Embedded Operating System

Linux

Team Emertxe

Contents

Contents

Embedded Systems Introduction

Embedded Operating System

Introduction to Embedded Systems (ES)

- What is ES
  - Examples
- GPS vs ES
- Components of ES
- Need of an OS in ES
- Introduction to Embedded Linux
- Typical Components of Embedded Linux System

Embedded Operating System

Introduction to ES - Definition

“Any combination of **Hardware** and **Software**

which is intended to do a

Specific Task

can be called as an **Embedded System**”

Embedded Operating System

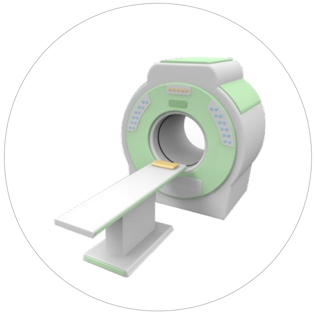
Introduction to ES - Examples

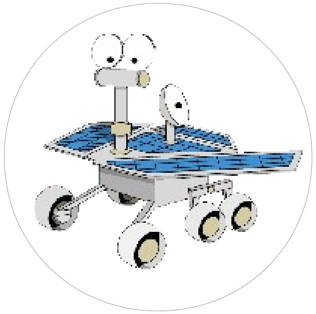








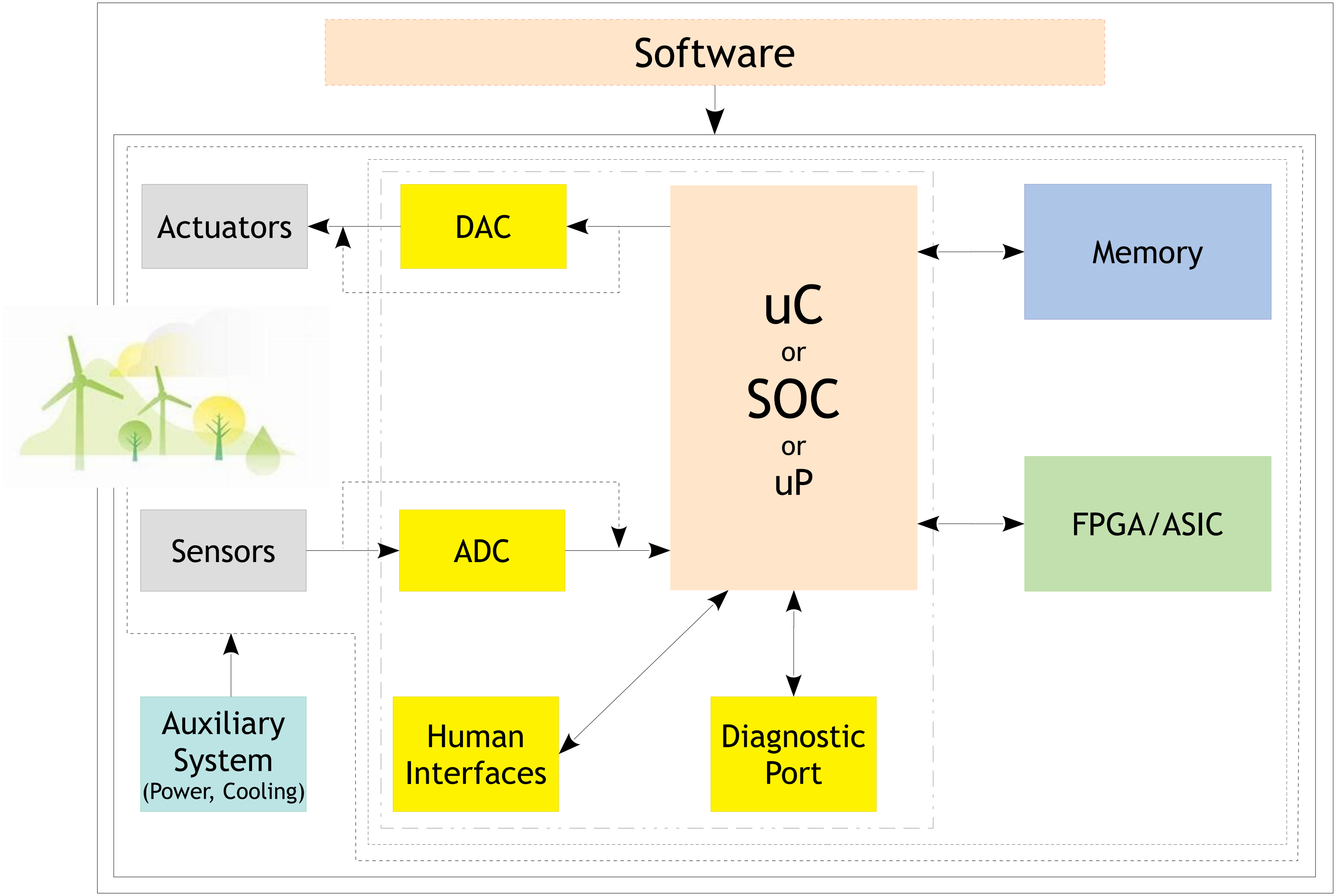



- Can you mention one example of a General Purpose System?
- Does the size of ES really matter?

Embedded Operating System

Components of Embedded System



Embedded Operating System

EOS – Why?

- Is it necessary to have an OS running in an Embedded System?

Linux as Embedded OS

Embedded System

EOS – Linux – Why?

