Embedded Linux on ARM

Building a custom Embedded OS

Team Emertxe

Introduction

Linux as an Embedded OS

- Open Source & Free Software Fundamentals
- Why to choose Linux
- Architecture
- Choices to Make

Open Source & Free Software Fundamentals

- Free Software – Freedom to Run, Change and Redistribute
- Free Software Licenses – GNU GPL and GNU FDL
- Open Source – Code is open to review but no Freedom

Choosing Linux

- Quality and Reliability of Code
- Availability of Code
- Hardware Support
- Communication Protocols and Software Stds
- Available Tools
- Community Support
- Licensing
- Vendor Independence
- Cost

Quality and Readability of Code

- Modularity and Structure
- Readability of Code
- Extensibility
- Configurable
- Predictability
- Error Recovery
- Longevity

Architecture

- Power PC
  - Intended for PC
  - Have become popular in embedded
- Strong ARM
  - Faster CPU – Higher Performance
  - PDAs, Setup box etc.,
- ARM
  - Suits well for Embedded
  - Include THUMB – reduce code bandwidth
  - High density code than PPC, x86.
- MIPS

Choices to Make

- Which kernel to use?
- Which development environment:
- Which compiler, debugger, dev boards?
- Which drivers and libraries?
- Support and training?

Embedded Development and Environment

Embedded Development and its Environment

- Requirements & Setup
  - Connectivity between Host and Target
    - Serial
    - Network
- The Embedded Environment Tools

Embedded Development and its Environment

- The Embedded Environment Tools
  - Serial Downloding Applications
    - minicom
    - gtkterm etc.,
  - tftp server
- Toolchains (Cross Compiler & Friends)

Embedded Development and its Environment

- Toolchains (Cross Compiler & Friends)
  - gcc
  - binutils

  and many more..
- Building your own Toolchain

Embedded Development and its Environment

- Building your own Toolchain
  - Buildroot
    - Down load buildroot from http://buildroot.uclibc.org
    - tar jxvf buildroot-snapshot.tar.gz
    - cd buildroot/
    - make menuconfig

Understanding Target

Understating Target

- Know your Target Controller
- Hardware Understanding
- Linux Startup Sequence
- Controller's Booting Sequence
- Working on the Board

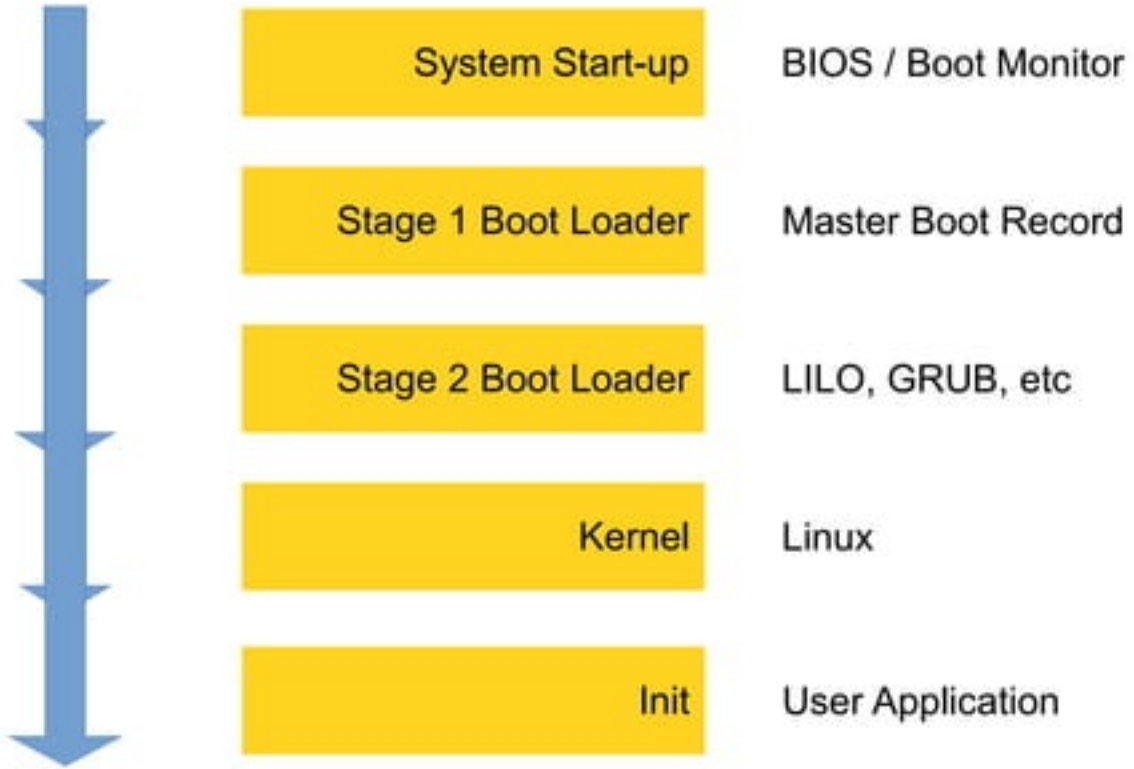
Know your Target Controller

- ARM920T
  - 200MIPS at 180MHz, MMU
  - 16KB Inst & Data Cache
  - ICE
- Memories
  - 128 ROM, 16 KB SRAM
  - EBI
  - SDRAM, SM,CF, NOR, NAND
- Ethernet MAX 10/100
- USB 2.0 FS Host
- USB 2.0 FS Device
- MCI
- SSC
- USART
  - Smart Card
  - RS485, RS232
  - IrDA
  - Modem
- I2C, SPI
- Debug Unit

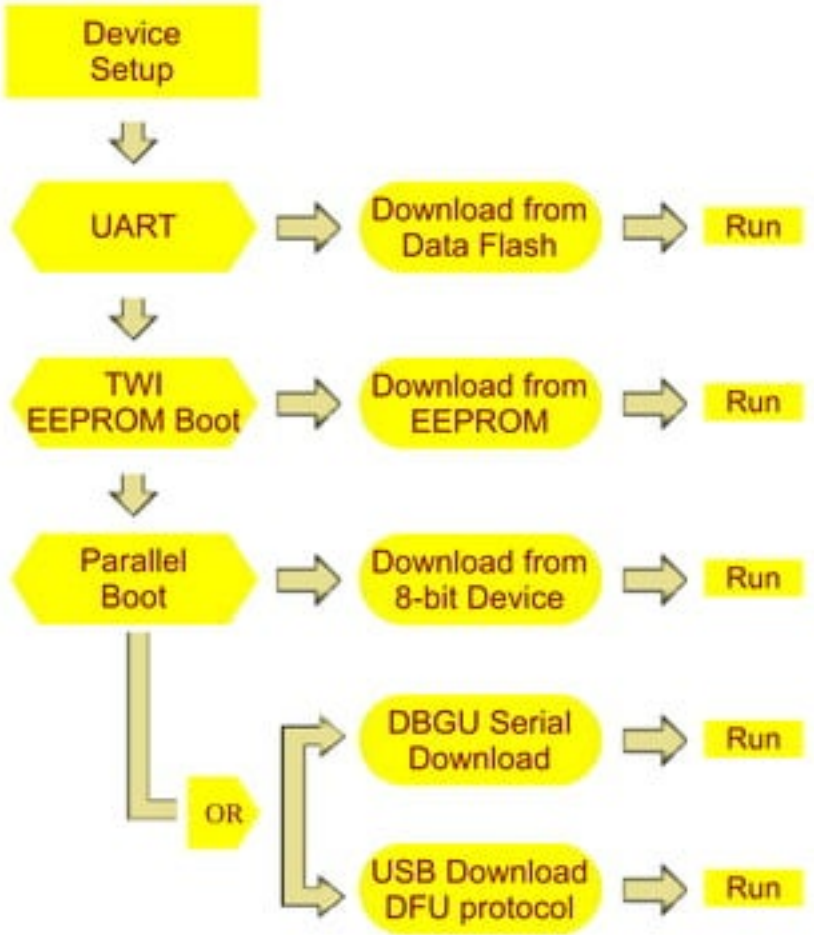
Target Board
Hardware Understanding

- Schematic
- Understand -
  - Memory mappings
  - Peripheral Mappings etc.,

Linux Booting Sequence



Controller's Booting Sequence



Hands on Target

- Factory Restoration
- Board Bring Up

Hands on Target
Factory Restoration

- Erasing the following from the Target
  - Kernel
  - U-Boot

Hands on Target
Board Bring Up

- Boot Up loader
- RAM Monitor
- Stage 1 Boot Loader
- Stage 2 Boot Loader

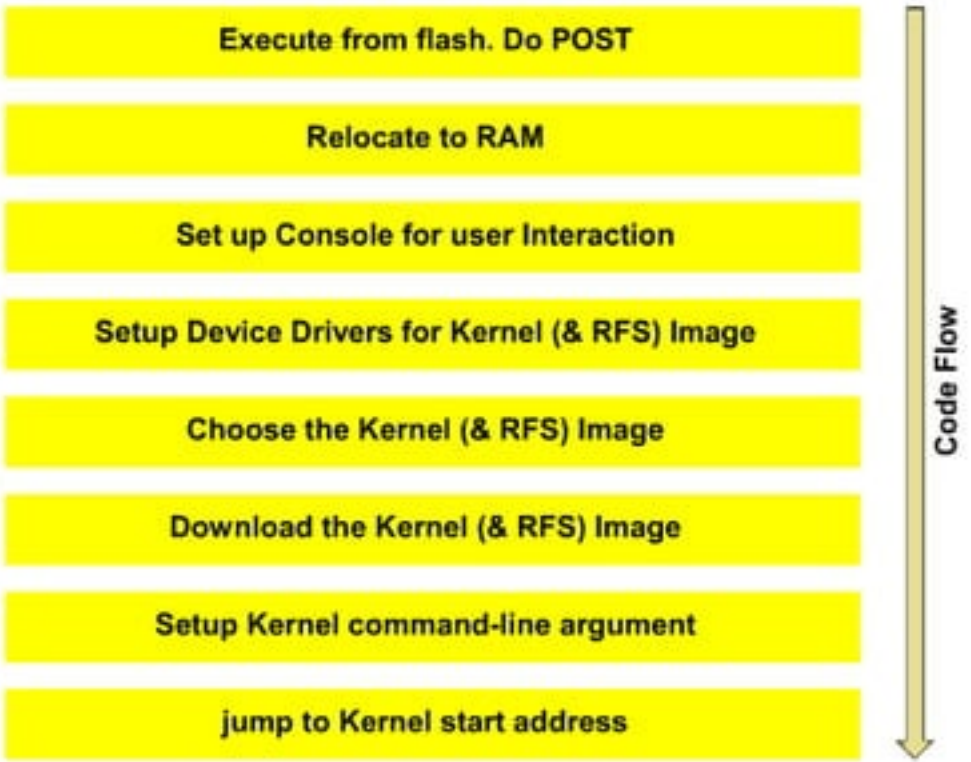
Stage 1 Boot Loader

- Pointer to Stage 2 Boot Loader

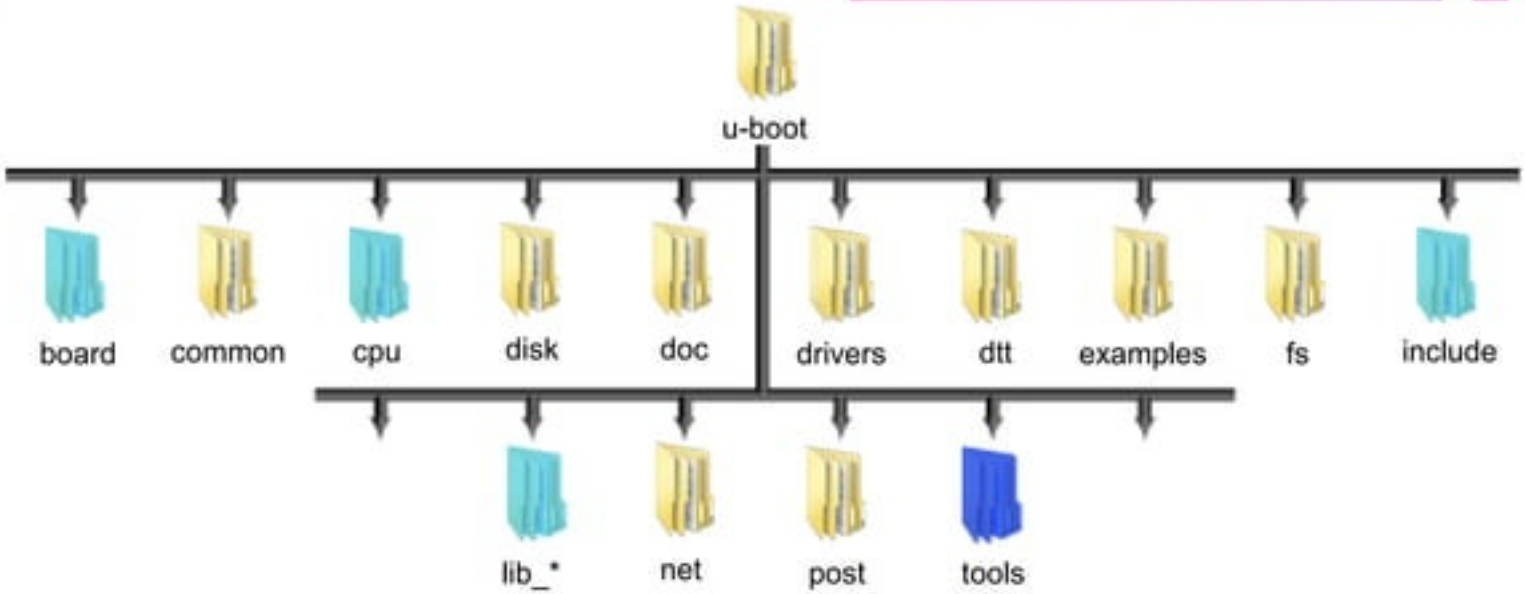
Stage 2 Boot Loader

- Pointer to Kernel Image
- We use u-boot as S2BL
- u-boot's responsibility
- u-boot

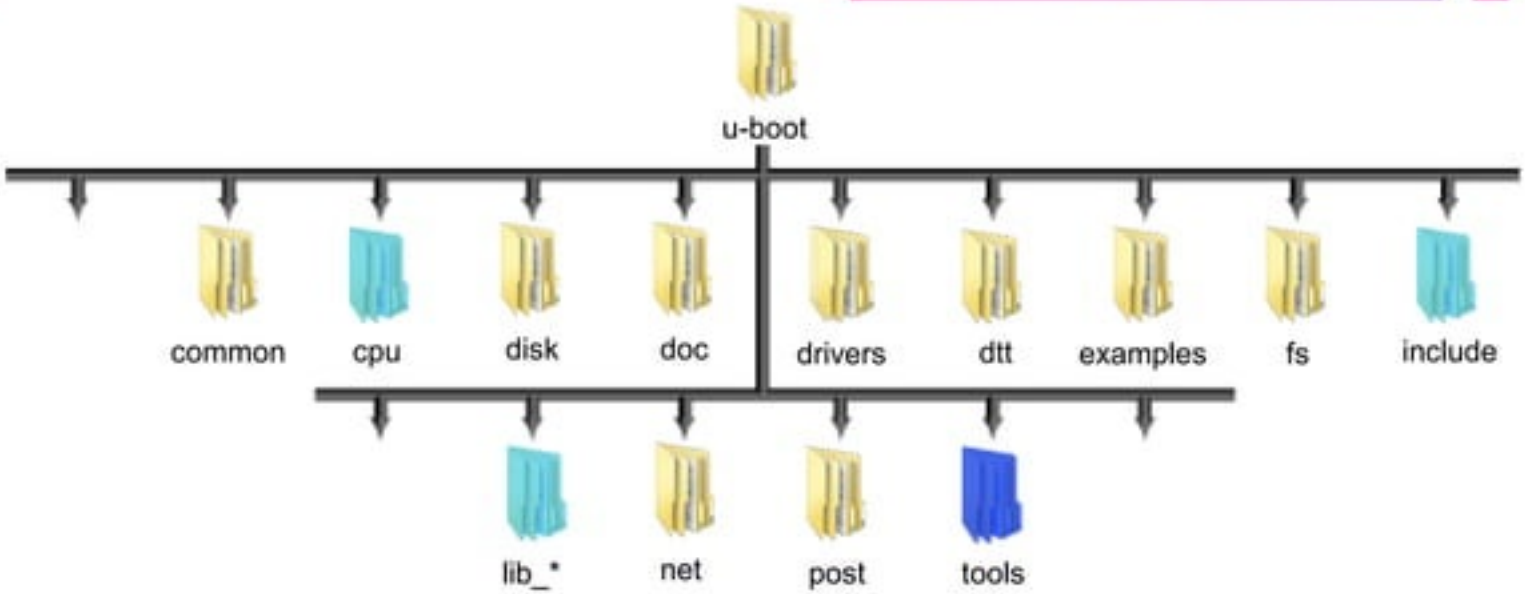
What does u-boot do?



u-boot



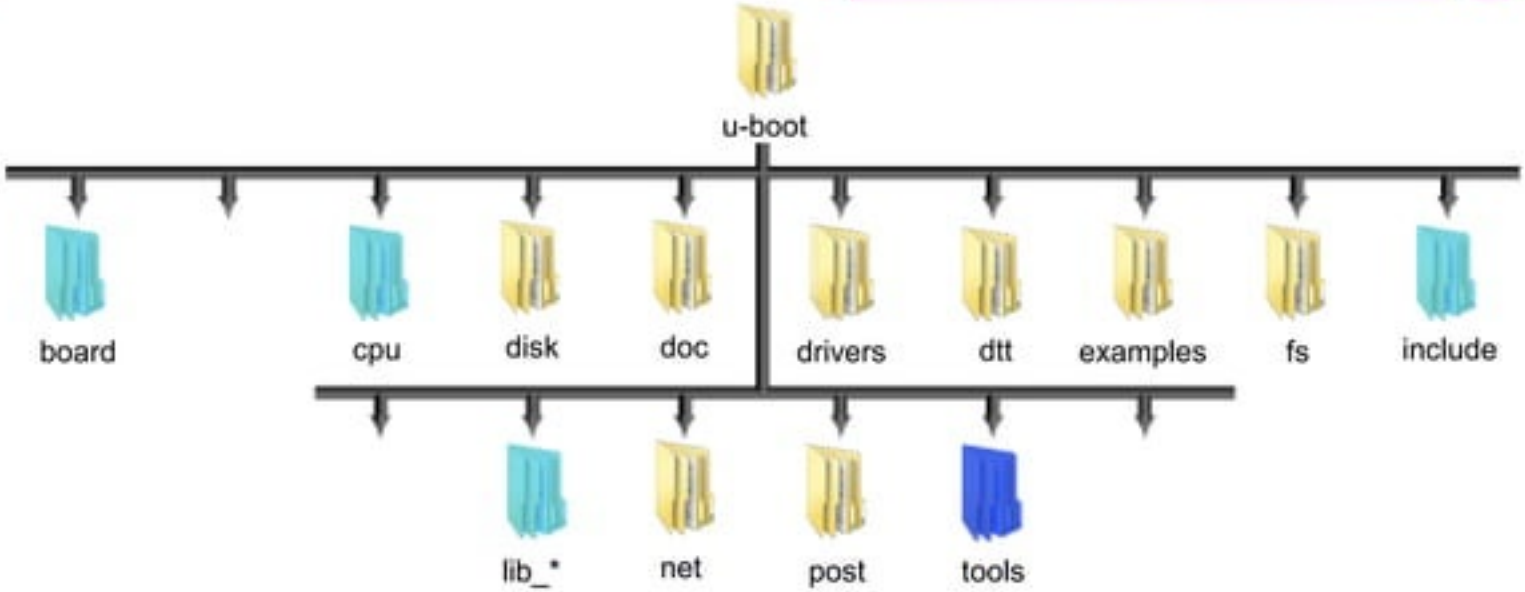
u-boot



board

- Platform, board level files. Eg, atmel, icecube,oxc etc.,
- Contains all board specific initialization
  - <boardname>/flash.c
  - <boardname>/<boardname>_emac.c
  - <boardname>/<boardname>.c
  - <boardname>/soc.h
  - <boardname>/platform.S

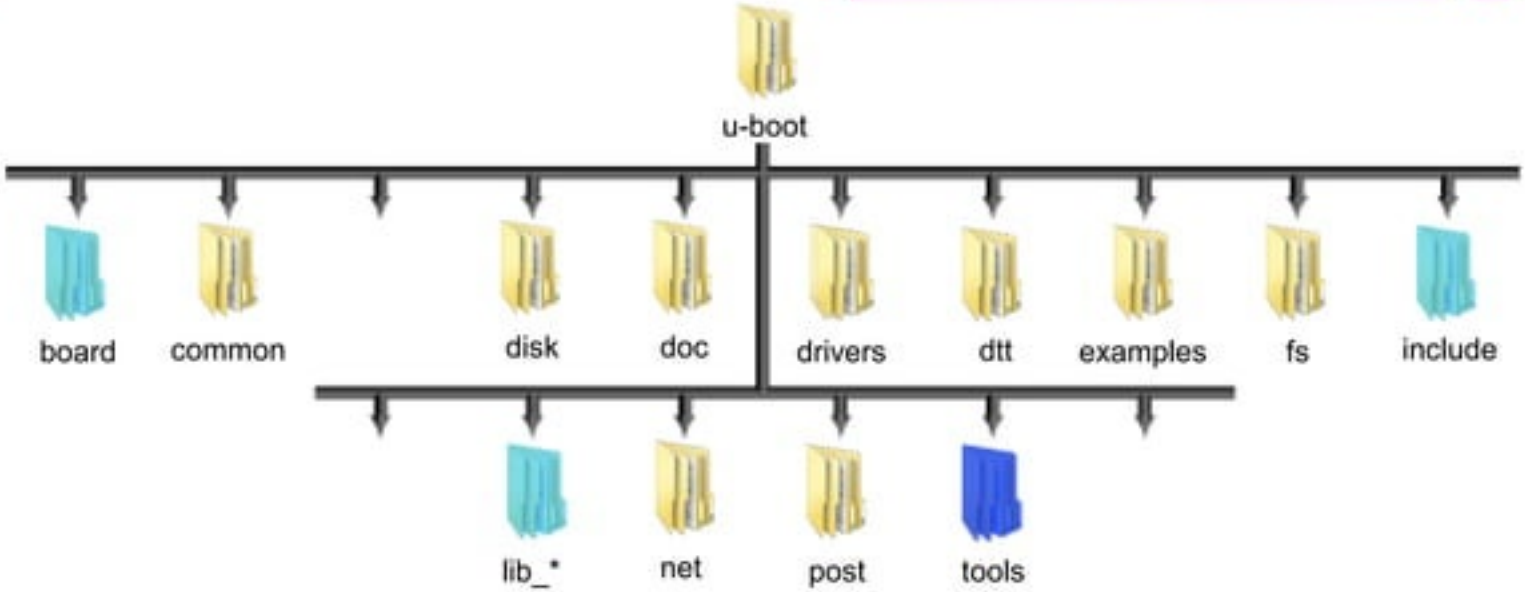
u-boot



common

- All acchitecture independent functions
- All the commands

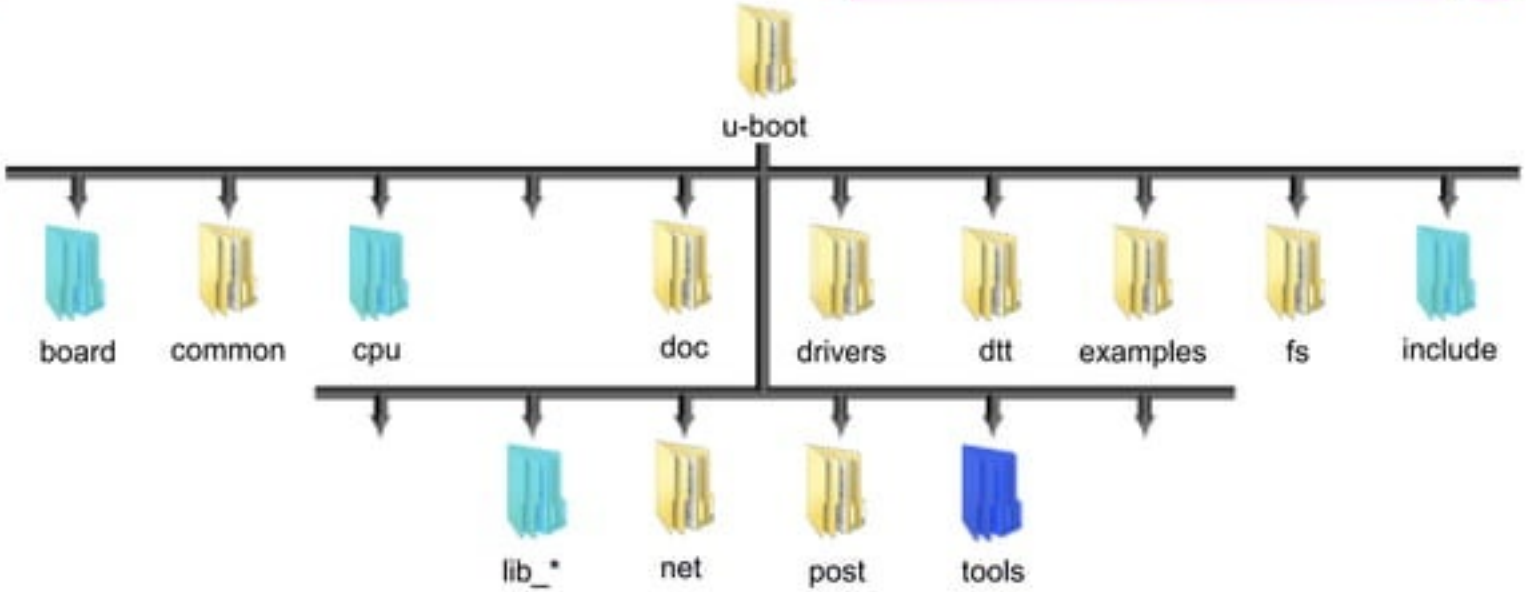
u-boot



cpu

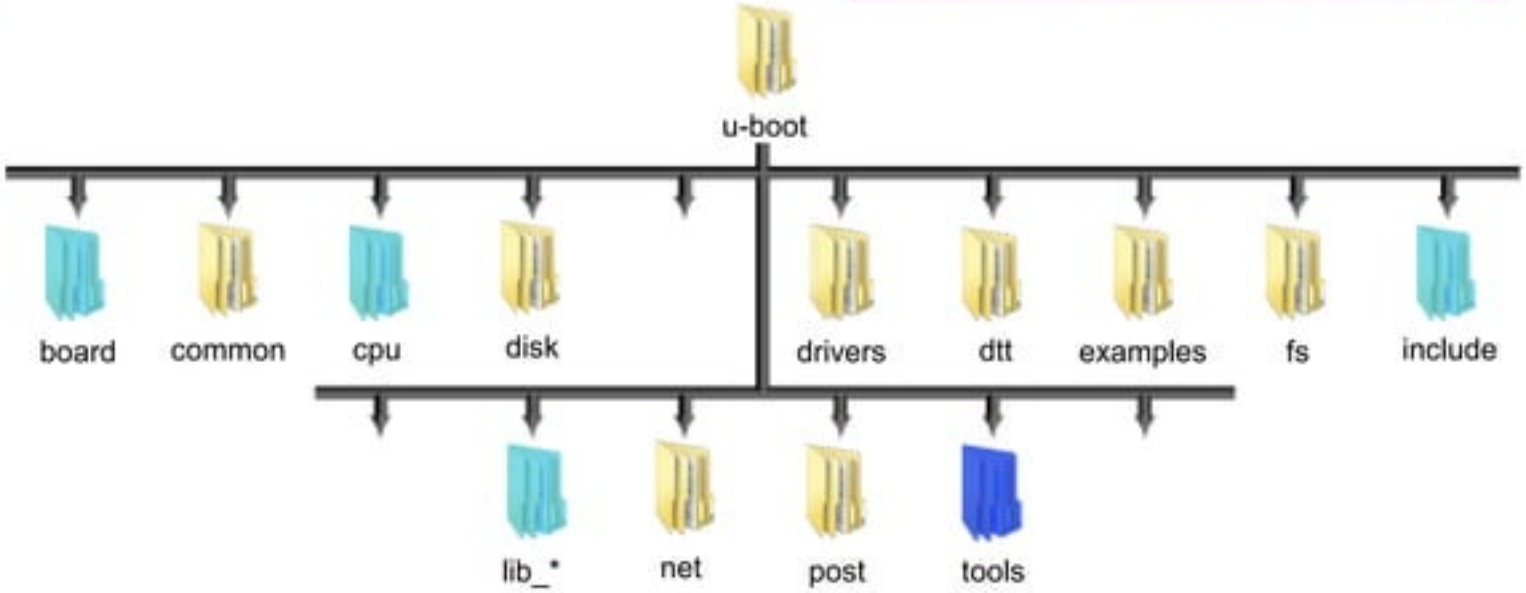
- CPU specific information
  - <core>/cpu.c
  - <core>/interrupt.c
  - <core>/start.S

