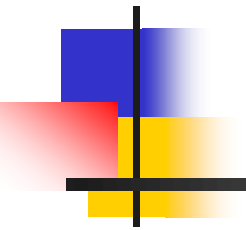
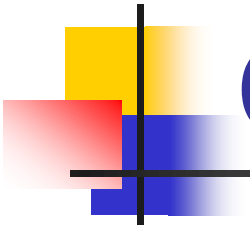


Embedded Linux

Unit- III

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

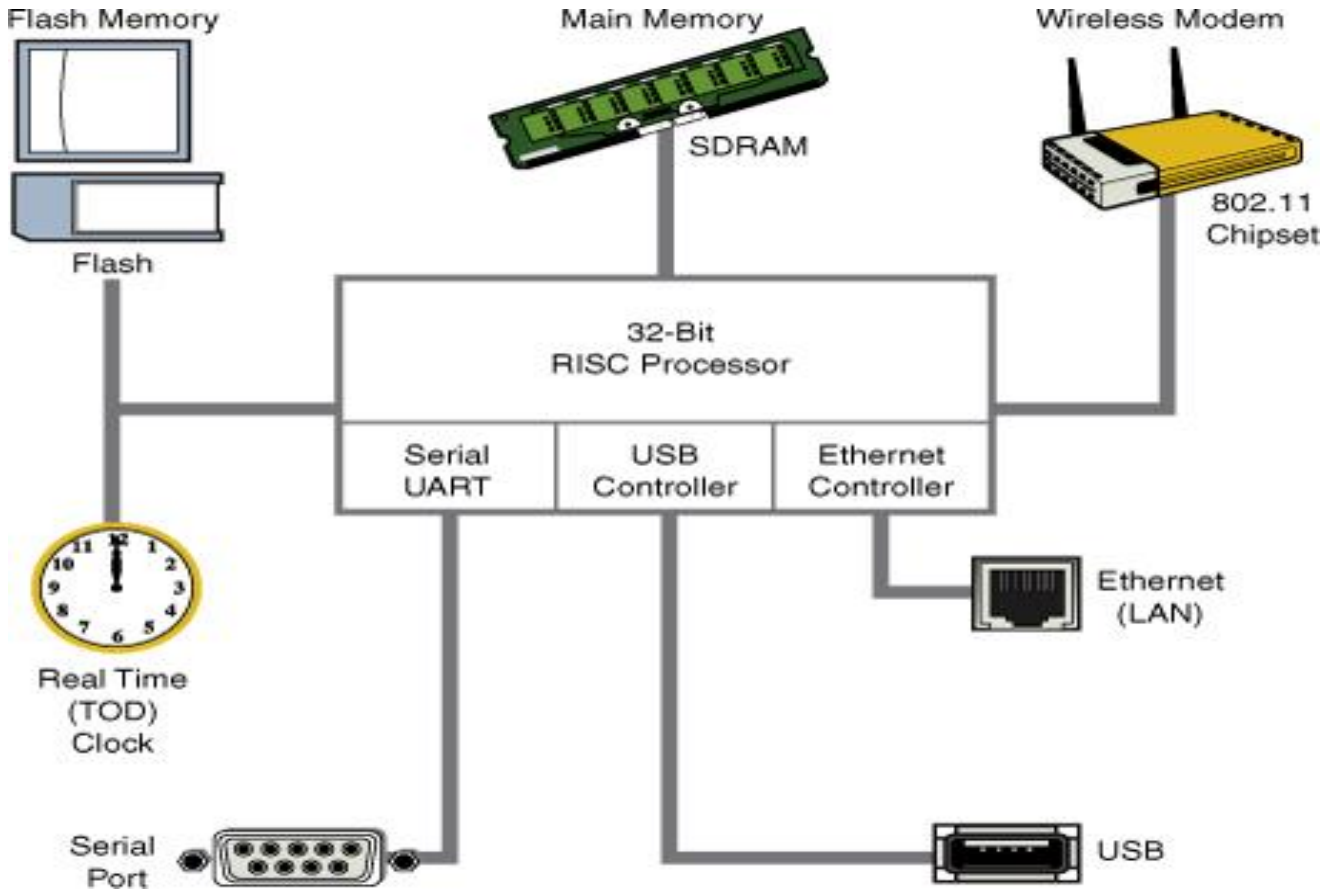


Outline

Contents of Unit III and IV from SPPU Syllabus prescribed for theory subject

310250 -Embedded Operating Systems

Embedded System



UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

Why Linux

• Linux supports a vast variety of hardware devices, probably more than any other OS.