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

Embedded Operating System

EOS – Linux – Ported Architectures

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

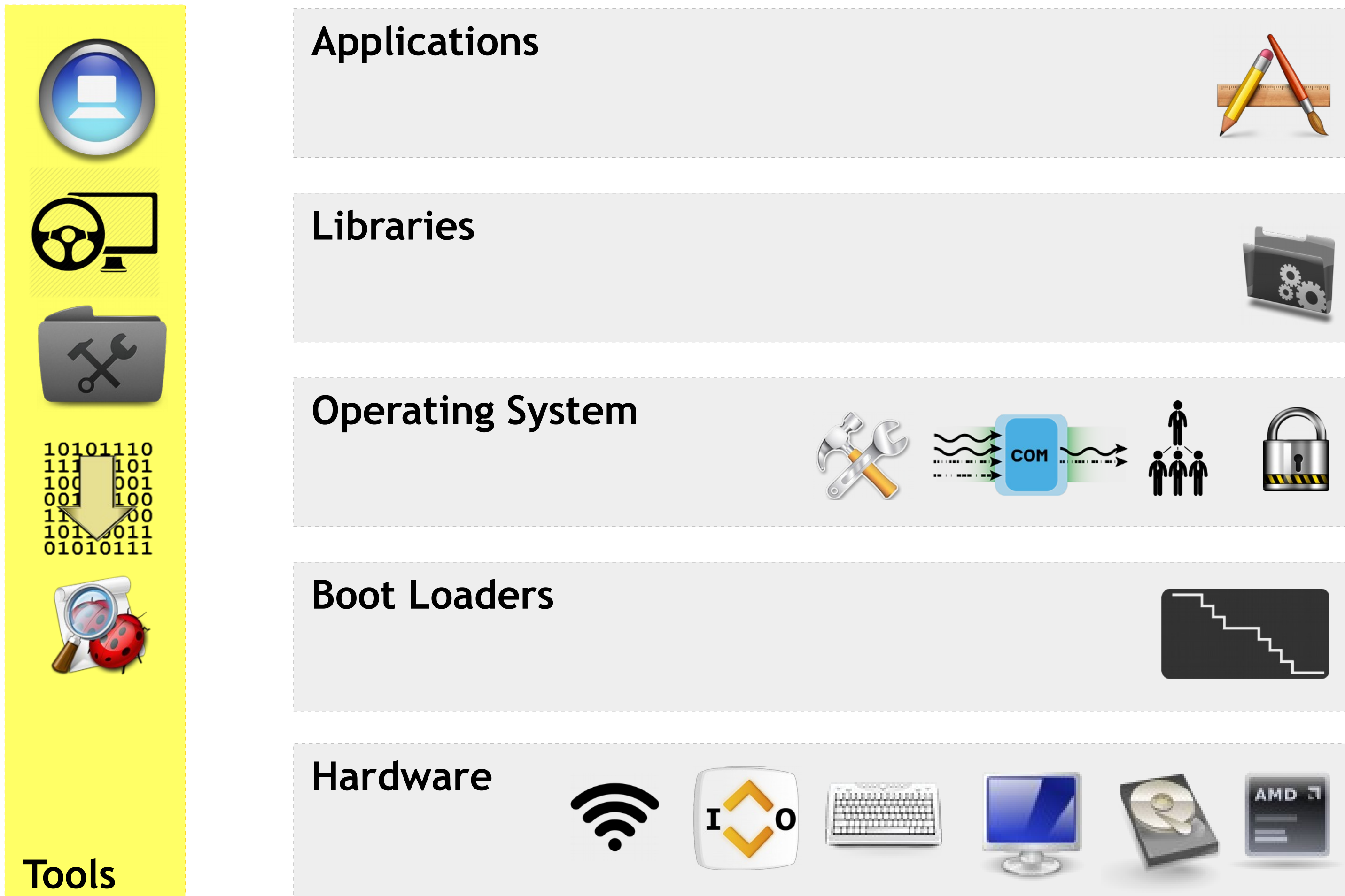
Embedded Operating System

EOS – Linux – Choices to make

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

Embedded Operating System

Typical System Components



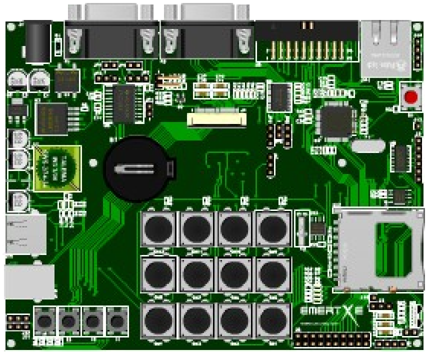
Target Development & Environment

Embedded Development & Environment

- Hardware Tools and Interfacing
- Software Environments Tools

Possible Connections

- Serial
- Network
- USB
- JTAG
- Emulators
- OTA



Embedded Development & Environment
Software Environment Tools

- **Serial**
  - **minicom**
  - **gtkterm**
- **Network**
  - **TFTP**
  - **NFS**

Toolchain

Introduction

Toolchain

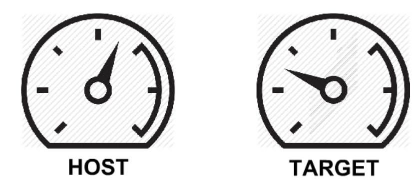
Introduction

- Set of applications used to perform a complex task or to create a product, which is typically another computer program or a system of programs.
- Tools are linked (or chained) together by specific stages
- Output or resulting environment state of each tool becomes the input or starting environment for the next one
- By default the host contains some development tools which ate called as native toolchain