u-boot



- Partition and device information for disks

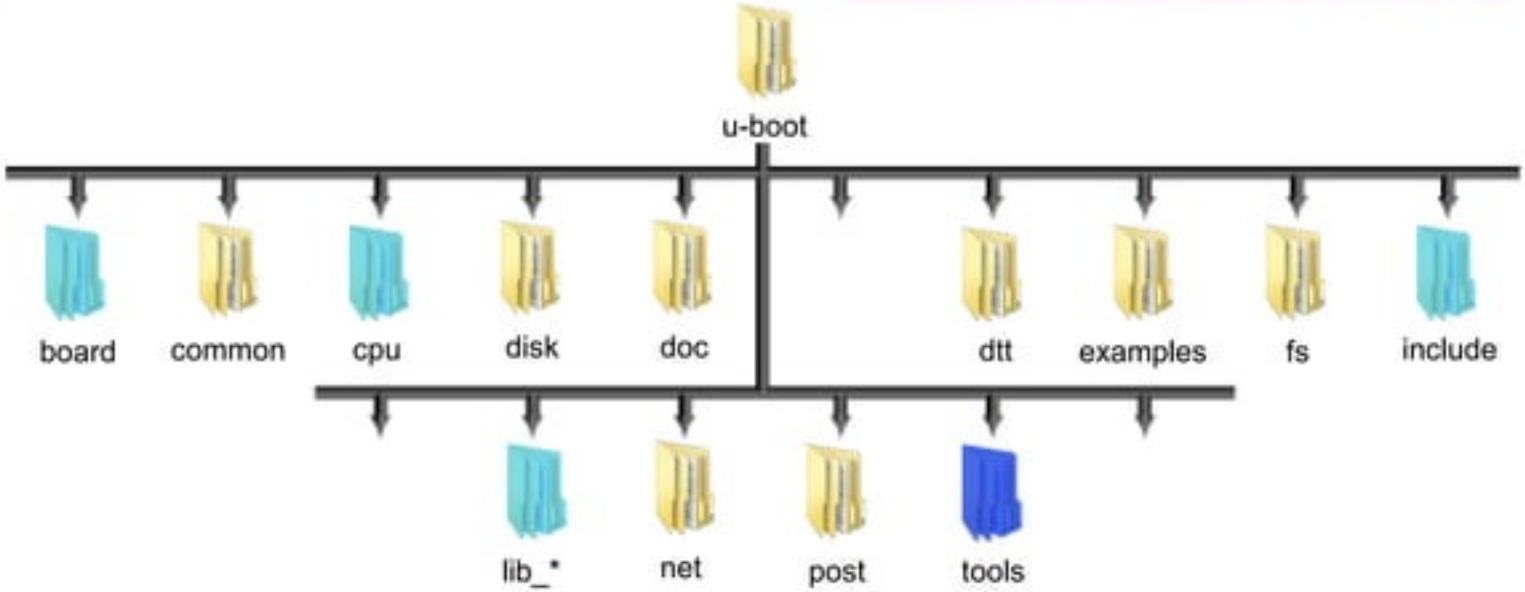
u-boot



doc

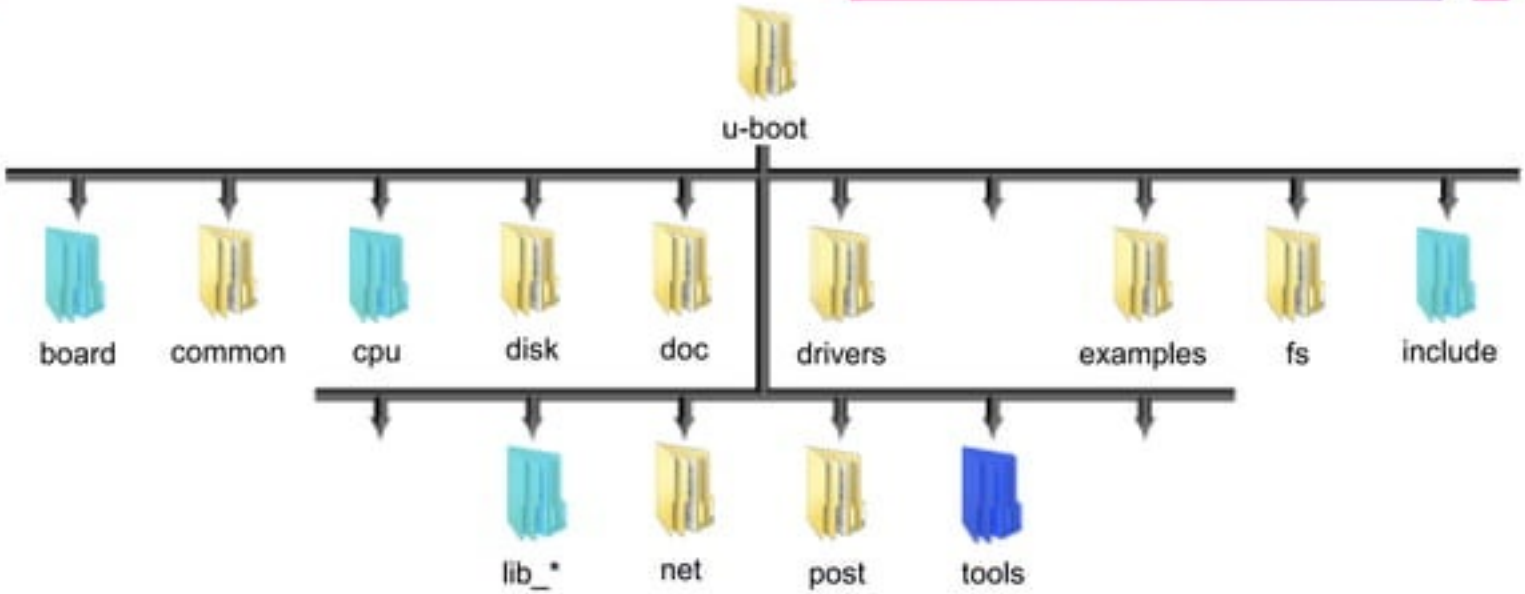
- You can find all the readme files here

u-boot



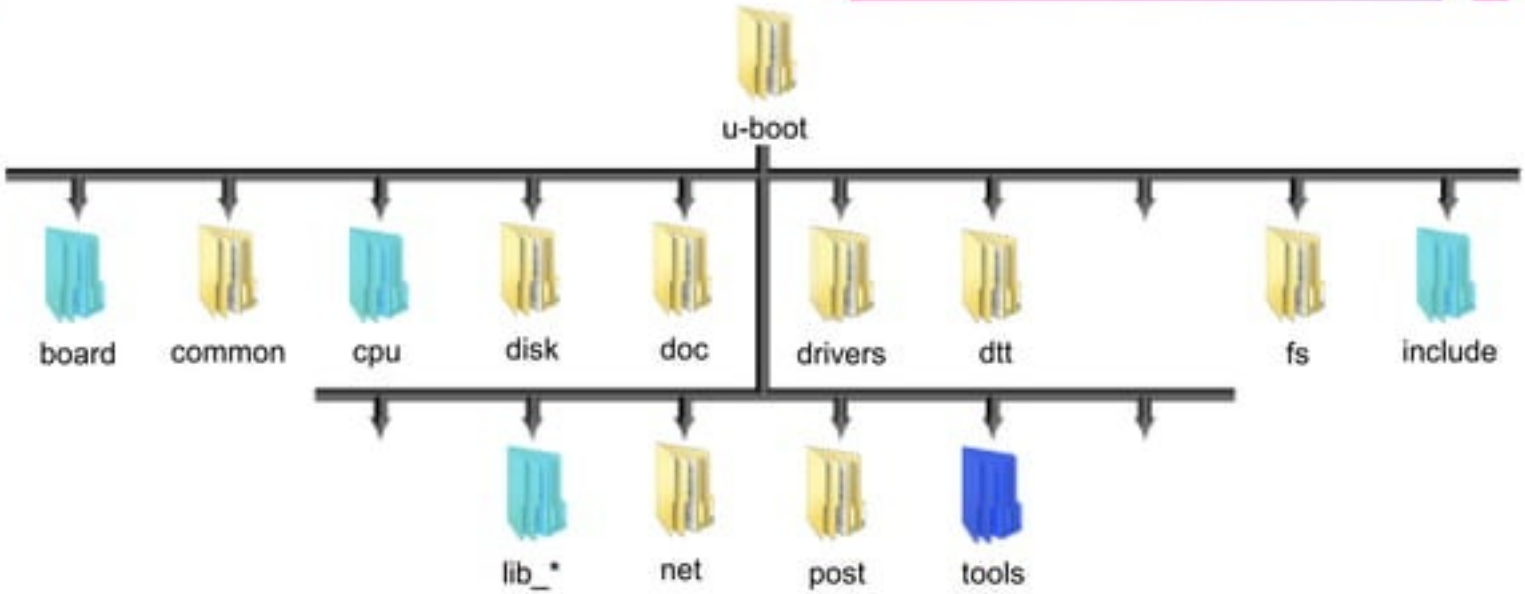
- Various device drivers files

u-boot



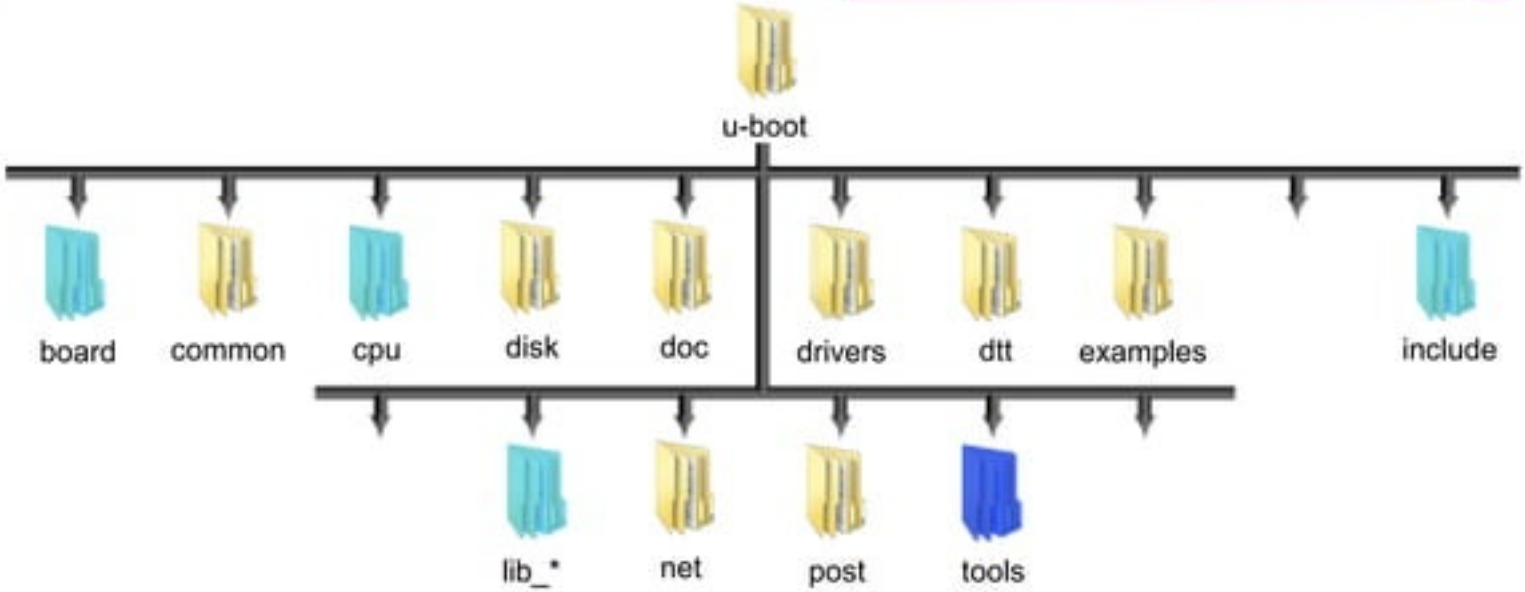
- Digital Thermometer and Thermostat drivers

u-boot



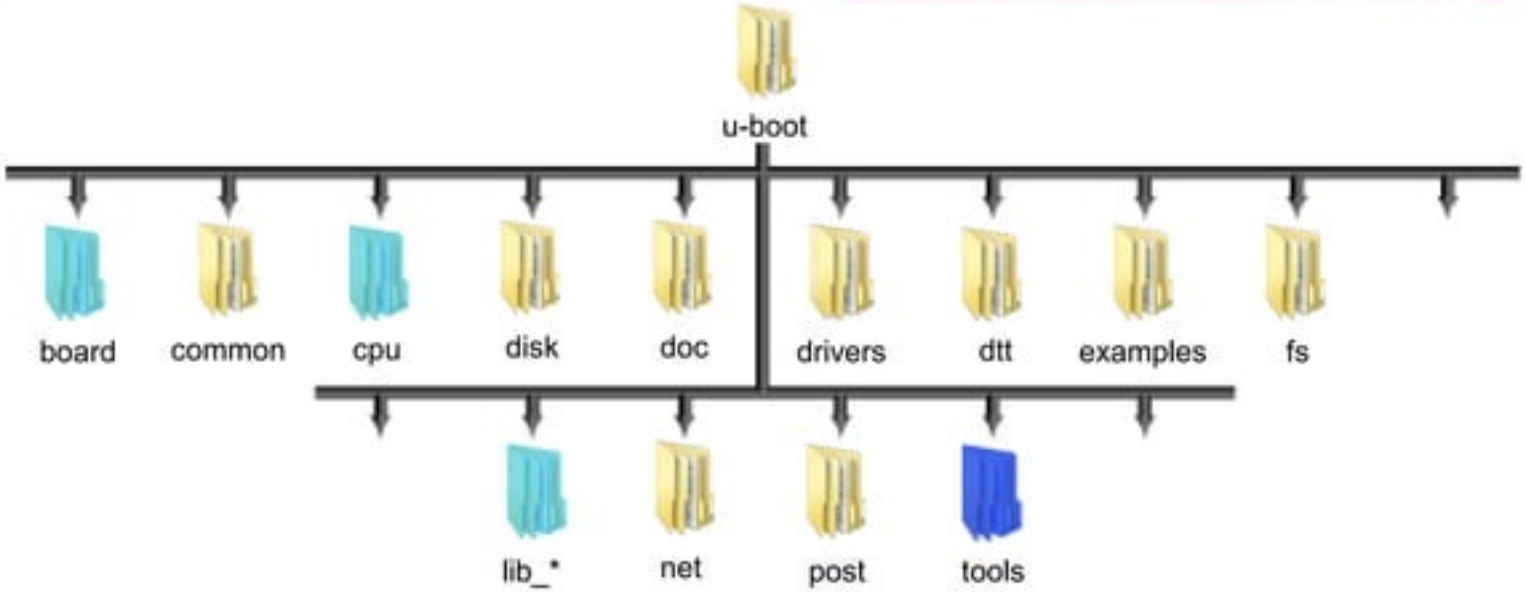
- Example code for standalone application

u-boot



- File system directories and codes

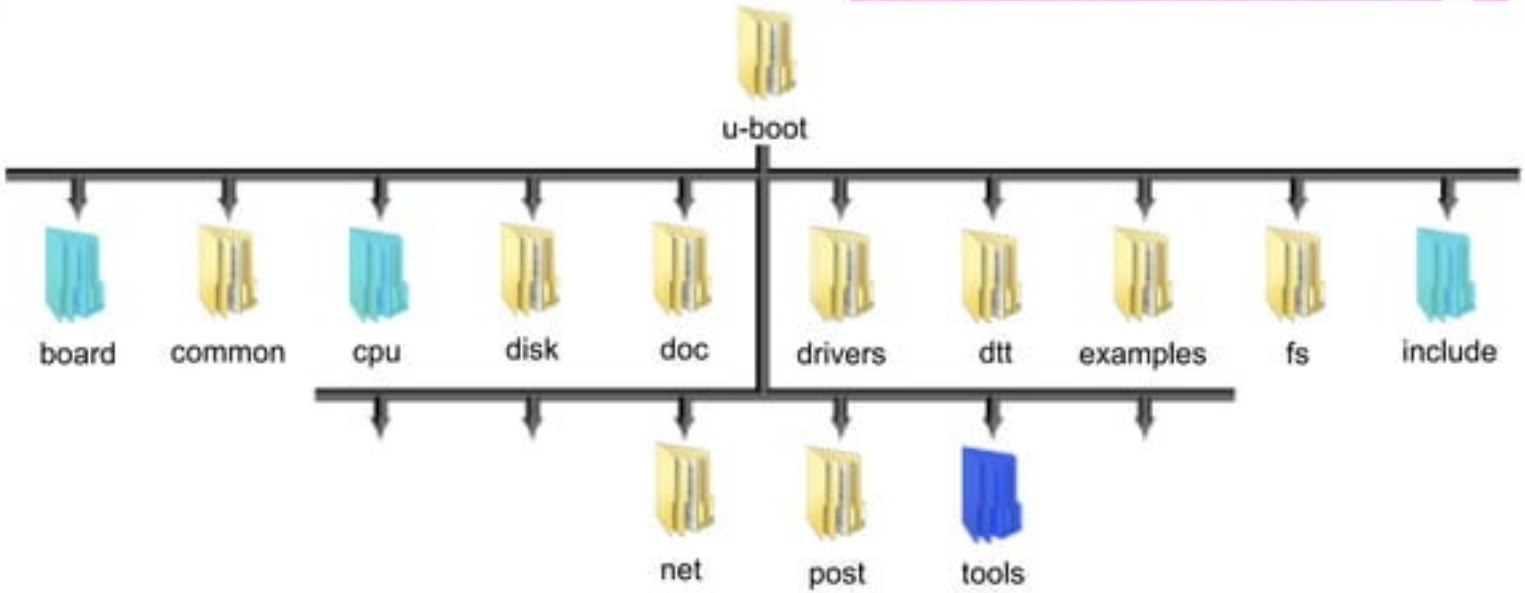
u-boot




include

- Various header files
  - configs/<boardname>.h
  - <core>.h

u-boot

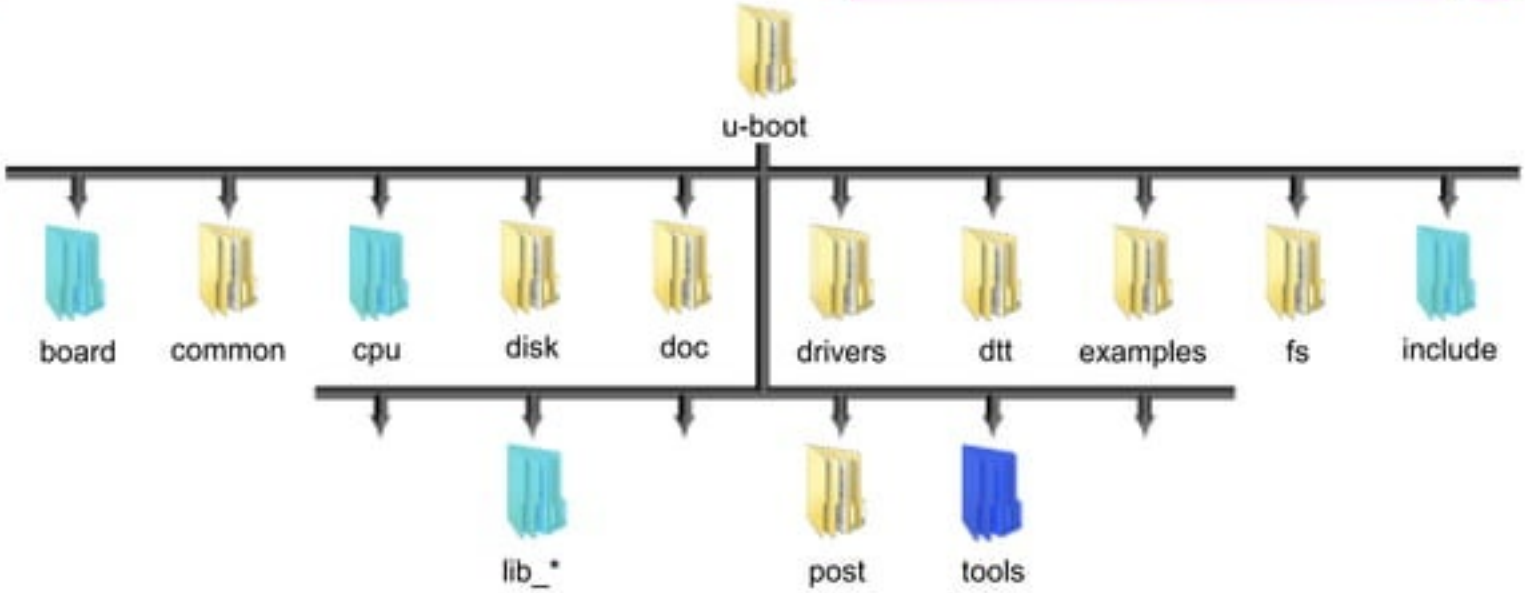


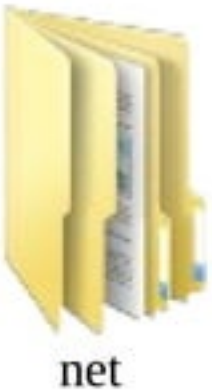
lib_*

- Processor specific libraries
  - board.c
  - <arch>linux.c
  - div0.c

EMERTXE

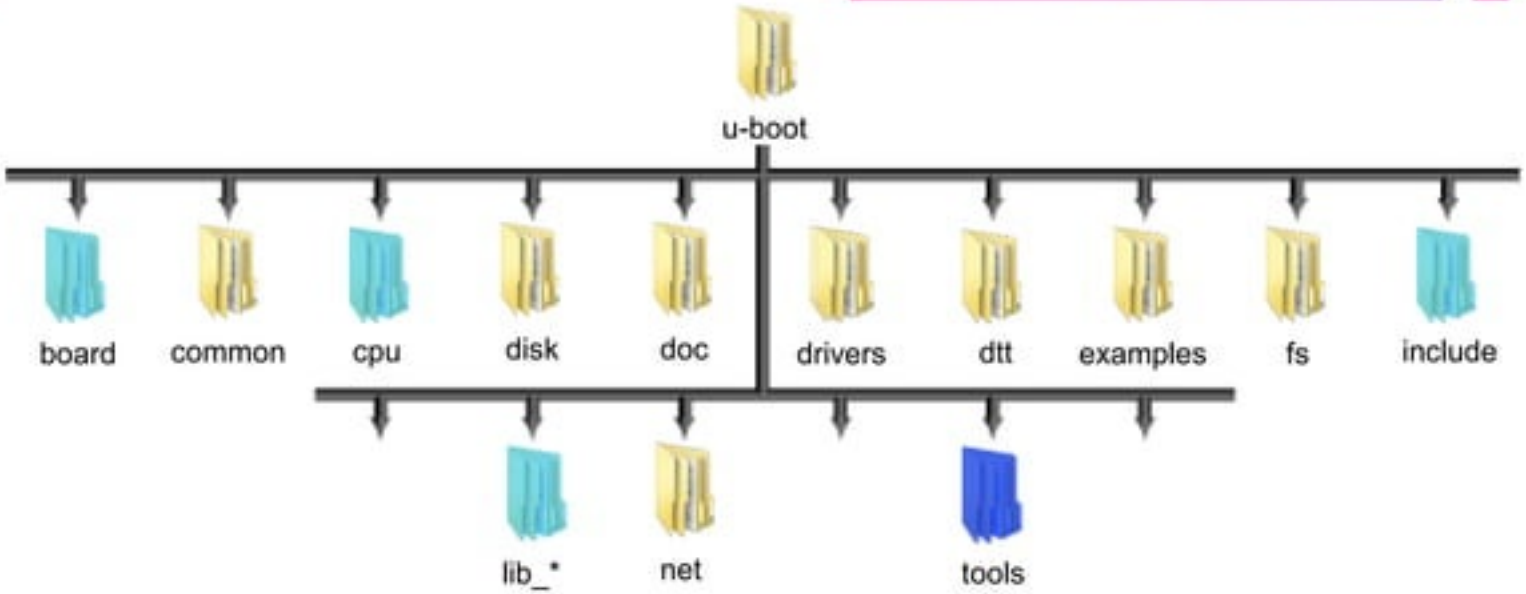
u-boot





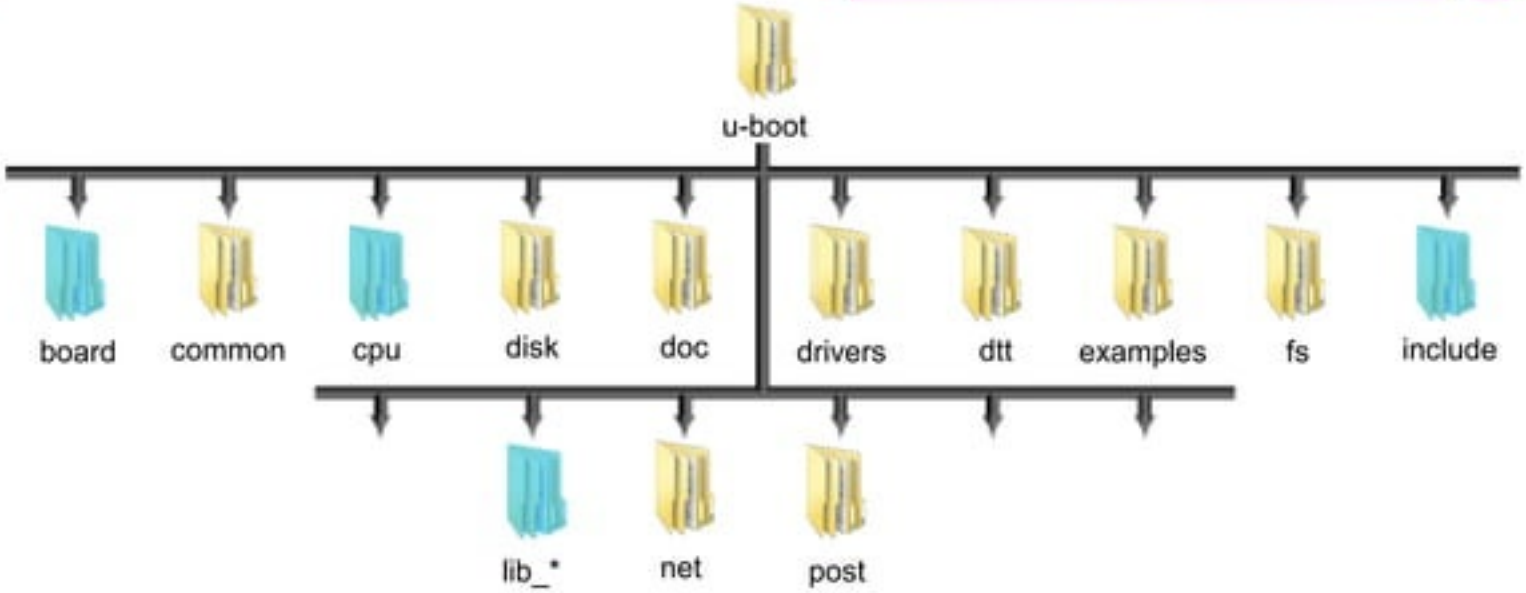
- Networking retated files.

u-boot



- Power On Self Test

u-boot



tools

- Various tools directories and files

Building Embedded Linux

Embedded Kernel

- Configuring the Kernel
- Configuration methods
- Kernel Image and its Arguments
- Booting the Kernel and Init
- File Systems

Configuration Methods

- make config
- make oldconfig
- make menuconfig
- make xconfig
- Building kernel

Configuration Methods

- make kb9202_defconfig
- make menuconfig
- make

Kernel Image and its Arguments

- Creating linux.bin – arm-linux-objcopy
- Creating kernel image - mkimage

Creating linux.bin

- arm-linux-objcopy -O binary vimlinux linux.bin

Creating kernel image

- mkimage -A arm -O linux -T kernel -C none -a 20008000 -e 20008000 -n “Embedded Linux” -d linux.bin uImage.arm

Stay connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046
T: +91 80 6562 9666
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

EMERTXE

Thank You

Buildroot
Introduction

- Can generate an (e)glibc or uClibc cross-compilation toolchain, or re-use your existing glibc, eglibc or uClibc cross-compilation toolchain
- Supports several hundreds of packages for userspace applications and libraries
- http://buildroot.uclibc.org

Buildroot
Configuration

Buildroot
Configuration

- Download buildroot package from http://buildroot.uclibc.org
- Untar the package and change directory to buildroot

```
$ tar xvf buildroot-<year>-<month>.tar.bz2
```

```
$ cd buildroot-<year>-<month>
```

- Buildroot supports Linux kernel like configuration options like menuconfig, xconfig etc.,
- To configure type

```
$ make menuconfig
```

- You should get a curses based configurator

Buildroot
Configuration

- Select the target architecture you want to work with
- You may select the toolchain components like
  - kernel headers
  - binutils
  - uclibc
  - gcc etc.,
- You can ignore selecting these components buy selecting the target architecture, but the default selected components would be used while building


Buildroot
Building

Buildroot
Building

- To start the build process just type

  ```
  $ make
  ```

- Make sure you don't use **`make -jN`** option. Instead you can use BR2_JLEVEL option to run compilation of each individual package with **`make -jN`**
- BR2_JLEVEL can be set while configuration at

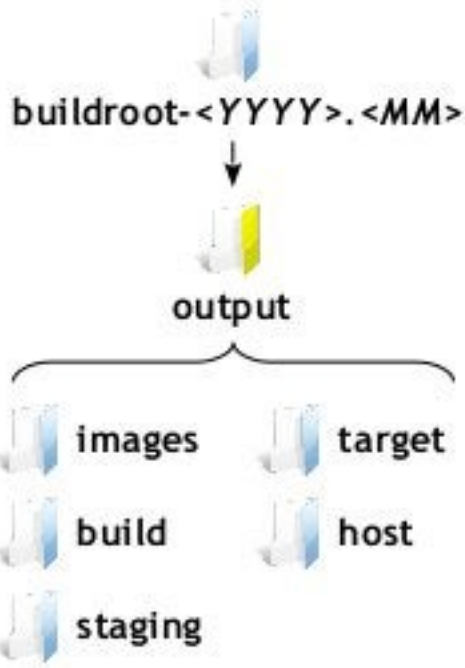
  Build options → Number of jobs to run simultaneously

Buildroot
Building

- The make command will generally perform:
  - Download source files (as required)
  - Configure, build and install the cross compilation toolchain, or simply import an external toolchain
  - Configure, build and install selected target packages
  - Build kernel, bootloader images if selected
  - Create the root filesystem in selected format
- Buildroot output is stored in a single directory named output/ which will be found in the root directory of buildroot

Buildroot

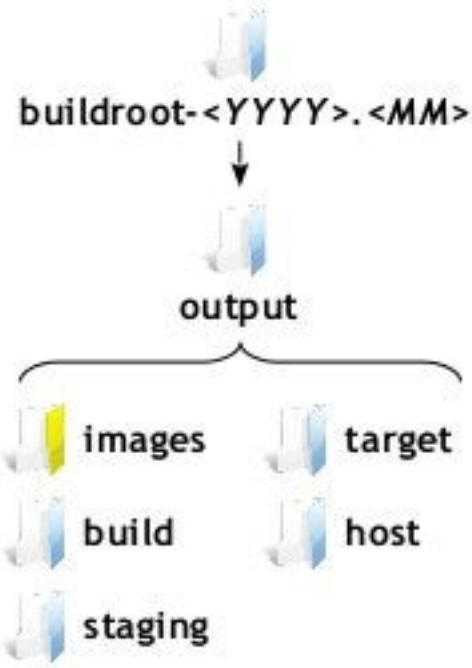
Building - Output



- The left side shows contents the output directory of the buildroot folder
- This directory contains several subdirectories
- The following slides discuss its contents

Buildroot

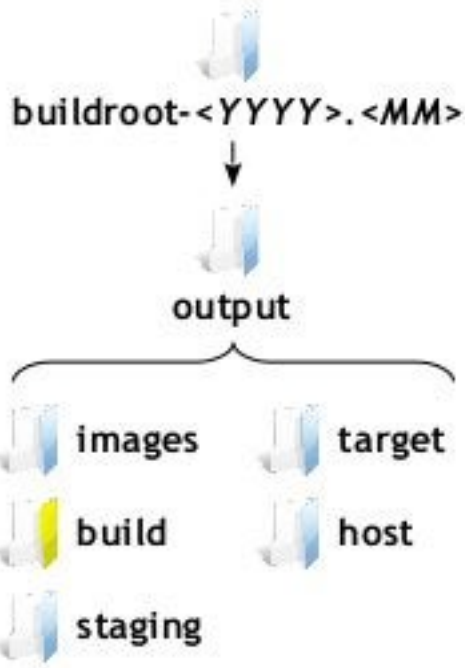
Building - Output



- All the built images like kernel, bootloader, filesystem are stored here
- These are the files you need to put on the target system

Buildroot

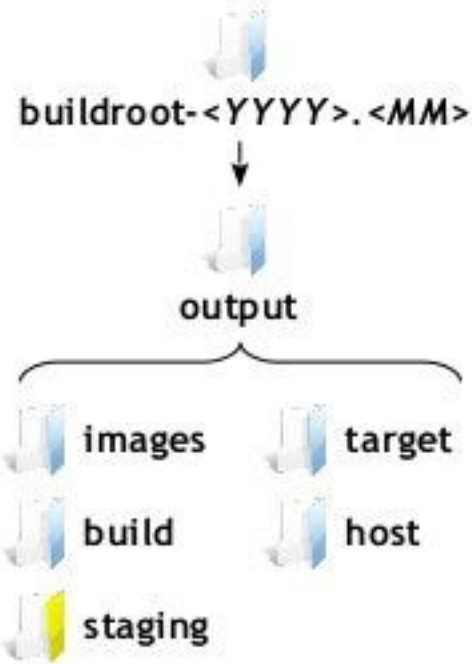
Building - Output



- All the components are built here (this includes tools needed by Buildroot on the host and packages compiled for the target)
- Contains one sub directory for each of these components

Buildroot

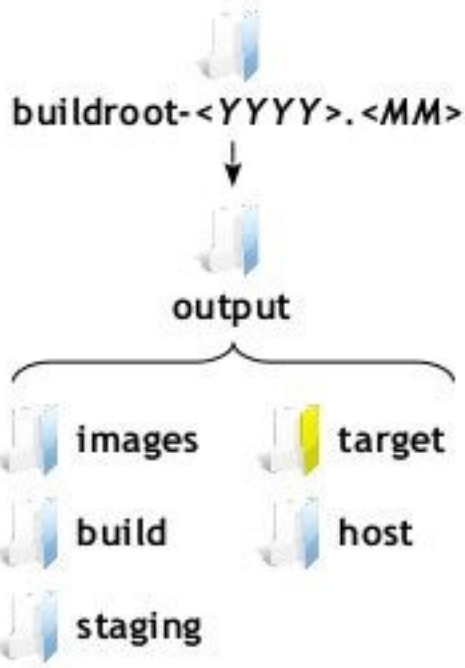
Building - Output



- Contains a hierarchy similar to a root filesystem hierarchy
- Contains the headers and libraries of the cross-compilation toolchain and all the userspace packages selected for the target
- This directory is not intended to be the root filesystem for the target: it contains a lot of development files, unstripped binaries and libraries that make it far too big for an embedded system
- These development files are used to compile libraries and applications for the target that depend on other libraries

Buildroot

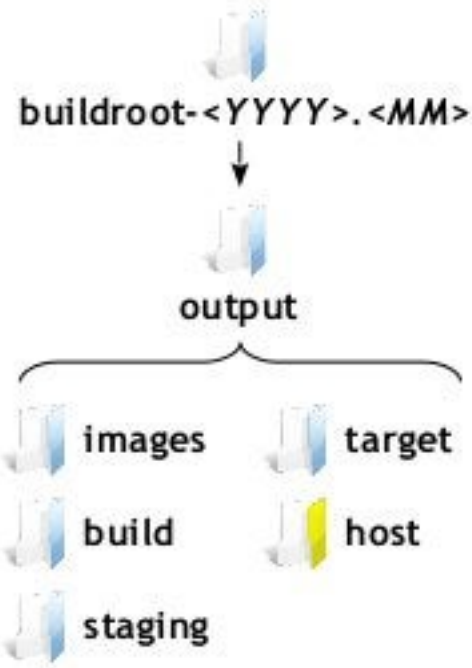
Building - Output



- Contains *almost* the complete root filesystem for the target: everything needed is present except the device files in /dev/ and It doesn't have the correct permissions
- Therefore, this directory **should not be used on your target**
- Instead, you should use one of the images built in the **images/** directory
- If you need an extracted image of the root filesystem for booting over NFS, then use the tarball image generated in **images/** and extract it as root
- Contains only the files and libraries needed to run the selected target applications: the development files (headers, etc.) are not present, the binaries are stripped

Buildroot

Building - Output



- Contains the installation of tools compiled for the host that are needed for the proper execution of Buildroot, including the cross-compilation toolchain

Buildroot

More Infos!!