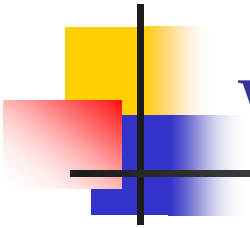
• Linux supports a huge variety of applications and networking protocols.

• Linux is scalable, from small consumer-oriented devices to large, heavy-iron, carrier-class switches and routers.

• Linux can be deployed without the royalties required by traditional proprietary embedded operating systems.

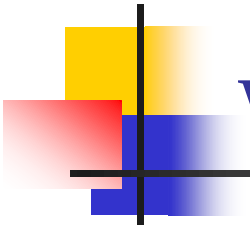
• Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.

• An increasing number of hardware and software vendors, including virtually all the top-tier chip manufacturers and independent software vendors (ISVs), now support Linux.



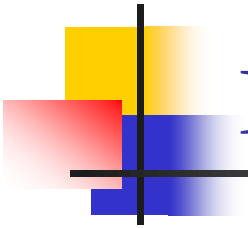
What is Embedded Linux?

- Porting the Linux kernel to run on a particular CPU and board which will be put into an embedded device.
- There are many companies that sell embedded Linux solutions.
- These usually include a ported Linux kernel with cross-development tools, and sometimes with real time extensions.
- The APIs and kernel codebase are the same for embedded Linux as desktop Linux



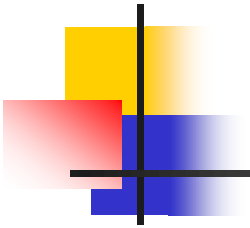
Why Embedded Linux?

- Royalty-free
- Strong networking support
- Has already been ported to many different CPU architectures
- Relatively small for its feature set
- Easy to configure
- Huge application base
- Modern OS (eg. memory management, kernel modules, etc.)



Embedded Linux System

- Boot loader
  - U-boot
  - GRUB
  - …
- Kernel
- File system
  - Many types



Open Source Development Labs

- **OSDL(2000) Goal :-** "to be the recognized center-of-gravity for the Linux industry.
- On January 22, 2007, OSDL and the Free Standards Group merged to form **The Linux Foundation.**

OSDL had established four Working Groups since 2002:

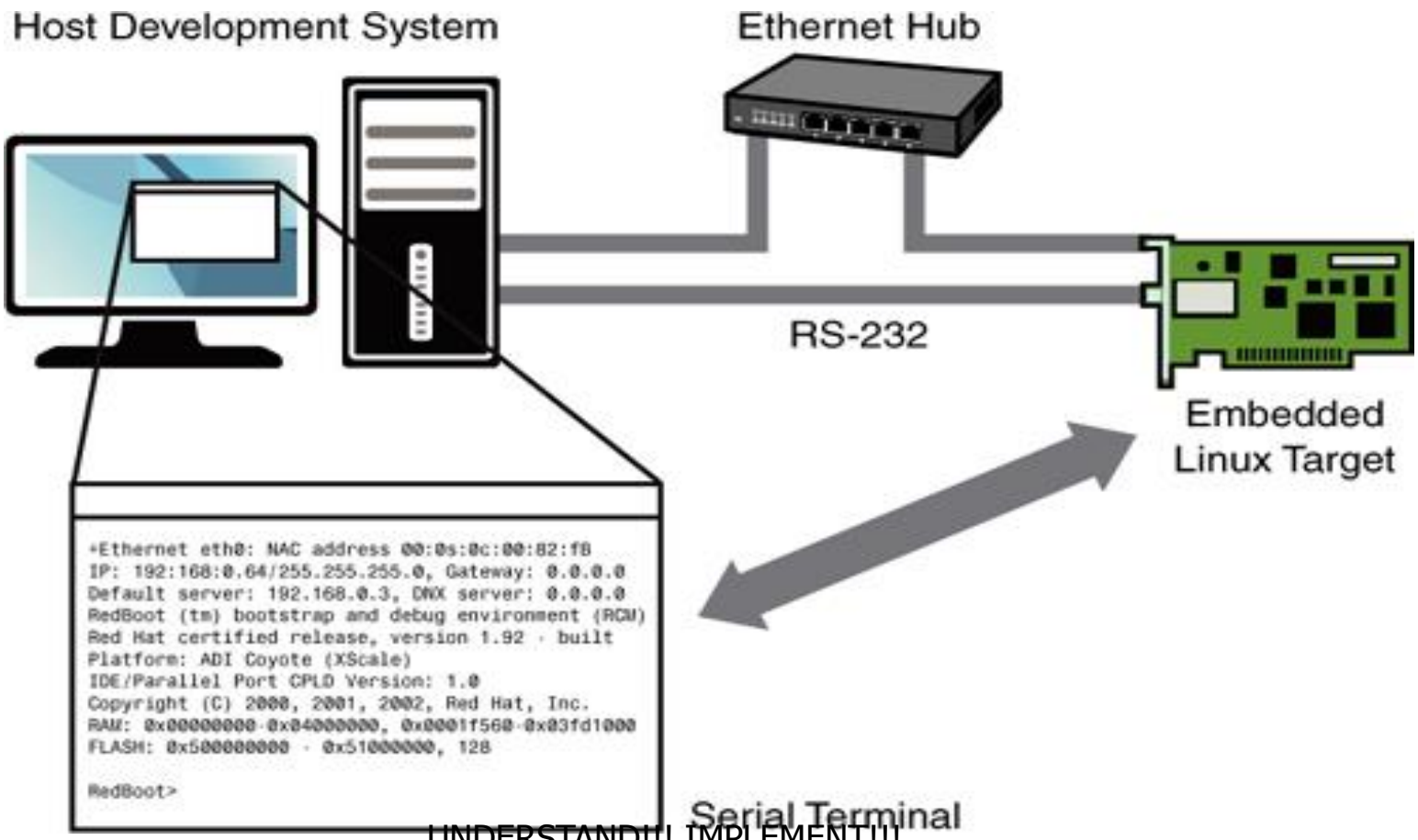
1. Mobile Linux Initiative (MLI)
2. Carrier Grade Linux (CGL)
3. Data Center Linux (DCL)
4. Desktop Linux (DTL)