Toolchain

Introduction

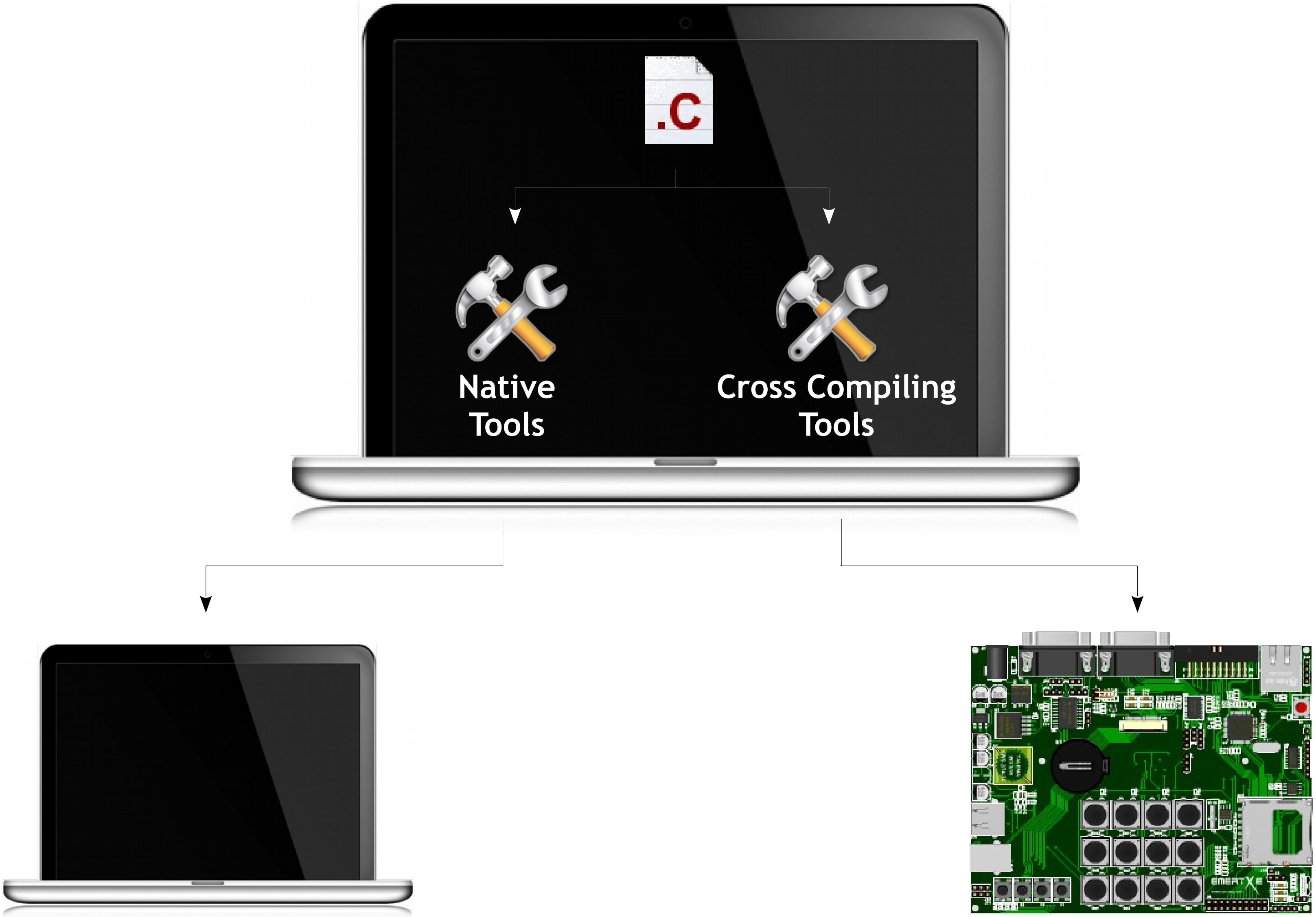
- When you use these tool the code generation would be for native architecture (say x86)
- In case of developing embedded systems these toolchains does not serve the purpose
- The targets some times might have the following restrictions too!
  - Less memory footprints
  - Slow compared to host
  - You may not wish to have in a developed system finally



- So cross compiling toolchain are used

Toolchain

Introduction - Flow



Toolchain

Components

Toolchain

Components

- Compiler
- Kernel Headers
- Binary Utilities
- Debuggers
- C / C++ Libraries

Toolchain

Components – GCC

- Abbreviated as GNU Compiler Collection, one of the famous free software compiler
- Can compile many programming languages and generate code for many different types of architecture

Toolchain

Components – Binary Utilities

- Set of programming tools for creating and managing binary programs, object files, libraries, profile data, and assembly source code
- Includes an assembler, linker and several other software tools
- Often used with a compiler and libraries to design programs for Linux
- **GNU Binary Utilities** are called **binutils**