- http://buildroot.uclibc.org/downloads/manual/manual.html

Target Overview

EmxDev Boards - EmxARM9A03

Target Overview

- Know your Target Controller
- Target Architecture
- Target Board

EMERTXE

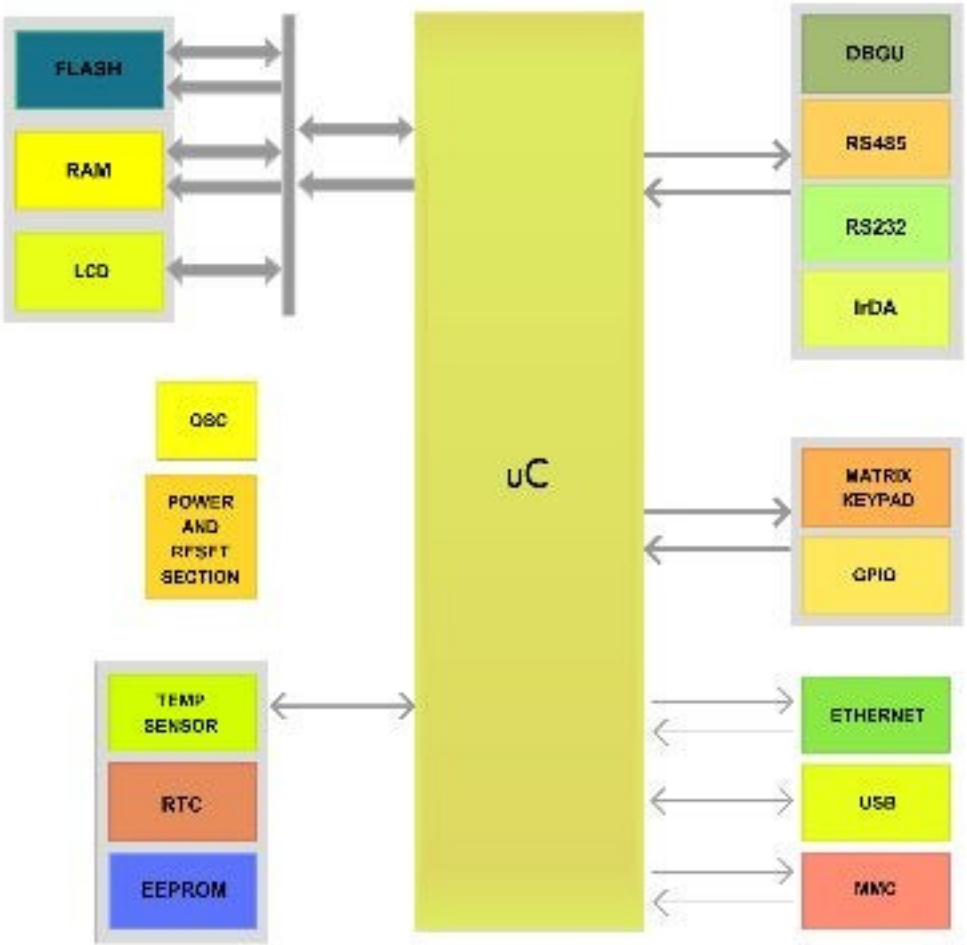
EmxDev Boards - EmxARM9A03

Target Overview - Target Controller

- **ARM920T**
  - 200MIPS at 180MHz, MMU
  - 16KB Inst & Data Cache
  - ICE
- **Memories**
  - 128 ROM, 16 KB SRAM
  - EBI
  - SDRAM, SM,CF, NOR, NAND
- **Ethernet MAX 10/100**
- **USB 2.0 FS Host**
- **USB 2.0 FS Device**
- **MCI**
- **SSC**
- **USART**
  - Smart Card
  - RS485, RS232
  - IrDA
  - Modem
- **I2C, SPI**
- **Debug Unit**

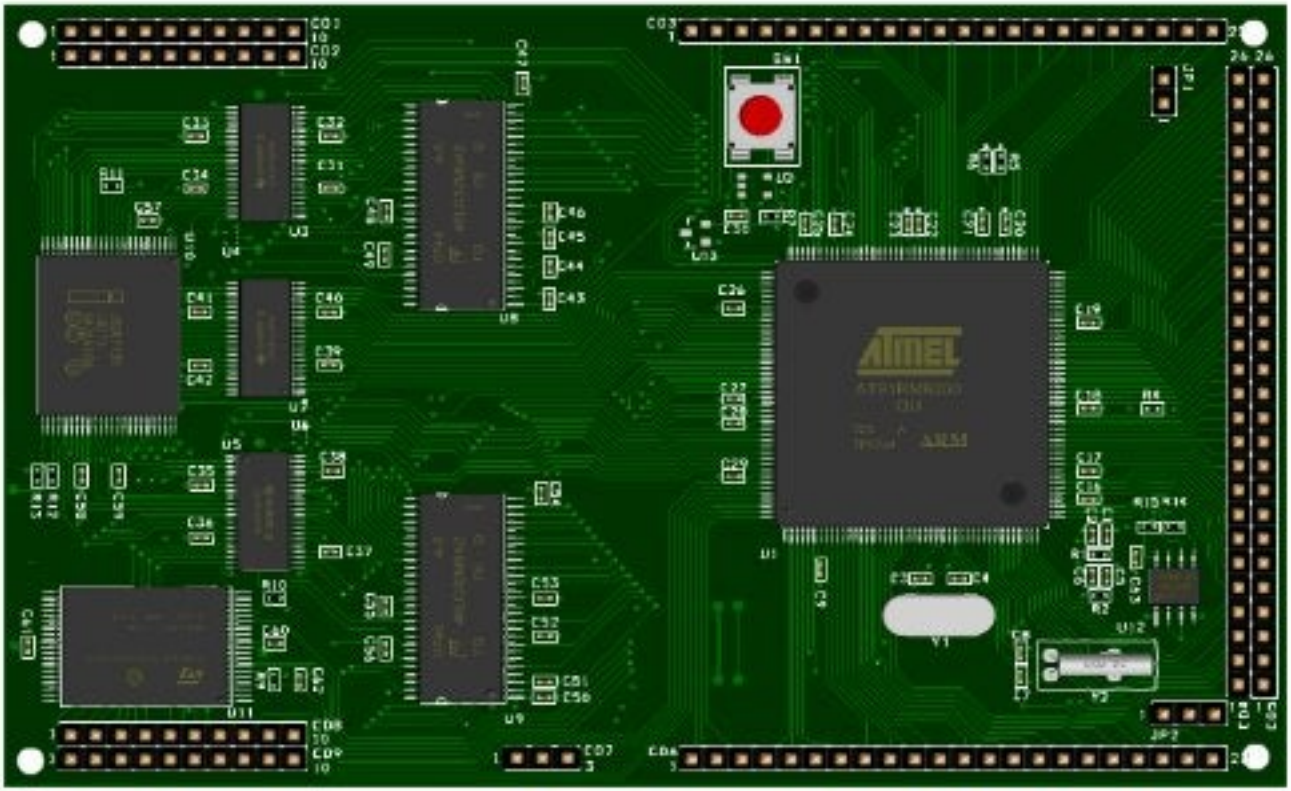
EmxDev Boards - EmxARM9A03

Target Overview - Target Architecture



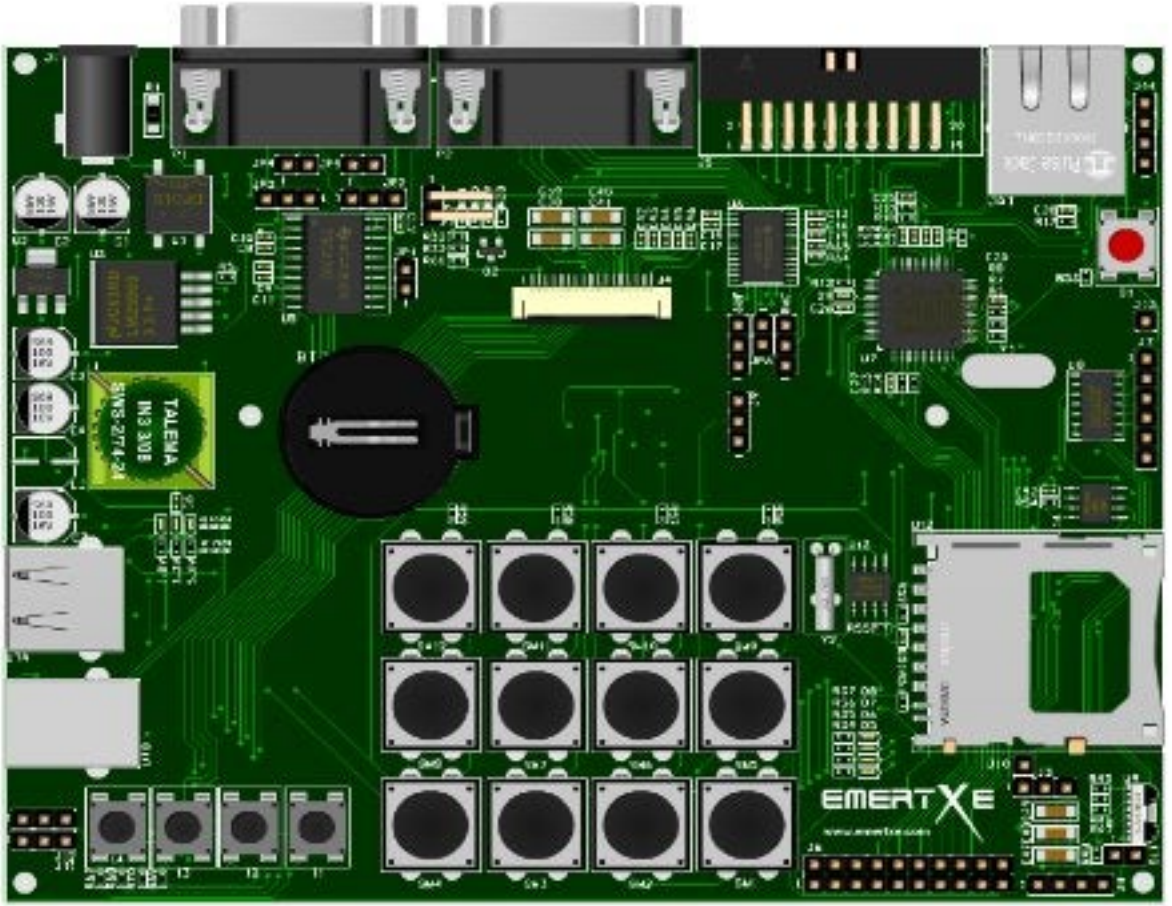
EmxDev Boards – EmxARM9A03

Target Overview - Base Board



EmxDev Boards – EmxARM9A03

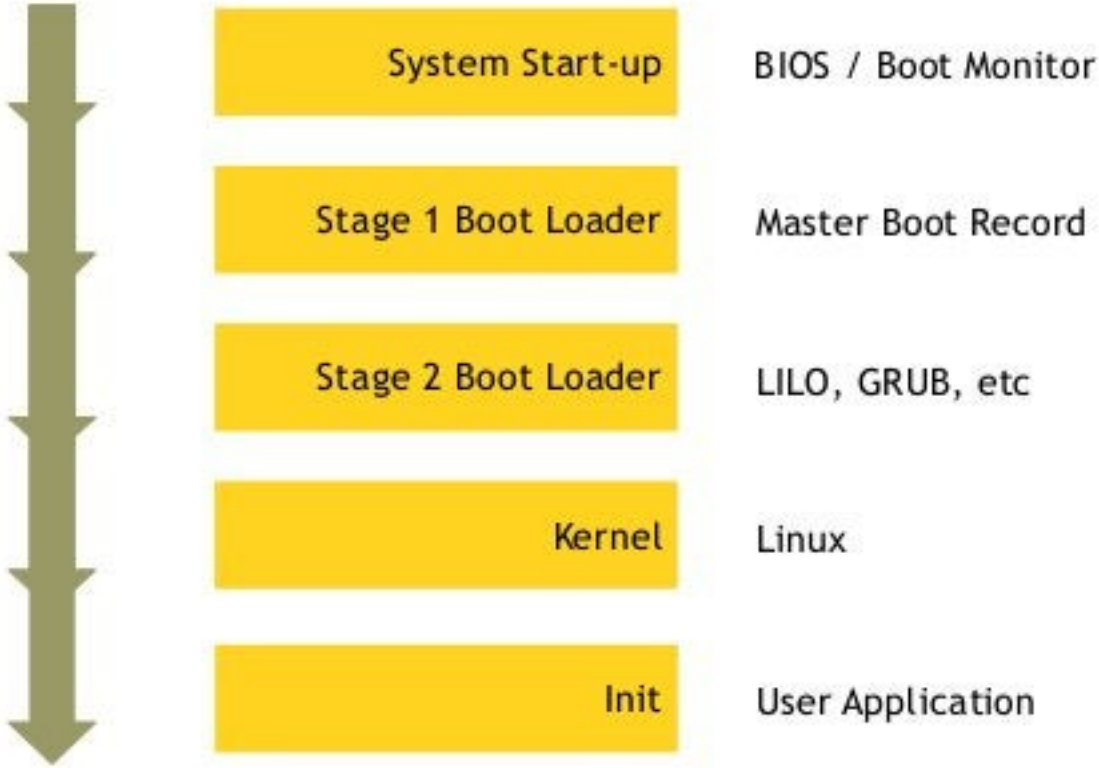
Target Overview - Application Board



Booting Sequences

Booting Sequence

Linux as general



EmxDev Boards - EmxARM9A03

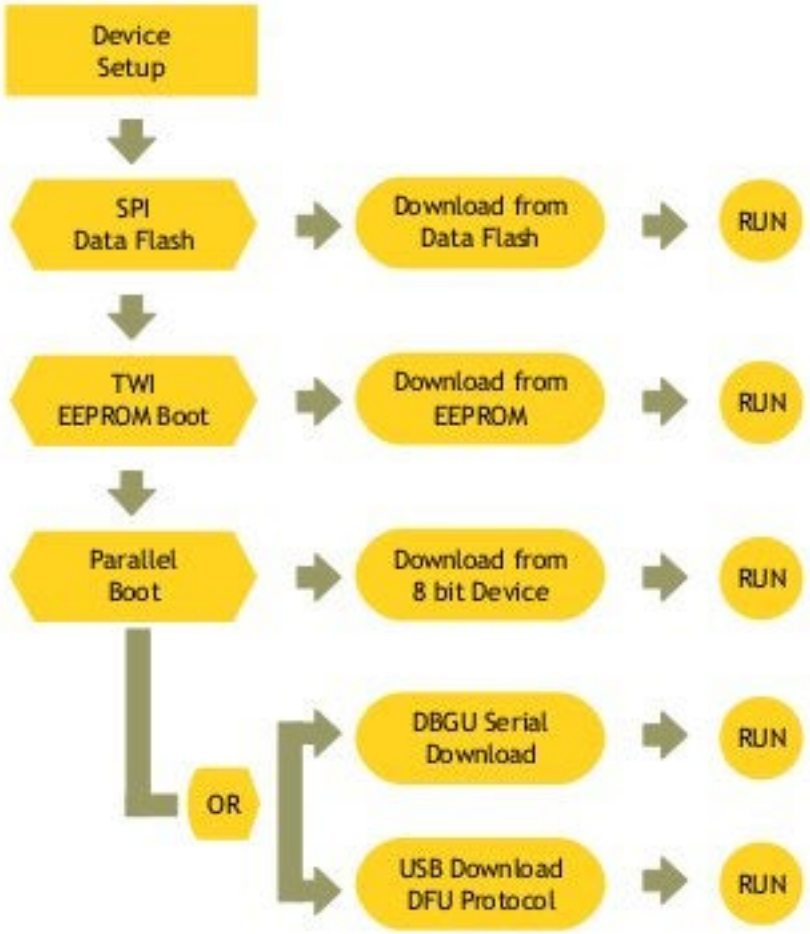
Booting Sequence

- Controller's Booting Sequence
- Boot Loader Stages

EMERTXE

EmxDev Boards – EmxARM9A03

Booting Sequence - Controller's Boot Sequence



ΣMERTXE

EmxDev Boards - EmxARM9A03

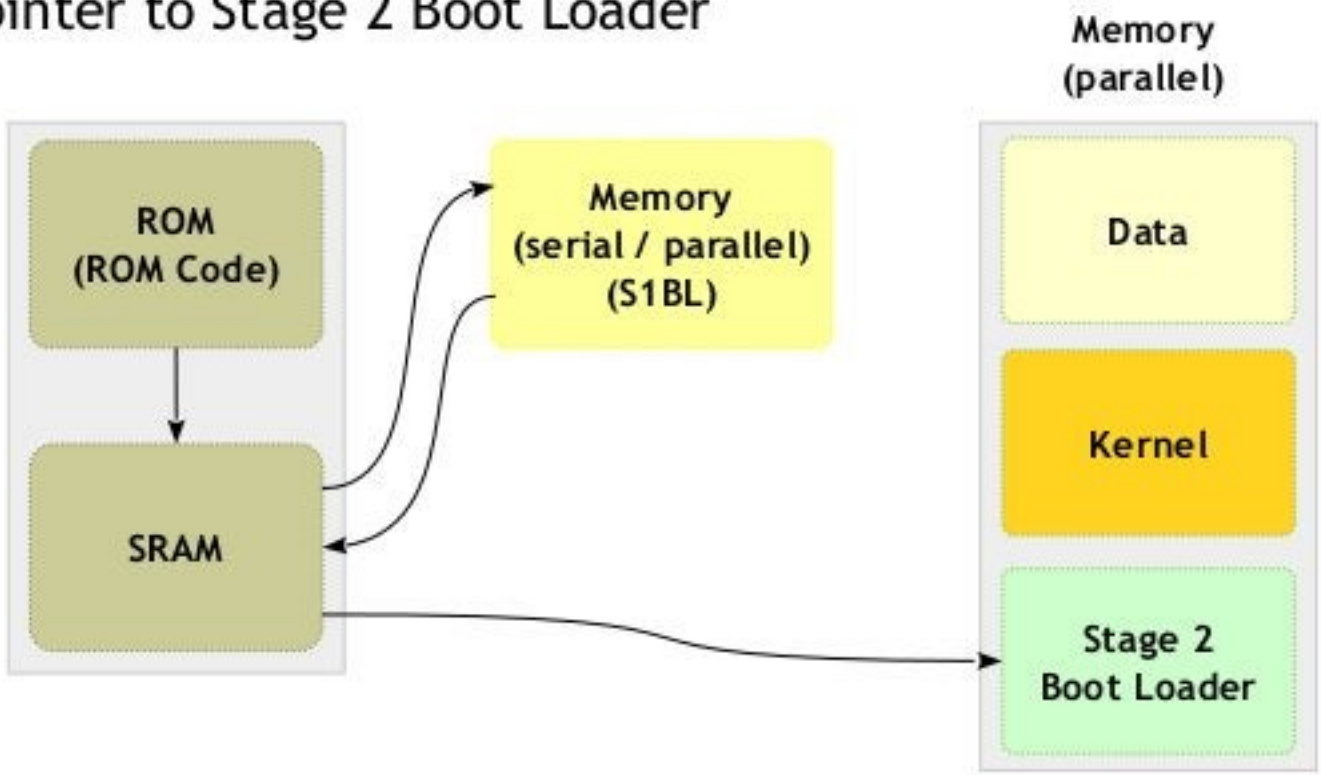
Booting Sequence - Boot Loader Stages

- Stage 1 Boot Loader
- Stage 2 Boot Loader

EmxDev Boards - EmxARM9A03

Booting Sequence - Stage 1 Boot Loader

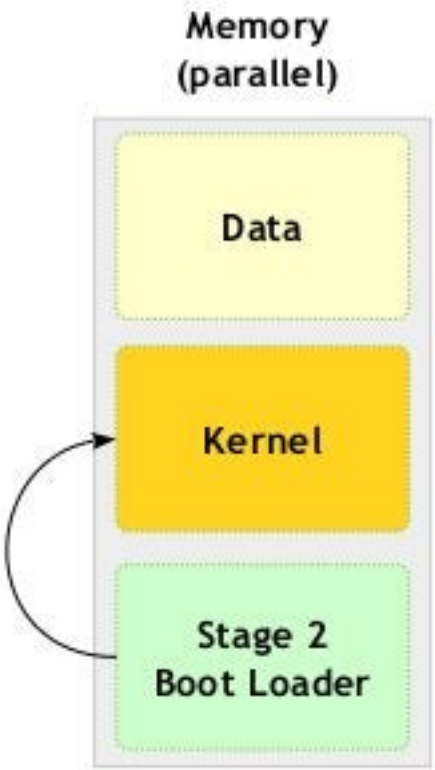
- Pointer to Stage 2 Boot Loader



EmxDev Boards - EmxARM9A03

Booting Sequence - Stage 2 Boot Loader

- Pointer to Kernel Image
- We use U-Boot as S2BL



EMERTXE

U-Boot Introduction

U-Boot

Introduction - General

- The "Universal Bootloader" ("Das U-Boot") is a monitor program
- Free Software: full source code under GPL
- Can get at: //www.denx.de/wiki/U-Boot
- Production quality: used as default boot loader by several board vendors
- Portable and easy to port and to debug
- Many supported architectures: PPC, ARM, MIPS, x86, m68k, NIOS, Microblaze

U-Boot

Introduction - General

- More than 216 boards supported by public source tree
- Simple user interface: CLI or Hush shell
- Environment variable storing option on different media like EEPROM, Flash etc
- Advanced command supports

U-Boot

Introduction - Design Principles

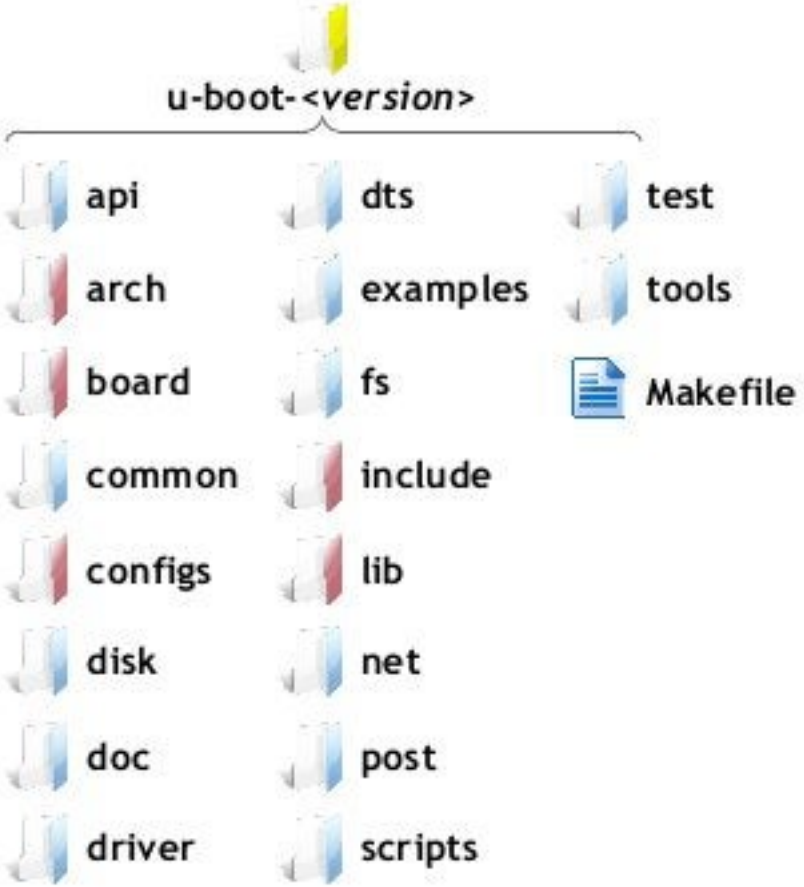
- Easy to port to new architectures, new processors, and new boards
- Easy to debug: serial console output as soon as possible
- Features and commands configurable
- As small as possible
- As reliable as possible

U-Boot Source Tree

U-Boot
Source Code Browsing

- Untar the U-Boot code
  - tar xvf u-boot-<*version*>.tar.bz2
- Enter the U-Boot directory
  - cd u-boot-<*version*>
- The following slide discuss the contents of the U-Boot directory

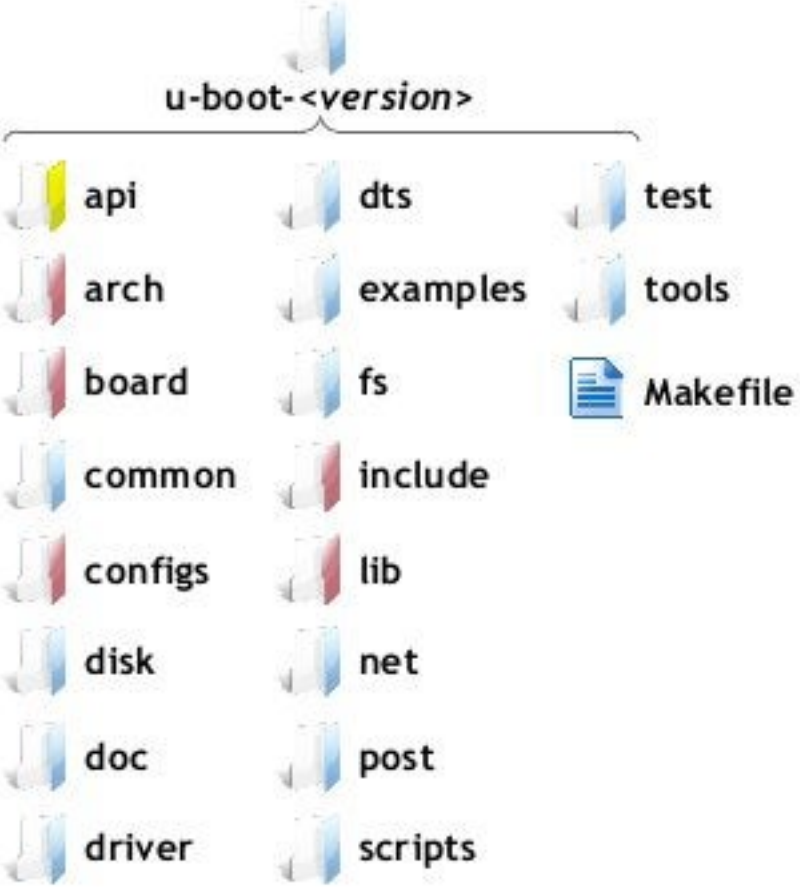
U-Boot
Source Tree



- The left side of the slide shows the source content of the U-Boot
- The directory structure **might vary** depending on the picked version.
- The considered version is

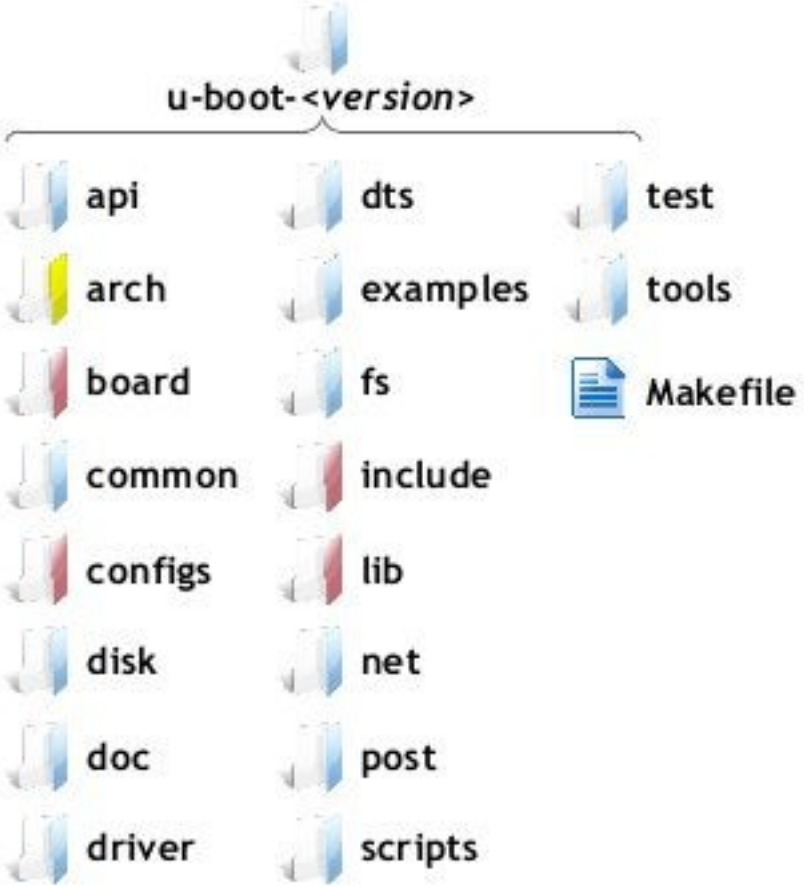
  u-boot-2015-01
- Lets us discuss some important directories and files