Linux Standard Base

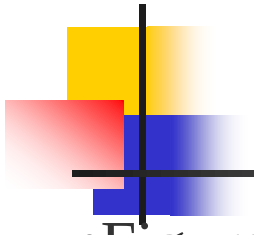
- The goal of the LSB is to establish a set of standards designed to enhance the interoperability of applications among different Linux distributions.
- The LSB spans several architectures, including IA32/64, Power Architecture 32- and 64-bit, AMD64, and others.
- The standard is divided into a core component and the individual architectural components.
- The LSB specifies common attributes of a Linux distribution, including object format, standard library interfaces, a minimum set of commands and utilities and their behavior, file system layout, system initialization, and so on.
- http://www.linuxfoundation.org/

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

Embedded Linux development setup



UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!



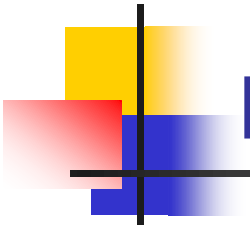
•Figure is a common arrangement.

•It shows a host development system, running desktop Linux distribution, such as Red Hat, Ubuntu Linux.

•The embedded Linux target board is connected to the development host via an RS-232 serial cable.

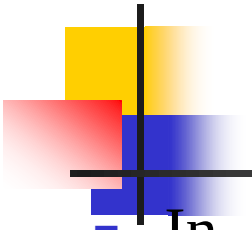
•You plug the target board's Ethernet interface into a local Ethernet hub or switch, to which your development host is also attached via Ethernet.

•The development host contains your development tools and utilities along with target files, which normally are obtained from an embedded Linux distribution.



BIOS Versus Bootloader

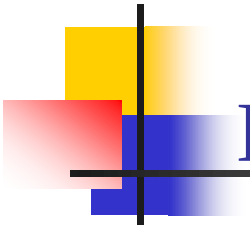
- BIOS takes control of the processor, when power is applied to PC.
- The BIOS might actually be stored in Flash memory.
- The BIOS is a complex set of system-configuration software routines that have knowledge of the low-level details of the hardware architecture.
- Its primary responsibility is to initialize the hardware, especially the memory subsystem, and load an operating system from the PC's hard drive.



- In a typical embedded system, a bootloader is the software program that performs the equivalent functions.
- In custom embedded system, bootloader specific to board the developed.
- There are several good open source bootloaders are available.
- The bootloader provides the foundation from which the primary system software is spawned.

Tasks bootloader performs on power-up

- Initializes critical hardware components, such as the SDRAM controller, I/O controllers, and graphics controllers.
- Initializes system memory in preparation for passing control to the operating system.
- Allocates system resources such as memory and interrupt circuits to peripheral controllers, as necessary.
- Provides a mechanism for locating and loading your operating system image.
- Loads and passes control to the OS, passing any required startup information. This can include total memory size, clock rates, serial port speeds, and other low-level hardware-specific configuration data.