Toolchain

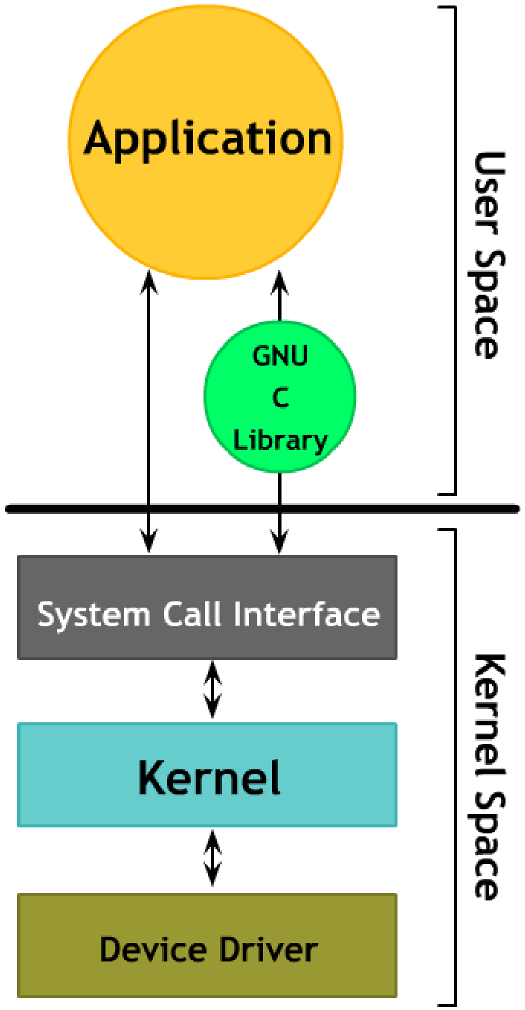
Components – binutils

- as - the assembler, that generates binary code from assembler source code
- ld - the linker
- ar, ranlib - to generate .a archives, used for libraries
- objdump, readelf, size, nm, strings - to inspect binaries. Very useful analysis tools !
- strip - to strip useless parts of binaries in order to reduce their size

Toolchain

Components - Libraries

- The C library is an essential component which provides interface between the applications and the kernel
- Provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services



Toolchain

Components - Libraries

- Several C libraries are available: glibc, uClibc, dietlibc, etc
- The selection of the library has to be made while generation the toolchain as gcc is compiled against the selected library
- Some common used libraries
  - glibc
  - uclibc

Toolchain

Components – Libraries - glibc

- C library from the GNU project
- Good performance optimization, standards compliance and portability
- Well maintained and used on all GNU / Linux host systems
- Require large memory footprint making it not suitable for small embedded systems
- Used in systems that run many different kernels and different hardware architectures
- License: LGPL
- Website: http://www.gnu.org/software/libc/