U-Boot
Source Tree



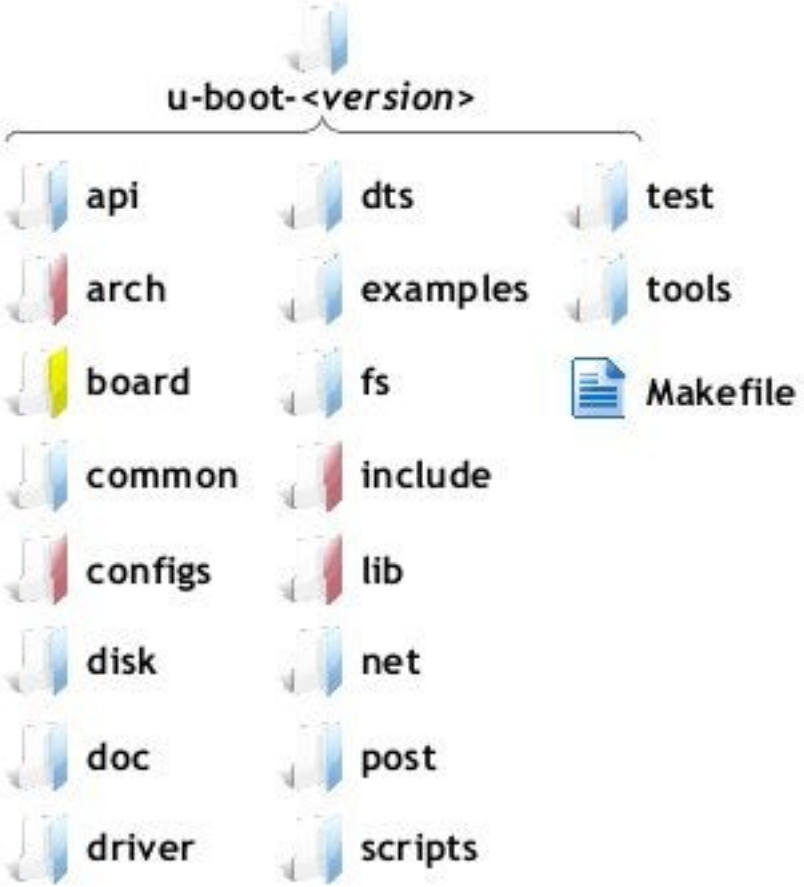
- Machine/arch independent API for external apps

U-Boot
Source Tree



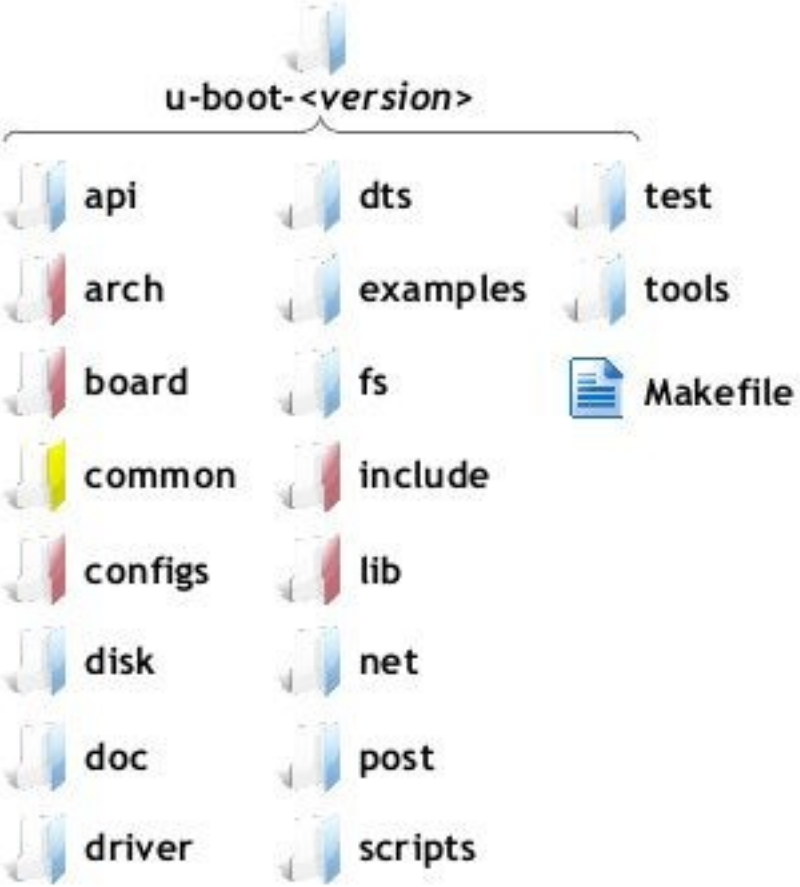
- All architecture dependent functions
- CPU specific information
  - <core>/cpu.c
  - <core>/interrupt.c
  - <core>/start.S

U-Boot
Source Tree



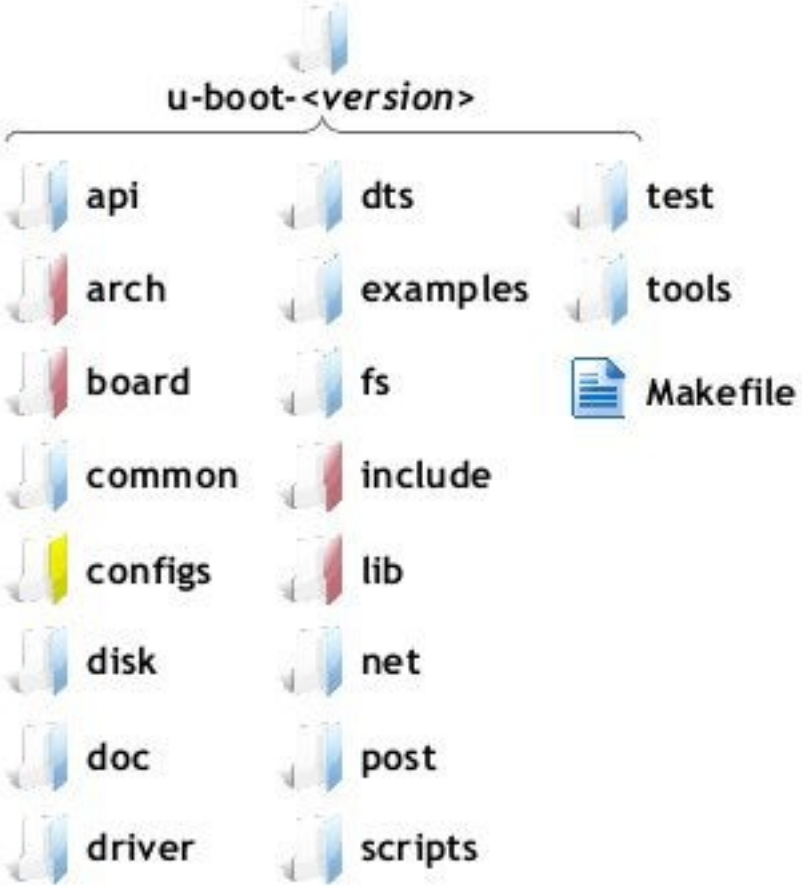
- Platform, board level files. Eg, atmel, icecube, oxc etc.,
- Contains all board specific initialization
  - <boardname>/flash.c
  - <boardname>/<boardname>_emac.c
  - <boardname>/<boardname>.c
  - <boardname>/soc.h
  - <boardname>/platform.S

U-Boot
Source Tree



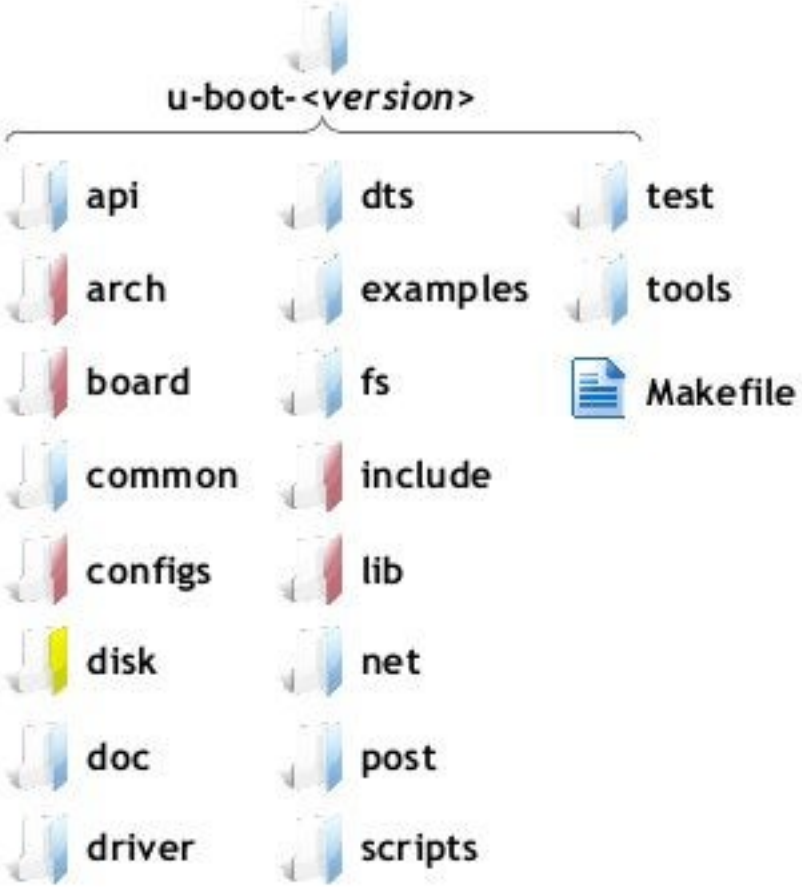
- All architecture independent functions
- All the commands

U-Boot
Source Tree



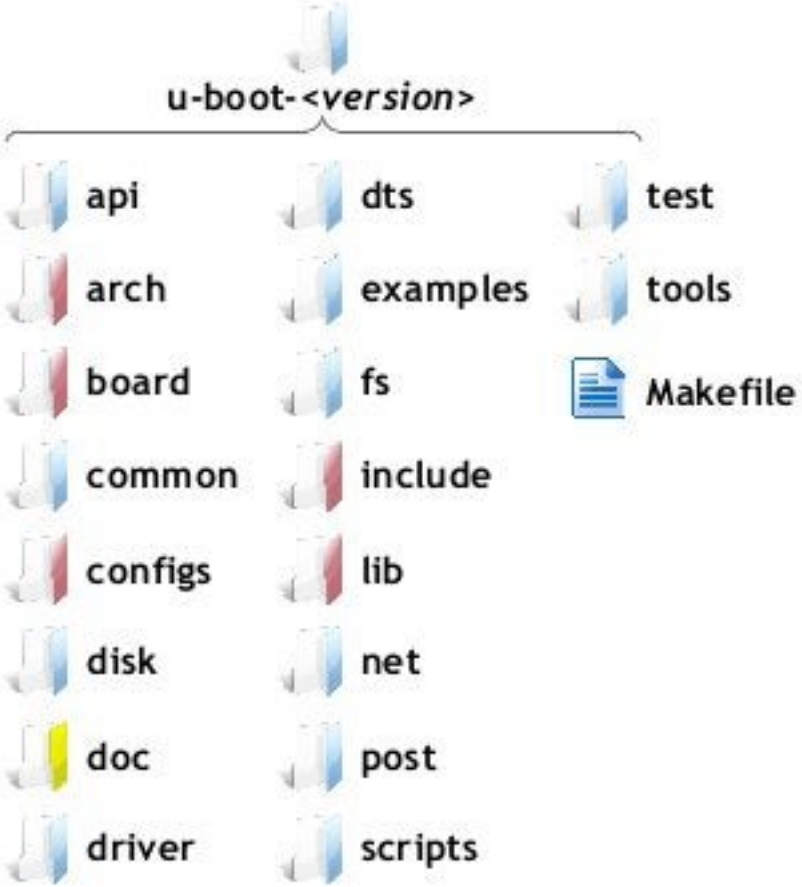
- Default configuration files for boards

U-Boot
Source Tree



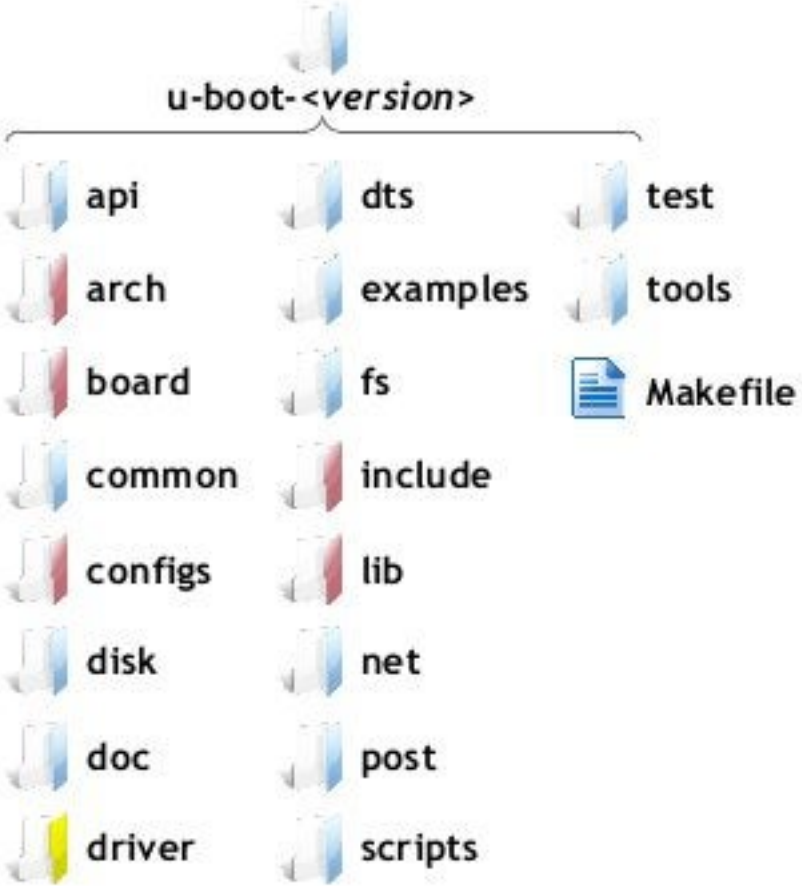
- Partition and device information for disks

U-Boot
Source Tree



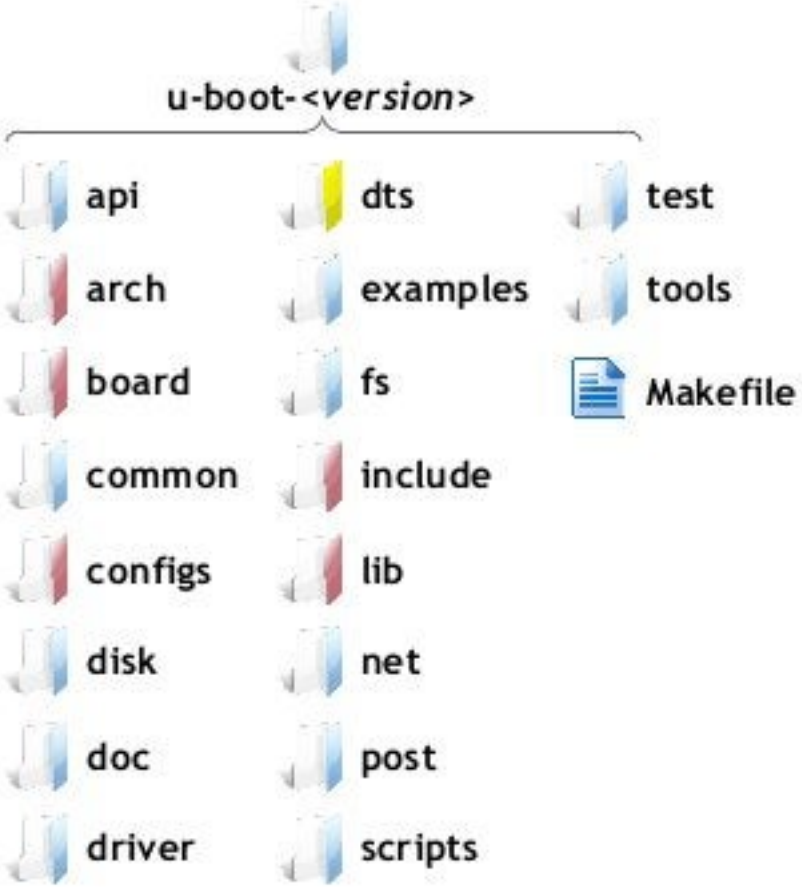
- You can find all the README files here

U-Boot
Source Tree



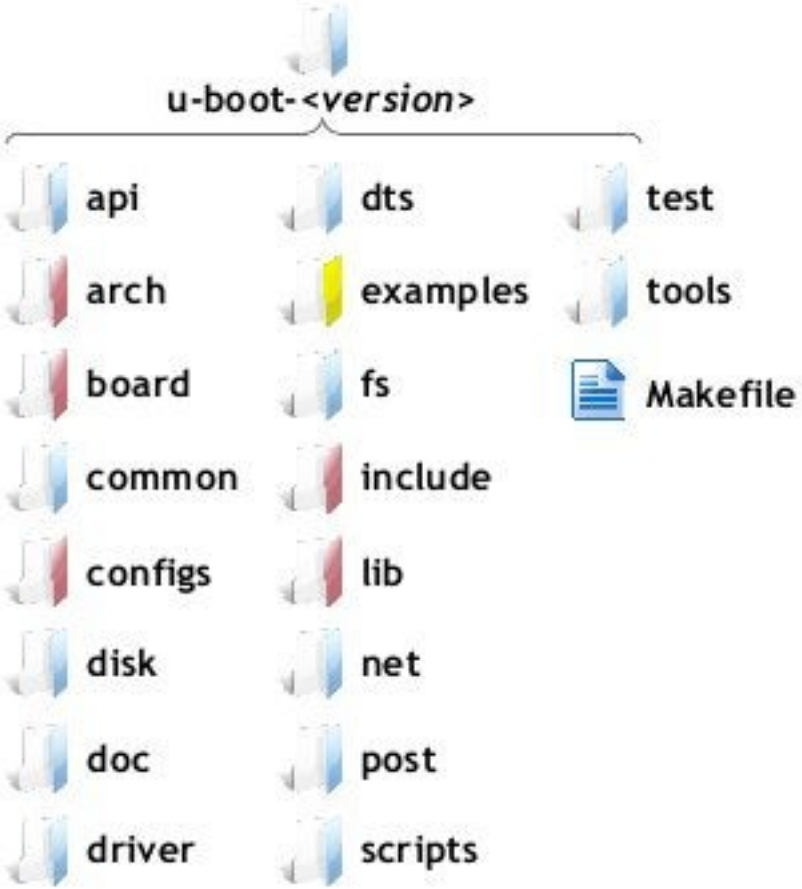
- Various device drivers files

U-Boot
Source Tree



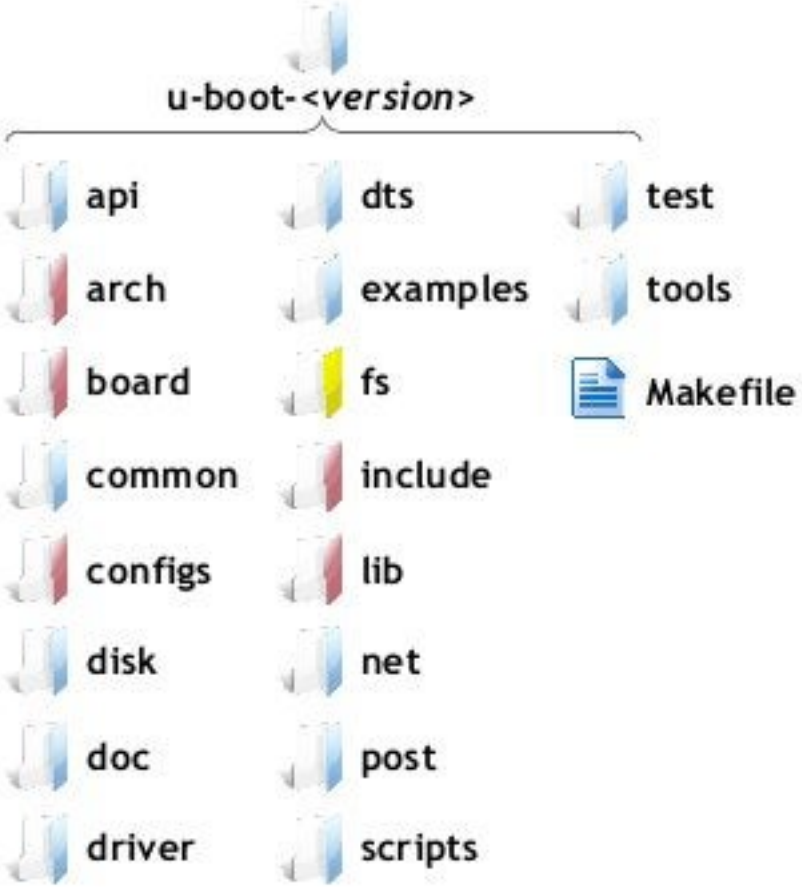
- Contains Makefile for building internal U-Boot fdt

U-Boot
Source Tree



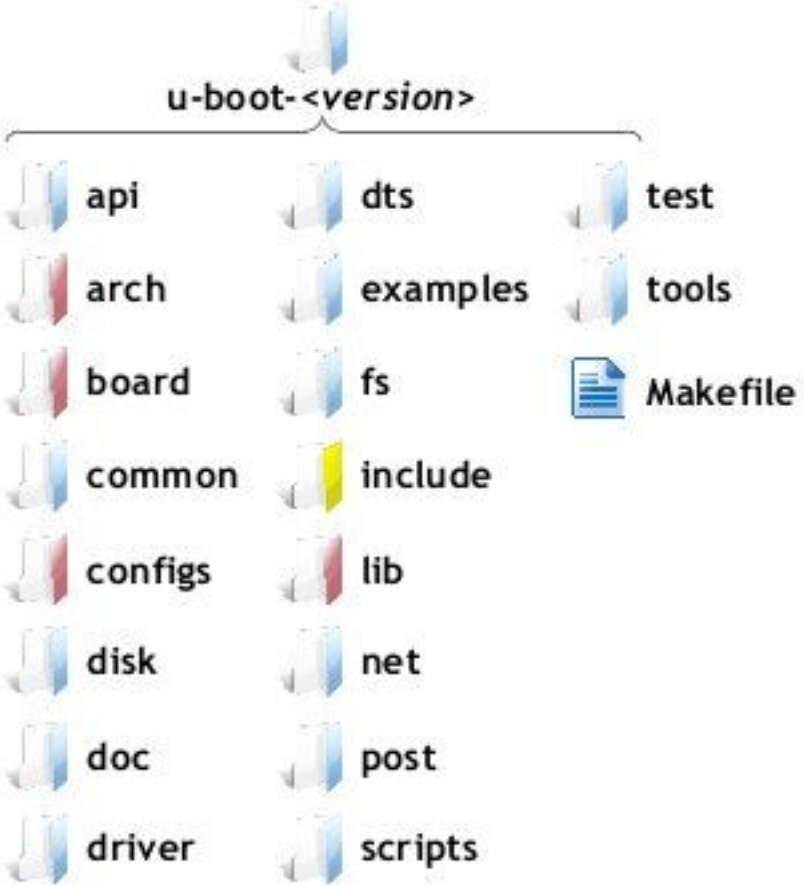
- Example code for standalone application

U-Boot
Source Tree



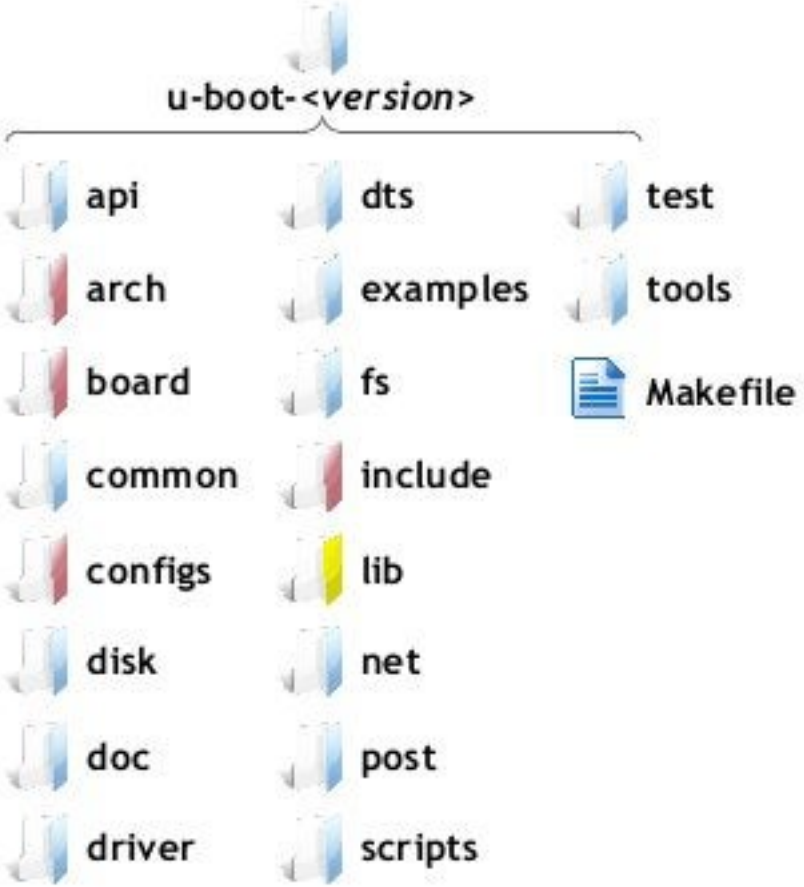
- File system directories and codes

U-Boot
Source Tree



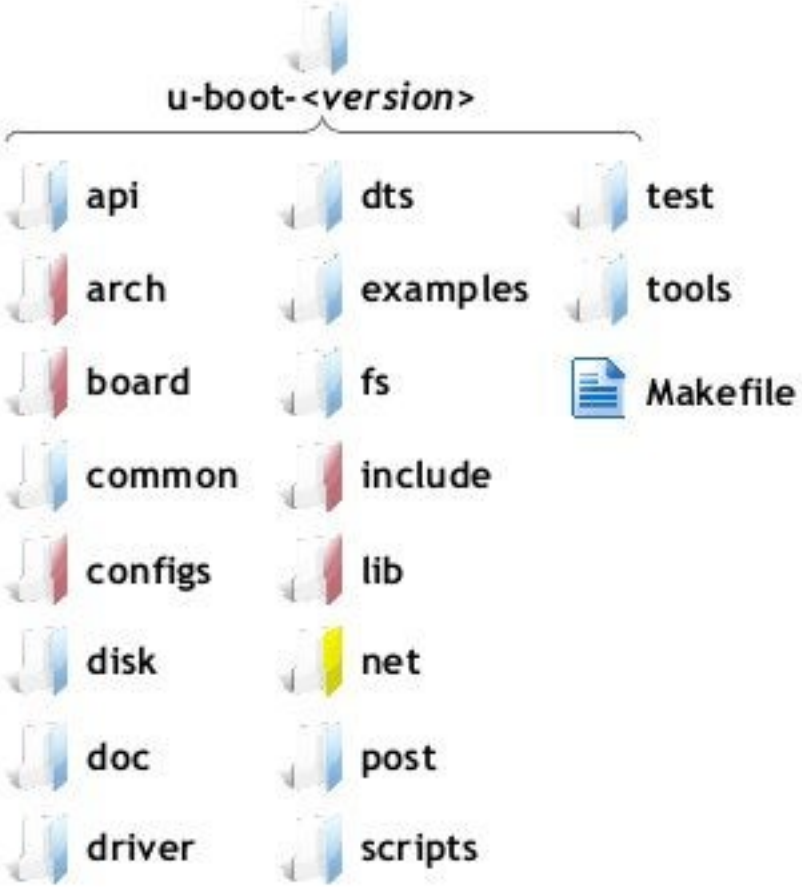
- Various header files
  - configs/<boardname>.h
  - <core>.h

U-Boot
Source Tree



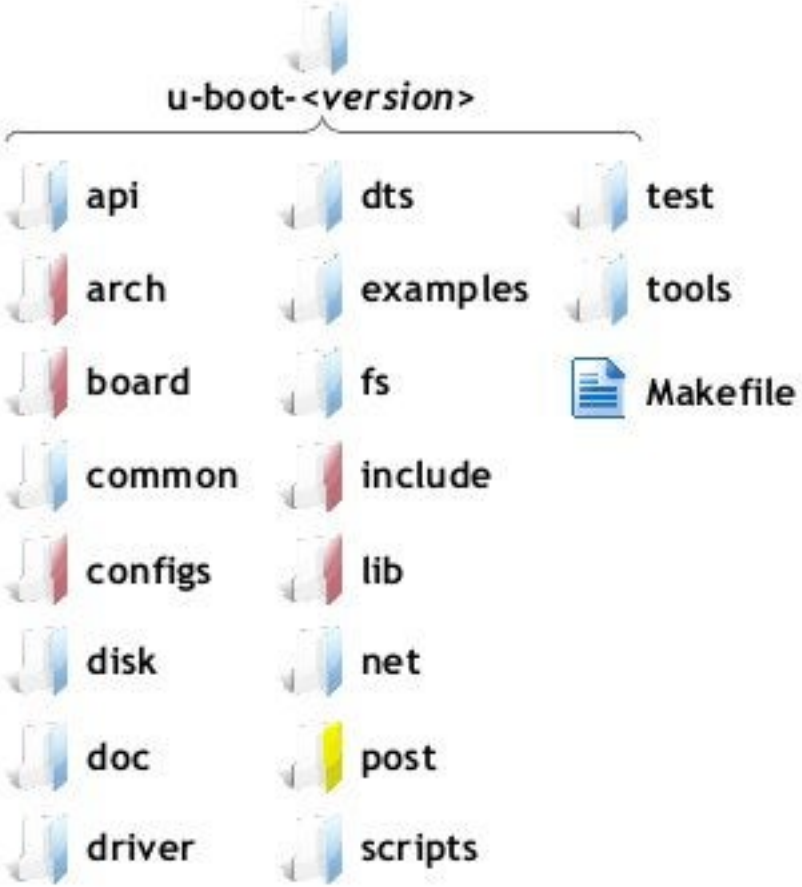
- Processor specific libraries
  - board.c
  - <arch>linux.c
  - div0.c

U-Boot
Source Tree



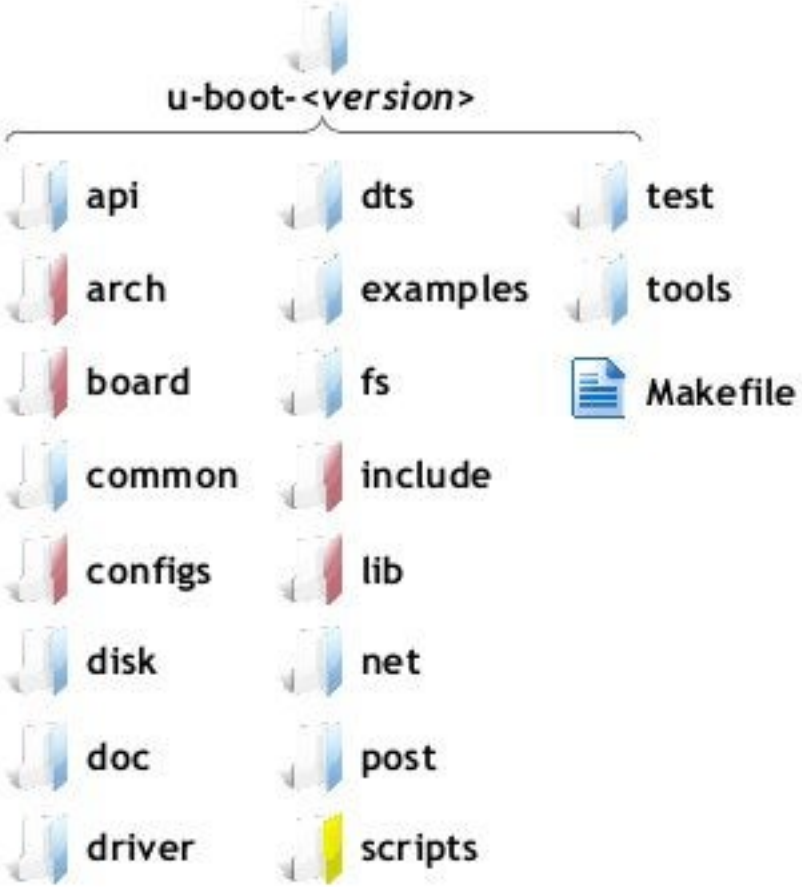
- Networking related files.

