somebody else will wake up. Making it possible for your sleeping process to be found is instead accomplished through a data structure called a **wait queue** (list of processes waiting for a specific event).
- **Wait queue** is managed by means of “wait queue head”, a structure of type ***wait_queue_head_t***.

Wait queues(1)

- A wait queue head can be defined and initialized statically with:

DECLARE_WAIT_QUEUE_HEAD(name);

or dynamically as follows:

wait_queue_head_t my_queue;

init_waitqueue_head(&my_queue);

- Any process that sleeps must check to be sure that the condition it was waiting for is really true when it wakes up again. ***wait_event***(with a few variants) is used for this purpose. It combines handling the details of sleeping with a check on the condition a process is waiting for. The forms of ***wait_event*** are:

wait_event(queue,condition)

wait_event_interruptible(queue, condition)

wait_event_timeout(queue, condition, timeout)

wait_event_interruptible(queue, condition, timeout)

- ***Condition*** is an expression that is evaluated before and after sleeping. Until the condition evaluates to true the process sleeps.

Wait queues(2)

- ***wait_event_interruptible*** can be interruptible by signals whereas if you use ***wait_event*** your process is put into an UN-interruptible sleep.
- ***wait_event_timeout*** and ***wait_event_interruptible_timeout*** waits for a limited time, after that time period expires.
- Some other threads of execution (a different process or interrupt handler) has to perform the wakeup for you, since your process is of-course asleep. The basic function that wakes up sleeping processes is called ***wake_up***.

void wake_up(wait_queue_head_t *queue)

void wake_up_interruptible(wait_queue_head_t *queue)

- ***wake_up*** wakes up all processes waiting on the given queue. The other form restricts itself to processes performing an interruptible sleep.

Wait queues(3)

- The following behaviour should be implemented in order to adhere to the standard semantics.
 - If a process calls read but no data is available, the process is awakened as soon as some data arrives, and that data is returned to the caller, even if there is less than the amount requested in the count argument to the method.
 - If a process calls write and there is no space in the buffer, the process must block, and it must be on a different wait queue from the one used for reading. When some data has been written to the hardware device , and space becomes free in the output buffer, the process is awakened and the write call succeeds, although the data may be only partially written if there isn't room in the buffer for the count bytes that we requested.

Poll & Select

Poll and Select support(1)

- The purpose of **poll** and **select** calls is to determine in advance if an I/O operation will block.
- Applications that wants to read from multiple devices may use poll() and select() system call to block on multiple devices simultaneously.
- To use these calls on devices, the device driver must support poll() method implementation.
- When application issues the poll()/Select() system call the kernel will call the driver's poll method.
- When ever driver's status changes (ex: rx data is available, tx buffer empty) the driver informs the kernel which makes poll/select to unblock. The driver method is in-charge of two steps.
 - Call **poll_wait** on one or more wait queues that could indicate a change in the poll status. If no file descriptors are currently available for I/O, the kernel causes the process to wait on the wait queues for all file descriptors passed to the system call.
 - Return a bit mask describing the operations taht could be immediately performed without blocking.

[select\(\)](#), [FD_SET\(\)](#), [FD_CLR\(\)](#), [FD_ISSET\(\)](#), [FD_ZERO\(\)](#)

- The select function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, select() blocks, up to the specified timeout interval, until the specified condition is true or false.
- The select function call supports regular file descriptors, console descriptors, pipe descriptors, FIFO descriptors, socket descriptors and communication port descriptors.

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

- The nfds argument specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to nfds-1.
- The ***readfds*** specifies the fd's to be checked for being ready to read.
- The ***writefds*** specifies the fd's to be checked for being ready to write.
- The ***exceptfds*** specifies the fd's to check for error condition.

On successful completion the objects pointed to by readfds, writefds and exceptfds arguments are modified to indicate

`select()`, `FD_SET()`, `FD_CLR()`, `FD_ISSET()`, `FD_ZERO()`

- On successful completion the objects pointed to by `readfds`, `writelfds` and `exceptfds` arguments are modified to indicate which file descriptors are ready to read, write or have an error condition pending.
- The timeout argument specifies a maximum interval to wait for the selection to complete.

File descriptor masks of type `fd_set` can be initialized and tested with `FD_CLR()`, `FD_ISSET()`, `FD_SET()`, and `FD_ZERO()` macros.

`FD_CLR(fd, &fdset)`: Clears the bit for the file descriptor `fd` in the file descriptor set `fdset`.

`FD_ISSET(fd, &fdset)`: Returns a non-zero value if the bit for the file descriptor `fd` is set in the file descriptor set pointed to by `fdset`, and 0 otherwise.

`FD_SET(fd, &fdset)`: Sets the bit for the file descriptor `fd` in the file descriptor set `fdset`.

`FD_ZERO(&fdset)`: Initializes the file descriptor set `fdset` to have zero bits for all file descriptors.

Underlying Data Structures

- Whenever user calls *poll*, *select* the kernel invokes the poll method of all files referenced by the system call, passing the same **poll_table** to each of them. The **poll_table** structure is just a wrapper around a function that builds the actual data structure. That structure for poll and select is a linked list memory pages containing **poll_table_entry** structures. Each **poll_table_entry** contains **struct file** and **wait_queue_head_t** passed to poll_wait, along with an associated wait queue entry.
- If none of the drivers being polled indicates that I/O can occur without blocking, the poll call simply sleeps until one of the wait queues it is on wakes it up.

Asynchronous Notification(1)

- Some applications may use signal driven I/O programming. In this model application will install SIGIO signal handler and enables the signal driven I/O by a device driver. Now onwards, whenever data is available with the device the driver will send a signal to this application process.
- In this model application process registers its process ID(PID) with the kernel. The driver sends the signal to this PID whenever it receives the data i.e., applications can receive a signal whenever data becomes available.
- **Signal Driver I/O Programming:** User programs have to execute two steps to enable asynchronous notification from an input file.
 - First specify a process as the owner of the file. When a process invokes the F_SETOWN command using fcntl system call, the process ID of the owner process is saved in filp->f_owner. This step is necessary for the kernel to know just whom to notify.

Asynchronous Notification(2)

- In order to actually enable asynchronous notification, the user programs must set the **FASYNC** flag in the device by means of the ***F_SETFL fcntl*** command.
- After these two calls have been executed, the input file can request delivery of SIGIO signal whenever new data arrives. The signal is sent to the process stored in `filp->f_owner`.
- Not all devices support asynchronous notification, and you can choose not to offer it. Applications usually assume that the asynchronous capabilities is available only for **sockets** and **ttys**.

Driver's point of view-asynchronous i/o(1)

- The below list of sequence of operations has to be implemented in driver.
 - When `F_SETOWN` is invoked, nothing happens, except that a value is assigned to `filp->f_owner`.
 - When `F_SETFL` is executed to turn on `FASYNC`, the driver's ***fasync*** method is called. This method is called whenever the value of `FASYNC` is changed in `filp->f_flags` to notify the driver of the change, so it can respond properly.
 - When data arrives, all the processes register for asynchronous notification must be sent to `SIGIO` signal.
- In the first step there is nothing to be done in the driver, whereas the other steps involves maintaining a dynamic data structure to keep track of the different asynchronous readers.
- ***Struct fasync_struct*** is used to keep track of the asynchronous readers which is dynamic.

Driver's point of view-asynchronous i/o(2)

- The two functions that the driver calls correspond to the following prototypes:

```
int fasync_helper(int fd, struct file *filp, int moe, struct  
fasync_struct **fa);
```

```
void kill_async(struct fasync_struct **fa, int sig, int band);
```

- *fasync_helper* function is invoked to add or remove entries from the list of interested processes when the FASYNC flag changes for an open file.

- When data arrives *kill_fasync* is used to signal the interested processes. Its arguments are the signal to send(usually **SIGIO**) and the **band**, which almost always **POLL_IN**.

Semaphores

- Semaphores are used for mutual exclusion between multiple processes trying to read or write simultaneously.
- Semaphores can be declared and initialized as shown below:

struct semaphore sem;

sema_init(struct semaphore *sem, int count);

- A process can acquire the semaphore token by using ***down()*** or ***down_interruptible()*** functions as shown below. If token count number is zero in the semaphore, the process will go into blocking state. The ***up()*** system call will send a token to semaphore, which cause the blocking process to wakeup.

down(struct semaphore *sem);

down_interruptible(struct semaphore *sem);

up(struct semaphore *sem)

Communicating with H/W

TIMERS

Time Management

- Real world drivers have to deal with issues such as timing, memory management, hardware access etc. we have some kernel resource that are available for drivers for ***time management***.
- Dealing with time involves the following tasks
 - Measuring time elapses and comparing times.
 - Knowing current time
 - Delaying operation for a specified amount of time.
 - Scheduling asynchronous functions to happen at a later time.
- The kernel keeps track of the flow of time by means of timer interrupts. Timer interrupts are generated by the system's timing hardware at regular intervals; these intervals are programmed at boot time by the kernel according to the value of HZ, which is an architecture defined value ***linux/param.h***. Most platforms run at 100 to 1000 interrupts per second.
- **Note:** HZ present in

/usr/src/kernels/2.6.35.6-45.fc14.i686/include/asm-generic/param.h

Time intervals in kernel: Jiffies

- Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to 0 when the system boots, so it represents the number clock ticks since last boot.
- This internal kernel counter is called as **jiffies**. Its a 64 bit variable. They are basically used to calculate long delays.
- Using the jiffies counter:
 - The counter and the utility functions to read it live in `<linux/jiffies.h>` although you usually include `linux/schedule.h` that automatically pulls `jiffies.h`

Knowing the Current Time(1)

- Kernel code can always retrieve a representation of the current time by looking at the value of **jiffies**. Usually, the fact that the value represents only the time since the last boot is not relevant to the driver, because its life is limited to the system uptime. As shown, drivers can use the current value of jiffies to calculate time intervals across events(for example, to tell double clicks from single click in input device driver or calculate time-outs).
- Looking at **jiffies** is always sufficient when you need to measure time intervals. If you need very precise measurements for short time lapses, processor specific register come to the rescue.
- Its quite unlikely that a driver will ever need to know the wall-clock time, expressed in months, days, and hours; the information is usually needed only by user programs such as **cron** and **syslogd**.

Cron: Enables users to schedule jobs(commands or shell scripts) to run periodically at certain date or time).

Syslogd: is the system logging utility.

Knowing the Current Time(2)

- Dealing with real-world time is usually best left to user space, where the C library offers better support;
- Kernel keeps the current time by reading a clock device.
- **gettimeofday()** is used to get access to current time.
- You won't have to deal with human-readable representation of time. There is a kernel function that turns a wall-clock time into a jiffies value.

```
#include <linux/time.h>
unsigned long mktime(unsigned int year, unsinged int mon,
                     unsigned int day, unsigned int hour, unsigned int min,
                     unsigned int sec);
```

- Sometimes you have to deal with absolute time stamps even in kernel space. **void do_gettimeofday(struct timeval *tv)** fills struct timeval pointer with seconds and microseconds. Precision is in microsec's
- To get current time **struct timespec current_kernel_time(void)** is used.

Long Delays

- If you want to delay execution by a multiple of the clock tick or you don't require strict precision(for example, if you want to delay an integer number of seconds), the easiest implementation is as following, also known as *busy waiting*.

```
unsigned long j=jiffies + jit_delay * HZ;  
while(jiffies < j);  
/*nothing*/
```

- A better solution that allows other processes to run during the time interval is the following, although it can't be used in hard real-time tasks or other time-critical situations.

```
while(jiffies < j)  
    schedule();
```

- The variable j in this example and the following ones is the value of *jiffies* at the expiration of the delay and is always calculated as just shown for *busy waiting*

Long Delays(1)

- If you want to delay execution by a multiple of the clock tick or you don't require strict precision(for example, if you want to delay an integer number of seconds), the easiest implementation is as following, also known as *busy waiting*.

```
unsigned long j=jiffies + jit_delay * HZ;  
while(jiffies < j);  
/*nothing*/
```

- A better solution that allows other processes to run during the time interval is the following, although it can't be used in hard real-time tasks or other time-critical situations.

```
while(jiffies < j)  
    schedule();
```

- The variable j in this example and the following ones is the value of **jiffies** at the expiration of the delay and is always calculated as just shown for *busy waiting*

Long Delays(2)

- Another easiest implementation is a loop that monitors the jiffy counter and uses *time_before* function. Here *j1* is the value of jiffies at the expiration of the delay:

```
while(time_before(jiffies,j1))  
    cpu_relax();
```

int time_after_eq(unsigned long a, unsigned long b);

int time_before_eq(unsigned long a, unsigned long b);

- The first evaluates true when a, as a snapshot of jiffies, represents a time after b, the second evaluates to true when time a is before time b.

Short Delays(1)

- When a device driver needs to synchronize with hardware or deal with latencies in hardware and the delays involved are usually microseconds at most. In this case, relying on the clock tick(using jiffies) is definitely not the way to go.
- The kernel functions `ndelay`, `udelay`, and `mdelay` serve well for short delays, delaying execution for the specified period of nanoseconds, microseconds and milliseconds respectively. Their prototypes are

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

```
void ndelay(unsigned long nsecs);
```

```
void udelay(unsigned long usecs);
```

```
void mdelay(unsigned long msecs);
```

- Every architecture implements `udelay`, but other functions may or may not be defined. `mdelay` is looped around `udelay`.
- The above three functions are busy-waiting:other tasks can't be running during the time lapse.

Short Delays(2)

Maximum allowable delay is nearly one second(since calculations overflow for longer delays), the suggested maximum value for udelay is 1000 microseconds(one millisecond). The function mdelay helps in cases where the delay must be longer than one millisecond.

- There is another way of achieving millisecond(and longer) delays that does not involve busy waiting. The file <linux/delay.h> declares this functions:

void msleep(unsigned int millisecs);

unsigned long msleep_interruptible(unsigned int millisecs);

void ssleep(unsigned int seconds);

- The first two functions puts the calling process to sleep for the given number of ***millisecs***. A call to sleep is ***uninterruptible***, you can be sure that the process sleeps for at least the given number of ***millisecs***.
- If your driver uses ***wait_queues*** and you want the ***wakeup*** to break the sleep then use ***msleep_interruptible***. ***ssleep*** puts the process into an un-interruptible sleep for th given number of seconds.

Processor specific registers(1)

- If you need to measure very short time intervals or you need extremely high precision in your figures, you can resort to platform-dependent resources.
- CPU manufacturers introduced a way to count clock cycles as an easy and reliable way to measure time lapses. Most modern processor include a counter register that is steadily incremented once at each clock cycle. The clock counter is the only reliable way to carry out high-resolution timekeeping tasks.
- Most renowned counter register is the TSC(timestamp counter). It is a 64 bit register that counts CPU clock cycles; it can be read from both kernel space and user space.
- After including <asm/msr.h> you can use one of the macros
 - rdtsc(low32,high32);***
 - rdtscl(low32);***
 - rdtscll(var64);***

Processor specific registers(2)

- The first macro automatically reads the 64-bit value into two 32 bit variables; the next one (“reads low half”) reads the low half of the register into a 32-bit value discarding the high half. The last reads the 64-bit value into along long variable, hence the name.
- As an example using only the low half of the register, the following lines measure the execution of the instruction itself;

```
unsigned long ini,end;  
rdtscl(ini); rdtscl(end);  
printk("time lapse: %li",end-ini);
```

- Some other platforms offer same functionality instead of rdtsc.

```
#include <linux/timex.h>  
cycles_t get_cycles(void)
```
- The above function returns 0 on platforms that have no cycle-counter register

Kernel Timers(1)

- Whenever you need to schedule an action(timer handler) to happen later, without blocking the current-process until that time arrives, kernel timers are the tool for you. They thus work differently from work queues and tasklets in that you can specify when in the future your function will be called, whereas you can't tell exactly when a queued task will be executed. On the other hand, kernel timers are similar to work queues in that a function register in a kernel timer is executed only once – timers aren't cyclic.
- You register your function once and the kernel calls it once when the timer expires. The kernel timers are organized in a double linked list. A timer is characterized by its timeout value(in jiffies) and the function to be called when the timer expires. The timer handler receives an argument which is stored in the data structure, together with a pointer to the handler itself.

Kernel Timers(2)

- The data structure of a timer looks like the following, which is extracted from <linux/timer.h>

```
struct timer_list{  
    struct timer_list next;      //never touch this  
    struct timer_list *prev;    //never touch this  
    unsinged long expires;    //the timeout in jiffies  
    unsigned long data;        //argument to the handler  
    void (*function)(unsigned long data); //handler  
    volatile int running;      //dont touch  
}
```

- The timeout of the timer is a value in jiffies. Thus timer->function will run when jiffies is equal to or greater than timer->expires. Timeout is generated by the current value of jiffies and adding the amount of the desired delay. These are the functions used to act on timers:

Kernel Timers(3)

void init_timer(struct timer_list *timer);

- This inline function is used to initialize the timer structure. Currently, it zeros the prev and next pointer. Programmers are strongly urged to use this function to initialize a timer and to never explicitly touch the pointers in the structure, in order to be forward compatible.

void add_timer(struct timer_list *timer);

- This function inserts a timer into the global list of active timers.

int mod_timer(struct timer_list *timer, unsigned long expires);

- If you want to modify the time at which a timer expires, **mod_timer** can be used. After the call, the new expires value will be used.

int del_timer(struct timer_list *timer);

- If a timer needs to be removed from the list before it expires, **del_timer** should be called. When a timer expires, on the other hand, it is automatically removed from the list.