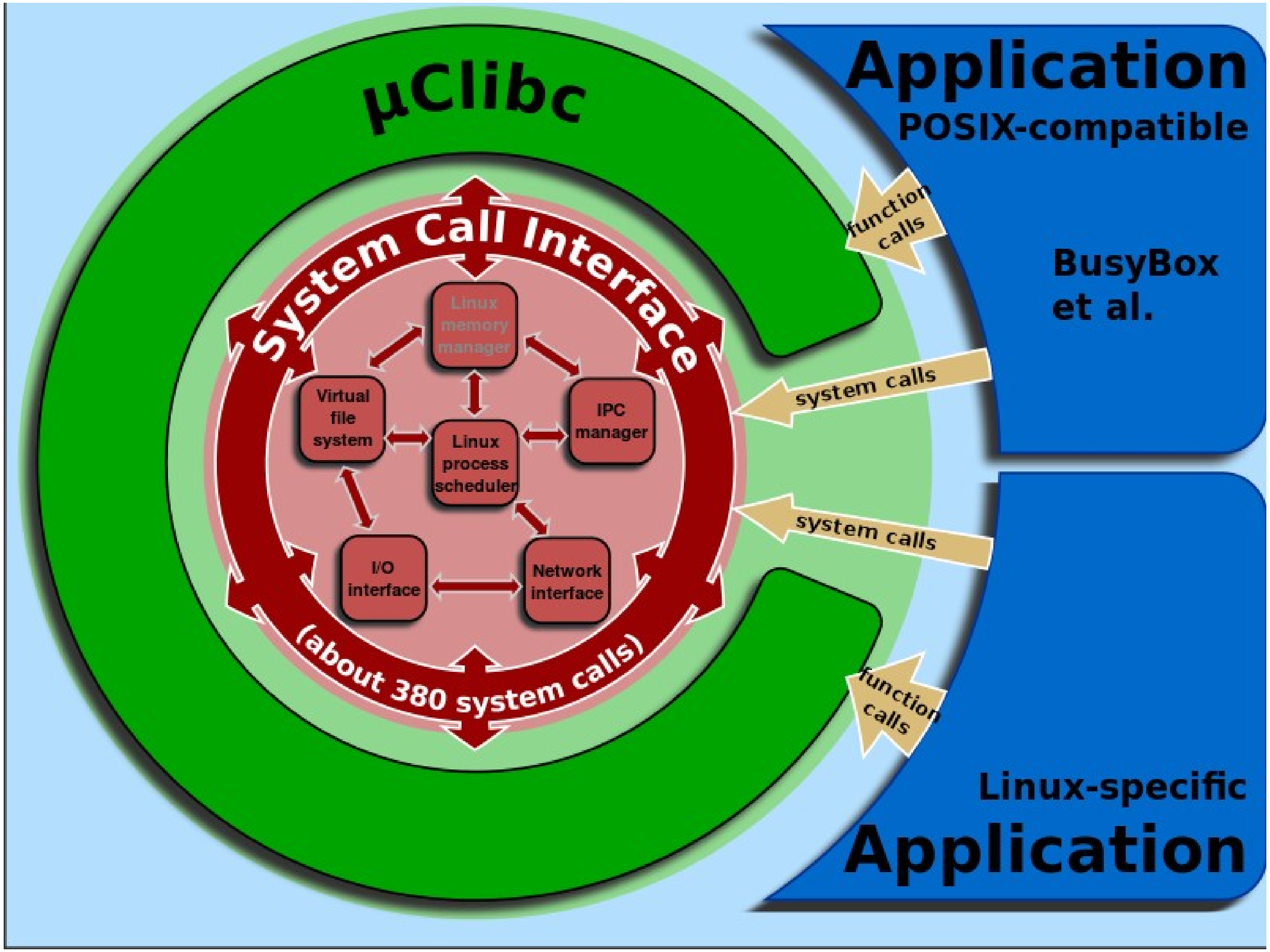
Toolchain

Components – Libraries - uclibc

- Smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc
- Recommended for lower footprint embedded systems
- Excellent configuration options at the cost of ABI differences for different configurations
- Features can be enabled or disabled according to space requirements
- Works on most embedded architectures with embedded Linux
- Focus on size rather than performance with less compile time

Toolchain

Components – Libraries - uclibc



Toolchain

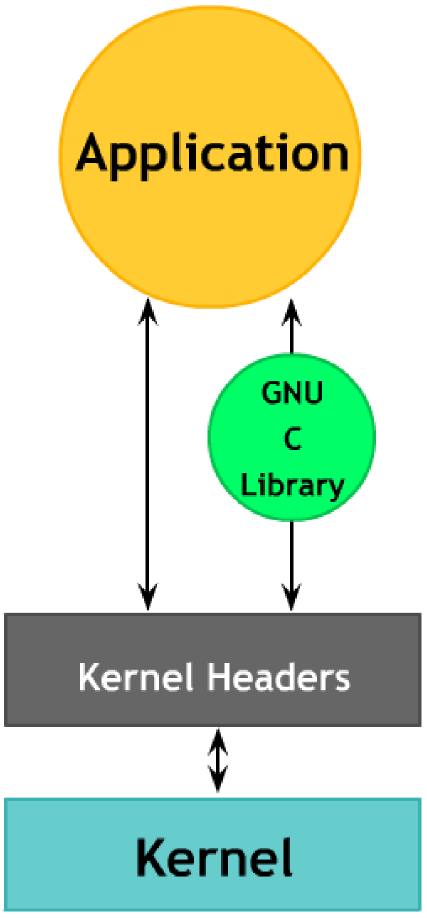