U-Boot
Source Tree



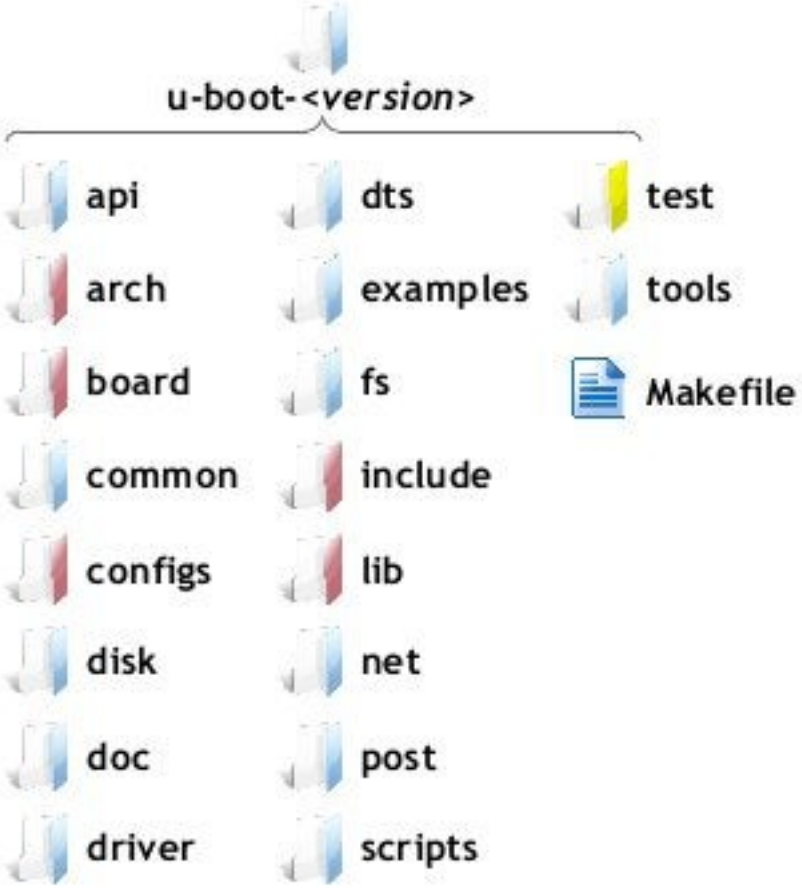
- Power On Self Test

U-Boot
Source Tree

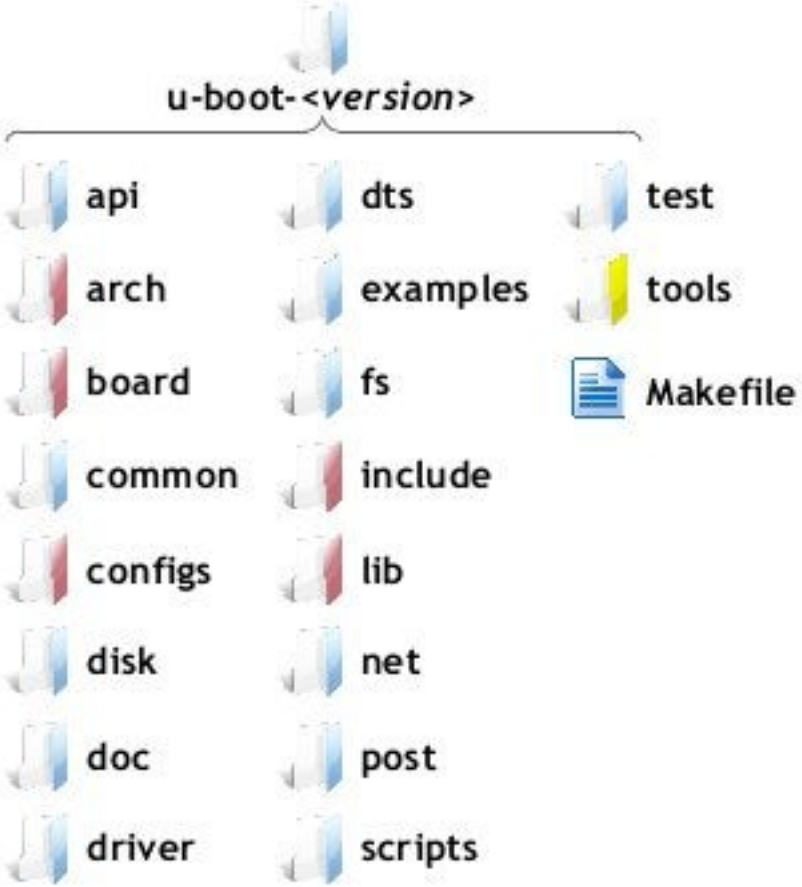


- Contains the sources for various helper programs

U-Boot
Source Tree

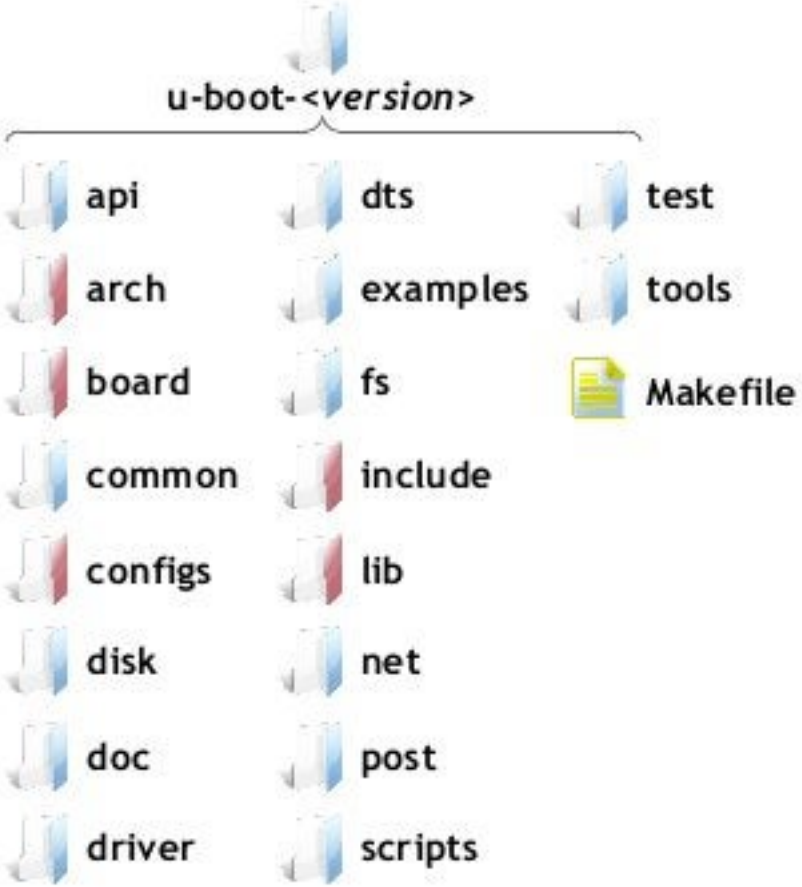


U-Boot
Source Tree



- Various tools directories and files

U-Boot
Source Tree



- Top level make file for Uboot build and configuration

U-Boot Building

U-Boot
Building

- The include/configs/ directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the Uboot features that should be compiled in, etc.
  - It is a simple .h file that sets preprocessor constants. See the README file for the documentation of these constants.
- Assuming that your board is already supported by Uboot, there should be a config corresponding to your board, for example include/configs/at91rm9200ek.h

U-Boot
Building

- We need to configure U-Boot for the required board which is generally done as

  `make <board_name>_config`

- The `board_name` can be found in include/configs/ directory
- The newer version supports kernel like configuration options like `make menuconfig`
- Compile Uboot, by specifying the cross compiler prefix.

  `make CROSS_COMPILE=<cross_compile_path>`

U-Boot
Building

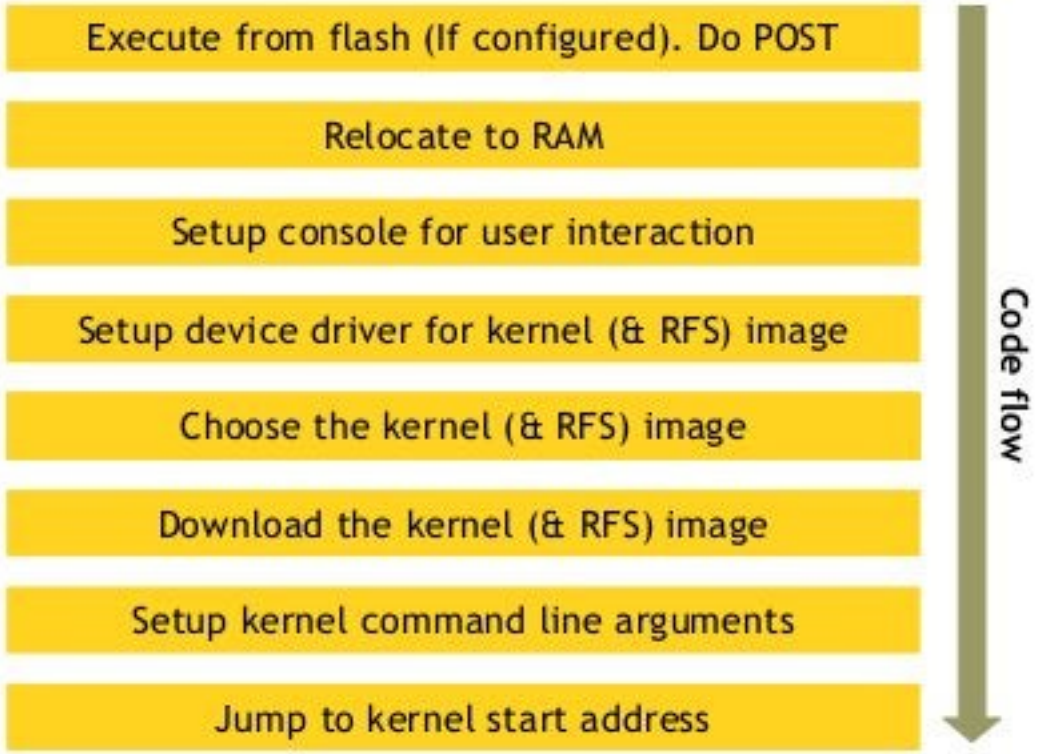
- ***cross_compile_path*** could be the command itself if already exported in PATH variable, else you can specify the installation path of command
- For example for arm platform it would look like

```
make CROSS_COMPILE=arm-linux-
```

- The result would be u-boot.bin which has to be stored in flash memory (in most of the cases)
- The invocation of the stored image depends on the target architecture. The memory used to store would play the role here

U-Boot Introduction

U-Boot
Responsibility



U-Boot Important Commands

U-Boot
Utilities

- Environment Variables
- Commands
  - Information
  - Environment
  - Network
  - Boot
  - Data Transfer
  - Memory

U-Boot
Environment Variables

- **bootcmd** : Contains the command that U-Boot will automatically execute at boot time after a configurable delay, if the process is not interrupted
- **bootargs** : contains the arguments passed to the Linux kernel
- **serverip** : Server (Host) ip address for network related commands
- **ipaddr** : Local ip address of the target
- **ethaddr** : MAC address. Will be set once

U-Boot

Important Environment Variables

- `netmask` : The network mask to communicate with the server
- `bootdelay` : Time in seconds to delay the boot process so that the u-boot can be interrupted before executing **`bootcmd`**
- `autostart` : If set the loaded image in memory will be executed automatically

U-Boot

Important Commands - Information

- `help` : Help command. Can be used to list all supported built commands
- `flinfo` : Display flash informations (NOR and SPI Flash)
- `nand info` : Display NAND flash informations

EMERTXE

U-Boot

Important Commands - Environment

- `printenv` : Print all set environment variables
- `setenv` : Set the environment variable
- `saveenv` : Save environment variable to configured memory

U-Boot

Important Commands - Network

- `ping` : Checks for network connectivity

U-Boot

Important Commands - Boot

- `boot` : Runs the default boot command, stored in bootcmd variable
- `bootm` : Boot memory. Starts a kernel image loaded at the specified address in RAM
  Example: `bootm <address>`

U-Boot

Important Commands - Data Transfer

- `loadb` : Load a file from the serial line to RAM
- `loads`
- `loady`
- `tftpboot` : Loads a file from the network to RAM
  Example: `tftpboot <address>`

U-Boot

Important Commands - Memory

- `erase` : Erase the content of NOR flash
- `protect` : Protect the content of NOR flash
- `cp` : Write to NOR flash
- `nand` : Erase, read, write to NAND flash

Embedded Linux Kernel

General Information

Embedded Linux Kernel

General Information

- Where to get?
- Kernel Subsystem
- Source Code Browsing

Embedded Linux Kernel

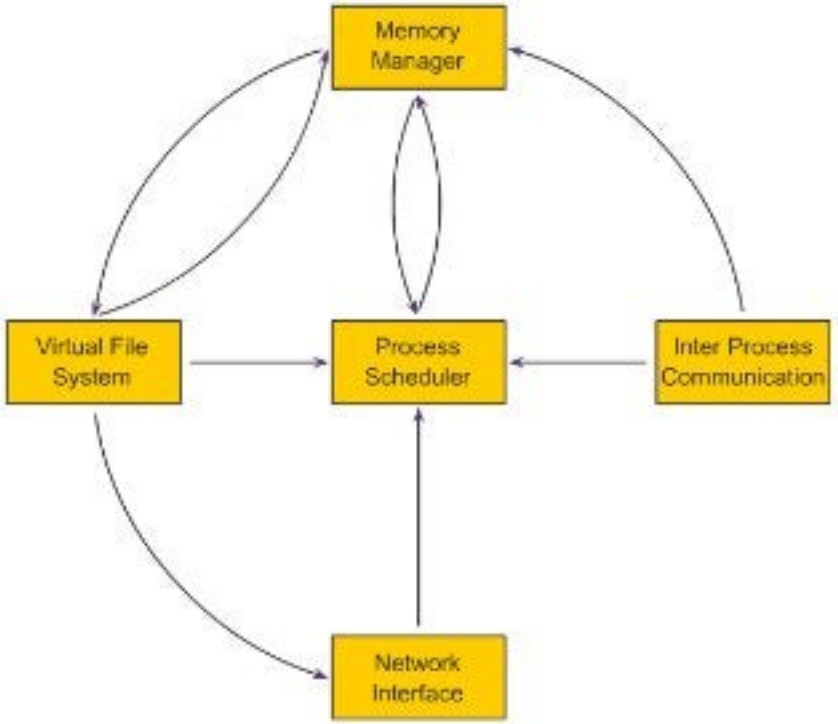
General Information - Where to get?

Note: Snapshot of www.kernel.org. Expect changes on updates

ΣMERTXE

Embedded Linux Kernel

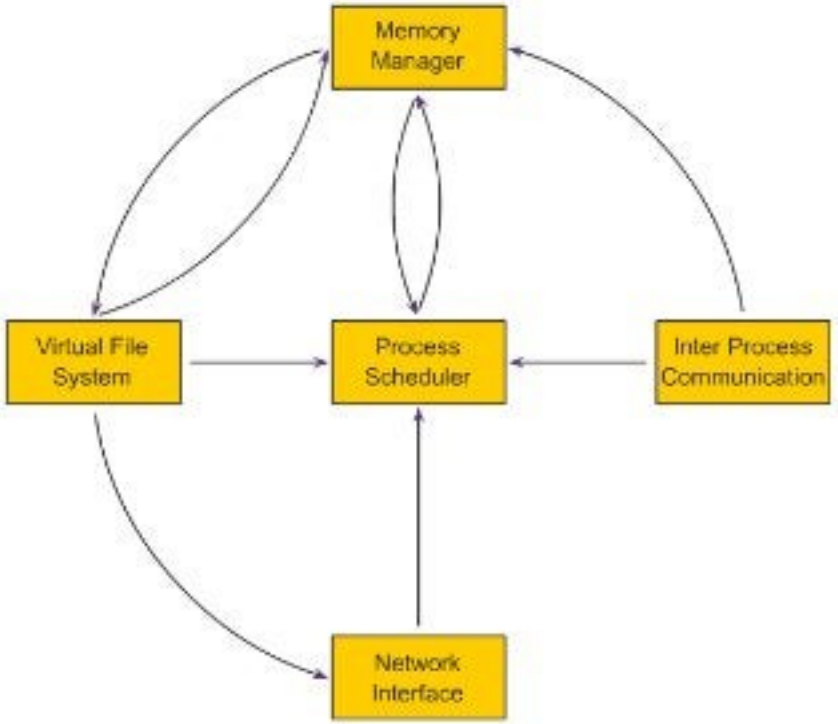
General Information - Kernel Subsystem



- Process Scheduler:
  - To provide control, fair access of CPU to process, while interacting with HW on time
- Memory Manager:
  - To access system memory securely and efficiently by multiple processes. Supports Virtual Memory in case of huge memory requirement
- Virtual File System:
  - Abstracts the details of the variety of hardware devices by presenting a common file interface to all devices

Embedded Linux Kernel

General Information - Kernel Subsystem



- Network Interface:
  - provides access to several networking standards and a variety of network hardware
- Inter Process Communications:
  - supports several mechanisms for process-to-process communication on a single Linux system

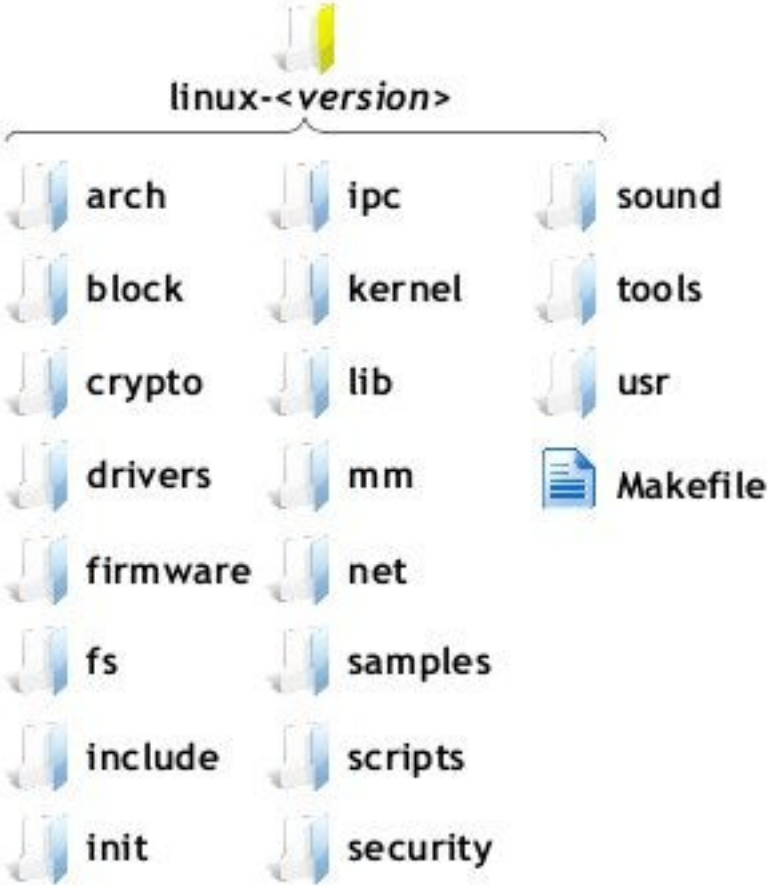
Embedded Linux Kernel

General Information - Source Code Browsing

- Untar the Linux kernel code
  - tar xvf linux-<*version*>.<*compression_format*>
- Enter the Linux kernel directory
  - cd linux-<*version*>
- The following slide discuss the contents of the Linux directory

Embedded Linux Kernel

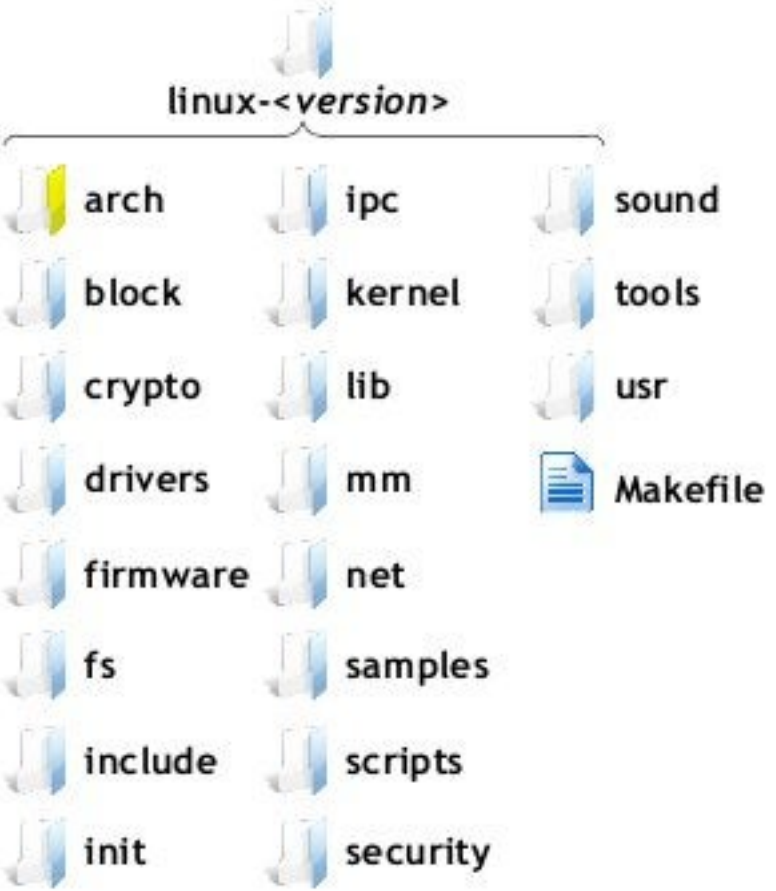
General Information - Source Code Browsing



- The left side of the slide shows the source content of the Linux kernel
- The directory structure might vary depending on the picked version.
- Lets us discuss some important directories and files

Embedded Linux Kernel

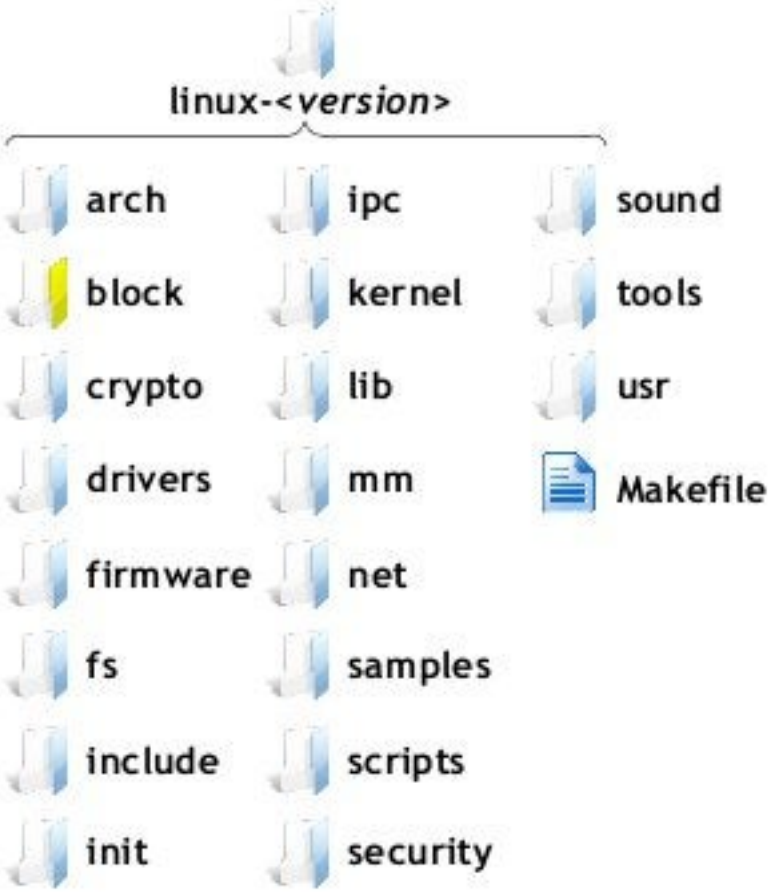
General Information - Source Code Browsing



- Architecture specific kernel code
- Has sub directories per supported architecture
- Example:
  - arm
  - powerpc
  - X86
- We can also find low level memory management, interrupt handling, early inits, assembly code and much more

Embedded Linux Kernel

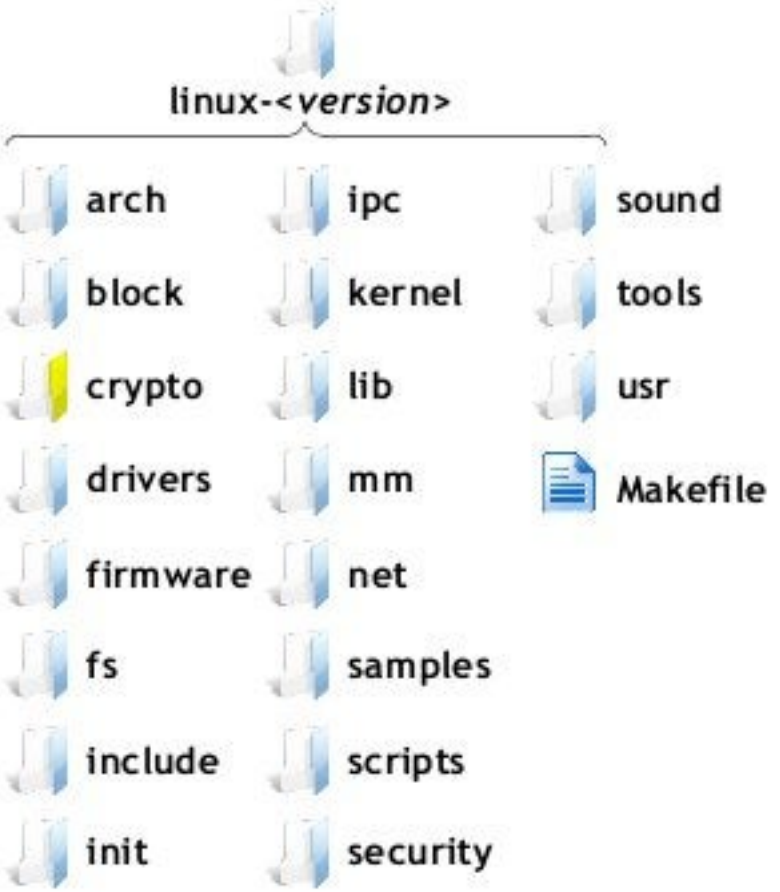
General Information - Source Code Browsing



- Contains core block layer files

Embedded Linux Kernel

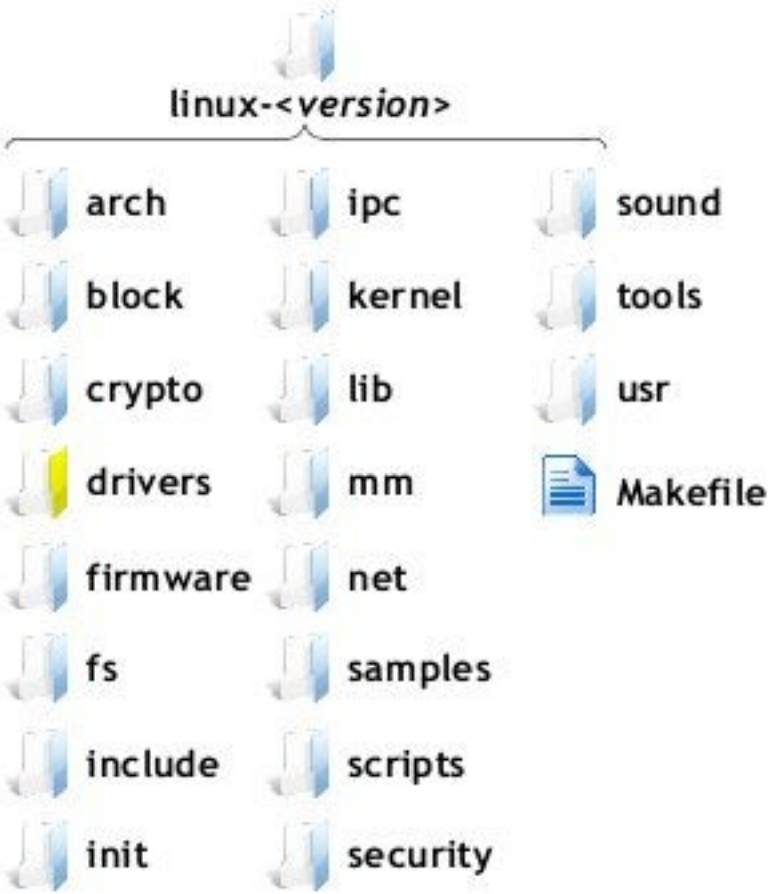
General Information - Source Code Browsing



- Cryptographic API for use by kernel itself

Embedded Linux Kernel

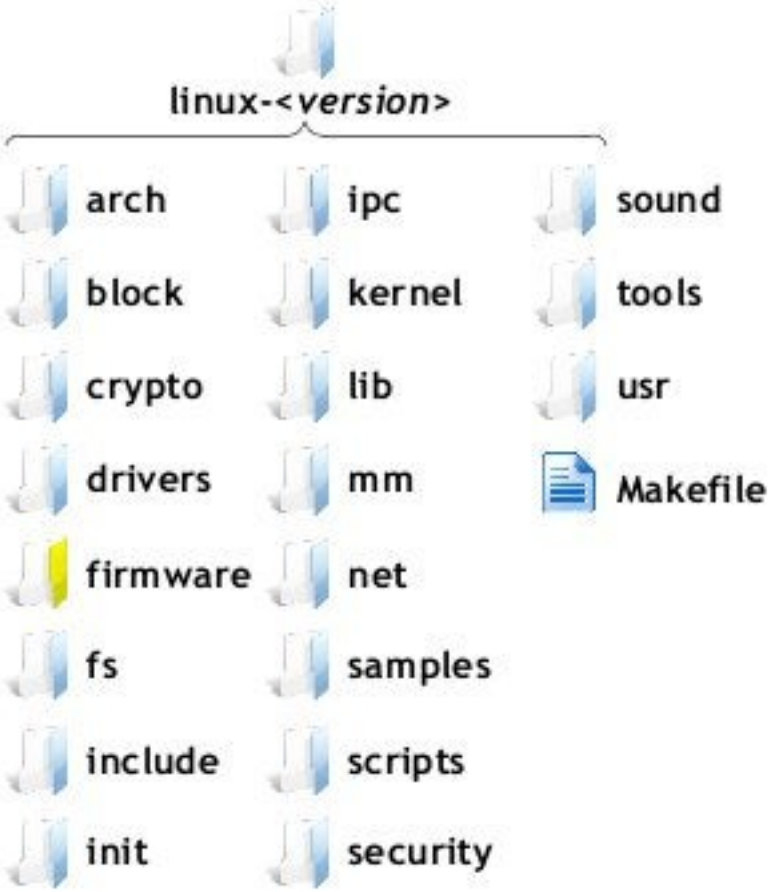
General Information - Source Code Browsing



- Contains system's device drivers
- Sub directories contain classes of device drivers like video drivers, network card drives, low level SCSI drivers etc.,