Tasklets(1)

- Another kernel facility related to timing issues is the ***tasklet*** mechanism. It is mostly used in interrupt management. This mechanism called ***tasklets***, is now the preferred way to accomplish bottom-half tasks; indeed, bottom halves themselves are now implemented with ***tasklets***.
- Each ***tasklet*** has associated with it a function that is called when the ***tasklet*** is to be executed. Tasklets resemble kernel times in some ways. They are always run at interrupt time and they always run on the same CPU that schedules them, and they receive an unsigned long argument. Unlike kernel timers. However you can't ask to execute the function at a specific timer. By scheduling a tasklet, you simply ask for it to be executed at a later time chosen by the kernel. This behaviour is especially useful with interrupt handlers, where the hardware interrupt must be managed as quickly as possible, but most of the data management can be safely delayed to a later time.

Tasklets(2)

- A tasklet just like a kernel timer is executed in the context of a “**soft interrupt**”, a kernel mechanism that executes asynchronous tasks with hardware interrupts enabled.
- A tasklet exists as a data structure that has to be initialized before use. Initialization can be performed by calling a specific function or by declaring the structure using certain macro.

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

```
struct tasklet_struct{
```

```
/*...*/
```

```
void (*func)(unsigned long);
```

```
unsigned long data;
```

```
}
```

```
void tasklet_init(struct tasklet_struct t, void (func)(unsigned  
long), unsigned long data);
```

```
DECLARE_TASKLET(name, func, data);
```

```
DECLARE_TASKLET_DISABLED(name, func, data);
```

Workqueues(1)

- **Workqueues** are similar to **tasklets**; they allow kernel code to request that a function be called at some future time. There are however, some significant differences.
- **Tasklets** run in software interrupt context with the result that all tasklet code must be atomic. Instead, workqueue functions run in the context of a special kernel process; as a result they have more flexibility. In particular, **workqueue** functions can sleep.
- **Tasklets** always run on the processor from which they were originally submitted. **Workqueues** work in the same way, by default.
- Kernel code can request that the execution of work queue functions can be delayed for an explicit interval.
- The key difference between the two is that tasklets execute quickly, for a short period of time, and in atomic mode, while workqueue functions may have higher latency but need not be atomic.

Workqueues(2)

- Workqueues has a type of struct workqueue_struct, which is defined in <linux/workqueue.h>. A workqueue must be explicitly created before use, using one of the following functions.

```
struct workqueue_struct *create_workqueue(const char *name);  
struct workqueue_struct *create_singlethread_workqueue(const  
char *name)
```

- Each workqueue has one or more dedicated processes (“kernel thread”), which run functions submitted to the queue.
- If you use *create_workqueue* you get a workqueue that has a dedicated thread for each processor on the system. In many cases all those threads are simply overkill; if a single worker thread will suffice, create the *workqueue* with *create_singlethread_workqueue* instead.
- To submit a task to a work queue, you need to fill in a *work_struct* structure. This can be done at compile time as follows.

```
DECLARE_WORK(name, void (*function)(void*),void *data)
```

Workqueues(3)

- Where name is the name of the structure to be declared, function is the function that is to be called from the workqueue, and data is a value to pass to that function. If you need to set up the `work_struct` structure at runtime, us the following two macros:

INIT_WORK(struct work_struct *work, void (*function)(void*), void *data);

PREPARE_WORK(struct work_struct *work, void (*function)(void*), void *data);

- ***INIT_WORK*** does more thorough job of initializing the structure, you should use it the first time that structure is set up.
PREPARE_WORK does almost the same job, but does not initialize the pointer used to link the `work_struct` structure into the work queue.

Workqueues(4)

- There are two functions for submitting work to a workqueue:
*int queue_work(struct workqueue_struct *queue, struct work_struct *work);*
*int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);*
- Either one adds work to the given queue. If queue_delayed_work is used, however, the actual work is not performed until at least delay jiffies have passed. The return value is 0 if the work was successfully added to the queue; a nonzero result means that this work_struct structure was already waiting in the queue, and was not added a second time. At some time in the future, the work function will be called with the given data value. The function will be running in the context of the worker thread, so it can sleep if need be—you should be aware of how that sleep might affect any other tasks submitted to the same workqueue.

Workqueues(5)

- If you want to cancel a pending workqueue entry , you may call
 - *int cancel_delayed_work(struct work_struct *work);*The return value is nonzero if the entry was canceled before it began execution. If cancel_delayed_work returns 0, however , the entry may have already been running on a different processor, and might still be running after a call to this function.
- To be absolute sure that the work funciton is not running anywhere in the system after cancel_delayed_work returns 0, you must follwo that call with a call to
 - void flush_workqueue(struct workqueue_struct *queue)*
- When you are done with a workqueue, you can get rid of it with:
 - void destroy_workqueue(struct workqueue_struct *queue)*

Interrupt Handling

Interrupts & Exceptions

Interrupts & Exceptions(1)

- An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events corresponds to electrical signals generated by hardware circuits which are inside and outside the CPU.
- **Interrupts** are often divided into **synchronous** and **asynchronous** interrupts.
- **Synchronous interrupts(Exceptions)** are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction. Exceptions on the other hand are caused by **programming errors** or by **anomalous conditions** that must be handled by the kernel. In the first case , the kernel handles the exception by delivering to the current process one of the **signals** familiar to **Unix** programmer. In the second case the kernel performs all the steps needed to recover from anomalous condition, such as a page fault or a request(via an int instruction) for a kernel service.

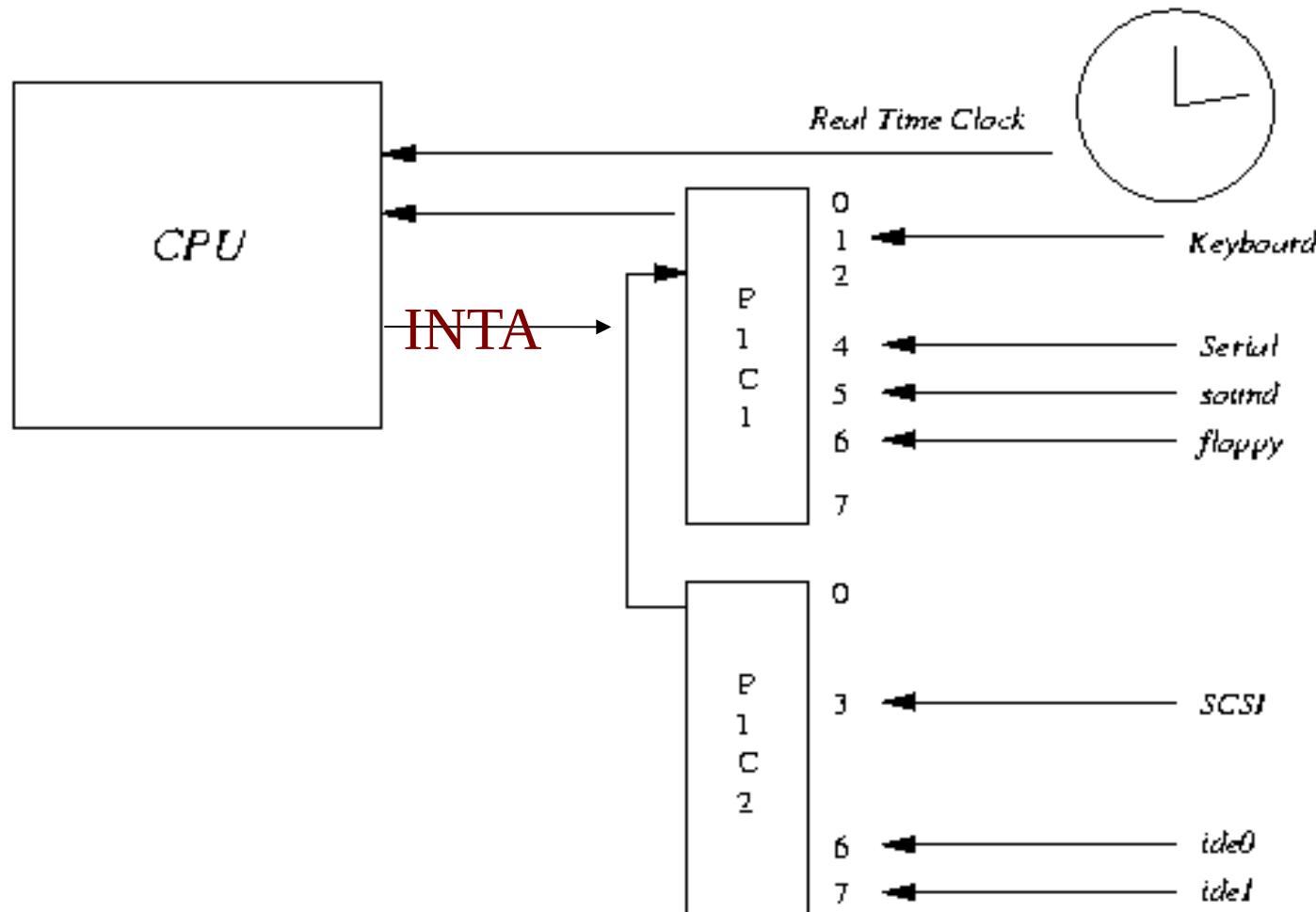
Interrupts & Exceptions(2)

- **Asynchronous interrupts**(**Interrupts**) are generated by other hardware devices at arbitrary times with respect to the CPU clock signals. Interrupts are issued by either ***interval timer*** or ***I/O devices***. For instance, the arrival of a keystroke from a user sets off an interrupt.
- Interrupt is an efficient mechanism for an I/O device to communicate with the CPU. Whenever any I/O device is ready for I/O operation, it interrupts the CPU. If interrupt mechanism does not exist, then CPU need to do more work by keep polling the I/O devices periodically.

Polling & Interrupts

- The primary purpose of a device driver is to read and write data to a device. In the last session we studies that the data transfers will happen by reading and writing in to the device's I/O registers. The read and write driver functions can be implemented in two possible modes. These are
 - **Polling mode.**
 - **Interrupt mode.**
- In ***polling mode*** the read function continuously read and checks the status register whether data is available or not. If status register indicates that the data is available it reads the data from the data register. This polling mode read wastes all the CPU time and never used in device drivers.
- In the **interrupt mode** the **I/O controller** itself activates its interrupt line, whenever it is ready with the data. This interrupt line from I/O controller is connected to the **Interrupt Controller(8295)**.

Interrupts Diagrams



Interrupts & Interrupt Handler(1)

- Interrupt controller is capable of receiving interrupts from various I/O Controller's. Interrupt controller in turn interrupts the CPU by activating CPU's interrupt line. CPU acknowledges the interrupt by activating INTA line. When CPU acknowledges the interrupt the interrupt controller will send a vector number as index into the interrupt vector table and takes interrupt handler address and jumps to it.
- The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board. Most hardware devices won't generate other interrupts until their "**interrupt-pending**" bit is cleared. Some devices don't have this pending bit.
- A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for such as arrival of new data.

Interrupts & Interrupt Handler(2)

- ***Reading data from device:*** The interrupt handler reads the data from the device and puts into the receive buffer maintained by the driver. Whenever application calls the driver's read function, read function supplies the data present in the read buffer.
- The programmer should be careful to write a routine that executes in a minimum time, independent of its being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a ***tasklet*** or ***task queue*** to schedule computation at a safer-time.

Installing an Interrupt handler(1)

- If you want to actually see interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the linux kernel hasn't been told to except your interrupt, it simply acknowledge and ignores it.
- Interrupt lines are a precious and often limited resources, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similarly to the registry of I/O ports. 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 expected to be able to share interrupt lines with other drivers. The following functions implement the interrupt registration functions. **#include <linux/interrupt.h>**

```
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);
```

Installing an Interrupt handler(2)

- The first parameter ***irq***, specifies the interrupt number or line to allocate. For some devices this interrupt number is fixed(for example in PC system timer's irq is 0 and keyboard's irq is 1 whereas for other devices such as PCI, irq need to be found dynamically).
- The second parameter, ***handler*** is a pointer to the actual interrupt handler that services the interrupt.
- The third parameter, ***irqflags***, may be either zero or bit mask of one or more of the following flags.
 - **SA_INTERRUPT**: This flags specifies that the given interrupt handler is a **fast interrupt handler**. For fast interrupt handlers, while it is running all other interrupts will be disabled. This enables a fast interrupt handler to complete quickly, without possible interrupts from other interrupts.
 - **SA_SHIRQ**: This flags specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag.

Installing an Interrupt handler(3)

- The fourth parameter, ***devname*** is the name of the device or driver requesting this interrupt.
- The fifth parameter, ***dev_id*** is used primarily for shared interrupt lines. Its a unique identifier used when an 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).
- The following kernel function is used to free the interrupt handler.

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

- 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 init function might sound like a good idea, the correct place to call ***request_irq*** is when the device is first opened, before the hardware is instructed to generate interrupts. The place to call ***free_irq*** is the last time the device is closed, after the hardware is told not to interrupt the process any more.

Shared interrupt handler

- It is possible to share the single interrupt line by multiple I/O devices. This is due to the shortage of total interrupt lines available. To write interrupt handler for a shared device we should take care of the following.
 - The ***SA_SHIRQ*** flag must be set in the ***flags*** arguments to ***request_irq()***.
 - The ***dev_id*** argument must be unique to each registered handler. A pointer to any per device structure is sufficient. You cannot pass NULL for a shared handler.
 - The interrupt handler must be capable of distinguishing whether is device actually generated an interrupt.

Interrupt Context(1)

- When executing an ***interrupt handler*** or ***bottom half***, the kernel is in interrupt context, which is different from process context. Kernel typically executes in process context when user process issues a system call or while running a kernel thread. A process running in kernel can issue a blocking/sleeping call so that the calling process will block.
- ***Interrupt context*** on the other hand is not associated with process. The current macro is not relevant. Unlike process, interrupt context cannot sleep. Therefore we cannot call certain functions. If a function sleeps we cannot use it in ***interrupt handlers***.
- Interrupt context is time critical because the interrupt handler interrupted other code(possibly another interrupt handler or a different line). Code should be quick and simple. Because of this asynchronous nature, all interrupt handlers are as quick and as simple as possible.

Interrupt Context(2)

- As much as possible, work should be pushed out from the interrupt handler and performed in ***bottom half***, which runs at a more convenient time.
- Finally, the ***interrupt handler*** does not receive its own stack, instead, it shares the kernel stack of the process it interrupted. If no process is running it uses the ***idle task's(swapper pid 0)*** stack. Because interrupt handlers share the stack they should be careful in use.
- **ProcFs interface:**
 - Interrupt statistics can be viewed through the proc file system interface. The following commands displays the interrupt statistics.

\$cat /proc/interrupts

0: 127 0 IO-APIC-edge timer

- The first column displays the interrupt line numbers. Second column shows the count of interrupts received on this line. The last column shows the name of the device.

Interrupts control & Status

- The linux kernel provides a set of functions for manipulating the state of interrupts on a machine. These API's enables us to disable the interrupt system from the current processor or mask out an interrupt line for the entire machine. These routines are all very architecture dependent and can be found in `<asm/system.h>` and `<asm/irq.h>`. Following is the complete list of those API's.

`local_irq_disable()` //Disables all interrupts on the local CPU

`local_irq_enable()` //Enables all interrupts on the local CPU

`local_irq_save(unsigned long flags)` //Saving interrupt state

`local_irq_restore(unsigned long flags)` //Restoring interrupt state

`disable_irq(unsigned int irq)`

`disable_irq_nosync(unsigned int irq)`

`enable_irq(unsigned int irq)`

`irqs_disable()`

`in_interrupt()`

`in_irq()`

Bottom Halves - Introduction

- Because of certain limitations the interrupt handlers perform only first(or top) half of the interrupt processing. Main limitation of the interrupt processing is that, it has to run and finish processing faster. Remaining, less critical half of the processing is deferred to later point when interrupts are enabled.
- Consequently managing interrupts is divided into ***two parts***, or halves, The first part(***top halves***), are executed by the kernel asynchronously in immediate response to a hardware interrupt as discussed in the last section. In this section we will study second part of the processing interrupt, ***bottom halves***.
- The job of bottom halves is to perform any interrupt related work not performed by the interrupt handler itself. In an ideal world, this is nearly all work(and in turn be as fast) as possible. We want interrupt handlers to return as quickly as possible.