Components – Libraries - uclibc

- Created for support uclinux, a Linux for MMU less system
- Actively maintained, large developer and user base
- Used on a large number of production embedded products, including consumer electronic devices

Toolchain

Components – Kernel Headers

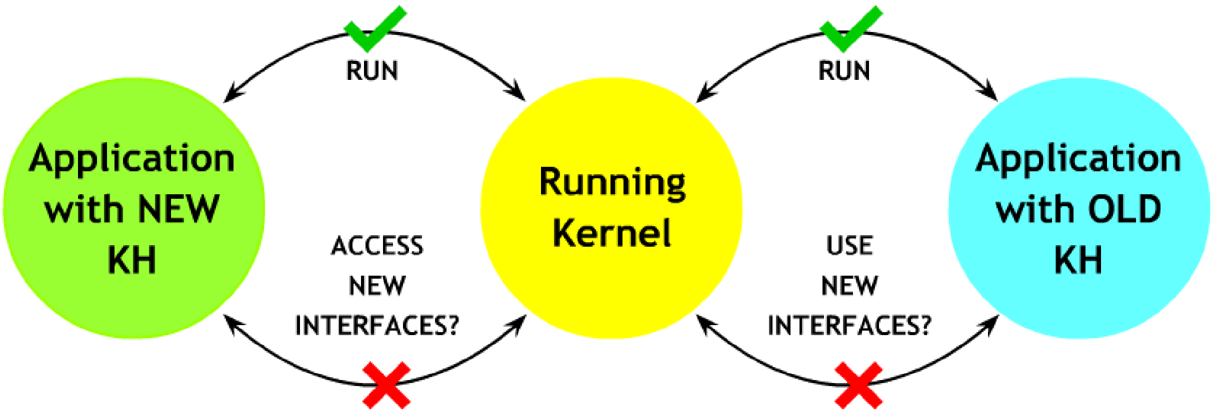
- The compiled applications and the libraries sometimes need to interact with the kernel
- So the Linux kernel need to expose the available interfaces for the libraries and applications to use, like
  - System calls and its numbers
  - MACRO definitions
  - Data structures, etc.