Embedded Linux Kernel

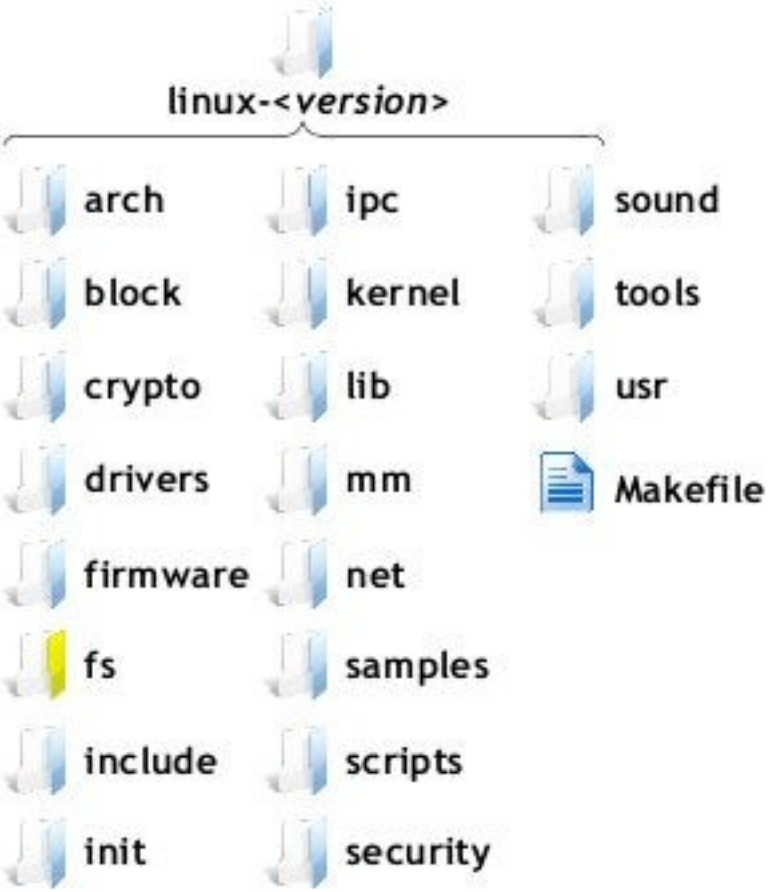
General Information - Source Code Browsing



- Contains the device firmwares which will be uploaded to devices with help of drivers

Embedded Linux Kernel

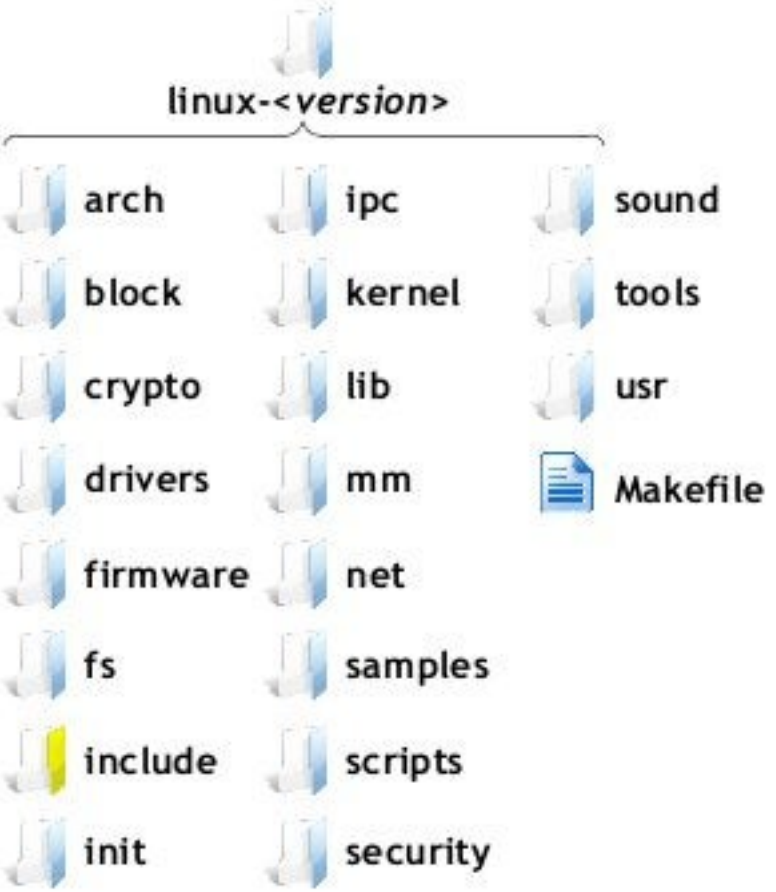
General Information - Source Code Browsing



- File system related code
- Contains both generic file system code (VFS) and different files system code
- Sub directories of supported file system
- Examples:
  - ext2
  - ext3
  - fat

Embedded Linux Kernel

General Information - Source Code Browsing

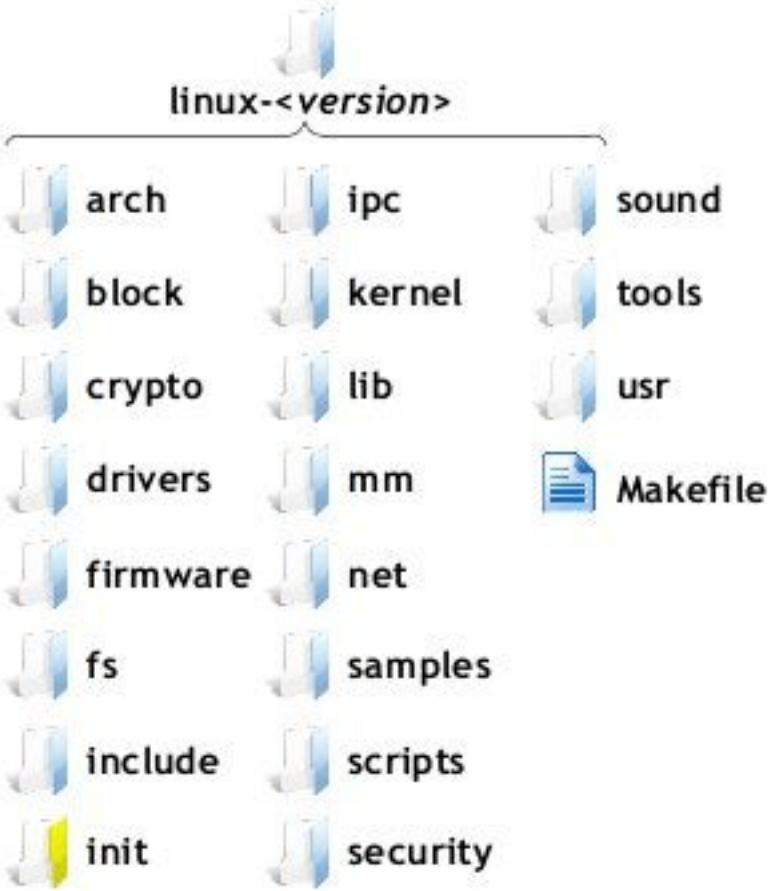


- Most of the header files used in the .c file of the kernel source
- It has further sub directories including asm-generic
- Architecture specific header file would be found in arch/<arch>/include/

Note: File level organization will vary based on different versions of kernel sources especially architecture and machine related header files

Embedded Linux Kernel

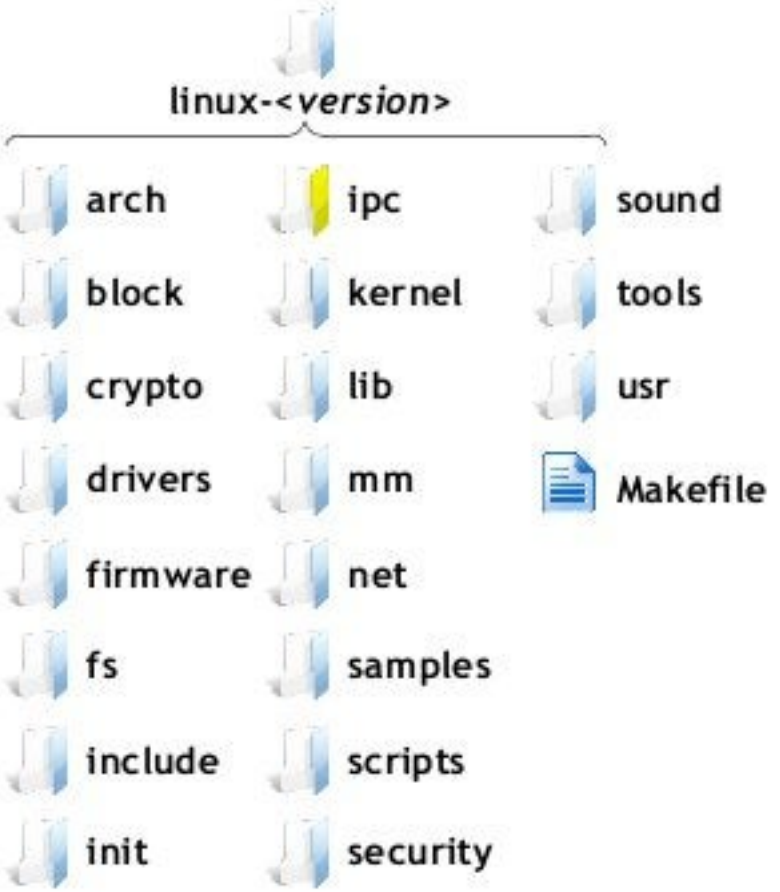
General Information - Source Code Browsing



- Initialization code for kernel
- Best directory to start with to know on how kernel works
- Has main.c of kernel

Embedded Linux Kernel

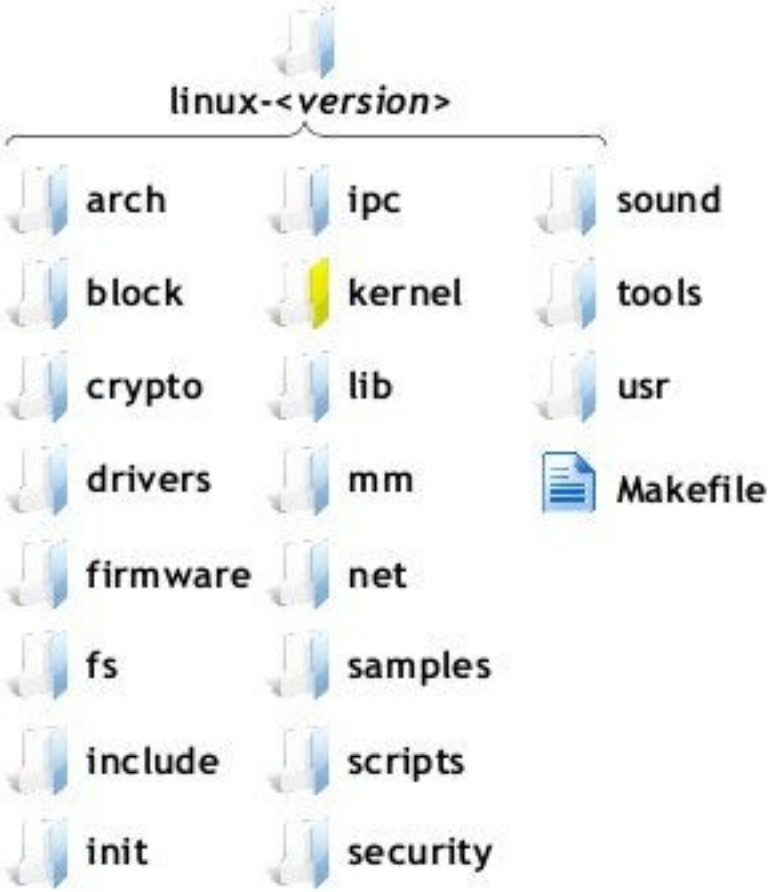
General Information - Source Code Browsing



- Contains kernel's inter process communication code like shared memory, semaphores and other forms

Embedded Linux Kernel

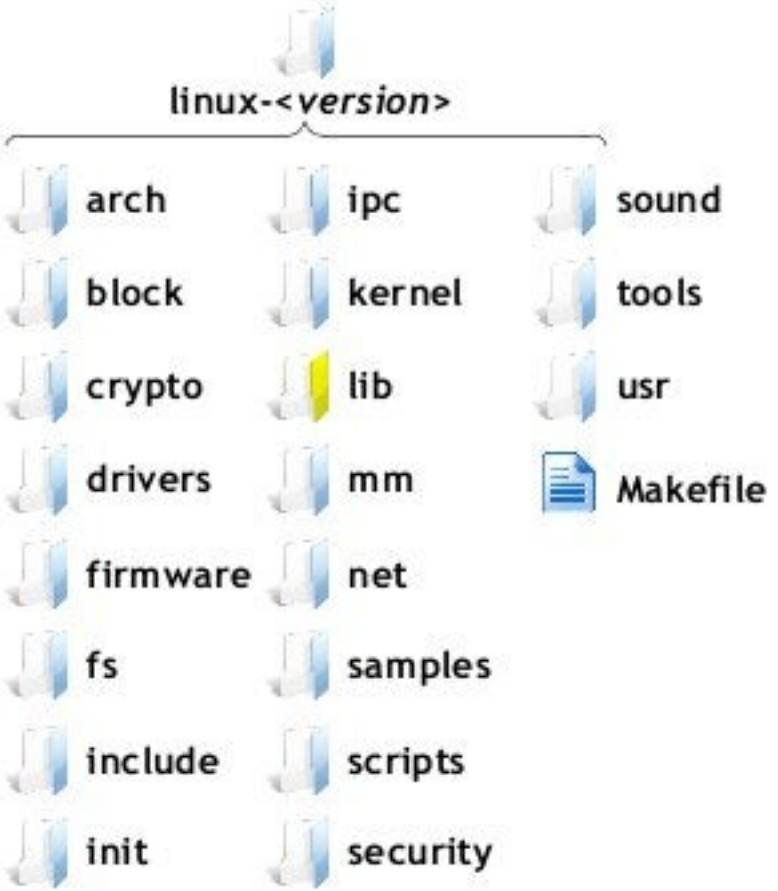
General Information - Source Code Browsing



- Generic kernel level code which can't fit anywhere else
- Contain upper level codes for signal handling, scheduling etc.,
- The architecture specific kernel code will be in arch/<arch_name>/kernel

Embedded Linux Kernel

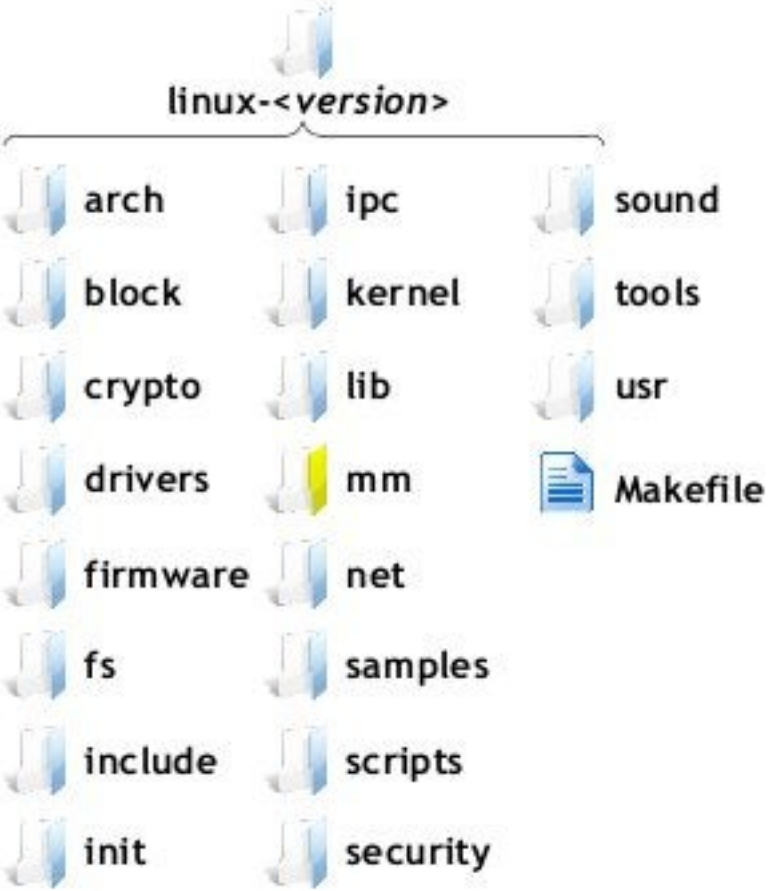
General Information - Source Code Browsing



- Contains kernel's library code
- Common string operations, code for debugging and command line parsing code can be found here
- The architecture specific library code will be in arch/<arch_name>/lib

Embedded Linux Kernel

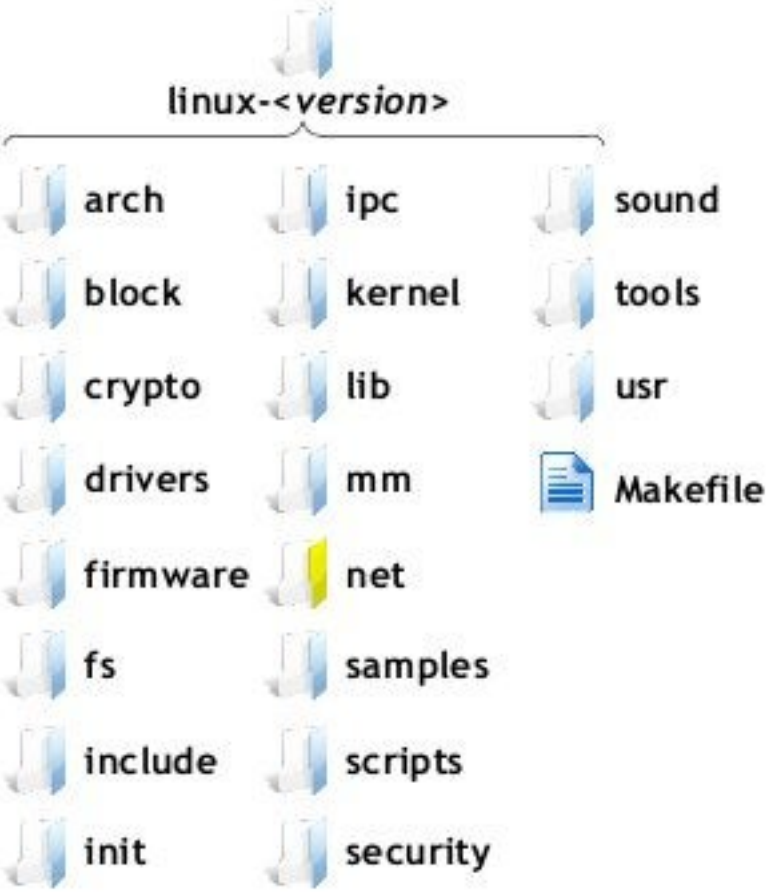
General Information - Source Code Browsing



- Contains memory management code
- The architecture specific memory management code would be found in arch/<arch_name>/mm
- Example:
  - arch/x86/mm/init.c

Embedded Linux Kernel

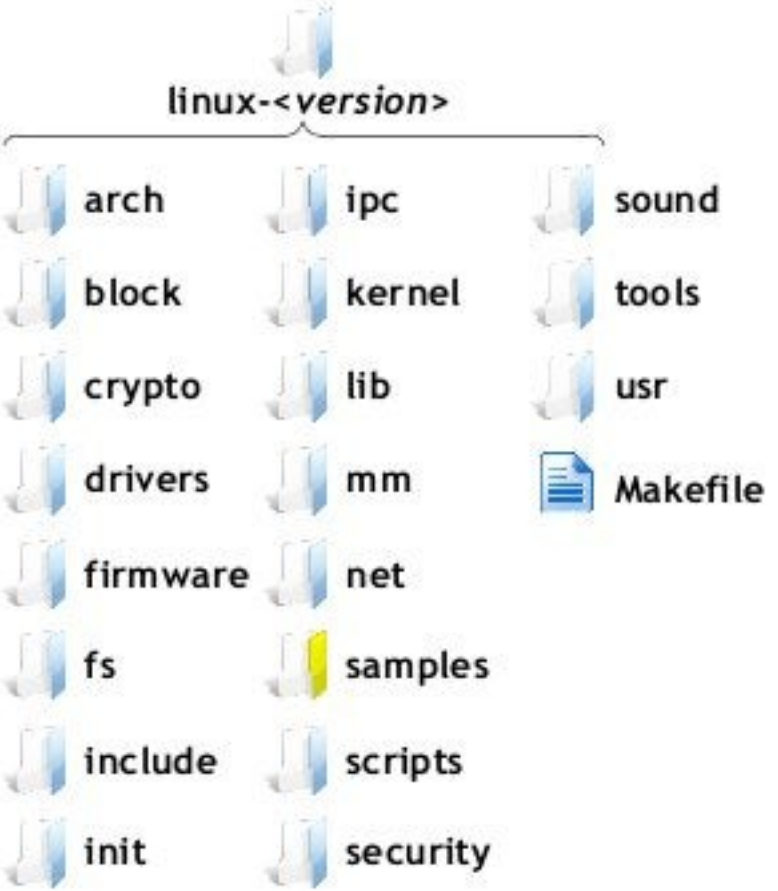
General Information - Source Code Browsing



- The kernels networking code

Embedded Linux Kernel

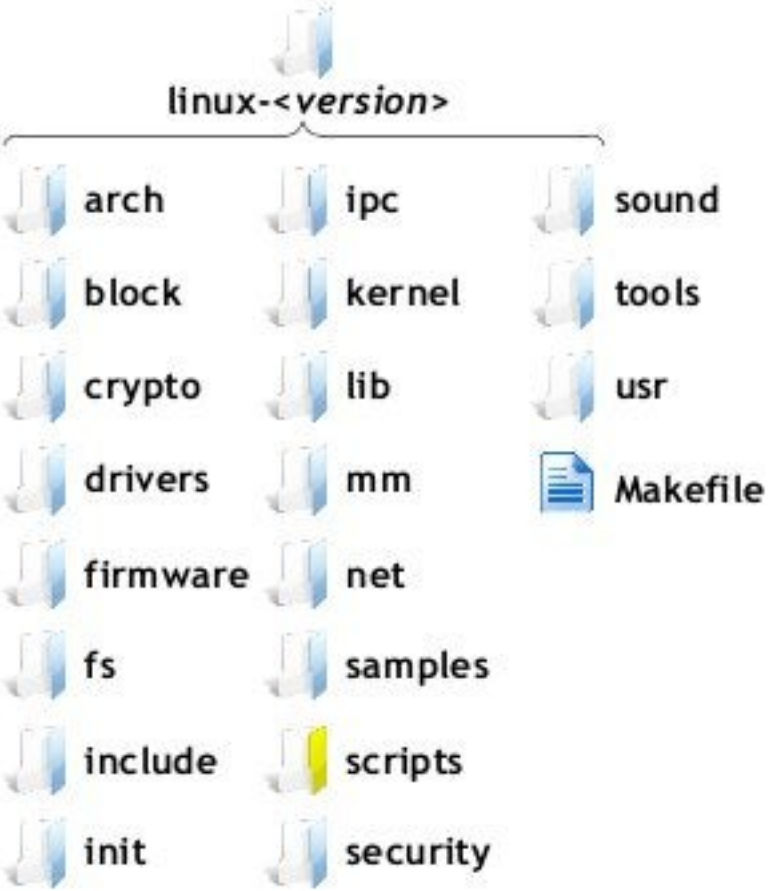
General Information - Source Code Browsing



- Some sample programs

Embedded Linux Kernel

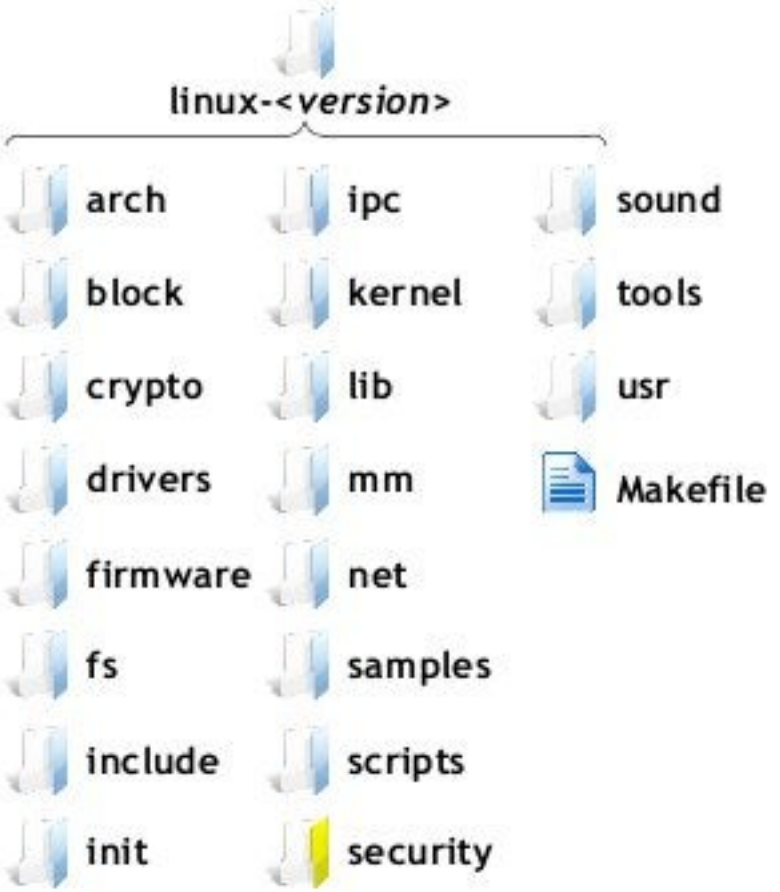
General Information - Source Code Browsing



- Contains scripts that are used while kernel configuration

Embedded Linux Kernel

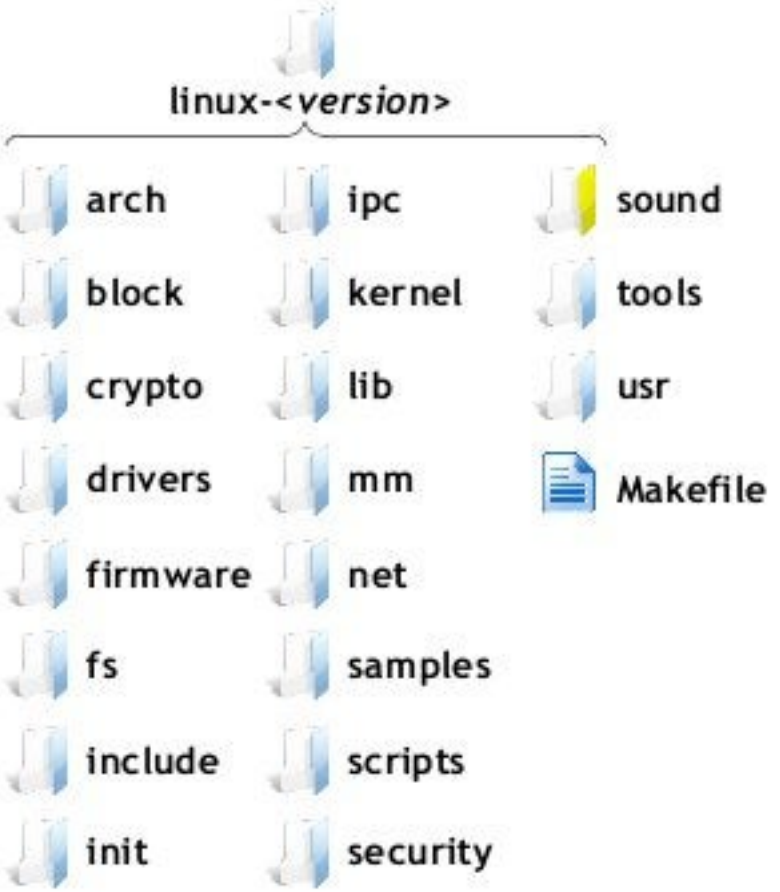
General Information - Source Code Browsing



- Contains code for different security models

Embedded Linux Kernel

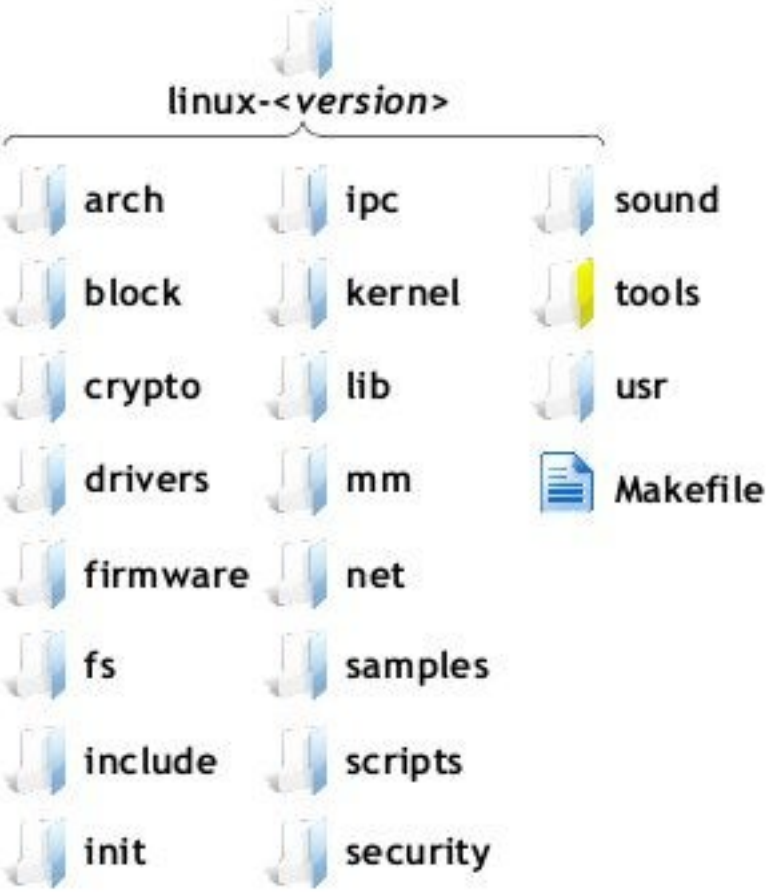
General Information - Source Code Browsing



- Contains all the sound card drivers

Embedded Linux Kernel

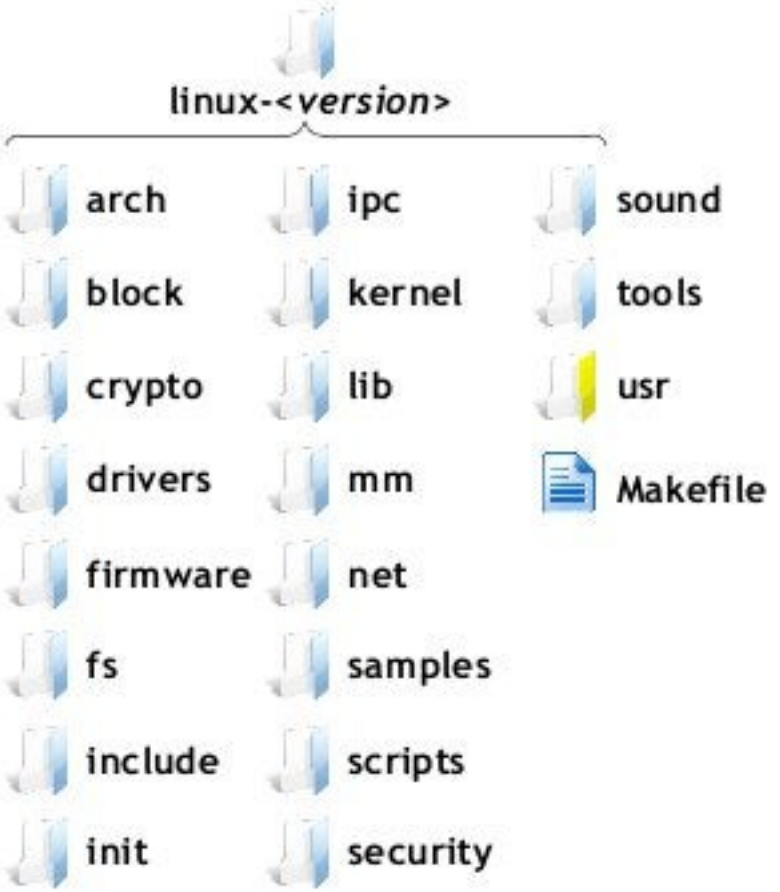
General Information - Source Code Browsing



- Certain configuration and testing tools

Embedded Linux Kernel

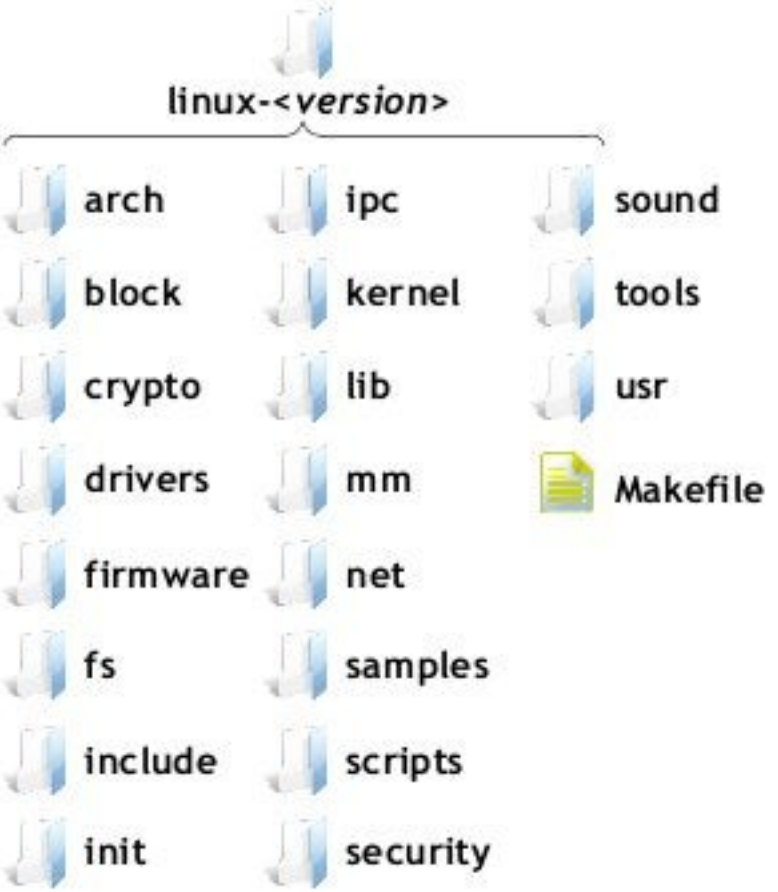
General Information - Source Code Browsing



- Contains code that builds a cpio-format archive containing a root file system image, which will be used for early userspace

Embedded Linux Kernel

General Information - Source Code Browsing



- This is top level Makefile for the whole source tree
- Contains useful rules and variables like default gcc compilation flags

Embedded Linux Kernel

Configuration

Embedded Linux Kernel
Configuration

- The kernel configuration is based on multiple **Makefiles**
- As discussed already the top level **Makefile** would be used for this purpose
- The configuration you should know the target. You can find of the target as mentioned below

```
$ cd linux-<version>
```

```
$ make help
```

- Now you may look for “Configuration targets:” section of the output and decide one

Embedded Linux Kernel
Configuration

- Once you decide on the target you may try the following command

```
$ make target
```

- The modified configurations would be saved on a file called as **.config** which can be found on the top level of the **linux-<*version*>** directory.
- All the target options use the same **.config** file, so you may use any interchangeably.