World of Bottom Halves(1)

- Unlike the top half which we implement entirely in the interrupt handler, multiple mechanisms are available for implementing a bottom half. These are
 - Original Bottom Half
 - Task Queues
 - Softirqs
 - Tasklets
 - Work Queues
- **Bottom halves** are original mechanism for deferring the interrupt processing to a later point of time. But main limitation with these bottom halves is that, they are globally synchronized, no two could run at the same time even on two different processors. These are simple to use but a bottleneck.
- Later on, the kernel developers introduced task queues both as a method of deferring work and as a replacement for the BH mechanism. Even though this task queues worked fairly well, it was

World of Bottom Halves(2)

still too inflexible to replace entirely the BH interface. It also was not lightweight enough for performance critical subsystems, such as networking.

- During 2.3 development series, the kernel developers introduced **softirqs** and **tasklets**. Softirqs are a set of 32 statically defined bottom halves that can run simultaneously on any processor, even two of the same type can run concurrently. Tasklets are flexible, dynamically created bottom halves that are built on top of softirqs. Two different tasklets can run concurrently on different processor, but two of the same type of tasklets cannot run simultaneously. Thus tasklets, are a good trade-off between performance and ease of use. For most bottom half processing tasklet is sufficient. Softirqs are useful when performance is critical, such as with networking. Using softirqs requires more care, however, because two of the same softirq can run at the same time. In addition softirqs must be registered statically at compile time. Tasklets can be registered dynamically.

World of Bottom Halves(3)

- While developing 2.5 kernel, the original BH interface was finally removed all existing BHs are converted to the other bottom half interfaces. Additionally task queue interface was replaced by the work interface. Work queues are a very simple yet useful method of queueing work to later be performed in process context.
- Consequently, today in 2.5 we have three bottom half mechanisms in the kernel.
 - Softirqs
 - Tasklets
 - Work queues

Softirqs

- Softirqs are rarely used, tasklets are a much more common form of bottom halves. But important point is that tasklets are built on top of softirqs. The softirq code lives in ***kernel/softirq.c***
- Softirqs are statically allocated at compile time. Unlike tasklets, we cannot dynamically register or destroy softirqs. Softirqs are represented by the ***softirq_action*** structure which is defined in `<linux/interrupt.h>`. An array of 32 of these structures are defined in `softirq.c` file. So maximum possible softirqs are 32. However in the current kernel, only six of 32 entry's are used. Following are these softirqs.

Softirq	Priority	Description
HI_SOFTIRQ	0	High priority tasklets
TIMER_SOFTIRQ	1	Time Bottom Half
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
SCSI_SOFTIRQ	4	SCSI-bottom half
TASK_SOFTIRQ	5	Tasklets

Tasklets

- **Tasklets** are bottom half mechanism built on top of softirqs. Tasklets are similar in nature and work in similar manner to softirqs; however they have simpler interface and relaxed locking rules.

```
DECLARE_TASKLET(name, func, data);
```

```
Struct tasklet_struct my_tasklet={NULL, 0, ATOMIC_INIT(0),  
task_handler, dev};
```

```
Struct tasklet_struct t;
```

```
tasklist_init(t, tasklet_handler, dev);
```

```
tasklet_schedule(struct tasklet_struct *);
```

```
tasklet_disable(struct tasklet_struct *);
```

```
tasklet_disable_nosync(struct tasklet_struct *)
```

```
tasklet_enable(struct tasklet_struct *)
```

Work Queues

- **Work queues** are a different form of deferring work from what we have looked so far. Work queues defer work into a kernel thread, the work always runs in process context. Thus code deferred to a work queue has all the usual benefits of process context. Most importantly, work queues are schedulable and can therefore sleep.

```
DECLARE_WORK(name, void (*function)(void*),void *data)
INIT_WORK(struct work_struct *work, void (*function)(void*),
void *data);
schedule_wrk(struct work_struct *)
schedule_delayed_work(struct work_struct *, unsigned long delay);
Void flush_schedule_work(void);
```

Kernel Synchronization

Critical Sections and Race conditions

- Code path that accesses and manipulate shared data are called ***critical regions***. This is usually unsafe for multiple threads of execution to access the same resource. Simultaneously, to prevent concurrent access during critical regions the programmers must ensure that the code executes ***atomically***, this is the code completes without interruptions as if the entire critical region is one indivisible instruction.
- It is a bug if it is possible for two threads of execution to be simultaneously in the same critical region. When this occurs we call it ***race condition***. Debugging race conditions is often very hard because they are not easily reproducible. Ensuring that concurrency (multiple path execution) is prevented and that race condition does not occur is called ***synchronization***.

Causes of concurrency(1)

- Concurrent access to shared resources needs to be synchronized to avoid race conditions. Before discussing various methods to achieve synchronization, let us discuss how concurrent paths are possible inside the kernel. First of all kernel is big common resource for all the user processes. All user processes execute kernel code concurrently. Following all the concurrent situations.
- **Interrupts:** An interrupt can occur asynchronously at almost any time, interrupting the currently executing task.
- **Kernel Pre-emption:** Because the kernel is pre emptive, one task in the kernel can pre-empt another.
- **Sleeping and synchronization with user space:** A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process. This sleep could occur while allocating memory (kmalloc) or accessing user space memory(that is paged out).
- **Symmetric multiprocessing:** Two or more processors can be executing code at the exact time.

Causes of concurrency(2)

- Pseudo concurrency and True concurrency:
 - **True concurrency** occurs when two or more processors try to execute same resources.
 - But concurrency is possible even between single processor. This happens when interrupt occurs, or current task is pre-empted by other task and interrupt or other task executes the same code. This is called pseudo concurrency.

Atomic integer operations(1)

- Atomic operations provide instruction that execute atomically, i.e without interruption. The kernel provides two sets of interfaces for atomic operations, one that operates on integers and other that operates on individual bits.
- The atomic integer methods operate on a special data type, `atomic_t`.

<asm/atomic.h>

```
ATOMIC_INIT(int i);
void atomic_set(atomic_t *v, int i)
int atomic_read(int i, atomic_t *v)
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
int atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
```

Atomic integer operations(2)

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_add_and_test(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

Following is the sample usage of atomic operations.

```
Atomic_t u;
atomic_t v=ATOMIC_INIT(0);
atomic_set(&v, 4);
atomic_add(2,&v);
atomic_inc(&v);
```

Atomic Bit-wise operations(1)

Following are the atomic bitwise operations. Because these functions operate on generic pointer, there is no equivalent of the atomic integers **atomic_t** type. Instead we can work with a pointer to whatever data we want.

<asm/bitpos.h>

*void set_bit(int nr, void *addr)*

*void clear_bit(int nr, void *addr)*

*void change_bit(int nr, void *addr)*

*Int test_and_set_bit(int nr, void *addr)*

*Int test_and_clear_bit(int nr, void *addr)*

*Int test_and_change_bit(int nr, void *addr)*

*Int test_bit(int nr, void *addr)*

*int find_first_bit(unsigned long *addr, unsigned int size)*

*int find_first_zero_bit(unsigned long *addr, unsigned int size)*

Atomic Bit-wise operations(2)

Following is the sample usage of atomic operations

Unsigned long word=0

set_bit(0,&word);

set_bit(1,&word);

*printf("%ul",word); /*should print 3*/*

Spin Locks(1)

- The atomic operations described above are useful only to modify simple variables. But most of the time we need to modify big data structures atomically. Then we need locks. The most common lock in the linux kernel is the spin lock. A spin lock is a lock that can be held at most by one thread of execution. If a thread of execution attempts to acquire a spin lock while it is contended(already held), the thread will busy loop(spinning) waiting for the lock to become available. If the lock is not contended, the thread can immediately acquire the lock and continue. The spinning prevents more than one thread of execution from entering the critical region at any one time.
- The fact that a contended spin lock causes threads to spin(essentially wasting processor time) while waiting for the lock to become available is important. It is no wise to hold spin lock for a long time.

Spin Locks(2)

<asm/spinlock.h>

spin_lock();

spin_lock_irq();

spin_lock_irqsave();

spin_unlock();

spin_unlock_irq();

spin_unlock_irqrestore();

spin_lock_init()

spin_try_lock()

spin_is_locked()

Spin Locks(3)

Following is sample usage:

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;  
spin_lock(&mr_lock);  
/**critical region**/  
spin_unlock(&mr_lock);
```

- When spin lock are used from interrupt handler also, we need to disable interrupts whenever spin lock is acquired. The kernel provides an interface an interface that conveniently disables the interrupts and acquires the lock. Following is sample usage.

```
spinlock_t mr_lock=SPIN_LOCK_UNLOCKED  
unsigned long flags;
```

```
spin_lock_irqsave(&mr_lock, flags);  
/*critical region*/  
spin_unlock_irqrestore(&mr_lock);
```

Reader – Writer Spin Locks(1)

- Concurrent access to a shared resource is not a problem if all concurrent accesses are read-only. That is no one is modifying it. So it is OK to allow concurrent read accesses to a shared resource. But if a write access is going on no other readers as well as writers should not be allowed. There may be instances, where some paths are doing only read operations other code paths are doing write operations. For such situations, the kernel provides Reader-Writer spin lock facility.
- The reader-writer spin lock can be acquired by multiple readers as long as no other writer is acquired it. If writer acquires the lock, then readers or writers should wait till writer releases the lock. Following are the kernel methods to access reader-writer spin lock.

Reader – Writer Spin Locks(2)

```
read_lock();  
read_lock_irq();  
read_lock_irqsave();  
read_unlock();  
read_unlock_irq();  
read_unlock_irqsave();  
write_lock();  
write_lock_irq();  
write_lock_irqsave();  
write_unlock();  
write_unlock_irq();  
write_unlock_irqrestore();  
write_trylock();  
rw_lock_init();  
rw_is_locked();
```

Reader – Writer Spin Locks(3)

- Following is the sample usage:

rwlock_t mr_rwlock = RW_LOCK_UNLOCKED

- Following is reader code path.

```
read_lock(&mr_rwlock);  
/** Critical region for read only**/  
read_unlock(&mr_rwlock);
```

- Following is writer code path

```
write_lock(&mr_rwlock);  
/** critical region for read only**/  
write_unlock(&mr_rwlock);
```

Semaphore(1)

- Semaphores in Linux are sleeping locks. When a task attempts to acquire a semaphore that is already held, the semaphore places the task onto a wait queue and puts the task to sleep. The processor is then free to execute other code. When the process holding the semaphore releases the lock, one of the tasks on the wait queue will be awakened up so that it can acquire the semaphore.
- We can draw some interesting conclusions from the sleeping behaviour of semaphore: Because the contending tasks sleep while waiting for the lock to become available, semaphores are well suited to locks that can be held a long time.
- Conversely semaphores are not optimal for locks that are held for very short periods because the overhead of sleeping can outweigh the total lock hold time. Because a thread of execution sleeps on lock contention, semaphores can only be obtained in process context, as interrupt context is not schedulable.

Semaphore(2)

<asm/semaphore.h>

DECLARE_SEMAPHORE_GENERIC(var_name, count)
DECLARE_MUTEX(var_name);
*sema_init(struct semaphore *, int);*
*init_MUTEX(struct semaphore *);*
*init_MUTEX_LOCKED(struct semaphore *);*
*down_interruptible(struct semaphore *);*
*down(struct semaphore *);*
*down_trylock(struct semaphore *);*
*up(struct semaphore *);*

- Following is sample usage:

```
static DECLARE_MUTEX(mr_sem);
if(down_interruptible(&mr_sem);
    /** Signal received semaphore not acquired **/
    /** critical region **/
    up(&mr_sem);
```

Reader-Writer Semaphores

- Semaphores, like spin locks, also come in a reader-writer flavour. The situation where reader-writer semaphores are preferred over standard. Semaphores are the same as with reader-writer spin locks versus standard spin locks.
- Reader-writer semaphores are represented by the struct `rw_semaphore` type. Which is defined in `<linux/rwsem.h>`

`<linux/rwsem.h>`

`DECLARE_RWSEM(var_name);`

`init_rwsem(strut rw_semaphore *sem);`

`down_read(struct rw_semaphore *sem);`

`down_read_trylock(struct rw_semaphore *sem);`

`up_read(struct rw_semaphore *sem);`

`down_write(struct rw_semaphore *sem);`

`down_write_trylock();`

`up_write(struct rw_semaphore *sem);`

Completion Variables(1)

- Completion variables are an easy way to synchronize between two tasks in the kernel, when one task needs to signal to the other that an event has occurred. One task waits on the completion variable, while another task performs some work. When the other task has completed the work, it uses the completion variable to wakeup any waiting task. These completion variables are somewhat similar to semaphores but much simpler to use.
- Completion variables are represented by the ***struct completion*** type, which is defined in ***<linux/completion.h>*** A statically created completion variable is created and initialized via
DECLARE_COMPLETION(mr_comp);
- A dynamically created completion variables is initialized via
init_completion();

Completion Variables(2)

- On a given completion variables, the task that want to wait call `wait_for_completion()`. After the event has occurred, calling `complete()` signals all waiting tasks to wakeup. Following is the list of kernel methods.

<linux/completion.h>

*DECLARE_COMPLETION(var_name)
init_completion(struct completion *)
wait_for_completion(struct completion *);
complete(struct completion *)*

Seq Locks(1)

- **Seqlocks** are a new type of lock introduced in 2.6 kernel. They provide a very simple mechanism for reading and writing shared data. They work by maintaining a sequence counter. Whenever data in question is written to, a lock is obtained and sequence number is incremented. Prior to and after reading data, the sequence number is read. If the values are the same, then a write did not begin in the middle of the read. Further if values are even, then a write is not under way(grabbing the write lock makes the value odd, while releasing makes it even ,since the lock starts at zero);
- Following is a sample usage:

seqlock_t mr_seq_lock = SEQLOCK_UNLOCKED;

- The write path is then

```
write_seqlock(&mr_seq_lock);  
/*Write lock is obtained*/  
write_sequnlock(&mr_seq_lock);
```

Seq Locks(2)

- This may looks like normal spin lock code. The oddness comes with the read path, which is quite bit different:

```
unsigned long seq;
do
{
    seq=read_seqbegin(&mr_seq_lock);
    /** read data here ....**/
} while(rad_seqretry(&mr_seq_lock);
```

- **Seqlocks** are useful to provide a very lightweight and scalable look for use with many readers and few writers.

Pre-emption Disabling

- It is possible to disable and enable the kernel pre-emption to achieve synchronization. Critical section during which we do not want to get pre-empted by kernel, we can disable the kernel pre-emption by using the following API's.

Preempt_disable();

Preempt_enable();

preempt_enable_no_resched();

preempt_count();

ACCESSING HARDWARE

Accessing I/O Ports(1)

- Device Driver is a set of function that talks to the I/O device or more precisely I/O controller. Each I/O Controller consists of a set of register called I/O ports. The driver function talk to the I/O Controller by reading and writing into these **I/O ports**. These I/O ports could be mapped to a memory address space or a separate I/O address space. Only few microprocessor like Intel x86 supports separate **I/O address space**. In all other microprocessors I/O ports are mapped to the memory address space. Then this memory is referred as **I/O memory**.
- Before accessing the I/O ports, the driver should check whether that I/O ports are available or already some other driver is using those I/O ports. For this purpose driver can use **check_region()** function. Once these I/O ports are not used by any other module, driver can request for these I/O ports by using **request_region()**. kernel maintains the list of I/O port ranges their driver names. we can see that list through **proc** file system by using the following command:

```
$cat /proc/ioports
```

Accessing I/O Ports(2)

- Finally a driver can release the I/O region by using **release_region()** function while it is exiting.

```
#include <linux/ioport.h>
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start, unsigned long
                                len,
char *name);
void release_region(unsigned long start, unsigned long len);
```

- Device driver access the I/O ports that mapped on to I/O addresses by using the following functions. The **inb()** is for reading the 8 bit I/O ports and **outb()** is for writing into 8 bit I/O ports. The **inw()** and **outw()** are for 16 bit; **inl()** and **outl()** are for 32 bit ports;

```
#include <asm/io.h>
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 long word, unsigned port);
```

Accessing I/O Ports(3)

- The following functions are for reading and writing the I/O port multiple times. That is to perform string I/O operations. These functions are optimized by using processor specific assembly instructions.

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned oprt, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

Accessing I/O Memory(1)

- When I/O ports are mapped to memory space, we call that memory as I/O memory. A driver should use separate request and release functions to use the I/O memory. These functions are given below. Similar to I/O ports that are being used, we can also view the I/O memory blocks currently being used by giving the following command.

```
$cat /proc/iomem  
#include <linux/ioport.h>  
int check_mem_region(unsigned long start, unsigned long len);  
int request_mem_region(unsigned long start, unsigned long len,  
    char *name);  
int release_mem_region(unsigned long start, unsigned long len);
```

- The driver can access the I/O ports that mapped to I/O memory by de-referencing a pointer variable. However for portability reasons one should use only the following function to access the I/O memory locations.

```
#include <asm/Io.h>  
unsigned redb(address);  
unsigned readw(address);  
unsigned real(address);
```

Accessing I/O Memory(2)

```
void writeb(unsigned value, address);  
void writew(unsigned value, address);  
void writel(unsigned value, address);
```

- Kernel always uses virtual addresses only. By default kernel will map only real physical memory (RAM) into kernel virtual address space. The physical I/O memory addresses are not mapped into kernel virtual address space. So driver may need to do this mapping. The following **ioremap()** function will map the given physical address range into virtual kernel address. The return value is the virtual address for the I/O memory physical address. Kernel should use only this logical address kernel will adjust its page tables to access the new physical I/O memory. The **iounmap()** unmaps the already mapped memory region.