Booting the Kernel

- Now that U-Boot has initialized the hardware, serial port, and Ethernet network interfaces, it has only one job left in its short but useful life span: to load and boot the Linux kernel.
- All bootloaders have a command to load and execute an operating system image.
- Figure shows one of the more common ways U-Boot is used to manually load and boot a Linux kernel.

```
=> tftp 600000 uImage
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.0.103; our IP address is 192.168.0.18
Filename 'uImage'.
Load address: 0x600000
Loading: ###################################################################
         ###################################################################
done
Bytes transferred = 1838553 (1c0dd9 hex)
=> tftp c00000 dtb
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.0.103; our IP address is 192.168.0.18
Filename 'dtb'.
Load address: 0xc00000
Loading: ##
done
Bytes transferred = 16384 (4000 hex)
=> bootm 600000 - c00000
## Booting kernel from Legacy Image at 00600000 ...
   Image Name:   MontaVista Linux 6/2.6.27/freesc
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
```

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

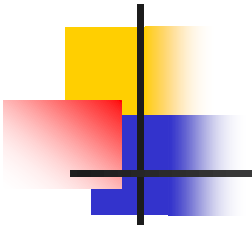
•The **tftp** command at the start of coding instructs U-Boot to load the kernel image uImage into memory over the network using the TFTP protocol.

•The kernel image, in this case, is located on the development workstation.

• The tftp command is passed an address that is the physical address in the target board's memory where the kernel image will be loaded.

•The second invocation of the **tftp** command loads a board configuration file called a *device tree*. It is also referred as *flat device tree* and *device tree binary* or dtb.

•This file contains board-specific information (such as memory size, clock speeds, onboard devices, buses, and Flash layout), that the kernel requires in order to boot the board.



•Next, the **bootm** (boot from memory image) command is issued, to instruct U-Boot to boot the kernel just loaded from the address specified by the tftp command.

•It instruct U-Boot to load the kernel at 0x600000 and pass the device tree binary (dtb) at 0xc00000 to the kernel.

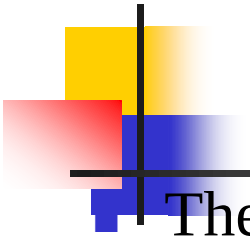
•This command transfers control to the Linux kernel.

•The only way to pass control back to the bootloader is to reboot the board.

•The kernel claims any memory and system resources that the bootloader previously used.

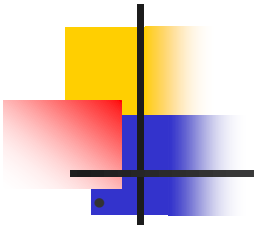
Kernel Initialization: Overview

- When the Linux kernel begins execution, it spews out numerous status messages during its boot process.
- In the example being discussed here, the Linux kernel displayed approximately 200 printk lines before it issues the login prompt.
- Shortly before issuing a login prompt on the serial terminal, Linux mounts a root file system.
- Many legacy embedded operating systems did not require a file system.
- A file system consists of a predefined set of system directories and files in a specific layout on a hard drive or other medium that the Linux kernel mounts as its root file system.
- Linux can mount a root file system from other devices.



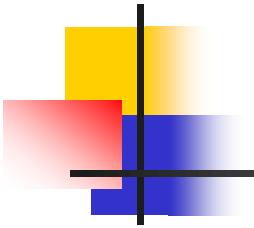
First User Space Process: init

- The Linux kernel spawn an application program called *init* after its internal initialization and mounted its root file system,
- When the kernel starts init, it is said to be running in *user space* or user space context.
- In this operational mode, the user space process has restricted access to the system and must use kernel system calls to request kernel services such as device and file I/O.



Storage Considerations

Embedded systems have limited physical resources.
Hard drives are bulky, have rotating parts, are sensitive to physical shock, and require multiple power supply voltages, which makes them unsuitable for many embedded systems.
The hard drive typically is replaced by smaller and less expensive nonvolatile storage devices.