Embedded Linux Kernel
Configuration

- Some most commonly used target are
  - make config
  - make menuconfig
  - make xconfig
- Configuring Architecture specific targets
- Configuring for specific architecture from scratch

Embedded Linux Kernel

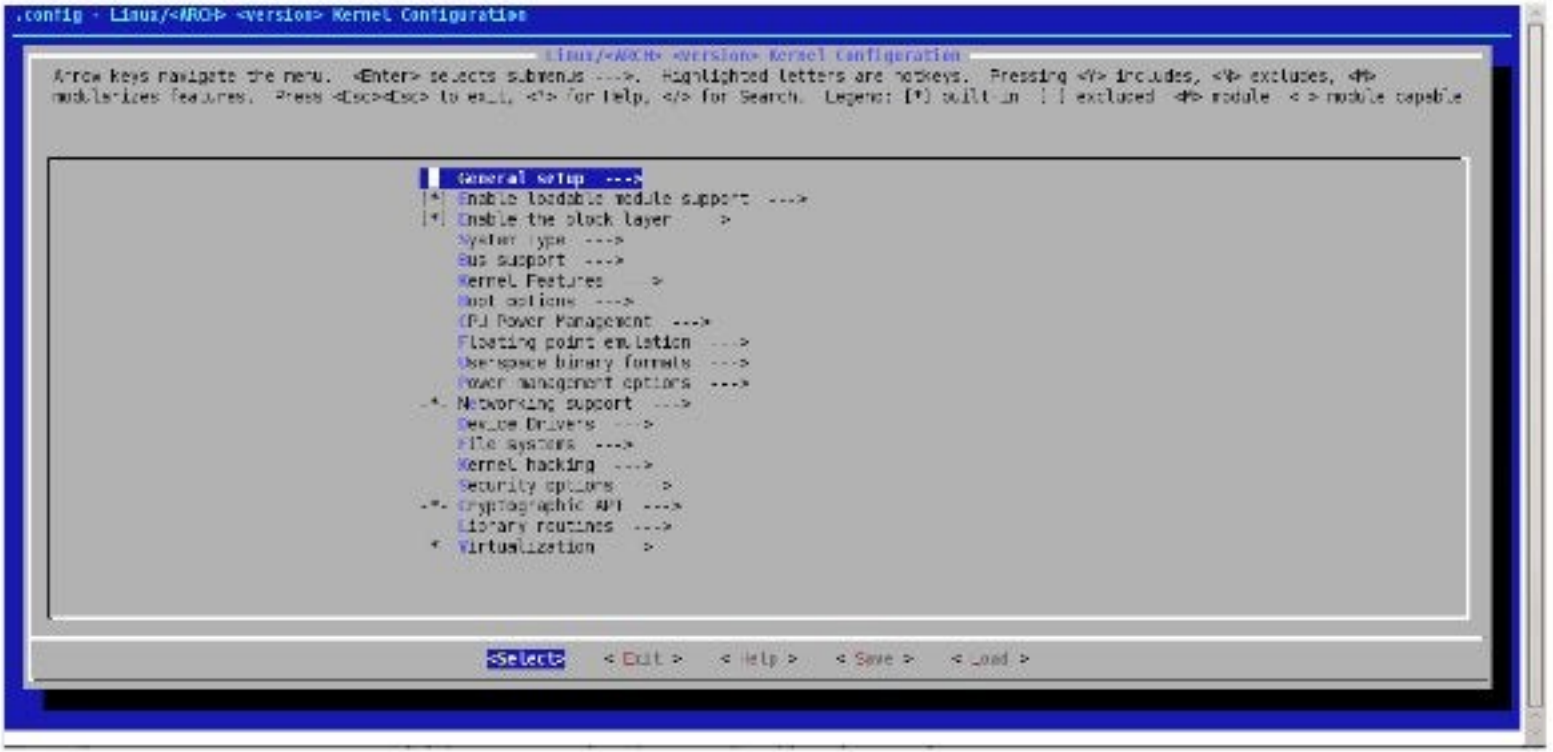
Configuration - make config

```
user@hostname:linux-<version>$ make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/<ARCH> <version> Kernel Configuration
*
Patch physical to virtual translations at runtime (ARCH_PATCH_PHYS_VIRT) [Y/n/?]
```

- The above image show snap shot typical output of make config command
- Updates current config utilizing a line-oriented program
- No user friendly approach. Could be used if you have limited host installations
- The problem with this approach is that, It force you to follow an sequence of questions while configuration.
- Have to use “Ctrl C” to exit

Embedded Linux Kernel

Configuration - make menuconfig



- The above image shows the snapshot of typical output of make menuconfig command

ΣMERTXE

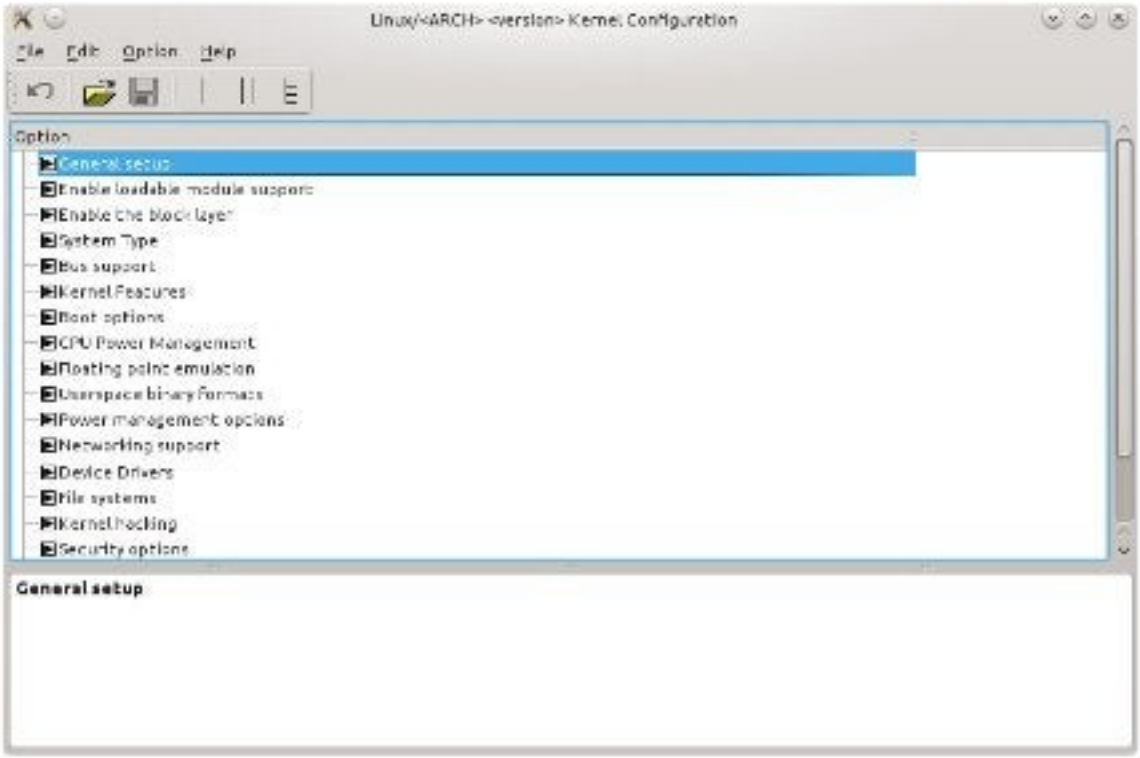
Embedded Linux Kernel

Configuration - make menuconfig

- Most commonly used method and simple method
- Can be used if graphics is unavailable
- Requires libncurses-dev installation
- Easy to navigate between options, using arrow keys
- Use <Help> to know more on menuconfig

Embedded Linux Kernel

Configuration - make xconfig



- The above image shows the snapshot of typical output of make xconfig command

ΣMERTXE

Embedded Linux Kernel

Configuration - make xconfig

- Most commonly used graphical method of configuration
- Easy to use, better search option
- Use Help menu to know more on xconfig
- Requires libqt-dev packages installation

Embedded Linux Kernel

Configuration - Architecture Specific

- Most preferably used in Embedded Linux configuration
- You can find then at `arch/<arch>/configs/`
- These files are best possible minimal .config file you can have for your board
- Just type the following on the command to know available target

```
$ make help
```

  - Now you may look for “Architecture specific targets:” section of the output to look for default configuration for your target architecture

- Now the following command

```
$ make <controller_name>_defconfig
```

Embedded Linux Kernel

Configuration - Architecture Specific

- The previous command would rewrite the existing .config file.
- Now you can use any of the general configuration method to discussed above to configure further if required
- If you feel the you are done and need to preserve your configuration then you can save it by

```
$ make savedefconfig
```

- The above line will create a file call **defconfig** on root of kernel source directory
- Now you can mv it to the config directory by the following command

```
$ mv defconfig /arch/<arch>/configs/my_defconfig
```

Embedded Linux Kernel

Configuration - From Scratch

- Its possible to configure a kernel from scratch without using any default configuration
- It would obvious if your a board vendor where you might have to do for your board
- Point to be kept in mind in this case
  - Make sure you alteast select a proper architecture for your board
  - Most of the architecture dependent basic things would be set by default, so just leave it as it is, unless you know what you change
  - Might have to change certain thing like select a correct device driver for your board

Embedded Linux Kernel

Building

Embedded Linux Kernel

Building - Compilation

- Assuming the required configuration are done, The next step would be to compile the kernel.
- Type the following command on the prompt to start the compilation

```
make
```

- Can use the below command if you have multicore CPU

```
make -j
```

- The above command will speed up your compilation process
- You may even specify the no of jobs you want to run simultaneously based on your CPU configuration

Embedded Linux Kernel

Building - Compilation

- Once the compilation is done you will get the kernel image in the following location `arch/<arch>/boot`
- `make install` this is rarely used in embedded dev as the kernel image is single file, But still can be done by modifying its behavior `arch/<arch>/boot/install.sh`
- You can install all the configured modules by the following command

```
make INSTALL_MOD_PATH=<dir>/ module_install
```

- The above line direct the module installation on the path provided by the INSTALL_MOD_PATH variable and this is important to avoid installation in host root path

Embedded Linux Kernel

Building - Kernel Image

- Most of the embedded system uses U-Boot as its second stage boot loader
- U-Boot require the kernel image to be converted into a format which it can load. This converted format is called as uImage
- The discussion done here is on how create the uImage from vmlinux
- vmlinux is the output of the kernel compilation which you would find on the root directory of the kernel directory
- vmlinux consists of multiple information like ELF header, COFF and binary

Embedded Linux Kernel

Building - Kernel Image

- So it required to extract the binary file from the vmlinux first, Which is done by the following command

```
arm-linux-objcopy -O binary vimlinux linux.bin
```

- After extraction the U-Boot header can be added using mkimage command, This is done by the following command

```
mkimage -A arm -O linux -T kernel -C none -a
20008000 -e 20008000 -n "Embedded Linux" -d
linux.bin uImage.arm
```

- After all the above steps the kernel image is ready for deployment on target

Embedded Linux Kernel

Deploy

Embedded Linux Kernel

Deploy

- Assuming the host is already configured with TFTP server and Target is running U-Boot with TFTP client
- Copy uImage.arm in /var/lib/tftpboot/
- Copy the kernel image to the target board as mentioned below

```
U-boot> tftp <TEXTBASE_ADDRESS> uImage.arm
```

- **TEXTBASE_ADDRESS** is defined configuring u-boot
- Once the image is transferred you can boot the image as

```
U-boot> bootm
```

- Your kernel should be loaded and executed now :)

File Systems

File Systems

- Introduction
- Building FS from scratch

Introduction

File Systems

Introduction

- File system is an approach on how the data can be organized in order to have a meaningful read or write in a system
- File systems provides a very easy way of identifying data like where it begins and ends
- The group of such data can be called as “Files”
- The method used to manage these groups of data can be called as “File systems”

File Systems
Introduction

- There are several kinds file system having different structure and logic, properties of speed, flexibility, security, size and more
- The most commonly used media to store the data are
  - Storage Devices
    - Magnetic Tapes
    - Hard Drive
    - Optical Discs
    - Flash Memories
    - RAM (Temporary)
  - Network like NFS
  - Virtual like procfs







File Systems
Introduction

- An OS can support more than one file system
- In this topic we are going to concentrate on Unix and Unix like file systems
- File systems can be discussed in the following context
  - Contents
  - Types
  - Partitions

File System Contents

Linux Directory Structure

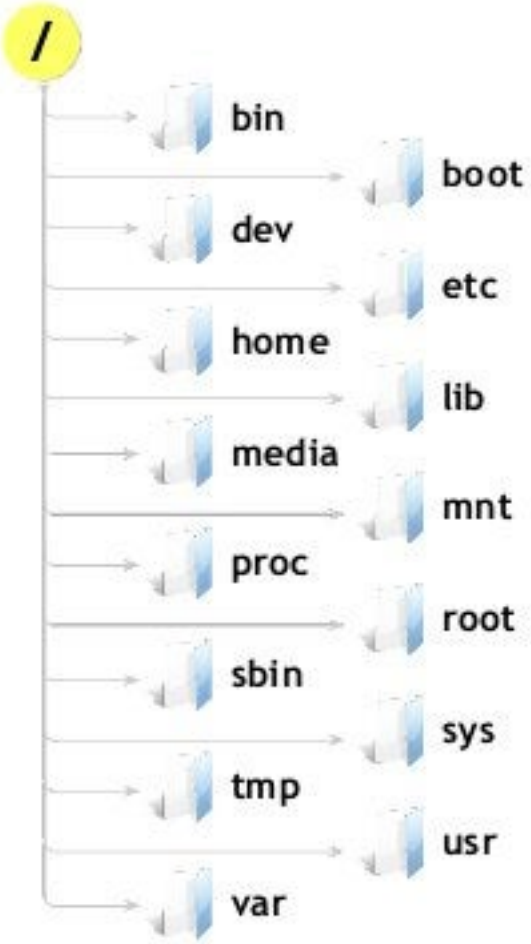
File System Contents

Linux Directory Structure

- When it is the matter of data, different operating system uses different approaches to organize it
- In Unix and Unix like systems, applications and users see a single global hierarchy of files and directories, which can be composed of several file systems.
- The access of the data are done using a process called as mounting
- The location on where the data should be located called as mount point is to be informed to the OS.
- The following slides discuss about the Linux Directory Structure

File System Contents

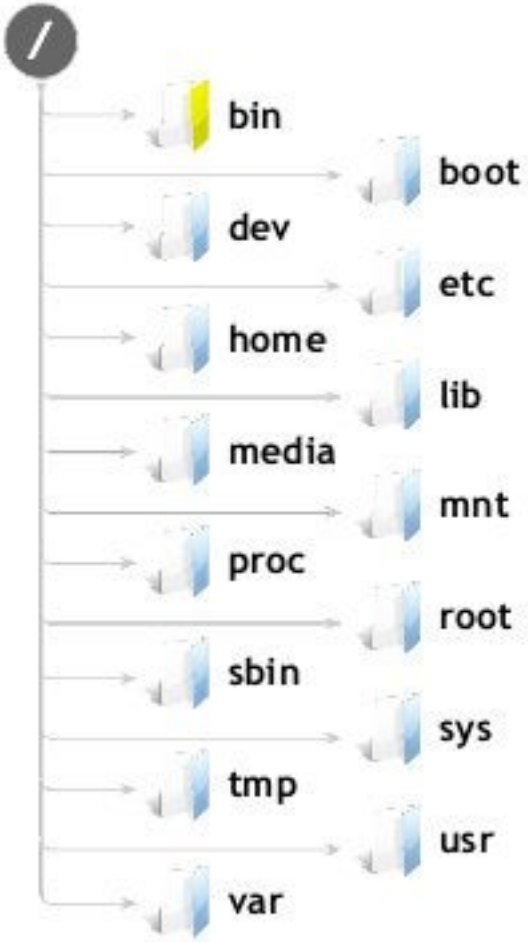
Linux Directory Structure



- The left side of the slide shows the most important files in Linux system
- The organization of a Linux root file system in terms of directories is well defined by the **Filesystem Hierarchy Standard**
- Most Linux systems conform to this specification
  - Applications expect this organization
  - It makes it easier for developers and users as the file system organization is similar in all systems

File System Contents

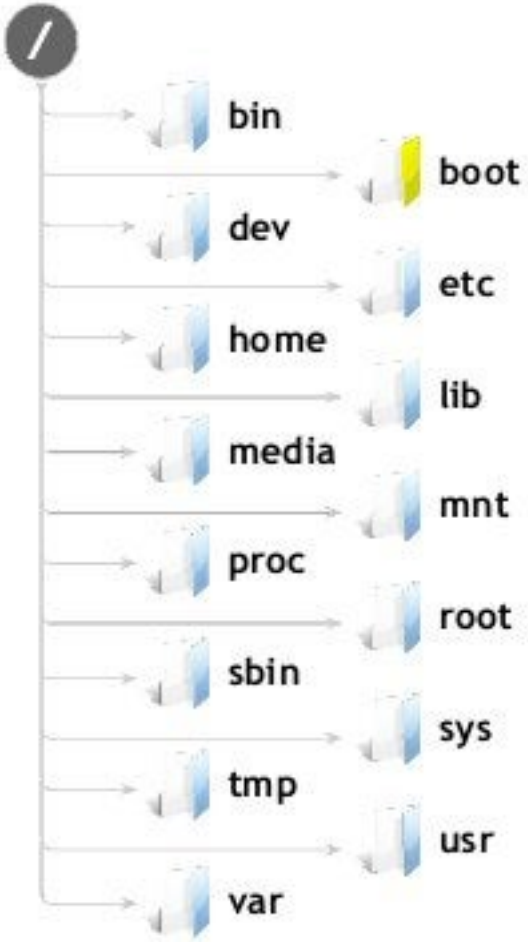
Linux Directory Structure



- Place for most commonly used terminal commands
- Common Linux commands you need to use in single-user modes are located under this directory.
- Commands used by all the users of the system are located here.
- Examples:
  - ls
  - ping
  - grep
  - cp

File System Contents

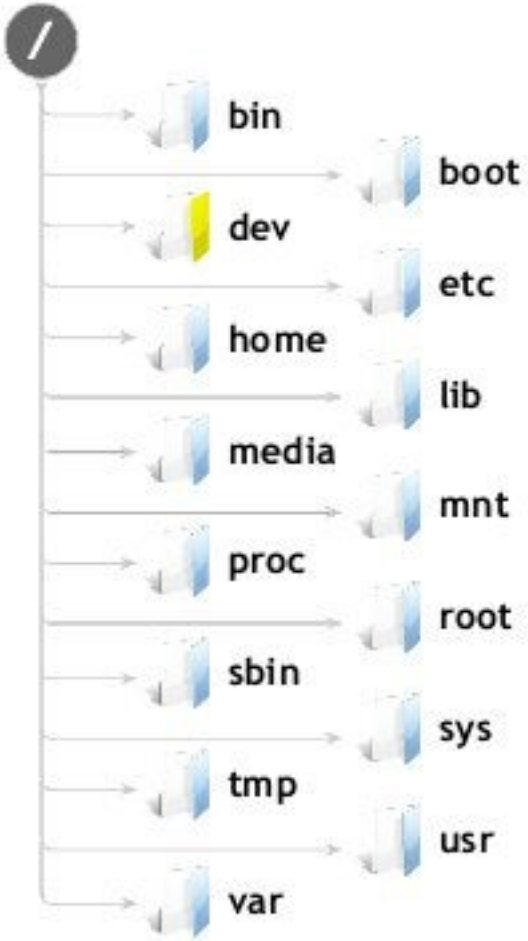
Linux Directory Structure



- Contains files needed to start up the system, including the Linux kernel, a RAM disk image and bootloader configuration files
- Kernel image (only when the kernel is loaded from a file system, not common on non-x86 architectures)

File System Contents

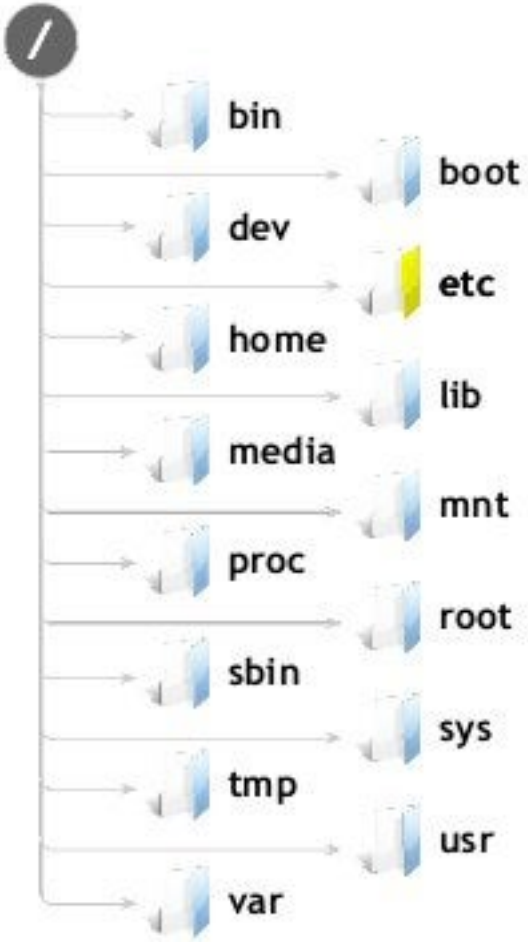
Linux Directory Structure



- Device files
- These include terminal devices, usb, or any device attached to the system.
- Examples:
  - /dev/tty1
  - /dev/usbmon0

File System Contents

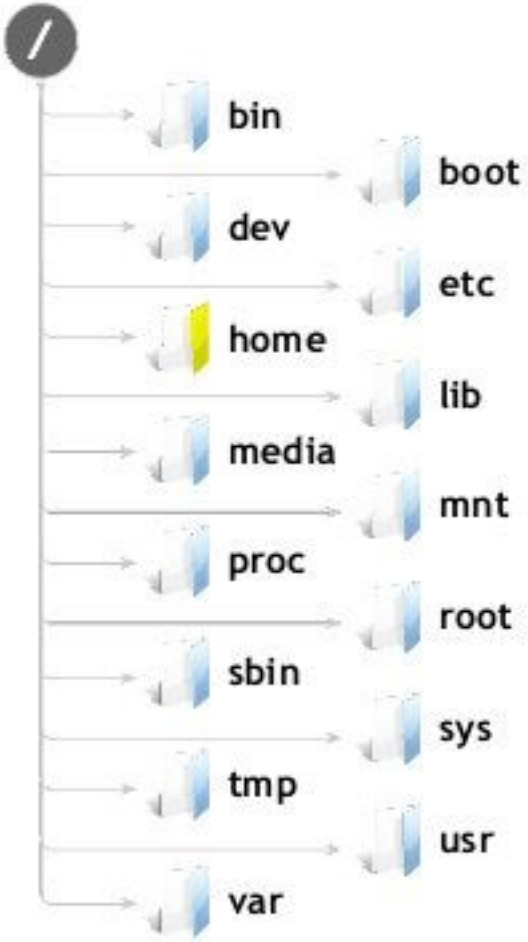
Linux Directory Structure



- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- Examples:
  - /etc/resolv.conf
  - /etc/logrotate.conf

File System Contents

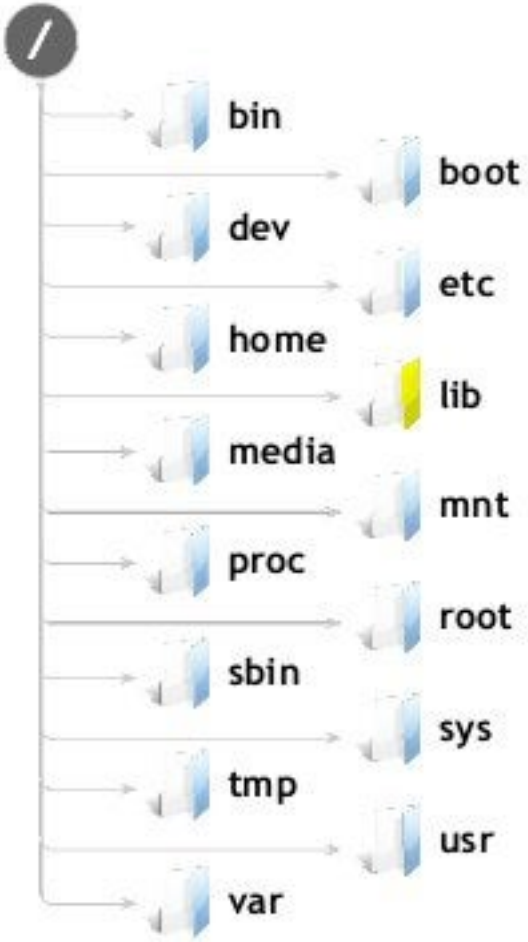
Linux Directory Structure



- Home directories for all users to store their personal files.
- Example:
  - /home/arun
  - /home/adil

File System Contents

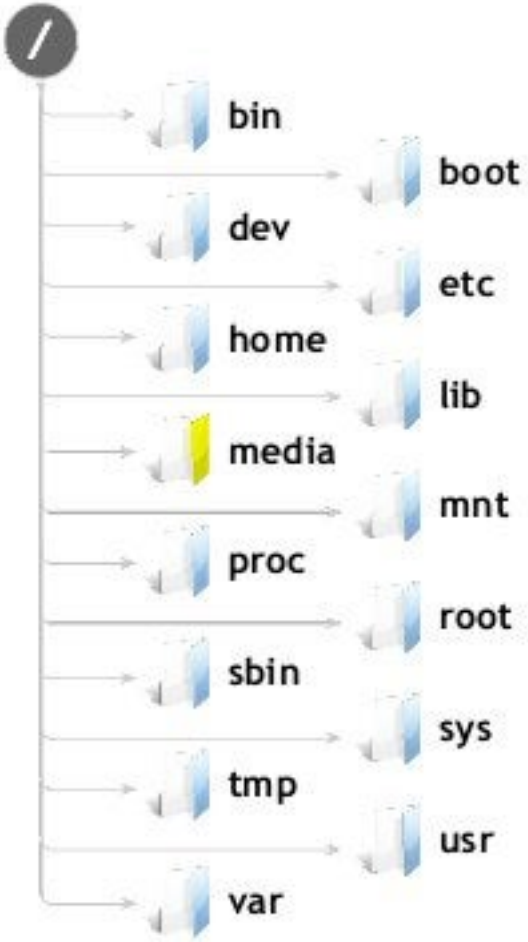
Linux Directory Structure



- Contains library files that supports the binaries located under /bin and /sbin
- Library file names are either ld* or lib*.so.*
- Examples:
  - ld-2.11.1.so
  - libncurses.so.5.7

File System Contents

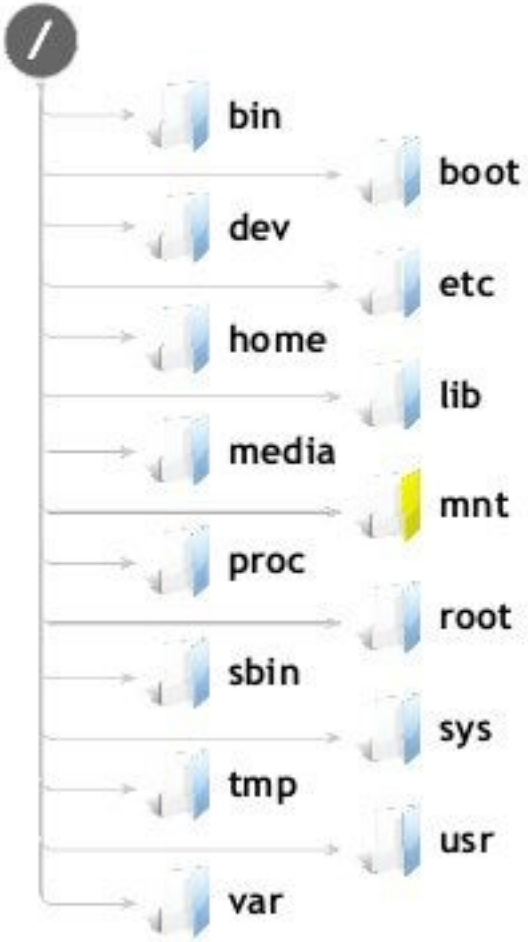
Linux Directory Structure



- Temporary mount directory for removable devices.
- Examples:
  - /media/cdrom
  - /media/floppy
  - /media/cdrecorder

File System Contents

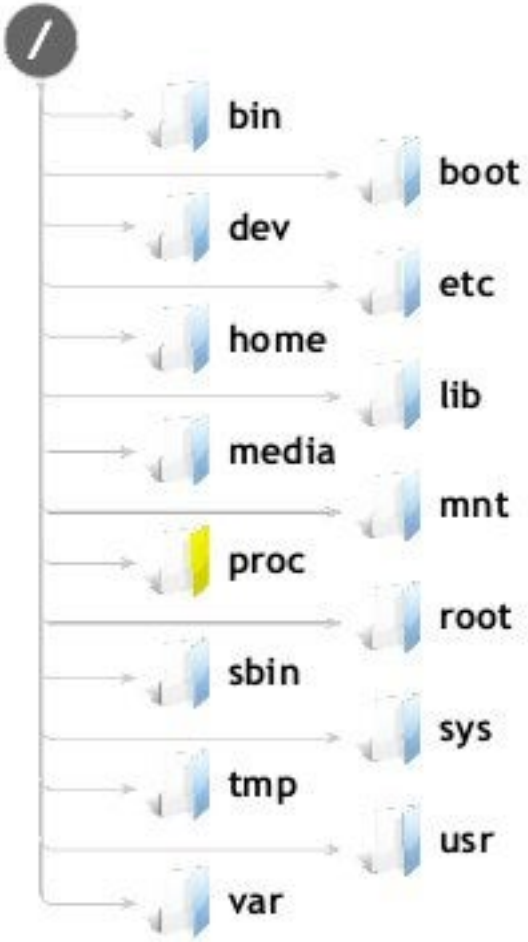
Linux Directory Structure



- Temporary mount directory where sysadmins can mount file systems.