```
void *ioremap(unsigned long phys_addrs, unsigned long size);  
void iounmap(void *addr);
```

Peripheral Component Interconnect (PCI)

PCI Bus & Interface(1)

- A bus is made of both electrical interface and a programming interface. Here we discuss only programming interfaces or kernel functions which accesses Peripheral Component Interconnect(PCI) peripherals, because these days the PCI bus is the most commonly used peripheral bus on desktops and bigger computers. This bus is the one that is best supported by the kernel.
- This discussion is mainly concerned with how a PCI driver can find its hardware and gain access to it. The probing technique discussed in the section **“Module Parameters”** and **“Auto-detecting the IRQ Number”** in Interrupt Handling can be used with PCI drivers. The specification offers an alternative that is preferable to probing.
- The PCI architecture is an alternative to ISA standard with three main goals:
 - 1) Better performance when transferring data between processor and peripherals. 2) Platform independent 3) Simplify adding and removing peripherals to the system.

PCI Bus & Interface(2)

- The PCI bus achieves **better performances** by using a higher clock rate than ISA. Its clock runs at 25 or 33MHz and recently 66MHz and 133MHz implementation have been deployed. Its equipped with 32 bit data bus and 64 bit extension is also provided with specification.
- **Platform independence** is often a goal in the design of a computer bus, and its an especially important feature of PCI, because the PC world is dominated by processor specific interface standards and is used in Alpha, PowePC, SPARC6 and IA-64 systems.
- PCI supports for **auto-detection** on interface boards. PCI devices are jumper-less and are automatically configured at boot time. Then the device drivers must be able to access configuration information in the device in order to complete initialization. This happens without the need to perform any probing.

PCI Addressing(1)

- Each **PCI peripheral** is identified by a **bus number**, a **device number**, and a **function number**. The PCI specification permits a single system to host up to 256 buses, but these buses are not sufficient for many larger systems, Linux now supports **PCI domains**. Each **PCI domain** can host up to **256 buses**. Each bus host up to **32 devices**, and each device can be a **multifunction** board(such as an audio device with an accompanying CD-ROM driver) with a maximum of **8 functions**.
- Each **function** can be identified at hardware level by a 16-bit address or a key. Device drivers written for Linux, though need to deal with those binary addresses, because they use a specific data structure, called **pci_dev** to act on devices.
- Workstation feature at least two PCI buses. Plugging more than one bus is accomplished by means of **bridges**, a special purpose PCI peripheral whose task is joining two buses. The overall layout is a tree where each bus is connected to upper layer bus, upto bus 0 at root

PCI Addressing(2)

- The 16 bit hardware address associated with PCI peripherals, although hidden in **struct pci_dev** object is still visible occasionally when list of devices are displayed using **lspci**(part of pciutils package available with most distributions) & layout of information in **/proc/pci** and **/proc/bus/pci**.
- The **sysfs** representation of PCI devices also shows this addressing scheme, with the addition of the PCI domain information. When the hardware address is shown it can be shown as two values(an 8-bit bus number and an 8-bit device and function number), as three values(bus, device and function), or as four(domain, bus,device and function); all the values are usually displayed in hexadecimal.
- **/proc/bus/pci/devices** uses 16 bit field. Use below commands
 - **\$lspci | cut -d: -f1-3(2,1)**
 - **Cat /proc/bus/pci/devices | cut f1**
 - **\$tree /sys/bus/pci/devices (four values in sysfs)**

PCI Addressing(3)

- *lspci* uses info from /proc file system.
- VGA video controller as an example , 0x00a0 means 0000:00:14:0 when split into **domain(16bits),bus(8bits), device(5bits) and function(3bits)**.
- The hardware circuitry of each peripheral board answers queries pertaining to three address spaces: **memory locations, I/O ports, and configuration registers**.
- As far as driver is concerned **memory and IO regions** are accessed in the usual ways like **inb** and **readb**. Configuration transactions on the other hand are performed by calling special **kernel functions** to access **configuration registers**.
- Every **PCI slot** has **four interrupt pins**,and each **device function** can use one of them without being concerned about how these pins are routed to CPU. PCI specification requires interrupt lines as shareable, even a processor with a limited number of IRQ lines and can host many PCI interface boards.

PCI Addressing(4)

- The IO space in a PCI bus uses a 32 bit address bus while the memory space can be accessed with either 32-bit or 64-bit addresses. 64Bit addresses are available on more recent platforms.
- Firmware initializes PCI hardware at system boot, mapping each region to a different address to avoid collisions. The addresses to which these regions are currently mapped can be read from the **configuration space** so Linux driver can access its devices without probing. After reading the configuration registers, the driver can safely access its hardware.
- The PCI configuration space consists of 256 bytes for each device function and PCI express device have 4KB of configuration space for each function, and the layout of configuration registers is standardized. Four bytes of the configuration space holds a unique function ID, so the driver can identify its device by looking for the specific ID for that peripheral. Difference between **ISA** and **PCI** is the addition of **configuration space** which avoids **probing**.

Boot Time(1)

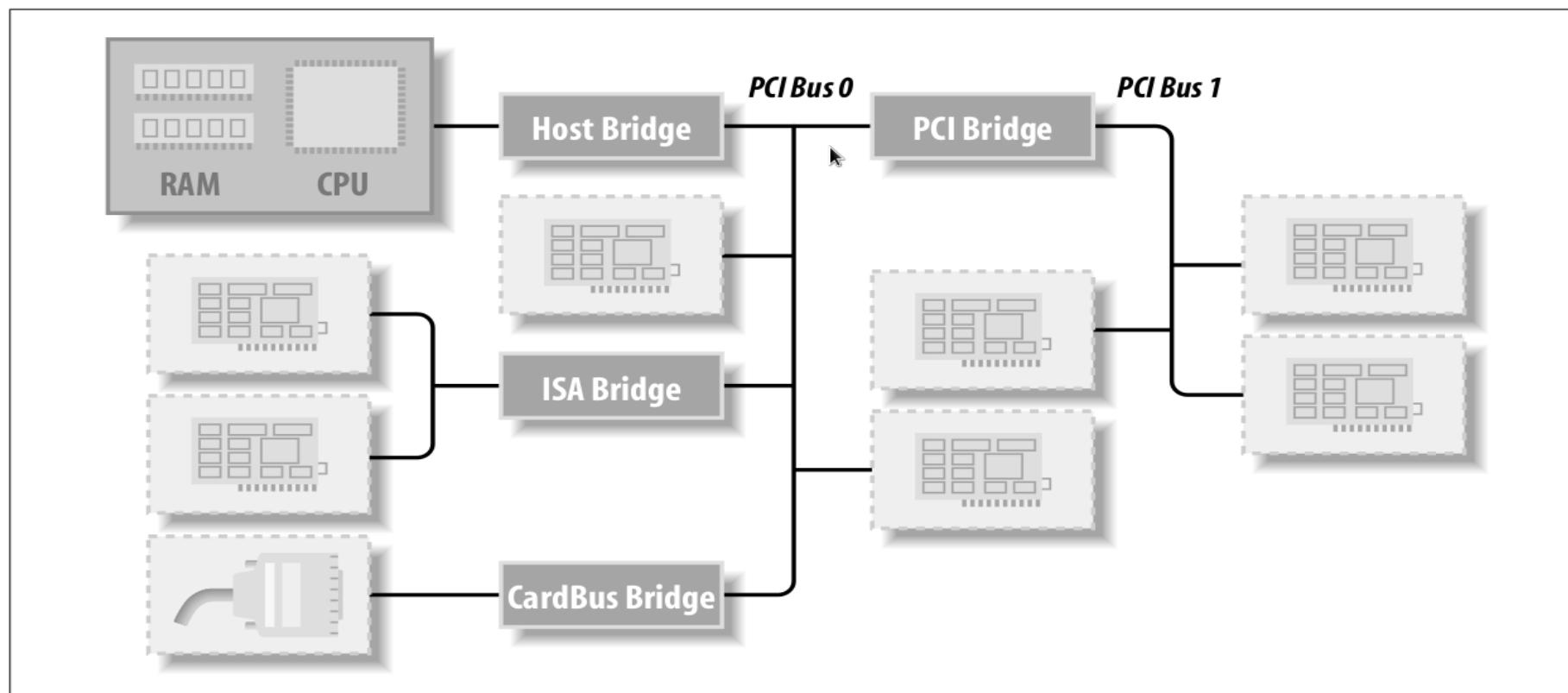
- When power is applied to PCI device, the hardware remains inactive. The device responds only to configuration requests. At power on, the device has no memory and no I/O ports mapped in the computers address space. Interrupt handling is even disabled.
- Every PCI motherboard is equipped with PCI-aware firmware called BIOS, NVRAM and PROM depending on the platform. The firmware offers access to device configuration address space by reading and writing to registers.
- At system boot the firmware(or the linux kernel) performs configuration transactions with every PCI peripheral in order to allocate a safe place for each region it offers. By the time driver accesses the device, its memory and IO regions have already been mapped into processor's address space. The driver can change this default assignment, but it never has to do that.

Boot Time(2)

- **/proc/bus/pci/devices** is a text file with device information.
- **/proc/bus/pci/*/*** is a binary files that report a the configuration of each device, one file per device.
- Individual PCI device directories in the sysfs tree can be found in **/sys/pci/pci/devices**. The files **vendor**, **device**, **subsystem_device**, **subsystem_vendor**, and class all refer to specific values of pci device. **IRQ** shows the current irq assigned to the PCI device, and file **resource** shows the current memory resources allocated by the device.

PCI Interface(1)

- Peripheral Component Interconnect (PCI), as its name implies is a standard which describes how to connect the peripheral components of a system together in a structured and controlled way. The standard describes the way that the system components are electrically connected and the way they should behave. Here we will look into how Linux kernel initializes the system's PCI buses and devices.



PCI Interface(2)

- The PCI buses and PCI-PCI Bridges are the glue connecting the system components together. The CPU is connected to PCI bus 0, the primary PCI bus as is the video device. A special PCI device, a PCI-PCI bridge connects the primary bus to the secondary PCI bus, PCI bus 1. In the jargon of the PCI specification, PCI bus 1 is described as being downstream of the PCI-PCI bridge and PCI bus is up-stream of the bridge. Connected to the secondary PCI bus are the SCSI and ether devices for the system. Physically the bridge, secondary PCI bus and two devices would all be contained on the same combination PCI card. The PCI-ISA bridge in the system supports older, legacy ISA devices and the diagram shows a super I/O controller chip which controls the keyboard, mouse and floppy.

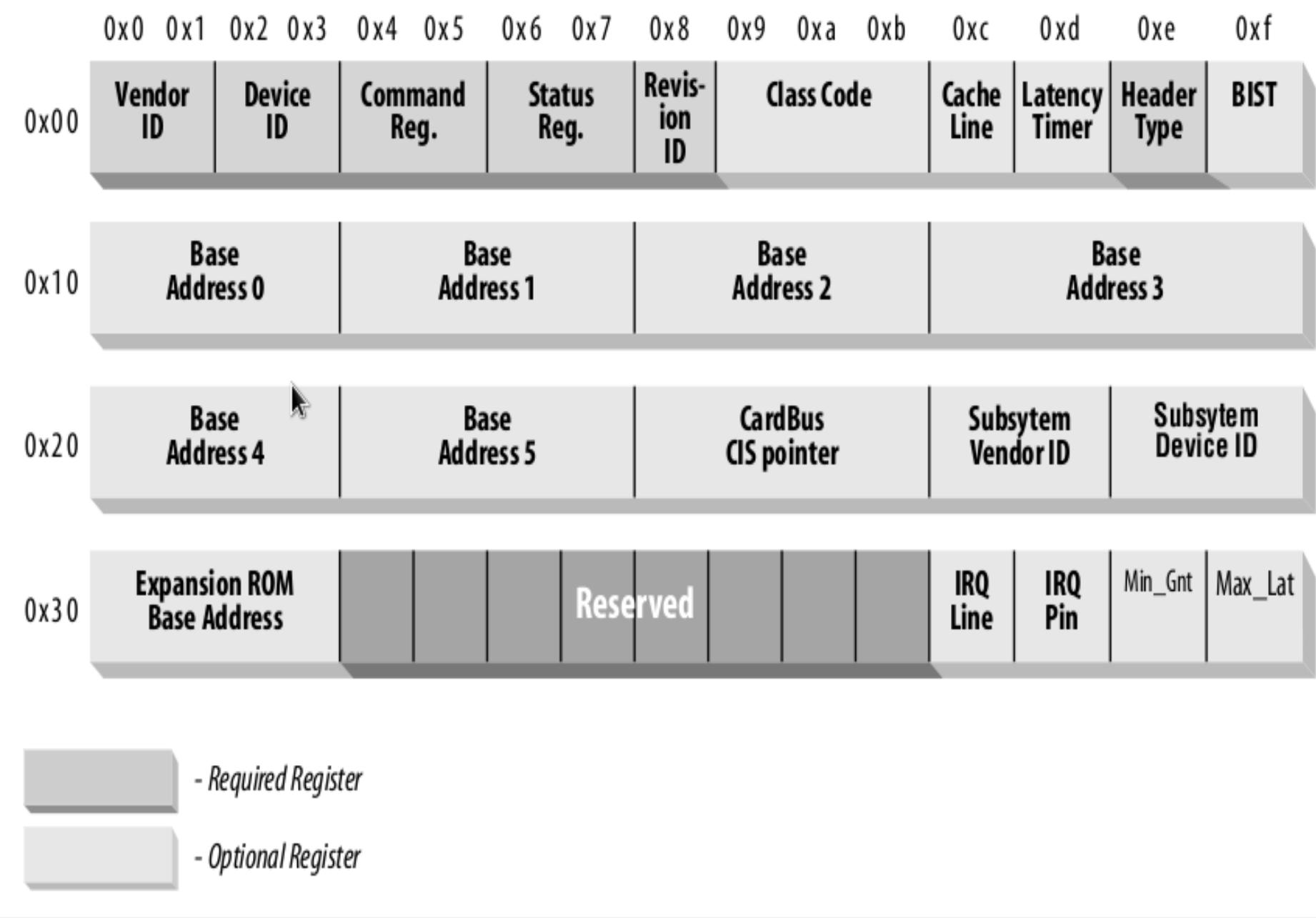
PCI Address spaces(1)

- The CPU and the PCI devices need to access memory that is shared between them. This memory is used by device drivers to control the PCI devices and to pass information between them. Typically the shared memory contains **control** and **status** registers for the devices. These devices are used to control the device and to read its status.
- The CPU's system memory could be used for the shared memory but if it were then every time a PCI device accessed memory, the CPU would have to stall waiting for the PCI device to finish. Access to memory is generally limited to one system component at a time. This would slow the system down. It is not a good idea to allow the system's peripheral devices to access main memory in an uncontrolled way. This would be dangerous; a rogue device could make the system very unstable.

PCI Address spaces(2)

- Peripheral devices have their own memory spaces which they use. The CPU can access these spaces but access by the devices into the system's memory is very strictly controlled using DMA(Direct Memory access) channels. ISA devices have access to two address spaces, ISA I/O(Input/Output) and ISA memory.
- PCI has access to three address spaces:
 - PCI I/O
 - PCI Memory
 - PCI Configuration
- All of these address spaces are also accessible by the CPU with the **PCI I/O** and **PCI Memory address** spaces being used by the **Device Drivers** and **PCI Configuration** space being used by the **PCI initialization code** within the Linux Kernel.

The Standard PCI Configuration Registers



PCI Configuration Space(1)

- All PCI devices features at least a 256-byte address space. The first 64 bytes are standardized, while the rest are device dependent. As the previous slide shows some registers are required and some are optional. Every PCI device must contain meaning full values whereas the optional registers depends on actual capabilities of the peripheral.
- PCI registers are always little endian – If you want conversion of data use functions in <asm/byteorder.h>.
- Three PCI registers identify the device: vendorID, deviceID and class. Every PCI manufacturer assigns proper values to this read-only registers. Additionally the fields **subsystem vendorID** and **subsystem deviceID** are sometimes set by the vendor.

PCI Configuration Space(2)

- Every PCI device in the system, including the PCI-PCI Bridges has a configuration structure that is somewhere in the PCI Configuration address space. The PCI Configuration header allows the system to identify and control the device. Exactly where the header is in the PCI Configuration address space depends on where in the PCI topology the device is.
 - Ex: A PCI video card plugged into one PCI slot on the PC mother board will have its configuration header at one location and if it is plugged into another PCI slot then its PCI slot then its header will appear in another location in PCI configuration memory.
- This does not matter, for where the PCI devices and bridges are, the system will find and configure them using the status and configuration registers in their configuration headers.

PCI Configuration Space(2)

- Typically systems are designed so that every PCI slot has its PCI configuration Header in an offset that is related to its slot on the board. So for example the first slot on the board might have its PCI Configuration at Offset 0 and the second slot at offset 256(all headers are of the same length 256 bytes) and so on.
- A system specific hardware mechanism is defined so that the PCI configuration can attempt to examine all possible PCI Configuration Headers for a given PCI bus and know which devices are present and which devices are absent simply by trying to read one of the fields in the header(usually Vendor Identification field) and getting some sort of error.
- The PCI Local Bus specification describes one possible error message as returning 0xFFFFFFFF when attempting to read the Vendor Identification and Device Identification fields for an empty PCI slot.

PCI Configuration Space(3)

- PCI Configuration Header Contains following fields:
 - **Vendor Identification:** A unique number describing the originator of the PCI device. Digital PCI Vendor identification is 0x1011 and intel's 0x8086.
 - **Device Identification:** A unique number describing the device itself. For example Digitals 21141 fast ether device has a device identification number of 0x0009.
 - **Status:** This field gives the status of the device with the meaning of the bits of this field set by the standard.
 - **Command:** By writing to this field the system controls the device, for example allowing the device to access PCI-I/O memory.
 - **Class code:** This identifies the type of device that this is. There are standard code for every sort of device; video, scsi and so on. The class code for SCSI is 0x0100.

PCI Configuration Space(4)

- **Base Address Registers:** These registers are used to determine and allocate the type, amount and location of PCI I/O and PCI memory space that the device can use.
- **Interrupt Pin:** Four of the physical pins on the PCI card carry interrupts from the card to the PCI bus. The standard label's these as A, B, C and D. The ***Interrupt Pin*** field describes which of these pins this PCI device uses. Generally it is hard-wired for a particular device. That is every-time the system boots the device uses the same interrupt pin. This information allows the interrupt handling subsystem to manage interrupts from this device.
- **Interrupt Line:** This field is used to pass an interrupt handle between the PCI initialisation code, the device driver and the Linux's interrupt handling subsystem. The number written there is meaningless to the device driver but it allows the interrupt handler to correctly route an interrupt from PCI device to correct device driver's interrupt handling code within linux kernel.

PCI Configuration Space(5)

- **PCI I/O and PCI Memory Address:** These two address spaces are used by the devices to communicate with their device drivers running in the Linux Kernel on the CPU. The DECchip 21141 fast Ethernet device maps its internal register into PCI I/O space. Its linux device driver then reads and writes those registers to control the device. Video drivers typically use large amount of PCI memory space to contain video information.
- Until the PCI system has been set up and the devices access to these address spaces have been turned on using the Command field in the PCI configuration header, nothing can access them. It should be noted that only PCI configuration code reads and writes PCI configuration address; the linux device drivers only read and write PCI I/O and PCI memory address.

Struct pci_device_id structure

- Using these different identifiers, a PCI driver can tell the kernel what kind of device it supports. The **struct pci_device_id** structure is used to define the list of different types of PCI devices that a driver supports. This structure contains, **vendor**, **device**, **subvendor**, **subdevice**, **class**, **class_mask**,**driver_data**.
- There are two helper macros that should be used to initialize a **struct pci_device_id**.
- **PCI_DEVICE(vendor,device)**: This creates a **struct pci_device_id** that matches only the specific vendor and device ID.
- **PCI_DEVICE_CLASS(device_class,device_class_mask)**: This creates a **struct pci_device_id** that matches a specific PCI class.
 - **/drivers/usb/host/ehci_hcd.c**, **/drivers/i2c/buses/i2c-i810.c** contains some example on how to use this macros.
- These examples create a list of **struct pci_device_id** structures. This array of Ids is used in the **struct pci_driver** and it is also used to tell user space which devices this specific driver supports.

Struct pci_device - MODULE_DEVICE_TABLE

- This **pci_device_id** structure needs to be exported to user space to allow **hotplug** and **module loading systems** to know what module works with what hardware. The macros **MODULE_DEVICE_TABLE** accomplishes this
- **MODULE_DEVICE(pci,i810_ids)**: This statement creates a local variable called **__mod_pci_device_table** that points to the list of **struct pci_device_id**. Later in the kernel build process the **depmod** program searches all modules for the symbol **__mod_pci_device_table**. If that symbol is found it pulls the data out of the module and adds it to the file **/lib/modules/KERNEL_VERSION/modules.pcimap**
- After **depmod** completes, all PCI devices that are supported by modules in the kernel are listed, along with their module names, in that file. When the kernel tells the **hotplug** system that a new PCI device has been found, the **hotplug** systems uses the **module.pcimap** file to find the proper driver to load.

Registering a PCI Device(1)

- The main structure that all PCI drivers must create in order to be registered with the kernel properly is **struct pci_driver**. This structure consists of a number of function callbacks and variables that describe the PCI driver to the PCI core. Here are the PCI fields that the PCI driver needs to be aware of:
 - **Const char *name:** The name of the driver. It must be unique among all PCI drivers in the kernel and is normally set to the same name as the module name of the driver. It shows up in **sysfs** under **/sysfs/bus/pci/drivers** when the driver is in the kernel.
 - **Const struct pci_device_id** table described earlier.
 - **Int (*probe)(struct pci_dev *dev, const struct pci_device_id *id):** Pointer to the probe function in the PCI driver. This function is called by the PCI core to **struct pci_dev**. The pointer to the **struct pci_device_id** that the PCI core used to make decision is also passed to this function.

Registering a PCI Device(2)

- **Void (*remove)(struct pci_dev *dev):** Pointer to the function that the PCI core calls when the struct pci_dev is being removed from the system, or when the PCI driver is being unloaded from the kernel.
- **Int (*suspend)(struct pci_dev *dev, u32 state):** Pointer to the function that the PCI core calls when the **struct pci_dev** is being suspended. The suspend state is passed in the state variable.
- **Int (*resume)(struct pci_dev *dev):** Pointer to the function that the PCI core calls when the struct pci_dev is being resumed. It is always called after suspend has been called.
- To create a proper struct pci_driver structure, only four fields need to be initialized

*Static struct pci_driver pci_driver={.name="pci_skel",
.id_table = ids,
.probe=probe,
.remove=remove}*

Registering a PCI Device(3)

- To register the **struct pci_driver** with the PCI core, a call to **pci_register_driver** is made with a pointer to the **struct pci_driver**. This is traditionally done in the module initialization code for the PCI driver:

```
static int __init pci_skel_init(void)
{return pci_register_driver(&pci_driver);}
```

- When the **PCI driver** is to be unloaded, the **struct pci_driver** needs to be unregistered from the kernel. This is done with the call to **pci_unregister_driver**.

```
Static void __exit pci_skel_exit(void)
{pci_unregister_driver(&pci_driver);}
```

Enabling the PCI Device

- In the probe function for the PCI driver, before the driver can access any device resources (IO region or interrupt) of the PCI device, the driver must call the **pci_enable_device** function.
- **Int pci_enable_device(struct pci_dev *dev):** This function actually enables the device. It wakes up the device and in some cases also assigns its interrupt line and IO regions.

Accessing the Configuration Space(1)

- After the driver has been detected the device, it usually needs to read from or write to the three address space: **memory**, **port** and **configuration**. Accessing configuration space is vital to the driver, because it is the only way it can find out where the device is mapped in memory and in the IO space.
- To access the configuration space the CPU must write and read registers in the PCI controller. Linux offers a standard interface to access the configuration space.
- *Int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val)*
*Int pci_read_config_word(struct pci_dev *dev, int where, u16 *val)*
*Int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val)*
word and dword functions convert value from little endian to native byte order.
*Int pci_write_config_byte(struct pci_dev *dev, int where, u8 *val)*
*Int pci_write_config_word(struct pci_dev *dev, int where, u16 *val)*
*Int pci_write_config_dword(struct pci_dev *dev, int where, u32 *val)*
Word and dword functions convert the value to little-endian before writing to the peripheral.

Accessing the Configuration Space(2)

- All the previous function are implemented as inline function that really call the following functions. These functions can be used if driver does not have access to a struct `pci_dev` at any particular moment.

*Int pci_bus_read_config_byte(struct pci_bus *bus, unsigned int devfn, int where, u8 val);*

*Int pci_bus_read_config_word(struct pci_bus *bus, unsigned int devfn, int where, u16 val);*

*Int pci_bus_read_config_dword(struct pci_bus *bus, unsigned int devfn, int where, u32 val);*

These are just like `pci_read_` functions, but `struct pci_bus *` and `devfn` variables are needed instead of `struct pci_dev *`.

*Int pci_bus_write_config_byte(struct pci_bus *bus, unsigned int devfn, int where, u8 val);*

*Int pci_bus_write_config_word(struct pci_bus *bus, unsigned int devfn, int where, u16 val);*

*Int pci_bus_write_config_dword(struct pci_bus *bus, unsigned int devfn, int where, u32 val);*

Accessing the IO and memory Spaces(1)

- A PCI device implements upto six I/O address regions. Each region consists of either memory or I/O locations. Most devices implement their I/O registers in memory regions. Unlike normal memory, I/O registers should not be cached by the CPU because each access can have side affects. The PCI device that implements I/O registers as a memory region marks the difference by setting a “memory-is-prefetchable” bit in the configuration register. If the memory region is marked as pre-fetchable, the CPU can caches its contents and do all sorts of optimizations. Nonprefetchable memory access can't be optimized because each access can have side effects, just as with IO ports. Peripherals that map their control registers to memory address range declare that range as nonprefetchable, whereas something like video memory on PCI boards is prefetchable.

Accessing the IO and memory Spaces(2)

- An interface board reports the size and current location of its regions using configuration registers – the six 32-bit registers whose symbolic names are PCI_BASE_ADDRESS_0 through PCI_BASE_ADDRESS_5. Function for getting region information consists of the following functions.
- ***unsigned long pci_resource_start(struct pci_dev *dev, int bar);***
The function return the first address(memory address or IO port number) associated with one of the six PCI I/O regions. The region is selected by the integer bar(the base address register), ranging from 0-5.
- ***unsigned long pci_resource_end(struct pci_dev *dev, int bar);*** The function returns the last address that is part of the I/O region number bar.
- ***unsigned long pci_resource_flags(struct pci_dev *dev, int bar);*** Resource flags are used to define some features of individual resources. All resource flags are define in <linux/ioport.h> Most important are **IORESOURCE_IO**, **IORESOURCE_MEM**, If the associated region exists, one and only one of these flags is set.
IORESOURCE_PREFETCH, IORESOURCE_READONLY: These flags tell whether memory region is prefetchable and or write protected.

PCI-ISA Bridges

- These bridges support legacy ISA devices by translating PCI I/O and PCI memory spaces accesses into ISA I/O and ISA Memory accesses. A lot of systems now sold contain several ISA bus slots and several PCI bus slots. Over time the need for this backwards compatibility will dwindle and PCI only systems will be sold. Where in the ISA address spaces(I/O and Memory) the ISA device(Floppy Disk) of the system have their register fixed in the dim mists of time by the early intel 8080 based PCs. The PCI specification copes with this by reserving the lower regions of the PCI I/O and PCI Memory address spaces for use by the ISA peripherals in the system and using a single PCI-ISA bridge to translate any PCI memory access to those regions into ISA accesses.

PCI-PCI Bridges(1)

- PCI-PCI Bridges are special PCI devices which glue the PCI buses of the system together. Simple systems have a single PCI bus but there is an electrical limit on the number of PCI devices that a single PCI bus can support. Using PCI-PCI bridges to add more PCI buses allow the system to support many more PCI devices. This is particularly important for a high performance server.
- Linux fully supports the use of PCI-PCI bridges. So that the CPU's PCI initialization code can address devices that are not on the main PCI bus, there has to be a mechanism that allows bridges to decide whether or not to pass configuration cycles from their primary interface to their secondary interface. A cycle is just an address as it appears on the PCI bus. The PCI specification defines two formats for the PCI configuration addresses. **Type 0** and **Type 1**: Type 0 PCI Configuration cycles do not contain a bus number and these are interpreted by all devices as being for PCI configuration addresses on the PCI bus. Bits 31:11 of the Type 0 are treated as device select field.

PCI-PCI Bridges(2)

- One way to design a system is to have each bit select a different device. In this case bit 11 would select the PCI in slot 0, bit 12 would select the PCI device in slot 1 and so on. Another way is to write the devices slot number directly into bits 31:11. Which mechanism is used in a system depends on the system's PCI memory controller.
- Type 1 PCI Configuration cycles contain a PCI bus number and this type of configuration cycle are ignored by PCI device except the PCI-PCI bridges. All of the PCI-PCI Bridges seeing Type 1 configuration cycles may choose to pass them to the PCI buses downstream of themselves. Whether the PCI-PCI Bridge ignores the type 1 configuration cycle or passes it onto to the downstream PCI bus depends on how the PCI-PCI Bridge has been configured.
- Every PCI-PCI bridge has a primary bus interface number, and a secondary bus interface number. The primary bus interface being the one nearest the CPU and the secondary bus interface being the one farthest away.

PCI-PCI Bridges(3)

- Each PCI-PCI bridge also has a subordinate bus number and this is the maximum bus number of all the PCI buses that are bridged beyond the secondary bus interface. Or to put it another way, the subordinate bus number is the highest numbered PCI bus downstream of the PCI-PCI bridge. When the PCI-PCI bridge sees a Type 1 PCI configuration cycles it does one of the following things.
 - Ignore it if the bus number specified is not in between the bridge;s secondary bus number and subordinate bus number(inclusive).
 - Convert it to a Type 0 configuration command if the bus number specified matches the secondary bus number of the bridge.
 - Pass it onto the secondary bus interface unchanged if the bus number specified is greater than the secondary bus number and less than or equal to the subordinate bus number. So if we want to address Device 1 on bus 3 of the topology figure 1, we must generate a Type 1 configuration command from the CPU.

PCI-PCI Bridges(4)

Bridge 2 passes this unchanged. Bridge 3 ignores it but converts it into a Type 0 configuration command and sends it out on Bus 3 where Device 1 responds to it.

- It is up to each individual operating system to allocate bus numbers during PCI configuration but whatever the numbering scheme used the following statement must be true for all the PCI-PCI bridges in the system:

“All PCI buses located behind a PCI-PCI bridge must reside between the secondary bus number and the subordinate bus number(inclusive).”

- If this rule is broken then the PCI-PCI Bridges will not pass and translate Type 1 PCI configuration cycles correctly and the system will fail to find and initialise the PCI devices in the system. To achieve this numbering scheme, Linux configures these special devices in a particular order.

Linux PCI Initialization(1)

- The PCI initialisation code in linux is broken into three logical parts.

- **PCI Device Driver:**

This pseudo-device driver searches the PCI system starting at Bus 0 and locates all PCI devices and bridges in the system. It builds a linked list of data structures describing the topology of the system. Additionally, it numbers all of the bridges that it finds.

- **PCI BIOS:**

This software layer provides the services described in **PCI BIOS ROM Specification**.

- **PCI Fixup:** System specific fixup code tidies up the system specific loose ends of PCI initialization.

Linux PCI Initialization(2)

- As the Linux Kernel initialises the PCI system it builds data structures mirroring the real PCI topology of the system. The above figure shows the relationships of the data structures that it would build for the PCI system in first slides.
- Each PCI device (including the PCI-PCI bridges) is described by a **pci_dev** data structure. Each PCI bus is described by a **pci_bus** data structure. The result is a tree structure of PCI buses each of which has a number of child PCI devices attached to them.
- As a PCI bus can only be reached using PCI-PCI Bridge(except the primary PCI bus, bus 0), each **pci_bus** contains a pointer to the PCI device(PCI-PCI Bridge) that it is accessed through. That PCI device is a child of the PCI Buses parent PCI bus.
- Not shown in the figure is a pointer to all of the PCI devices in the system, **pci_device**. All of the PCI devices in the system have their **pci_dev** data structures queued onto this queue. This queue is used by linux kernel to quickly find all the PCI devices in the system.

Linux Device Driver(1)

- The PCI device driver is not really a device driver at all but a function of the operating system called at system initialization time.
- The PCI initialisation code must scan all of the PCI buses in the system looking for all PCI device in the system(including PCI-PCI bridge devices). It uses the PCI BIOS code to find out if every possible slot in the current PCI bus that it is scanning is occupied. If the PCI slot is occupied, it builds a **pci_dev** data structure describing the device and links into the list of known PCI devices(pointer at by **pci_devices**).
- The PCI initialisation code starts by scanning PCI Bus 0. It tries to read the Vendor Identification and Device Identification fields for every possible PCI device in every possible PCI slot. When it finds an occupied slot it builds **pci_dev** data structure describing the device. All of the **pci_dev** data structure built by the PCI initialisation code(including all of the PCI-PCI Bridges) are linked into a singly linked list: **pci_devices**.

Linux Device Driver(2)

- If the PCI device that was found was PCI-PCI bridge then a **pci_bus** data structure is built and linked into the tree of **pci_bus** and **pci_dev** data structures pointed at by **pci_root**. The pci initialisation code can tell if the PCI device is a PCI-PCI Bridge because it has a class code of 0x060400.
- The linux kernel then configures the PCI bus on the other (downstream) side of the PCI-PCI Bridge that it has just found. If more PCI-PCI Bridges are found then these are also configured. This process is known as **depth-wise** algorithm. The system's PCI topology is fully mapped **depth-wise** before searching breadthwise.
- In second slide Linux would configure PCI Bus 1 with its Ethernet and SCSI before it configured the video device on PCI Bus 0.
- As Linux searches for downstream PCI buses it must also configure the intervening PCI-PCI bridges secondary and subordinate bus numbers.

Finding out How much PCI I/O and PCI Memory Space a Device Needs

- Each PCI device found is queried to find out how much PCI I/O and PCI Memory address space it requires To do this, each Base Address Register has all 1's written and then read. The device will return 0's in the don't-care address bits, effectively specifying the address space required.
- There are two basic types of Base Address Register, the first indicates within which address space the device reside; either PCI I/O or PCI memory space. This is indicated by Bit 0 of the register. The above figure shows the two forms of the Base Address Register for PCI Memory and for PCI I/O. To find out just how much of each address space a given Base Address Register is requesting, you write all 1's into the register and read it back .The device will specify zero's in the don't care address bits. Effectively specifying the address space required. This design implies that all address spaces are a power of two and are naturally aligned.

Configuring a PCI System

- For PCI-PCI bridges to pass PCI I/O, PCI Memory or PCI Configuration address space reads and writes across them, they need to know the following.
 - **Primary Bus Number:** The bus number immediately upstream of the PCI-PCI Bridge.
 - **Secondary Bus Number:** The bus number immediately downstream of the PCI-PCI Bridge.
 - **Subordinate Bus Number:** The highest Bus Number of all of the buses that can be reached downstream of the bridge.

Allocating Memory

Getting hold of memory

- We have used **kmalloc** and **kfree** for the allocation and freeing of memory. The Linux kernel offers a richer set of memory allocation primitives, however. In this chapter we look at other ways of making use of memory in device drivers and at how to make and at how to make it the best use of your system's memory resources. We will not get into how the different architectures actually administer memory. Modules are not involved in issues of segmentation, paging and so on, since the kernel offers a unified memory management interface to the drivers.
- The **kmalloc** allocation engine is a powerful tool , and easily learned because of its similarity to **malloc**. The function is fast unless it blocks and it doesn't clear the memory it obtains the allocated region still holds its previous content. The allocated region is also contiguous in physical memory.