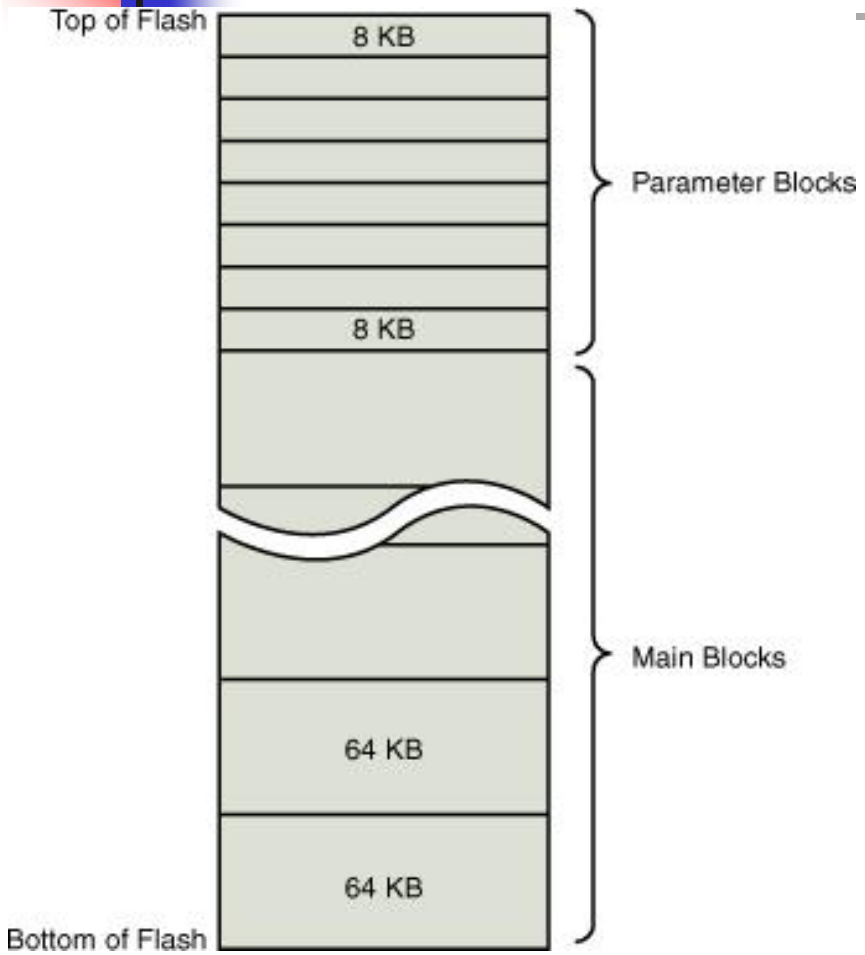
Flash Memory

Flash memory technology, can be thought of as solid-state hard drives, capable of storing many megabytes—and even gigabytes—of data .

Flash memory can be written to and erased under software control. Rotational hard drive technology remains the fastest writable medium.

Flash write and erase time is still considerably slower.

Flash memory is divided into relatively large erasable units, referred to as erase blocks.



Boot block Flash architecture

- A typical NOR Flash memory device contains many erase blocks.

Flash memory is also available with non uniform erase block sizes, to facilitate flexible data-storage layouts.

- These are called boot block or boot sector Flash chips.
- The bootloader is stored in the smaller blocks, and the kernel and other required data are stored in the larger blocks.

Limitations of NOR Flash

•To modify data stored in a Flash memory array, the block in which the modified data resides must be completely erased. Even if only 1 byte in a block needs to be changed, the entire block must be erased and rewritten.

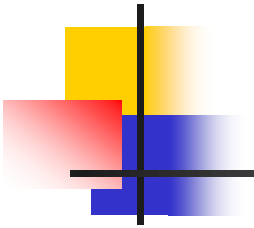
•Flash block sizes are relatively large compared to traditional hard-drive sector sizes.

•Write times for updating data in Flash memory can be many times that of a hard drive.

•Flash memory cell write lifetime. A NOR Flash memory cell has a limited number of write cycles before failure. (100,000 cycles per block)

NAND Flash

- NAND Flash is a relatively new Flash technology.
- NAND Flash offers smaller block sizes, resulting in faster and more efficient writes and generally more efficient use of the Flash array.
- NAND devices present an operational model more similar to that of a traditional hard drive and associated controller.
- Data is accessed in serial bursts, which are far smaller than NOR Flash block size.
- Write cycle lifetime for NAND Flash is an order of magnitude greater than for NOR Flash, although erase times are significantly smaller.