Toolchain

Components – Kernel Headers

- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- You may find these in <linux/...> and <asm/...> and a few other directories corresponding to the ones visible in include/ in the kernel sources
- Compatibility with running kernel has to be considered



Toolchain

Building

Toolchain
Building

- Toolchain building is one of the time consuming process
- A very clear picture of the system architecture is sometimes required
- We might have to identify three systems in this case, like
  - Build System – which is used to create the toolchain
  - Host – which will be used to execute the created toolchain
  - Target – which will execute the binaries created by the toolchain
- So by considering the above points some possible build options are provided in next slide

Toolchain
Building

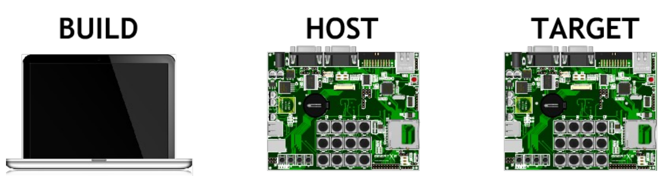
- Possible Build System
- Build Considerations
- Toolchain Build Possibilities
  - Prebuilt
  - Home Built
  - Automated Tools
    - Crosstool NG
    - Buildroot

Toolchain

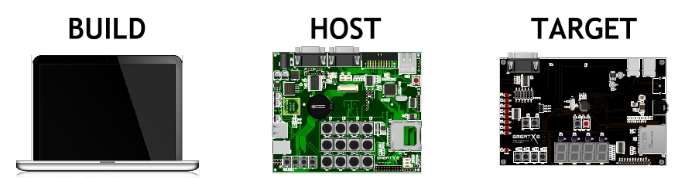
Building – Build Systems

Native Build

Cross Build



Cross Native Build



Canadian Build

Toolchain

Building - Considerations

- Before building the toolchain following decisions have to be made
  - Which library to be used?
  - What version of the components to selected?
  - Certain important configurations like
    - Architecture features like floating point
    - ABI selections
    - Networking features etc.,
- So you might have to put good amount of time in investigations

Toolchain

Building – Pre-built

- Get a pre-built open toolchain either from a web or from your hardware vendor
- This is the easiest solution, as the toolchain is already built, and has supposedly been tested by the vendor
- The drawback is that you don't have flexibility on your toolchain features (which C library? hard-float or soft-float? which ABI?)

Toolchain

Building – Home Built

- Building toolchain from scratch is one of the tiring and time consuming task which might go upto several weeks.
- Several information to be know. Lots of components to be considered to build.
- Proper decision on the components and its configuration have to be made
- Need kernel headers and C library sources
- There are version dependency issues, patches required to make something work etc.
- The order of build is to be known, but have complete freedom of choice

Toolchain

Building – Home Built

- Can get some idea from the crosstool matrix by Kegel (Note this is outdated)