```
void *kamloc(size_t size, int flags);
```

```
void kfree(void *obj);
```

The Flags Argument(1)

- The first argument to **kmalloc** is the size of the block to be allocated. The second argument, the allocation flgs, is much more interesting, because it controls the behaviour of **kmalloc** in a number of ways. The most used flags, **GFP_KERNEL**, means that the allocation (internally performed by calling, eventually, **get_free_pages**, which is the source of the GFP_prefix) is performed on behalf of a process running in kernel space. In other words, this means that the calling function is executing a system call on behalf of a process. Using **GFP_KERNEL** means that **kmalloc** can put the current process to sleep waiting for a page when called in low-memory situations. A function that allocates memory using **GFP_KERNEL** must therefore be re-entrant. While the current process sleeps, the kernel takes proper action to retrieve a memory page, either by flushing buffers to disk or by swapping out memory from a user process.
- **GFP_KERNEL** isn't always the right allocation flag to use; sometimes **kmalloc** is called from outside a process' context.

The Flags Argument(2)

- This type of call can happen, for instance in interrupt handlers, task queues, and kernel timer. In this case, the current process should not be put to sleep, and the driver should use a flag of **GFP_ATOMIC** instead. The kernel normally tries to keep some free pages around in order to fulfil atomic allocation. When **GFP_ATOMIC** is used, **kmalloc** can use even the last free page. If that last page does not exist, however the allocation will fail.
- Other flags can be used in place of or in addition to **GFP_KERNEL** and **GFP_ATOMIC**, although those two cover most of the needs of device drivers. All the flags are defined in `<linux/mm.h>` individual flags are prefixed with a double underscore, like `__GFP_DMA`; collections of flags lack the prefix and are sometimes called allocation priorities.
- **GFP_KERNEL**: Normal allocation of kernel memory. May sleep.

The Flags Argument(3)

- **GFP_BUFFER**: Used in managing the buffer cache, this priority allows the allocator to sleep. It differs from **GFP_KERNEL** in that fewer attempts will be made to free memory by flushing dirty pages to disk; the purpose here is to avoid deadlocks when the I/O subsystems themselves need memory.
- **GFP_ATOMIC**: Used to allocate memory from interrupt handlers and other code outside of a process context. Never sleeps.
- **GFP_USER**: Used to allocate memory on behalf of the user. It may sleep, and is a low-priority request.
- **GFP_HIGHUSER**: Like GFP_USER, but allocates from high memory, if any.
- **GFP_DMA**: This flag requests memory usable in DMA data transfers to/from devices. Its exact meaning is platform dependent, and the flag can be ORed to either GFP_KERNEL or GFP_ATOMIC.
- **GFP_HIGHMEM**: The flag requests high memory, a platform-dependent feature that has no effect on platforms that don't support it. It is part of the GFP_HIGHUSER mask and has little use elsewhere.

Memory Zones:(1)

- Both `_GFP_DMA` and `_GFP_HIGHMEM` have a platform dependent role, although their use is valid for all platforms.
- Version 2.6 of kernel knows about three memory zones: **DMA-capable memory, normal memory, and high memory**. While allocation normally happens in the normal zone, setting either of the bits just mentioned requires memory to be allocated from a different zone. The idea is that every computer platform that must know about special memory ranges(instead of considering all RAM equivalent) will fall into this abstraction.
- **DMA-capable** memory is the only memory that can be involved in DMA data transfers with peripheral devices. This restriction arises when the address bus used to connect peripheral devices to the processor is limited with respect to the address bus used to access RAM. For example ,on the x86, devices that plug into the ISA bus can only address memory from 0 to 16MB.
- High memory is a memory that requires special handling to be accessed. It made its appearance in kernel memory management when support for the

Memory Zones:(2)

pentium II virtual Memory extension was implemented during 2.3 development to access up-to 64 GB of physical memory. High memory is a concept that only applies to the x86 and SPARC platform's and two implementation are different.

- Whenever a new page is allocated to fulfil the **kmalloc** request, the kernel builds a list of zones that can be used in the search. If **__GFP_DMA** is specified, only the DMA zone is searched. If no memory is available at low addresses, allocation fails. If no special flag is present, both normal and DMA memory is searched: if **__GFP_HGHMEM** is set, then all three zones are used to search a free page.
- If the platform has no concept of high memory or it has been disabled in thee kernel configuration, **__GFP_HGIHMEM** is defined as 0 and has no effect.

The mechanism behind memory zones is implemented in **mm/page_alloc.c**, while initialization of the zone resides in platform files, usually in **mm/init.c** within the **arch** tree.

The Size Argument:(1)

- The kernel manages the system's physical memory, which is available only in page sized chunks. As a result **kmalloc** looks rather different than a typical user-space **malloc** implementation A simple heap-oriented allocation technique would quickly run into trouble It would have hard time working around the page boundaries. Thus the kernel uses a special page-oriented allocation technique to get he best use from the system's RAM.
- Linux handles memory allocation by creating a set of pools of memory objects of fixed sizes. Allocation request are handled by going to a pool that holds sufficiently large objects, and handing an entire memory chunk back to the requester. The memory management scheme is quite complex, and the details of it are not normally all that interesting to device drier writer,. After all , the implementation can change -- as it did in the 2.1.38 kernel - without affecting the interface seen by the rest of the kernel.
- The one thing driver developers should keep in mind, though is that the kernel can allocate only certain pre-defined fixed-size byte arrays If you ask for an arbitrary amount of memory, you're likely to get slightly more

The Size Argument:(2)

than you asked for, up to twice as much. Also programmers should remember that the minimum memory that **kmalloc** handles as big a s32 or 64 depending on the page size used by their current architecture.

- The data sizes available are generally power of two. In the 2.0 kernel, the available sizes were actually slightly less than power of two, due to control flags added by the management system if you keep this fact in mind, you use memory more efficiently. For example , if you need a buffer of about 2000 bytes and run Linux 2.0 you're better off asking for 2000 bytes, rather than 2048 Requesting exactly a power of two is the worst possible case with any kernel older than 2.1.38 the kernel will allocate twice as much as you request. This is why char used 4000 bytes per quantum instead of 4096.
- You can find the exact values used for the allocation blocks in **mm/kmalloc.c** (with the 2.0 kernel) or **mm/slab.c** (in current kernels). but remember that they can change again without notice. the trick of allocating less than 4KB works well for chat with all 2.x kernels, but its not guaranteed to be optimal in the future.

More memory allocation functions(1)

- In any case, the maximum size that can be allocated by **kmalloc** is 128KB- slightly less with 2.0 kernels. If you need more than a few kilobytes, however, there are better ways than **kmalloc** to obtain memory.
- ***get_free_page***: If a module needs to allocate big chunks of memory, it is usually better to use a page-oriented technique.

To allocate pages, the following functions are available:

- ***get_zeroed_page***: Returns a pointer to a new page and fills the page with zeros.
 - ***__get_free_page***: Similar to *get_zeroed_page*, but doesn't clear the page.
 - ***__get_free_pages***: Allocates and returns a pointer to the first byte of a memory area that is several (physically contiguous) pages long, but doesn't zero the area.
 - ***_get_dma_pages***: Similar to *get_free_pages*, but guarantees that the allocated memory is DMA capable. The prototypes for the function follow:
- ***unsigned long get_zeroed_pages(int flags);***
 - ***unsigned long __get_free_page(int flags);***
 - ***unsigned long __get_free_pages(int flags);***
 - ***unsigned long __get_dma_pages(int flags, unsigned long order);***

More memory allocation functions(2)

- The flags argument works in the same way as with kmalloc; usually either **GFP_KERNEL** or **GFP_ATOMIC** is used. they can be freed with one of the following functions. The first function is a macros that falls back on the second:

void free_page(unsigned long addr);

void free_pages(unsigned long addr, unsigned long order);

- Its worth stressing that get_free_pages and the other functions can be called at any time, subject to the same rules we saw for **kmalloc**. The function can fail to allocate memory in certain circumstances, particularly when **GFP_ATOMIC** is used. Therefore the program calling these allocation function must be prepared to handle an allocation failure.

vmalloc

- **vmalloc** allocates a contiguous memory region in the virtual address space. Although the pages aren't necessarily consecutive in physical memory (each page is retrieved with a separate call to `__get_free_page`), the kernel sees them as a contiguous range of addresses. **malloc** returns 0 (the NULL address) if an error occurs, otherwise, it returns a pointer to an linear memory area of size at least size. the prototypes of the function and its relatives are as follows;

```
#include <linux/vmalloc.h>
void vmalloc(unsigned long size);
void vfree(void *addr);
```

- Its worth stressing that memory addresses returned by **kmalloc** and **get_free_pages** are also virtual addresses Their actual value is still messaged by the MMU.

Block Device Drivers

Block Devices & Drivers(1)

- Devices which do data transfers in terms of blocks of data are known as **Block Devices**. Block devices are random access Devices Ex: Hard disk, pendisk, memory cards.
- **Block driver** is a driver which drives block devices.
- **Registration:** Block drivers, like char drivers, must use a set of registration interfaces to make their devices available to the kernel. The concepts are similar, but the details of block device registration are all different. You have a whole new set of data structures and device operations to learn.
- **Block Driver Registration:** The first step taken by most block drivers to register themselves with the kernel. The below function is used for registration which is declared in <linux/fs.h>

int register_blkdev(unsigned int major, const char *name)

- *The arguments are the major number that your device will be using and the associated name(which the kernel will display in /proc/devices). If major is passed as 0, the kernel will allocate a new*

Block Devices & Drivers(2)

Major number and return it to the caller. As always, a negative return value from **register_blkdev** indicates that an error has occurred. The corresponding function for cancelling a block device registration is:

int unregister_blkdev(unsigned int major, const char *name);

- Here the arguments must match those passed in the **register_blkdev**, or the function returns -EINVAL and not un-register anything.
- Like character drivers, block drivers in the kernel are identified by major numbers. Block major numbers are entirely distinct from character major numbers, however. A block device with major number 32 can coexist with a character device using the same major number since the two ranges are separate.
- One difference is already evident: **register_chrdev** took a pointer to a **file_operations**, but **register_blkdev** uses a structure of type **block_device_operations** instead as it has since kernel version 2.3.38. The structure is still sometimes referred to by the name **fops** in block drivers: we will call it **bdops**.

Block Devices & Drivers(3)

- The definition of this structure is as follows:

```
struct block_device_operations{  
    int (*open)(struct inode *inode, struct file *filp);  
    int (*release)(struct inode *inode, struct file *filp);  
    int (*ioctl)(struct inode *inode, struct file *filp unsigned  
                 command, unsigned long argument);  
    int (*check_media_change) (kdev_t dev);  
    int (*revalidate)(kdev_t dev);  
};
```

- The open, release, and ioctl methods listed here are exactly the same as their char device counterparts. For the purposes of block device registration, however, we must tell the kernel where our request method is. This method is not kept in the **block_device_operations** structure, for both historical and performance reasons; instead, it is associated with the queue of pending I/O operations for the device. By default there is one such queue for each major number.

Block Devices & Drivers(4)

- Note that there are no read or write operations provided in the `block_device_operations` structure. All I/O to block devices is normally buffered by the system user processes do not perform **direct I/O** to these devices. User mode access to the block devices usually is implicit in file system operations they perform, and those operations clearly benefit from I/O buffering. However, even “direct I/O to a block device, such as when a filesystem is created, goes through Linux buffer cache.
- Clearly a block driver must eventually provide some mechanism for actually doing block I/O to a device. In Linux, the method used for these I/O operations is called **request**; it is the equivalent of the **strategy function** found on many Unix systems. The request method handles both read and write operations and can be somewhat complex.

The gendisk structure(1)

- Struct gendisk (declared in <linux/genhd.h>) is the kernel's representation of an individual disk device. In fact, the kernel also uses gendisk structures to represent partitions, but driver authors need not be aware of that. There are several fields in **struct gendisk** that must be initialized by a block driver:
 - Int major;
 - Int first_minor;
 - Int minor;
- Fields that describe the device numbers used by the disk. At a minimum, a driver must use at least one minor number. If your drive is to be partitionable, however (and most should be), you want to allocate one minor number for each possible partition as well.
- **Char disk_name[32]** - Field that should be set to the name of the disk device. It shows up in **/proc/partitions** and **sysfs**.

The gendisk structure(2)

- Struct block_device_operations *bdops;
 - Set of device operations from the previous section.
 - Struct request_queue *queue;
 - Int flags
- A set of **flags** describing the state of the driver. If your device has removable media, you should set **GENHD_FL_REMOVABLE**. CD-ROM drives can set **GENHD_FL_CD**. If for some reason, you do not want partition information to show up in /proc/partitions, set **GENHD_FL_SUPPRESS_PARTITION_INFO**.
 - **Sector_t capacity:** The capacity of this drive, in 512-byte sectors. The **sector_t** type can be 64 bits wide. Drivers should not set this field directly; instead pass the number of sector's to **set_capacity**.
 - **Void *private_data:** Block drivers may use this field for a pointer to their own internal data.

The gendisk structure - Functions(1)

- The kernel provides a small set of functions for working with **gendisk structures**. We introduce them here, then see how **sbulk** uses them to make its disk devices available to the system. **Struct gendisk** is dynamically allocated structure that requires special kernel manipulation to be initialized. Drivers cannot allocate the structure on their own. Instead, you must call

struct gendisk *alloc_disk(minors);

- The minors argument should be the number of minor number this disk uses; note that you cannot change the minors field later and expect things to work proeperly. When disk is not longer needed, it should be freed with;

void del_gendisk(struct gendisk *gd);

- A gendisk is a reference-counted structure (it contains a kobject). There are **get_disk** and **put_disk** functions available to manipulate the reference count, but drivers should never need to do that. Normally , the call to **del_gendisk** removes the final reference to a

The gendisk structure - Functions(2)

Gendisk, but there is no guarantees of that. Thus it is possible that the structure could continue to exist (and your methods could be called) after a call to `del_gendisk`. If you delete the structure when there are not users (that is, after the final release or in your module cleanup function), however, you can be sure that you will not hear from it again.

- Allocating a gendisk structure does not make the disk available to the system. To do that you must initialize the structure and call **add_disk**.

Void add_disk(struct gendisk *gd);

- As soon as you call **add_disk**, the disk is live and its methods can be called at any time. In fact, the first such calls will probably happen even before **add_disk** returns; the kernel will read the first few blocks in an attempt to find a partition table. So you should not call **add_disk** until your driver is completely initialized and ready to respond to requests on that disk.

The gendisk structure - Functions(3)

For the purposes of block device registration, however, we must tell the kernel where our **request** method is. This method is not kept **block_device_operations** structure, instead it is associated with the queue of pending I/O operations for the device. By default there is one such queue for each major number. A block driver must initialize that queue with **blk_init_queue**. Queue initialization and clean-up is defined as follows:

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