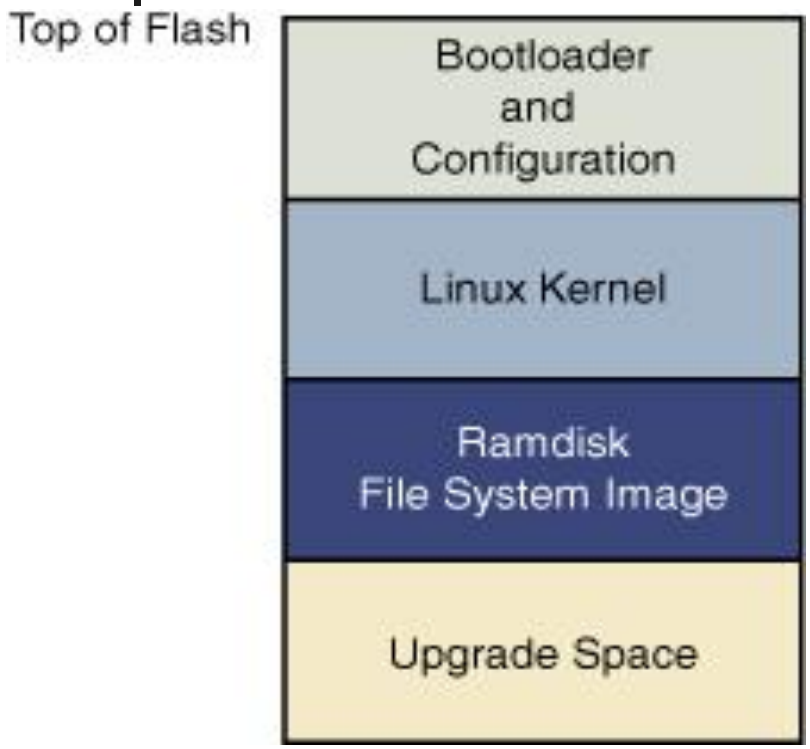


Flash Usage

•An embedded system designer has many options in the layout and use of Flash memory.

•In the simple systems, raw binary data can be stored on the Flash device. When booted, a file system image stored in Flash is read into a Linux ram disk block device, mounted as a file system, and accessed only from RAM.

•Following Figure illustrates a common Flash memory organization that is typical of a simple embedded system in which nonvolatile storage requirements of dynamic data are small and infrequent.



•The bootloader is often placed in the top or bottom of the Flash memory array.

•Following the bootloader, space is allocated for the Linux kernel image and the ramdisk file system image, which holds the root file system.

•The Linux kernel and ramdisk file system images are compressed, and the bootloader handles the decompression task during the boot cycle.

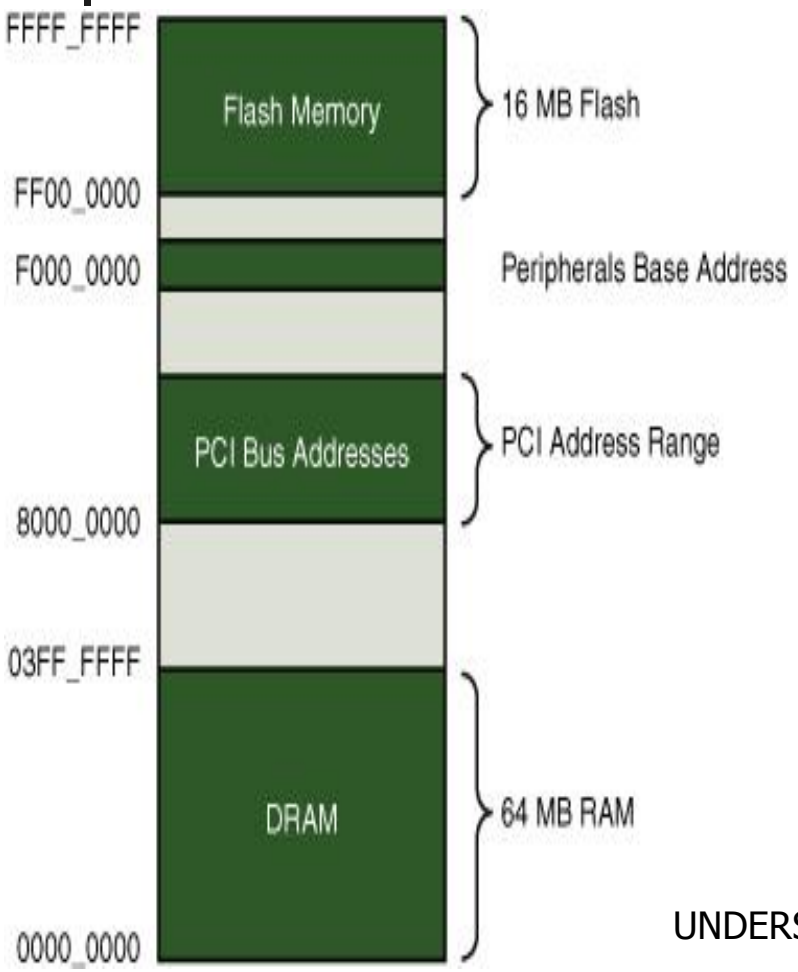
Typical Flash memory layout

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

Flash File Systems

- The limitations of the simple Flash layout scheme just described can be overcome by using a Flash file system to manage data on the Flash device in a manner similar to how data is organized on a hard drive.
- The enhancements to Flash file systems was the incorporation of wear leveling.
- Wear-leveling algorithms are used to distribute writes evenly over the physical erase blocks of the Flash memory in order to extend the life of the Flash memory chip.