File System Contents

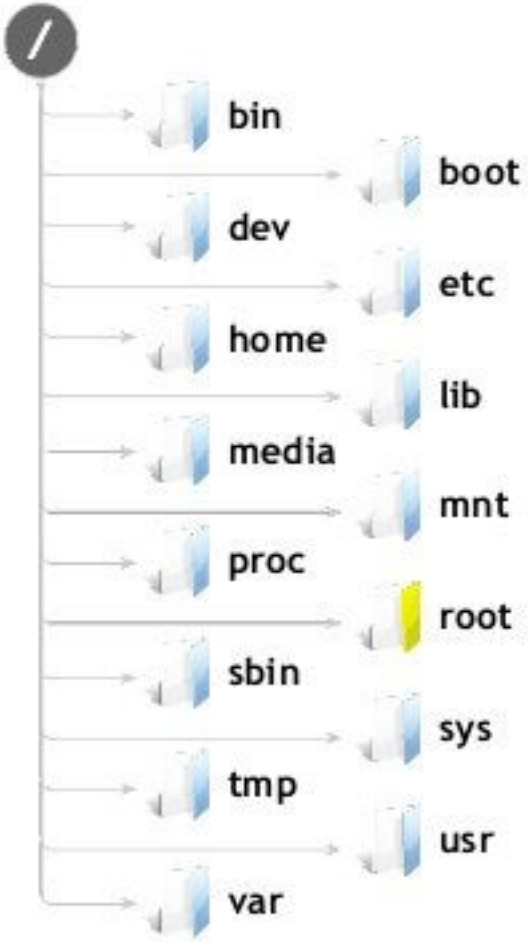
Linux Directory Structure



- Contains information about system process.
- This is a pseudo file system contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
- This is a virtual file system with text information about system resources. Examples:
  - /proc/uptime

File System Contents

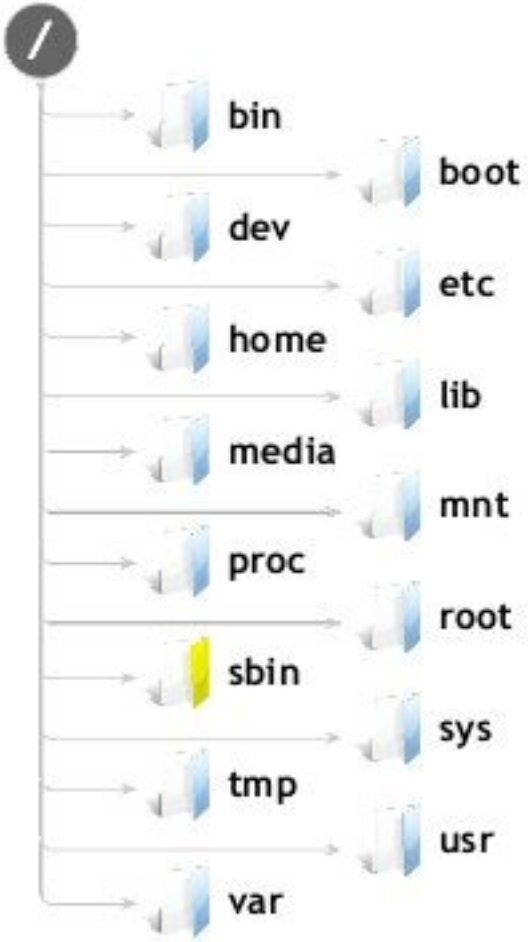
Linux Directory Structure



- Root user's home directory

File System Contents

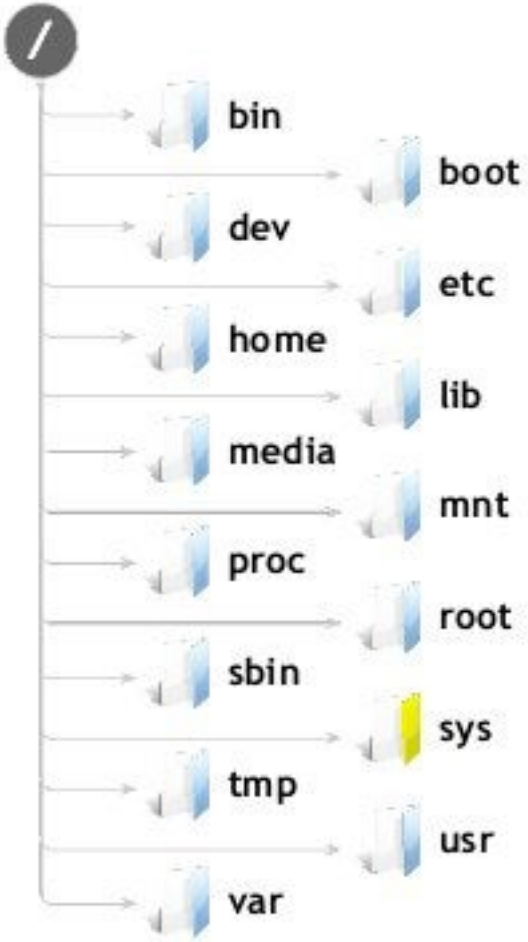
Linux Directory Structure



- Just like /bin, /sbin also contains binary executables
- But, the Linux commands located under this directory are used typically by system administrator, for system maintenance purpose.
- Examples:
  - iptables
  - reboot
  - fdisk
  - ifconfig
  - swapon

File System Contents

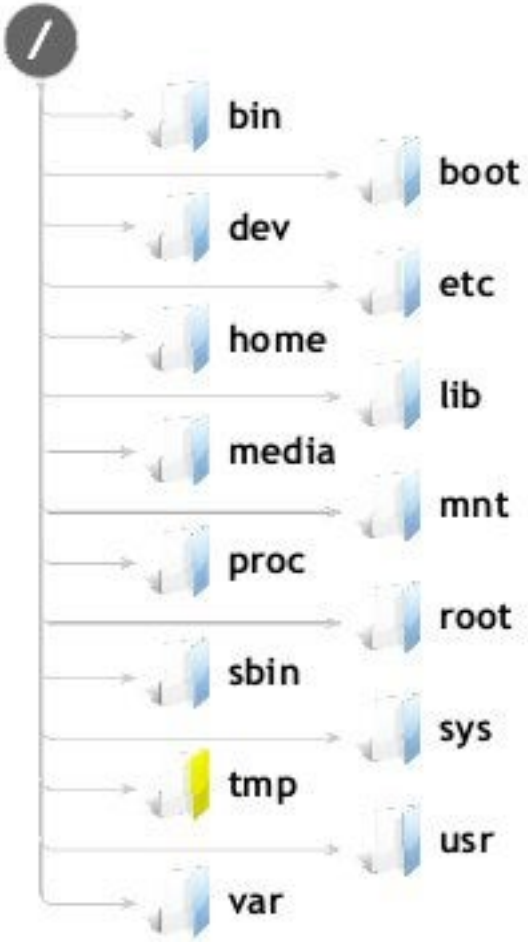
Linux Directory Structure



- Mount point of the sysfs virtual file system

File System Contents

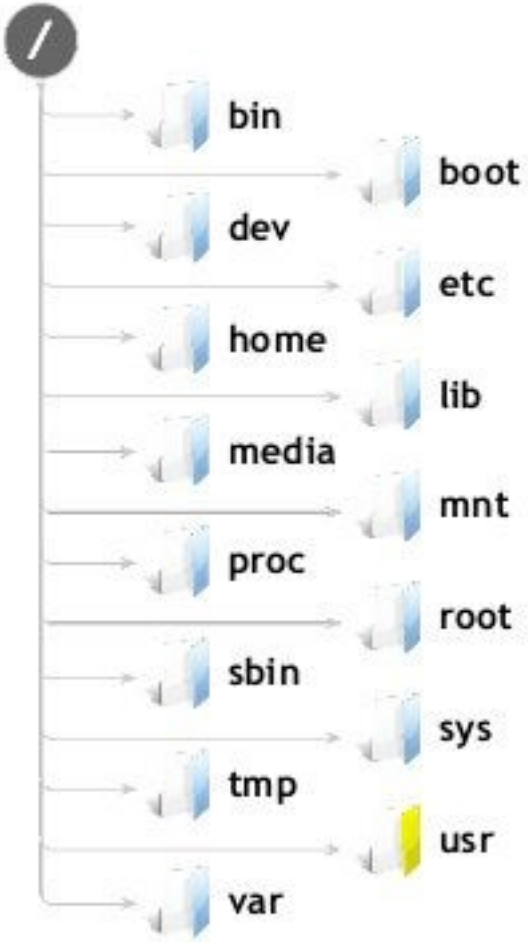
Linux Directory Structure



- Directory that contains temporary files created by system and users.
- Files under this directory are deleted when system is rebooted.

File System Contents

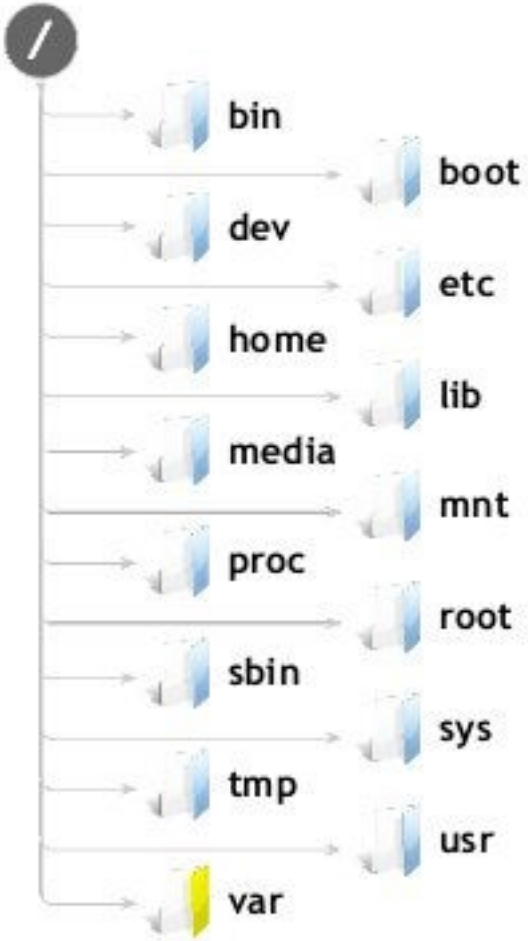
Linux Directory Structure



- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For Examples: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, look under /usr/sbin. For Examples: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2

File System Contents

Linux Directory Structure



- var stands for variable files.
- Content of the files that are expected to grow can be found under this directory.
- This includes —
  - /var/log - system log files
  - /var/lib - packages and database files
  - /var/mail - emails
  - /var/spool - print queues
  - /var/lock - lock files
  - /var/tmp - temp files needed across reboots

File System Types

File System Types
Introduction

- Variety of choices available based on
  - Speed
  - Size
  - Reliability
  - Access restrictions
- Some types are specifically targeted based on the storage media
- The following slide discuss some of the most commonly used types

File System Types
Introduction

- RAM File Systems
  - initrd
  - initramfs
- Disk File Systems
  - ext2
  - ext3
  - ext4
  - Flash File Systems
    - Intro to MTD
    - JFFS2

File System Formats
Variants

- Distributed File Systems
  - NFS
- Special Purpose File Systems
  - squashfs

File System Types

RAM File Systems

File System Types

RAM File Systems - initrd

- Abbreviated as initial RAM disk
- RAM based block device with fixed size formatted and mounted like disk at /dev/ram
- The size problem
  - Sometimes might be a waste of space if its not fully utilized
  - You can extend the limit later on run time as it has to be reformatted
- Requires a file system driver compiled statically into kernel to interpret data at run time
- ext2 format mostly used for creation

File System Types
RAM File Systems - initrd

- Typical boot process using initrd is as followed
  - Bootloader loads the kernel and initrd into ram
  - Kernel converts the initrd into a normal RAM disk and frees the memory used by initrd
  - initrd is mounted as read write as root
  - /linuxrc is executed (could be a executable, shell script)
  - linuxrc mounts the real root file system
  - linuxrc places the root file system at the root directory using the pivot_root system call
  - The normal boot is done on the root file system (run /sbin/init)
  - Initrd file system is removed

File System Types

RAM File Systems - initramfs

- Abbreviated as initial RAM file system
- Successor of initrd, with an advantage of flexible size which can grow and shrink and does not need a driver
- Gets integrated with kernel image
- cpio archive of initial file system that gets loaded into memory during Linux startup process
- Kernel mounts the file system and starts init
- Faster booting as the it is loaded at boot time, hence applications can start faster

File System Types

RAM File Systems - initramfs

- Could be used as intermediate stage before loading real file system
- The path of file system to be included has to be set while kernel configuration
- The kernel build process would take care of integrating it with the kernel image
- Just search INITRAMFS on the configuration menu to know where to set the path
- Just build the kernel once the path is set

File System Types

Disk File Systems

File System Types

Disk (Block) File System

- Has ability to store data randomly at addresses in a short amount of time
- Multiple users (or processes) access to various data on the disk without regard to the sequential location of the data
- Disk file systems are usually block-oriented.
- Files in a block-oriented file system are sequences of blocks, often featuring fully random-access read, write, and modify operations
- No erasing required before write

File System Types
Disk (Block) File System

- Examples
  - Hard drives, floppys, RAM Disks
  - Compact Flash, SD Cards
    - Though this device are based of flash devices, the integrated controllers take care of the low level handling of data like block devices

File System Types

Disk (Block) File System - ext2

- Abbreviated as Second Extended File System
- Matured and stable GNU/Linux file system
- Does not support journaling support or barriers
- Not recommend as for / and /home partitions since it take long time for file system check
- Most compatible choice for /boot partition
- Recommend for very small partitions (say < 5 MB), because other file system types need more space for metadata
- Used to be the default file system in several Linux distribution

File System Types

Disk (Block) File System - ext2

- ext2 is still preferred choice of flash based storage due the absence of journal

File System Types

Disk (Block) File System - ext3

- Abbreviated as Third Extended File System
- An extension of ext2 with journaling support and write barriers
- Backward compatible with ext2 and extremely stable
- ext2 can be converted to ext3 directly without backup

File System Types

Disk (Block) File System - ext4

- Abbreviated as Fourth Extended File System
- Compatible with both ext2 and ext3 and used as default file system in many distributions
- Supports huge individual file data and overall file system size
- ext4 allows unlimited number of sub directories compared to ext3
- Has many new features like
  - Journal checksum
  - Fast fsck etc.
- Can turn off the Journal feature if required

File System Types

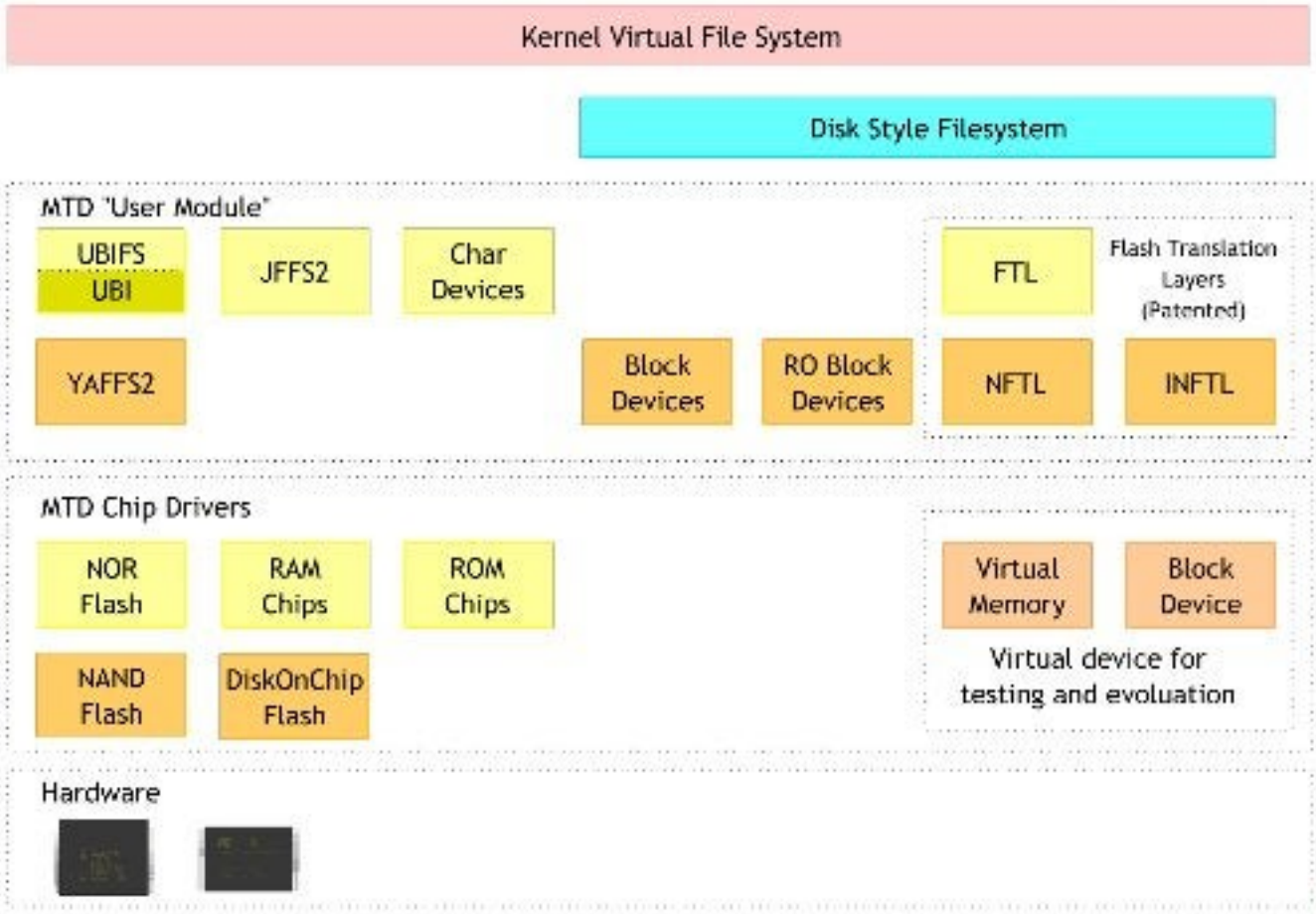
Disk (Block) File System - Flash File System

- Flash file system is designed for storing files on flash memory-based storage devices
- They are optimized for the nature and characteristics of flash memory for the following reasons
  - Erasing blocks
  - Random access
  - Wear leveling
- Examples
  - NAND and NOR Flashes
- In Linux the flash memories use a device file called MTD for interaction

File System Types
Flash File System - MTD

- MTD stands form Memory Technology Device
- Created to have abstraction layer between the hardware specific device drivers and higher-level applications
- MTD helps to use the same API with different flash types and technologies
- MTD does not deal with block devices like MMC, eMMC, SD, Compact Flash etc., Since these are no raw flashes but they have Flash Translation layer. The FTL make these devices look like block devices

File System Types
Flash File System - MTD



File System Types
Flash File System - MTD

- MTD devices can be looked at /proc/mtd
- Two types of drivers created
  - mtdchar drivers
    - Can be seen in /dev as mtd<*n*> or mtd<*n*>ro with major number 90
    - Even minor number for **rw** type and odd minor number for **ro** type
  - mtdblock drivers
    - Can be seen in /dev as mtdblock<*n*> allowing rw in block level
    - Does not handle bad block check and any wear-leveling

File System Types
Flash File System - MTD

- MTD device are usually partitioned
- These partitions can be used fro different purposes like boot code area, kernel image, data partitions etc
- The MTD partitions can be described by
  - Specifying in board device tree
  - In kernel code (Hard code)
  - Through kernel command line (Modification)
- Each partition becomes a separate MTD device

File System Types
Flash File System - MTD - Partitions

- Using Device Tree

File System Types

Flash File System - JFFS2

- Abbreviated as Journalling Flash File System 2
- Standard filesystem for MTD flash
- Compression support for zlib, rubin, rtime and lzo
- Better performance, power down reliability, wear leveling and ECC

File System Types

Distributed File Systems

File System Types

Distributed File System

- A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.
- Data accessed by all users can be kept on a central host, with clients mounting this directory at boot time
- Data consuming large amounts of disk space may be kept on a single host

File System Types

Special Purpose File Systems

File System Types
Special Purpose File System

- Sometimes the way we handle data might depend on the requirements
- There some file system types which can offer different features like
  - Compressions
  - Backups
  - Access etc.

File System Types

Special Purpose File System - squashfs

- Read-only compressed file system for block devices
- Can be used for read only data like binaries, kernel etc
- Very good compression rate and read access performance
- Mostly used in live CDs and live USB distros
- Can support compressions like
  - gzip
  - LZMA
  - LZO etc

File System Partitions

File System Partitions

- Logically dividing the available storage space into sections that can be accessed independently to one another for
  - File management
  - Multiple users
  - Data Security
  - Efficiency and reliability etc.,
- Entire storage can be created as a single partition if required, But multiple partition can help in number of way like dual or multiboot system, swap etc.

File System Partitions
Linux System

- Two major partitions
  - Data partition - Linux data partition
    - A root partition and one pr more data partition
  - Swap partition - Expansion of physical memory, extra memory on storage (like virtual RAM)
    - Rarely used in embedded system because of the nature of the storage devices used

File System Partitions

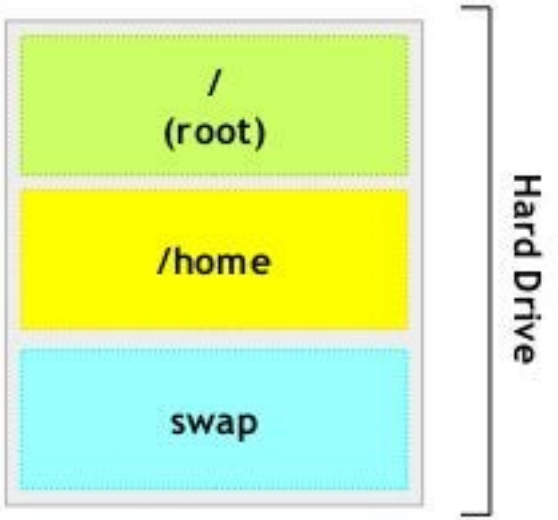
Mount Points

- All partitions are attached to the system via a **Mount Point**
- A mount point defines the place of a particular data set in a file system
- Usually all the partitions are connected through the **root (/)** partition
- The / will have empty directories which will be the starting point of partitions
- Some core partitions can be mounted on startup by describing it in **/etc/fstab**

File System Partitions

Schemes - Desktop - Example 1

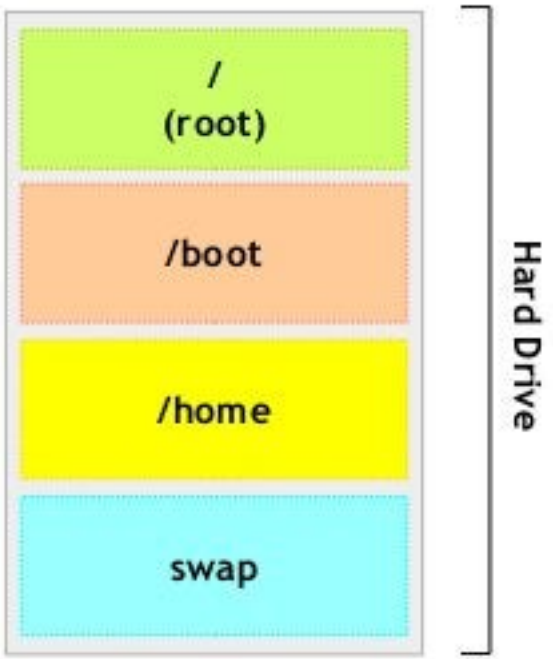
- One possible Desktop scheme is shown here
- Simplest and standard one to have
- Partition for OS which get mounted as / (root)
- Data partition mounted as /home
- Augment to RAM, referred and mounted as swap



File System Partitions

Schemes - Desktop - Example 2

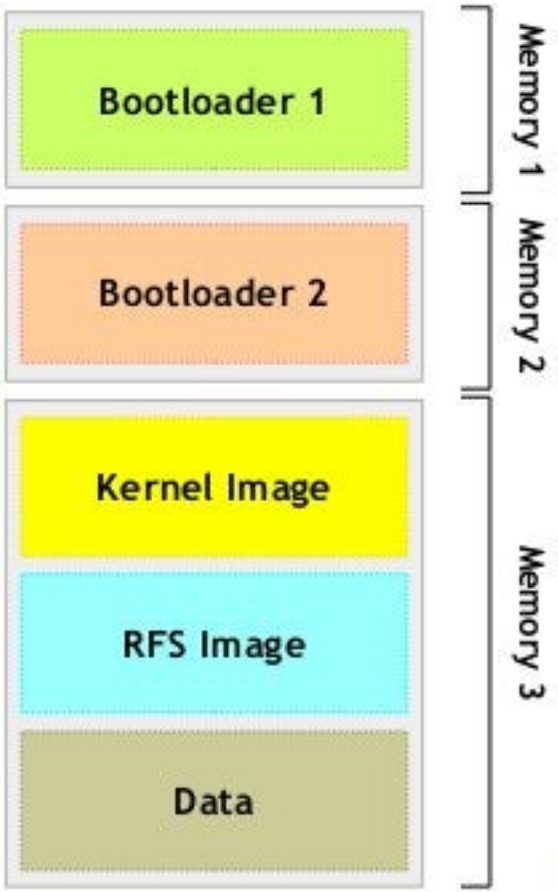
- Similar to previous one with a separate boot partition which will contain the boot images



File System Partitions

Schemes - Embedded - Example 1

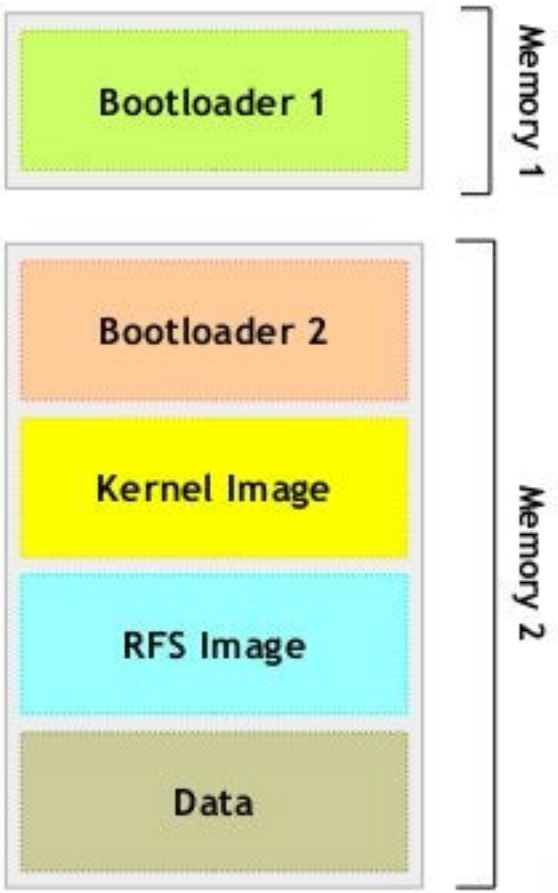
- The partition scheme in embedded systems depends on the system architecture
- This scheme shows two stages of boot loaders each in different memories
- Rest all the sections goes in other memory



File System Partitions

Schemes - Embedded - Example 2

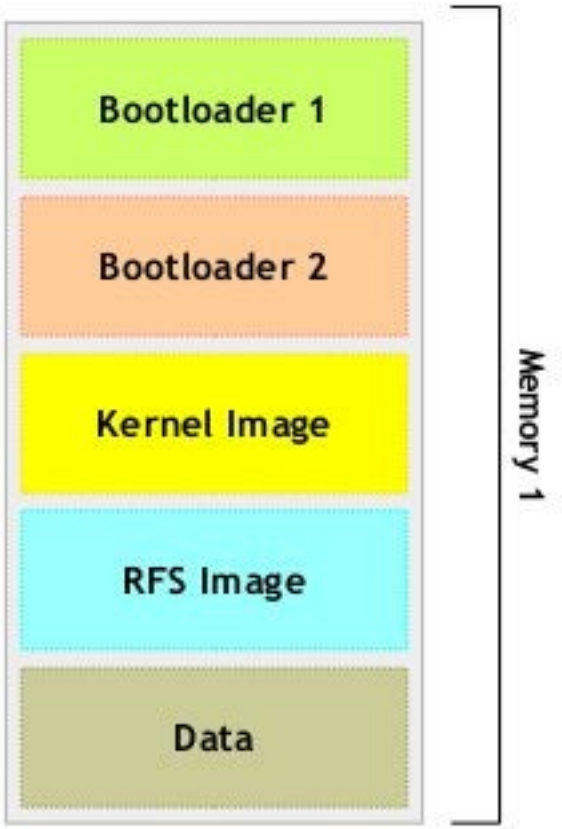
- This scheme shows stage 1 of boot loader in different memory
- Rest all the sections including stage 2 boot loader goes in other memory



File System Partitions

Schemes - Embedded - Example 3

- This scheme shows all the contents share in a single memory



EMERTXE

File Systems
Building from scratch

- File systems can be created manually but
  - Would be time consuming
  - Lots of dependencies
  - Lots of components have to be integrated like binaries, libraries, scripts, configuration files etc.
  - Customization for embedded would be challenging
  - Many more..
- So BusyBox is an alternative solution

BusyBox
Introduction

BusyBox
Introduction

- Often called as the Swiss Army Knife of Embedded Linux
- BusyBox combines tiny versions of many common UNIX utilities into a single small executable
- It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc
- Written with size-optimization and limited resources in mind

BusyBox
Introduction

- The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; with expected functionality and behave very much like their GNU counterparts
- Provides a fairly complete environment for any small or embedded system
- Extremely modular so you can easily include or exclude commands (or features) at compile time
- This makes it easy to customize your embedded systems
- Sizes less than < 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc)


BusyBox
Building

BusyBox

Building - Configuration

- Download the latest stable sources from http://busybox.net
- You may try the following targets for configuration

  `make defconfig`

- Configures all options for regular users.

  `make allnoconfig`

- Unselects all options. Good to configure only what you need.
- Linux kernel like configuration, `make menuconfig` or `make xconfig` also available

BusyBox

Building - Compilation

- BusyBox, by specifying the cross compiler prefix.

  ```
  make CROSS_COMPILE=<cross_compile_path>
  ```

- *cross_compile_path* could be the command itself if already exported in PATH variable, else you can specify the installation path of command
- For example for arm platform it would look like

  ```
  make CROSS_COMPILE=arm-linux-
  ```

- The cross compiler prefix can be set in configuration interface if required

  BusyBox Setting → Build Option → Cross Compiler prefix

BusyBox
Installation

- To install BusyBox just type

  ```
  make install
  ```

- The default installation path would the current directory, will should see _install directory
- The installation path can be customized in configuration if required as mentioned below

  BusyBox Setting → Installation Option → BusyBox installation prefixs

- The installation directory will contain Linux like directory structure with symbolic links to busybox executable

Stay connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Branch office:
Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046

Corporate Head Office:
Emertxe Information Technologies,
No-83, 1st Floor, Farah Towers,
MG Road,
Bangalore, Karnataka 560001
T: +91 809555 7333 (M), +91 80 4128 9576
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

EMERTXE

Thank You