```
blk_init_queue(request_queue_t *queue, spinlock_t *lock);
```

- **blk_cleanup_queue(request_queue_t *queue);**
- The init function sets up the queue, and associates the driver's request function(passed as the second parameter) with the queue.

It is necessary to call `blk_cleanup_queue` at module cleanup time. The block driver initializes its queue with this line of code.

- The **request_queue** member contains the I/O request queue that we have just initialized.

The Request Queue(1)

- When the kernel schedules a data transfer, it queues the request in a list, ordered in such a way that it maximizes system performance. The queue of requests is then passed to the driver's request function, which has the following prototype:
 - Void request_fn(request_queue_t *queue);
- The request function should perform the following task for each request in the queue:
 - (1.) Check the validity of the request. This test is performed by the macro INIT_REQUEST, defined in blk.h; the test consists of looking for problems that could indicate a bug in the system's request queue handling.
 - (2.) Perform the actual data transfer. The CURRENT variable (a macro, actually) can be used to retrieve the details of the current request. CURRENT is a pointer to struct request, whose fields are described in the next section.

The Request Queue(2)

- (3.)Clean up the request just processed. This operation is performed by end_request, a static function whose code resides in blk.h
end_request handles the management of the request queue and wakes up rprocesseds waiting on the I/O operation. It asls manages the CURRENT varaible, ensuring that it points to the next unsatisfied request. The driver passes the function a single argument, which is 1 in case of success and 0 in case of failure. When end_request is valled with an argument of 0, and “I/O error” message is delivered to the system logs(via printk).
- (4.)Loop back to the beginning, to consume the next request.
Based on the previous description, a minimal request function, which does not actually transfer any data, would look like this:

The Request Queue(3)

• Request Code:

```
/*Function to process data transfer request's lined up in the request
queue*/
static void my_request(request_queue_t *q)
{
    struct request *req;
    printk("DRIVER: INSIDE REQUEST FUNCTION\n");
    while((req=elv_next_request(q))!=NULL)
    {
        struct mydev *dev=>req->rq_disk->private_data;
        if(!blk_fs_request(req))
        {
            printk("DRIVER: skip non-fs request\n");
            end_request(req,0);
            Continue;
        }
        my_transfer(dev,req-<sector, req->currentnr_sectors, req->buffer, rq_data_dir(req));
        end_request(req,1);
    }
}
```

The Request Structure(1)

The Request Structure:

- ***sector_t sector***: The index of the beginning sector on our device. Remember that this sector number, like all such numbers passed between the kernel and the driver, is expressed in 512-byte sector. If your hardware uses a different sector size, you need to scale sector accordingly. If your hardware uses a different sector size, you need to scale sector accordingly. For example, if the hardware uses 2048-byte sectors, you need to divide the beginning sector number by four before putting it into a request for the hardware.
- ***unsigned long nr_sectors*** – The number of (512-byte) sectors to be transferred.
- ***Char *buffer***: A pointer to the buffer to or from which the data should be transferred. This pointer is a kernel virtual address and can be dereferenced directly by the driver if needed.

The Request Structure(2)

rq_data_dir(struct request *req):

This macro extracts the direction of the transfer from the request.

A zero return value denotes a read from device, a non zero value denotes a write to the device.

- The Linux I/O request queue is represented by a structure of type **request_queue**, declared in **<linux/blkdev.h>** The **request_queue** structure looks somewhat like **file_operations** and other such objects, in that it contains pointers to a number of functions that operate on the queue.

Handling Request: The Detailed View(1)

- The block driver as described earlier works very well. In simple situations, the macros from <linux/blk.h> can be used to easily set up a request function and get a working driver. As already been mentioned, however, block drivers are often a performance critical part of the kernel. Drivers based on the simple code shown earlier will likely not perform very well in many situations, and can also be a drag on the system as a whole. In this section we get into the details of how the I/O request queue works with an eye towards writing a faster, more efficient driver.

The I/O Request Queue:

- Each block driver works with at least one I/O request queue. This queue contains, at any given time, all of the I/O operations that the kernel would like to see done on the driver's devices. The management of this queue is complicated; the performance of the system depends on how it is done.

Handling Request: The Detailed View(3)

- The queue is designed with physical disk drives in mind. With disks, the amount of time required to transfer a block of data is typically quite small. The amount of time required to position the head(seek) to do that transfer, however, can be very large. Thus the linux kernel works to minimize the number and extent of seeks performed by the device.
- Two things are done to achieve those goals. One is the clustering of requests to adjacent sectors on the disk. Most modern file-systems will attempt to lay out files in consecutive sectors; as a result, requests to adjoining parts of the disk are common. The kernel also applies an “elevator” algorithm to the requests. An elevator in a skyscraper is either going up or down; it will continue to move in those directions until all of its “request”(people wanting on or off) have been satisfied. In the same way, the kernel tries to keep the disk head moving the same direction as long as possible; this approach tends to minimize seek times while ensuring all requests get satisfied eventually.

USB Device Drivers

Introduction

- There are different types of USB devices as they can be used for different purposes. First a device can be self powered, bus powered or both. The USB can provide a power supply up to 500mA for its devices. If there are only bus powered devices on the bus the maximum power dissipation could be exceeded and therefore self powered devices exist. They need to have their own power supply. Devices that support both power types can switch to self powered mode when attaching an external power supply.
- Even maximum communication speed can differ for particular USB devices. The USB Specification decides between low speed and full speed devices. Low speed devices(such as mice, keyboards, joysticks etc.) communicate at 1.5MBit/s and have only limited capabilities. Full speed devices(such as audio and video system) can use up to 90% the 12Mbit/s which is about 10Mbit/s including protocol overhead.

Data Flow Types(1)

- The communication on the USB is done in two directions and uses 3 different transfer types. Data directed from the host to a device is called downstream or OUT transfer. The other direction is called upstream or IN transfer. Depending on the device type different transfer variants are used.
- **Control transfers** are used to request and send reliable shot data packets. It is used to configure devices and every one is required to support a minimum set of control commands. Here is a list of standard commands:
 - GET_STATUS, CLEAR_FEATURE, SET_FEATURE,
SET_ADDRESS, GET_DESCRIPTOR, SET_DESCRIPTOR,
GET_CONFIGURATION, SET_CONFIGURATION,
GET_INTERFACE, SET_INTERFACE, SYNCH_FRAME

Data Flow Types(2)

- **Bulk transfer** are used to request or send reliable data packets up to the full bus bandwidth. Devices like scanners or scsi adapter use this transfer type.
- **Interrupt Transfers** are similar to bulk transfers which are polled periodically. If an interrupt transfer was submitted the host controller driver will automatically repeat this request in a specified interval(1ms – 255ms).
- **Isochronous Transfers** send or receive data streams in real-time with guaranteed bus bandwidth but without any reliability. In general these transfers types are used for audio and video devices.

Enumeration and Device Descriptors

- Whenever a USB device is attached to the bus it will be enumerated by the USB subsystem – i.e., an unique device number (1-127) is assigned and then the device descriptor is read. Such a descriptor is read. Such a descriptor is a data structure which contains information about the device and its properties. The USB standard define a hierarchy of descriptors.
- <Image>

Standard Descriptors(1)

- A **Device Descriptors** describes general information about a USB device. It includes information that applies globally to the device and all of the devices configurations A USB device has only one device descriptor.
- The **Configuration descriptor** gives information about a specific device configuration. A USB device has one or more configuration descriptors Each configuration has one or more interfaces and each interfaces has zero or more endpoints. An endpoint is not shared among interfaces within a single configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configuration without restriction. Configuration can be activated exclusively by the standard control transfer **set_configuration**. Different configurations can be used to change global device settings like power consumption.

Standard Descriptors(2)

- An **Interface Descriptor** describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptor describing g a unique set of endpoints within the configuration. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. Alternate settings can be selected exclusively by the standard control transfer **set_interface**. For example a multifunctional devices like a video camera with an internal microphone could have three alternate settings to change the bandwidth allocation on the bus.
 - Camera activated
 - Microphone activated
 - Camera and Microphone activated.

Standard Descriptors(3)

- An **Endpoint descriptor** contains information required by the host to determine the bandwidth requirements of each endpoint. An endpoint represents a logical data source or sink of a USB Device. Endpoint zero is used for all standard control transfers and there is never a descriptor for this endpoint. The USB specification uses the term pipe for an endpoint too.
- **String Descriptors** are optional and provide additional information in human readable Unicode format. They can be used for vendor and device names or serial Device Classes.
- The standard **device** and **interface descriptors** contain fields that are related to classification: sub-class and protocol. These fields may be used by a host system to associate a device or interface to a driver, depending on how they are specified by the class specification. Valid values for the class fields of the device and interface descriptors are defined by the **USB Device Working Group**.

Standard Descriptors(4)

- Grouping devices or interfaces together in classes and then specifying the characteristics in a class specification allows the development of host software which can manage multiple implementations based on that class. Such host software adapts its operation to a specific device or interface using descriptive information presented by the device. A class specification serves as a framework defining the minimum operation of all devices or interfaces which identify themselves as members of the class.

The Linux USB Subsystem

- In Linux there exists a subsystem called “**The USB Core**” with a specific API to support USB devices and host controllers. Its purpose is to abstract all hardware or device dependent parts by defining a set of data structures, macros and functions.
- The **USB Core** contains routines common to all **USB device drivers** and **host controller drivers**. These functions can be grouped into an upper and lower API layer. As shown in next figure. There exists an API for USB device drivers and another one for host controllers. The following section concentrates on the USB device driver layer, because the development for host controller drivers is already finished.
- This next section will give an overview of the **USB framework** by explaining entry points and the usage of API functions. If you are not very familiar with linux device drivers the following section might not be very useful.

The USB Device Driver Framework(1)

- USB Devices Drivers are registered and de registered at the subsystem. A driver must register 2 entry points and its name. For specific USB devices(which are not suitable to be registered at any other subsystem) a driver may register a couple of file operations and a minor number. In this case the specified minor number and the 15 following numbers are assigned to the driver. This makes it possible to serve up-to 16 similar USB devices by one driver. The major number of all USB devices is 180.
- Framework Data Structure:** All USB related functions or data structures follow the same naming convention and start with **usb_**. Following shows the structure needed to register a USB device driver at the subsystem.

Framework Data Structure(1)

Struct usb_driver

{

```
const char *name;
void *(*probe)(struct usb_device , unsigned int, const struct
usb_device_id *id_table);
void (*disconnect)(struct usb_device *, void *);
struct list_head driver_list;
struct file_operations *fops;
int minor;
struct semaphore serialize;
int (*ioctl)(struct usb_device *dev,unsigned int code,void *buf);
const struct usb_device_id *id_table;
```

}

Framework Data Structure(2)

- **Name:** Usually the name of the module.
- **Probe:** the entry point of the probe function.
- **Disconnect:** The entry point of the disconnect function.
- **Driver_list:** For internal use of the subsystem – initializes to {NULL, NULL}.
- **Fops:** The usual list of file operations for a driver.
- **Minor:** The base minor number assigned to this device (the value has to be a multiple of 16).
- **Serialize;**
- **Ioctl;**
- **id_table**

Framework Entry Points(1)

- The USB Driver framework adds two entry points to normal device drivers:

*void *probe(struct usb_device *dev, unsigned int interface, const struct usb_device_id *id_table);*

- This entry point is called whenever a new device is attached to the bus. Then the device driver has to create a new instance of its internal data structure for the new device.
- The **dev** argument specifies the device context, which contains pointers to all USB descriptors. The interface argument specifies the interface number. If a USB driver wants to bind itself to a particular device and interface it has to return a pointer. This pointer normally references the device driver's context structure.
- Probing normally is done by checking the vendor and product identifications or the class and subclass definitions. If they match the interface number is compared with the ones supported by the driver.

Framework Entry Points(2)

- When probing is done on class based it might be necessary to parse some more USB descriptors because the device properties can differ in a wide range. A simple probe routine is shown below.

```
void *probe(struct usb_device *dev, unsigned int interface, const struct
usb_device_id *id_table)
{
    struct driver_context *context;
    if(dev->descriptor.idVendor == 0x0547 && dev->descriptor.idproduct
    == 0x2131 && interface == 1)
    {
        MOD_INC_USE_COUNT;
        /*allocate resources for the instance */
        context = allocate_driver_resources();
        /*return pointer to instance context*/
        return context;
    }
}
```

Framework Entry Points(3)

void disconnect(struct usb_device *dev, void *drv_context);

- This function is called whenever a device which was served by this driver is disconnected. The argument **dev** specifies the device context and the **driver_context** returns a pointer to the previously register **driver_context** of the probe function. After returning from the disconnect function the **USB framework** completely deallocates all data structures associated with this device. So especially the **usb_device** structure must not be used any longer by the usb driver.
- A simple disconnect function is shown below.

Framework Entry Points(4)

- Static void dabusb_disconnect(struct usb_device *usbdev, void *drv_context){
 /*get a pointer to our driver context */
 Struct driver_context *s=driver_context;
 /*set remove pending flags*/
 s->remove_pending =1 ;
 /*wake up all sleeping parts of the driver */
 wake_up(&s->wait);
 /*wait until driver is ready to release device associated structures*/
 sleep_on(&s->remove_ok);
 /*deallocate resources used by this instance*/
 free_driver_resources(s);
 MOD_DEC_USE_COUNT;
}

Framework Functions(1)

int usb_register(struct usb_driver *drv);

- This function is used to register a new **USB device driver** at the subsystem. The argument **drv** points to a completely initialized **usb_driver** structure. On success 0 is returned otherwise an error value is returned.

void usb_deregister(struct usb_driver *drv);

- This function deregister a formerly registered USB device driver at the subsystem.,

void usb_driver_claim_interface(struct usb_driver *driver, struct usb_interface *iface, void *drv_context);

- This function is intended to be used by USB Device drivers that need to claim more than one interface on a device at once when probing. The argument **driver** points to a completely initialized **usb_driver** structure. The **iface** argument points to a **usb_interface** structure which is part of the **usb_config_descriptor** which is accessible from the **usb_device** structure(given in the probe function). The **drv_context** pointer normally references the device driver's context structure

int usb_interface_claimed(struct usb_interface *iface);

- This function is used to check if another device driver already has claimed the specified interface. The return value is 0 if the interface was not claimed by any driver.

Framework Functions(2)

```
void usb_driver_release_interface(struct usb_driver *driver, struct  
usb_interface *iface);
```

- If a driver wants to release a previously claimed interface it has to call this function. In the disconnect function you do not have to release any interfaces that were additionally claimed in the probe function.

```
Const struct usb_device_id *usb_match_id(struct usb_device *dev, struct  
usb_interface *interface, const struct usb_device_id *id);
```

- **Configuring USB Devices:**

- The API includes a set of functions to select or query descriptors, configurations and alternate settings of devices. All these standard operation are done via control transfer to the device.

Descriptor Data Structures(1)

- The Linux USB Subsystem describes the hierarchical structure of descriptors by extending or embedding the standard USB descriptors with or in a subsystem specific structure. This structure helps storing pointers to the selected configuration and interfaces.
- The elements of these structures are only explained in detail as far as they are necessary for subsequent API calls.