Memory Space



All embedded operating systems view and manage system memory as a single large, flat address space. e.g. a microprocessor's address space exists from 0 to the top of its physical address range. Hardware designs commonly place DRAM starting at the bottom of the range, and Flash memory from the top down. Unused address ranges between the top of DRAM and bottom of Flash would be allocated for addressing of various peripheral chips on the board. This design approach is often dictated by the choice of microprocessor. Figure shows a typical memory layout for a simple embedded system.

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

Execution Context

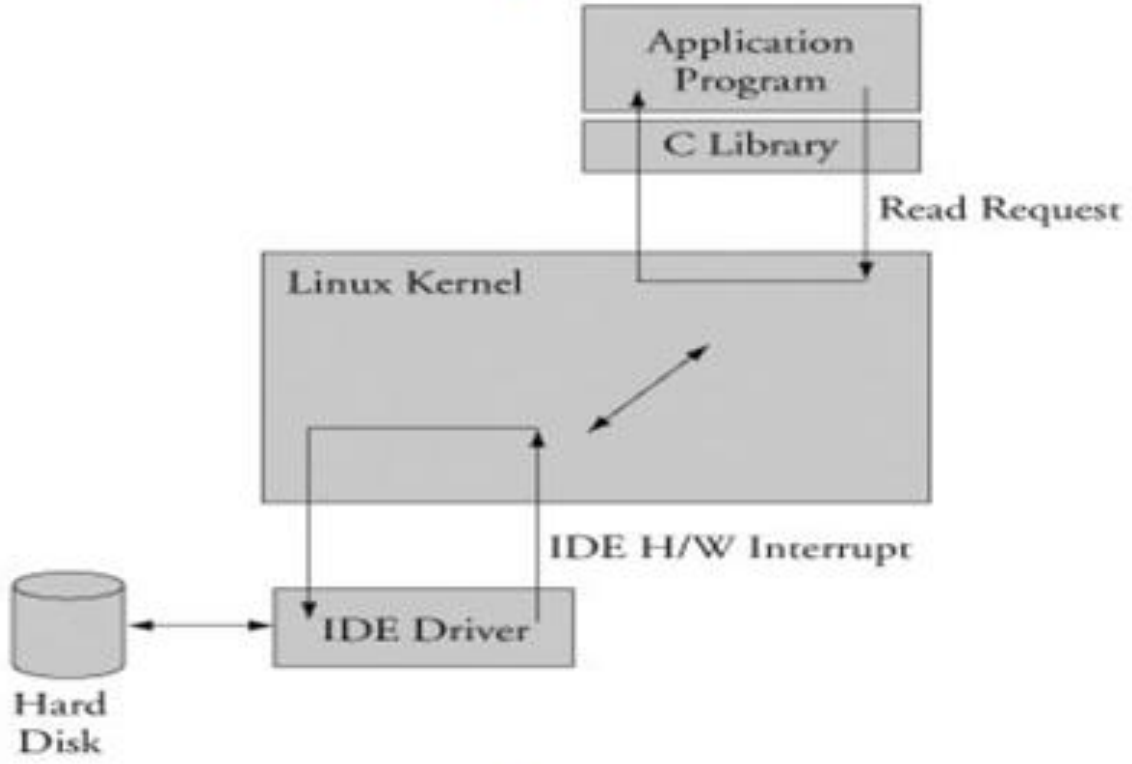
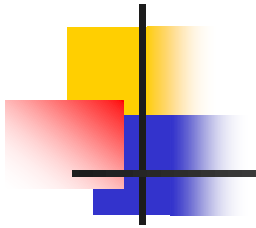


Figure : Simple file read request

UNDERSTAND!!! IMPLEMENT!!!
ANALYZE!!!

Process Virtual Memory



- When a process is spawned for example, when the user types ls at the Linux command prompt the kernel allocates memory for the process and assigns a range of virtual-memory addresses to the process.
- The resulting address values bear no fixed relationship to those in the kernel, nor to any other running process.
- Furthermore, there is no direct correlation between the physical memory addresses on the board and the virtual memory as seen by the process.
- In fact, it is not uncommon for a process to occupy multiple different physical addresses in main memory during its lifetime as a result of paging and swapping.

Cross-Development Environment

Cross-development environment requires that the compiler running on your development host output a binary executable that is incompatible with the desktop development workstation on which it was compiled.

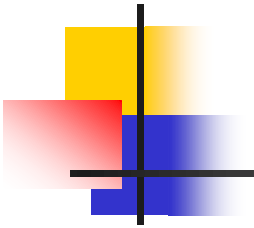
The primary reason these tools exist is that it is often impractical or impossible to develop and compile software natively on the embedded system because of resource (typically memory and CPU horsepower) constraints.

When a given program is compiled, the compiler often knows how to find include files, and where to find libraries that might be required for the compilation to succeed.

e.g. :-

gcc -Wall -o hello hello.c

Compilers have built-in defaults for locating include files.
A similar process exists for the linker to resolve the reference to the external symbol printf( ).
This default behavior is built into the toolchain.
e.g. :- Building an application targeting a Power Architecture embedded system. you will need a cross-compiler to generate binary executables compatible with the Power Architecture processor. If you issue a similar compilation command using your cross-compiler to compile the preceding hello.c example, it is possible that your binary executable could end up being accidentally linked with an x86 version of the C library on your development system, attempting to resolve the reference to printf(). Of course, the results of running this hybrid executable, containing a mix of Power Architecture and x86 binary instructions, are predictable: **crash!**
The solution to this predicament is to instruct the cross-compiler to look in nonstandard locations to pick up the header files and target specific libraries.



Embedded Linux Distributions

•Root file system.

• Startup scripts launch a number of programs and utilities that the system requires.

•These programs often invoke other programs to do specific tasks, such as spawn a login shell, initialize network interfaces, and launch a user's applications.

•All programs has specific requirements (often called dependencies) that must be satisfied by other components in the system.

•Small embedded Linux system needs many dozens of files populated in an appropriate directory structure on a ***root file system***.

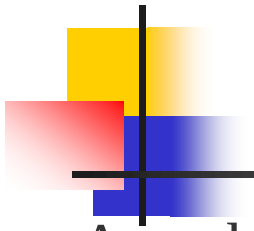
•Packages that are usually grouped by functionality.

•Package manager. **Red Hat's Package Manager (rpm)**

For Red Hat, and Fedora series,

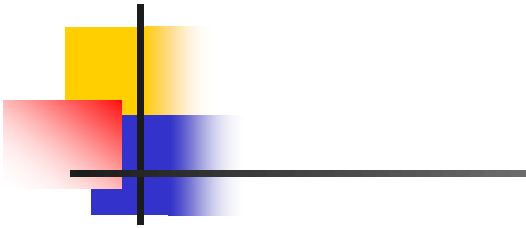
$ rpm –qa

$dpkg -l for Ubauntu



A package can consist of many files
e.g. :- Packages you might find on an embedded Linux distribution, and their purpose:

- initscripts :- contains basic system startup and shutdown scripts.
- apache :- implements the popular Apache web server.
- telnet-server :- contains files necessary to implement telnet server functionality, which allows you to establish telnet sessions to your embedded target.
- glibc: - implements the Standard C library.
- busybox:- contains compact versions of dozens of popular command-line utilities commonly found on UNIX/Linux systems.



• The executable target binaries from an embedded distribution will not run on your PC, but are targeted to the architecture and processor of your embedded system.

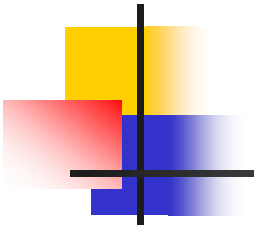
•A desktop Linux distribution tends to have many GUI tools .

•An embedded Linux distribution typically omits these components.

•An embedded distribution typically contains cross tools, as opposed to native tools.

e.g. The gcc toolchain that ships with an embedded Linux distribution runs on your x86 desktop PC but produces binary code that runs on your target system, often a non-x86 architecture.

Many of the other tools in the toolchain are similarly configured:

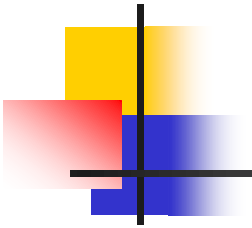


Commercial Linux Distributions

The leading embedded Linux vendors have been shipping embedded http://elinux.org/Embedded_Linux_Distributions.

- **Do-It-Yourself Linux Distributions**

•You can choose to assemble all the components you need for you embedded project on your own.
•You have to decide whether the risks are worth the effort.
•This approach might be a good one for academic project.
•Spend a significant amount of time assembling all the tools and utilities your project needs and making sure they all interoperate.



You need a toolchain. gcc and binutils (binary utilities) are available from www.fsf.org and other mirrors around the world.
Both are required to compile the kernel and user-space applications for your project.
Patches are often required to the most recent “stable” source trees of these utilities, especially when they will be used beyond the x86/IA32 architecture.