```
struct usb_device{  
    ....  
    struct usb_config_descriptor actconfig; /*the active configuration*/  
    ....  
    struct usb_device_descriptor descriptor; /*Descriptor*/  
    struct usb_config_descriptor config; /*All of the configs*/  
}
```

- The **usb_device** structure is the root of all USB specific descriptor. Sometimes it is necessary to parse the descriptors within a driver to configure the device or to set-up transfer requests properly.
 - Accessing all available configuration descriptors can be done like this;
- ```
for(i=0;i< dev->descriptor.bNumConfigurations;i++){
 struct usb_config_descriptor *cfg = &dev->config[i];.....}
```

# Descriptor Data Structures(2)

- Accessing all available interface descriptors of a particular configuration is done like this:

```
for(j=0;j<cfg->bNumInterfaces;j++){
 struct usb_interface *ifp = &cfg->interface[j];....}
```

To start the parsing of the active configuration simply use the **dev->actconfig** pointer.

- Accessing all alternate settings of a particular interface can be done like this:

```
for(k=0;k<ifp->num_altsetting;k++){
 struct usb_interface_descriptor *as = &ifp->altsetting(k); ...}
```

- The active alternate setting can be accessed via  
**\*as = &ifp->altsetting(ifp->act\_altsetting)**

- Accessing all endpoint descriptors of a particular alternate setting can be done like this;

```
for(i=0;i<as->bNumEndpoints;i++){
 struct usb_endpoint_descriptor *ep=&as->endpoint[k];...}
```

# Standard Device Structures(1)

- To query or set a particular configuration or alternate setting there exist a number functions. These commonly used functions set-up standard device request(control transfers for a specified device):

*int usb\_set\_configuration(struct usb\_device \*dev, int configuration);*

To activate a particular configuration use this function. The argument is of *0<= configuration < dev->descriptor.bNumConfigurations.*

Configuration 0 is selected by default after the device is attached t the bus.

*int usb\_set\_interface(struct usb\_device \*dev, int interface, int alternate);*

- This function actives an alternate setting of aspecified interface. The argument interface is of *0<=interface < dev->actconfig->bNumInterfaces.*

The argument alternate is of

*0<=alternate < dev->actconfig->interface[interface].num\_altsetting*

*int usb\_get\_device\_descriptor(struct usb\_device \*dev);*

- This function rereads the complete descriptor tree from a particular device. It is called automatically whenever a device is attached to the bus or it may be called whenever a USB Descriptor is changed.

# Standard Device Structures(2)

*Int usb\_get\_descriptor(struct usb\_device \*dev, unsigned char desc\_type, unsigned char desc\_index void \*buf, int size);*

- Single USB Descriptor can be read as raw data from a devie. This funciton can be used to parse extended or vendor specific descriptors.

*int usb\_get\_string(struct usb\_device \*dev, unsigned short langid, unsigned char index, void \*buf, int size);*

- If a device, configuration or interface descriptor references a string index value this function can be used to retrieve the string descriptor. According to the specification USB strings are coded as unicode. If successful the function return 0 otherwise an error code is returned.

*int usb\_string(struct usb\_device \*dev, int index, char \*buf, size\_t size);*

- This function simplifies **usb\_get\_string** by converting unicode strings into ASCII strings

*int usb\_get\_status(struct usb\_device \*dev, int type, int target, void \*data);*

*int usb\_clear\_halt(struct usb\_device \*dev int pipe);*

- If an endpoint is stalled call this function to the clear STALL. STALL indicates that a function is unable to transmit or receive data or control pipe request isn't supported. The argument endpoint defines a pipe handle.

# Standard Device Structures(3)

*Int usb\_get\_protocol(struct usb\_device \*dev, int ifnum);*

- This HID USB control request is documented.

*int usb\_set\_protocol(struct usb\_device \*dev, int protocol, int ifnum);* This HID USB control request is also documented.

*int usb\_get\_report(struct usb\_device \*dev, unsigned char type, unsigned char id, int ifnum, void \*buf, int size);*

- This HID USB Control is documented.

*int usb\_set\_idle(struct usb\_device \*dev, int ifnumb, int duration, int report\_id);*

- This HID USB control request is documented.

# USB Transfer Data Structures & Macros(1)

This section will give an overview of all data structure; macros and function related to data transfers on the bus. Further it will be explained how to actually set up, submit and process transfer request.

- The linux USB Subsystem uses only one data stransfer structure called USB Request Block(URB). This structure contains all parameter to setup any USB Transfer type. All transfer request are sent asynchronously to the USB Core and the completion of the request is signaled via a callback function.

**Typedef struct**

{

```
unsigned int offset; //offset to the transfer_buffer
unsigned int length; //expected length
unsigned int actual_length; //actual length after processing
unsigned int status; //status after processing
```

}

**Struct urb**

**Typedef void (urb\_complete\_t) (struct urb \*);**

# USB Transfer Data Structures & Macros(2)

```
Typedef struct urb
{
 spinlock_t lock;
 void *hcpriv; //Private data host cntrl(don't care)
 struct list_head urb_list; //list pointer to all active URB's (dc)
 struct urb*next; //pointer to next URB
 struct usb_device *dev; //pointer to associated usb device
 unsigned int pipe ; //pipe information
 int status; //returned status
 unsigned int transer_flags; //USB_DISABLED_SPD|USB_ISO_ASAP
 void * transfer _buffer; //associated data buffer
 int rtransfer_buffer_length; //data buffer length
 int actual_length; //actual data buffer length
 int bandwidth; //allocated bandwidth
 unsigned char *setup_packet; //control transfer only)
 int start_fram; //iso/irq only
 int number_ofpackets; //## of packets in this request (iso)
 int interval; //polling interval (irq only)
 int erro_count; //## of errors in this transfer(iso)
 int timeout; //timeout in jiffies
 void *context; //context for completion routine
 usb_complete_t complete; //pointer to complete routine
 iso_packet_descriptor_t iso_frame_desc[0]; } urb+t, *purb_t; //optional iso descriptor
```

# USB Transfer Data Structures & Macros(3)

- As shown in the URB structure contains elements common to all transfer types. There are several macros to set-up the right parameters but first the common elements will be explained as they are very important.
- **dev**: this element is a pointer to the **usb\_device** structure.(introduced in framework function probe).
- **pipe**: The pipe element is used to encode the endpoint number and properties. There exist several macros to create an appropriate pipe value.

– ***pipe=usb\_sndctrlpipe(dev,endpoint)***

***pipe=usb\_rcvctrlpipe(dev,endpoint)***

Creates a pipe for downstream(snd) or upstream(rcv) control transfers to a given endpoint. The argument dev is a pointer to a **usb\_device** structure. The argument endpoint is usually 0.

– ***pipe=usb\_sndbulkpipe(dev,endpoint)***

***pipe=usb\_rcvbulkpipe(dev,endpoint)***

Creates a pipe for downstream(snd) or upstream(rcv) bulk transfers to given endpoint. The endpoint  $1 \leq \text{endpoint} \leq 15$ (depending on active endpoint descriptors).

# USB Transfer Data Structures & Macros(4)

– *pipe=usb\_sndintpipe(dev,endpoint)*

*pipe=usb\_rcvintpipe(dev,endpoint)*

Creates a pipe for downstream(snd) or upstream(rcv) interrupt transfers to given endpoint. The endpoint  $1 \leq \text{endpoint} \leq 15$  (depending on active endpoint descriptors).

– *pipe=usb\_sndisopipe(dev,endpoint)*

*pipe=usb\_rcvisopipe(dev,endpoint)*

Creates a pipe for downstream(snd) or upstream(rcv) isochronous transfers to given endpoint. The endpoint  $1 \leq \text{endpoint} \leq 15$  (depending on active endpoint descriptors).

• **transfer\_buffer:** This element is a pointer to the associated transfer buffer which contains data transferred from or to a device. This buffer has to be allocated as a non-pageable contiguous physical memory block(simply use `void *kmalloc(size_t, GFP_KERNEL);`)

• **Transfer\_buffer\_length:** This element specifies the size of the transfer buffer in bytes. For interrupt and control transfers the value has to be less or equal the maximum packet size of the associated endpoint. The

# USB Transfer Data Structures & Macros(5)

maximum packet size can be found as element **wMaxPacketSize** of an endpoint descriptor. Because there is no endpoint descriptor for the default endpoint 0 which is used for all control transfer the maximum packet size can be found in element **maxPacketSize** of **usb\_device** structure.

Bulk transfer which are bigger than **wMaxPacketSize** are automatically split into smaller portions.

- **Complete:** As noted above the USB subsystem processes request asynchronously. This element allows to specify a pointer to a caller supplied handler function which is called after the request is completed. The purpose of this handler is to finish the caller specific part of the request as fast as possible because it is called out of the host controller's hardware interrupt handler. This even implies all other restriction that apply for code which is written for interrupt handlers.
- **Context:** Optionally a pointer to a request related context structure can be given following code sample shows a simple completion handler.

```
void complete(struct urb *purb){
 struct device_context *s = purb->context;
 wake_up(&s->wait) }
```

# USB Transfer Data Structures & Macros(6)

• **transfer\_flags**: A number of transfer flags may be specified to change the behaviour when processing the transfer request.

**USB\_DISABLE\_SPD**: This flag disable short packets. A short packet condition occurs if an upstream request transfer less data than maximum packet size of the associated endpoint.

**USB\_NO\_FSBR**; **USB\_ISO\_ASAP**: When scheduling isochronous request this flag tell the host controller to start the transfer as soon as possible. If **USB\_ISO\_ASAP** is not specified a start frame has to be given. It is recommended to use this flag if isochronous transfer do not have to be synchronized with the current frame number. The current frame number is a 11 bit counter that increments every millisecond.

**USB\_ASYNC\_UNLINK**: When a URB has to be cancelled it can be done synchronously or asynchronously. Use this flag to switch on asynchronous URB unlinking.

**USB\_TIMEOUT\_KILLED**: This flag is set by the boost controller to mark the URB as killed by time-out. The URB status carries the actual error which caused the time-out.

**USB\_QUEUE\_BULK**: This flag is used to allow queueing for bulk transfer. Normally only one bulk transfer can be queue for an endpoint of a particular device.

# USB Transfer Data Structures & Macros(7)

- **Next:** It is possible to link several URBs in a chain by using the next pointer. This allows you to send a sequence of USB transfer request to the USB Core. The chain has to be terminated by a NULL pointer or the last URB has to be linked with the first. This allows to automatically reschedule a number of URBs to transfer a continuous data stream.
- **Status:** This element carries the status of an ongoing or already finished request. After successfully sending a request to the USB core the status is -EINPROGRESS. The successful completion of request is indicated by 0. There exists a number of error conditions which are documented.
- **actual\_length:** After a request has completed this element counts the number of bytes transferred.

# USB Transfer Data Structures & Macros(8)

- The remaining element of the URB are specific to the transfer type
- **Bulk Transfers:** Not additional parameter have to be specified
- **Control Transfers:**
  - **setup\_packet:** Control transfer consists of 2 or 3 stages .The first stage is the downstream transfer of the set-up packet. This element takes the pointer to a buffer containing the set-up data. This buffer hast o be allocated as a non-pageable contiguous physical memory block(simply use **void \*kmalloc9size\_t< GFP\_KERNEL);**).
- **Interrupt Transfers:**
  - **start\_frame:** This element is returnend to indicate the first fram number the interrupt is scheduled.
  - **Interval:** This element specifies the interval in milliseconds of the interrupt transfer allowed values are  $1 \leq \text{interval} \leq 255$ .Specifying an interval of 0ms causes an one shot interrupt(no automatic rescheduling is done. You can find the interrupt interval as element **bInterval** of an endpoint descriptor for interrupt endpoints.

# USB Transfer Data Structures & Macros(9)

## • ISOCHRONOUS TRANSFER:

- **start\_frame:** This element specifies the first frame number the isochronous transfer is scheduled. Setting the **start\_frame** allows to synchronize transfer to or from a endpoint. If the USB\_ISO\_SASAP flag is specified this element is returned to indicate the first frame number the isochronous transfer is scheduled.
- **number\_of\_packets:** Isochronous transfer request are sent to the USB core as a set of single request. A single request transfers a data packet up-to the maximum packet size of the specified endpoint (pipe). This element sets the number of packets for transfer.
- **error-count:** After the request is completed(URB Status is!= -EINPROGRESS) this element counts the number of erroneous packet. detailed information about the single transfer request scan be found in the **iso\_frame\_desc** structure.
- **Timeout:** A timeout in jiffies can be specified to automatically remove a URB form the host controller schedule. If a time-out happens the transfer flag **USB\_TIMEOUT\_KILLED** is set. The actual transfer status carries the UGR

# USB Transfer Data Structures & Macros(10)

- **iso\_frame\_desc**: This additional array of structure sat he end of every isochronous URB sets up the transfer parameter for every single request packet.
  - **Offset**: Specifies the offset address to the transfer\_buffer for a single request.
  - **length**: Specifies the length of the data buffer for a single packet. If length is set out 0 for a single request eh USB frame is skipped and not transfer will be initiated This option can be used to synchronize isochronous data streams.
  - **actual\_length**: returns the actual number of bytes transferred by this request.
  - **status**: Return the status of the request.

# USB Functions(1)

- There are four function of the USB core that handle URBs.

- ***purb\_t usb\_alloc\_urb(int iso\_packets);***

Whenever a URB structure is needed this function has to be called. The argument ***iso\_packets*** is used to specify the number of ***iso\_fram\_desc*** structures at the end of the URB structure when setting up isochronous transfers. If successful the return value is a pointer to a URB structure preset to zero otherwise a NULL pointer is returned.

- ***void usb\_free\_urb(purb\_t purb);***

To free memory allocated by `usb_alloc_urb` simply call this function.

- ***int usb\_submit\_urb(purb\_t purb);***

This function sends a transfer request asynchronously to the USB core. The argument ***purb*** is a pointer to a previously allocated and initialized URB structure. If successful the return value is 0 otherwise an appropriate error code is returned. The function returns always non-blocking and it is possible to schedule several URBs for different endpoints without waiting. On isochronous endpoints it is even possible to schedule more URBs for one endpoint. This limitation is caused due to error handling and retry mechanism of the usb protocol.

# USB Functions(2)

- `int usb_unlink_urb(purb_t purb);`

This function cancels a scheduled request before it is completed. The argument **purb** is a pointer to a previously submitted URB structure. The function can be called synchronously or asynchronously depending on the transfer\_flag **USB\_ASYNC\_UNLINK**. Synchronously called the function waits for 1ms and must not be called from an interrupt or completion handler. The return value is 0 if the function succeeds Asynchronously called the function returns immediately. The return value is -EINPROGRESS if the function was successfully started. When calling **usb\_unlink\_urb** the completion handler is called after the function completed. The URB status is marked with -ENOENT(synchronously called) or -ECONNRESET(asynchronously called) .

**usb\_unlink\_urb** is also used to stop an interrupt transfer **URB**. As documented interrupt transfers are automatically rescheduled. Call **usb\_unlink\_urb** even for “one shot interrupts”.

# USB Macros

- To initialize URB structures for different transfer types there exist some macros;
  - *FILL\_CONTROL\_URB(purb, dev, pipe, setup\_packet, transfer\_buffer, transfer\_buffer\_length, complete, context);*
  - *FILL\_BULK\_URB(purb, dev, pipe, transfer\_buffer, transfer\_buffer\_length, complete, context);*
  - *FILL\_INT\_URB(purb, dev, pipe, transfer\_buffer, transfer\_buffer\_length, complete, context, interval)*
  - *FILL\_CONTROL\_URB\_TO();*
  - *FILL\_BULK\_URB\_TO();*
- The macros self explaining – more documentation can be found in the include file **usb.h**.

# Compatibility Wrappers

- The USB core contains a number of higher level functions which were introduced as compatibility wrappers for the older APIs. Some of these functions can still be used to issue blocking control or bulk transfers.
- *int usb\_control\_msg(struct usb\_device \*dev, unsigned int pipe, \_\_u8 request, \_\_u8 requesttype, \_\_u16 value, \_\_u16 index, void \*data, \_\_u16 size, int timeout);*  
Issues a blocking standard control request. A timeout in jiffies has to be specified. If successful the return value is a positive number which represents the bytes transferred otherwise an error code is returned.
- *Int usb\_bulk\_msg(struct usb\_device \*usb\_dev, unsigned int pipe, void \*data, int len, unsigned long \*actual\_length, int timeout);*  
Issues a blocking bulk transfer. The standard arguments should be self-explaining **actual\_length** is an optional to a variable which carries the actual number of bytes transferred by this request. A **timeout** in jiffies has to be specified.

# Kernel Debugging Techniques

# Challenges to kernel debugging(1)

---

- Debugging modern operating system involves many challenges below are the list of challenges you will encounter while debugging Linux kernel code.  
Pipeline architectures hide important code-execution details. This is because memory access on bus can be ordered differently from code execution. It is not always possible to correlate external bus activity to internal processor execution.
  - Linux Kernel code is highly optimized for speed of execution in many areas.
  - Compilers use optimization techniques that complicate the correlation of c source to actual machine instruction flow. Inline functions are good example of this.
  - Single stepping through compiler optimized code often produces unusual unexpected results.
  - Virtual memory isolates user space memory from kernel memory and can make various debugging scenarios especially difficult.
  - Some code cannot be stepped through with traditional debuggers.
  - Start-up code can be especially difficult because of its proximity to the hardware and the limited resource available(no console, limited memory mapping and so on).

# Challenges to kernel debugging(2)

- The Linux kernel has matured into a very high-performance operating system that can compete with the best commercial operating systems. Many areas within the kernel do not lend themselves easy analysis by simply reading the source code. Knowledge of the architecture and detailed design are often necessary to understand the code flow.
- GCC is an optimizing compiler. By default, the Linux kernel is compiled with the -O2 compiler flag. This enables many optimization algorithms that can change the fundamental structure and order of your code. For example the Linux kernel makes heavy use of **inline** functions which results in the function's being included directly in the execution thread instead of generating a function call with the associated overhead. **Inline** functions require a minimum of -O1 optimization level. Therefore, you cannot turn off optimization, which would be desirable to facilitate debugging.
- Single stepping through code is difficult or impossible where code paths that modify virtual memory settings. When an application makes a system call that results in entry into the kernel(change in address space seen by process.).

# Using KGDB for kernel debugging

---

- Two popular methods enable symbolic source-level debugging within the Linux Kernel:
  - Using KGDB as a remote GDB agent.
  - Using a hardware JTAG probe to control the processor.
- KGDB(kernel GDB) is a set of linux kernel patches that provide an interface to GDB through its remote serial protocol. KGDB implements a GDB stub that communicates with a cross-GDB running on your host development workstation. Until recently KGDB on the target required **serial connection** to the development host. Some targets support KGDB connection via **Ethernet** and **USB**. Complete support for KGDB is still not in the mainline **kernel.org**. You need to port KGDB to your chosen target or obtain an embedded Linux distribution for your chosen architecture and platform that contains KGDB support.