	gcc-2.95.3 cgcc-2.95.3 glibc-2.1.3 binutils-2.15 linux-2.4.26	gcc-2.95.3 cgcc-2.95.3 glibc-2.2.2 binutils-2.15 linux-2.4.26 hdrs-2.6.12.0	gcc-2.95.3 cgcc-2.95.3 glibc-2.2.5 binutils-2.15 linux-2.4.26	gcc-3.2.3 cgcc-3.2.3 glibc-2.2.5 binutils-2.15 linux-2.4.26	gcc-3.2.3 cgcc-3.2.3 glibc-2.3.2 binutils-2.15 linux-2.4.26	gcc-3.2.3 cgcc-3.2.3 glibc-2.3.2 binutils-2.15 linux-2.6.9 tls	gcc-3.3.6 cgcc-3.3.6 glibc-2.1.3 binutils-2.15 linux-2.4.26	gcc-3.3.6 cgcc-2.95.3 glibc-2.2.2 binutils-2.15 linux-2.4.26 hdrs-2.6.12.0	g 3. cg 3. gl 2. bin 2 lin 2.
alpha	FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL	FAIL gdb FAIL	ok gdb
arm	kernel fail	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	FAIL	kernel fail gdb ok	ker fail gdb
arm9tdmi	FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	FAIL	FAIL gdb FAIL	ker fail gdb
arm-iwmmxt	FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL	FAIL gdb FAIL	FA gdb FA
arm-	kernel	kernel	kernel	kernel	kernel	kernel		kernel	ker

Toolchain

Building – Automated Tools

- Lots of open community has built several scripts or more elaborate systems to ease the process of building a toolchain
- More easy procedure since no need of breaking your heads on dependency resolutions
- Provides easy configuration mechanism
- Recipes and patches needed to build a toolchain made of particular versions of the various components are shared and easily available
- Some common automated tools are discussed in following slides

Toolchain

Building – Automated Tools – Crosstool-NG

- Updated version of Crosstool, with a kernel-like menuconfig like configuration system
- Supports uClibc, glibc, eglibc, hard and soft float, many architectures
- Support different target OS: Linux, bare metal
- Debug facilities like native and cross gdb, gdb server
- Actively maintained
- Targeted at building toolchains
- http://crosstool-ng.org/

Toolchain

Building – Automated Tools – Buildroot

- Buildroot is a set of Makefiles and patches that makes it easy to generate a complete embedded Linux system
- Generates root file system images ready to be used
- Complete build system based on the Linux Kernel configuration system and supports a wide range of target architectures
- It automates the building process of your embedded system and eases the cross-compilation process
- Supports **multiple filesystem types** for the root filesystem image

Toolchain

Building – Automated Tools – Buildroot

- 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

Toolchain

Buildroot - Introduction

- Buildroot is a set of Makefiles and patches that makes it easy to generate a complete embedded Linux system
- Generates root file system images ready to be used
- Complete build system based on the Linux Kernel configuration system and supports a wide range of target architectures
- It automates the building process of your embedded system and eases the cross-compilation process
- Supports **multiple filesystem types** for the root filesystem image

Toolchain

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

Toolchain

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

Toolchain

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

Toolchain

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

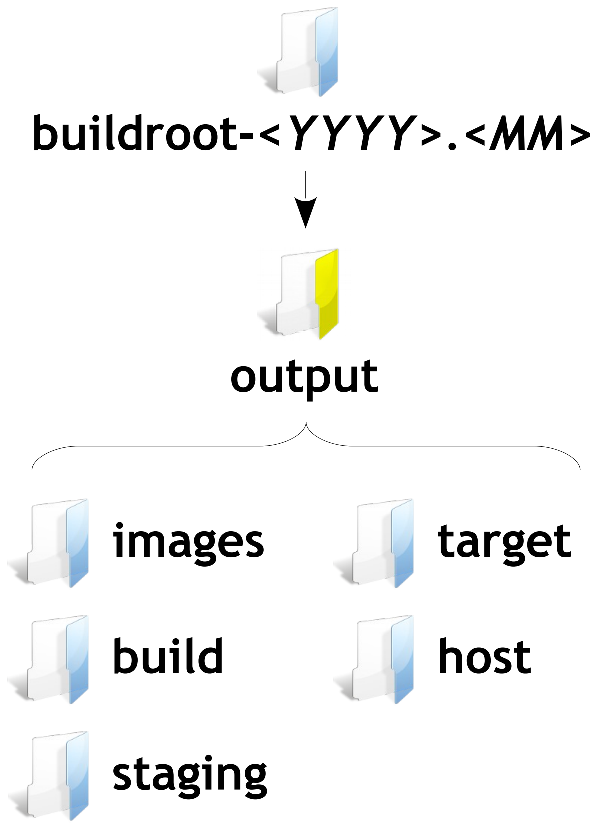
Toolchain

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

Toolchain

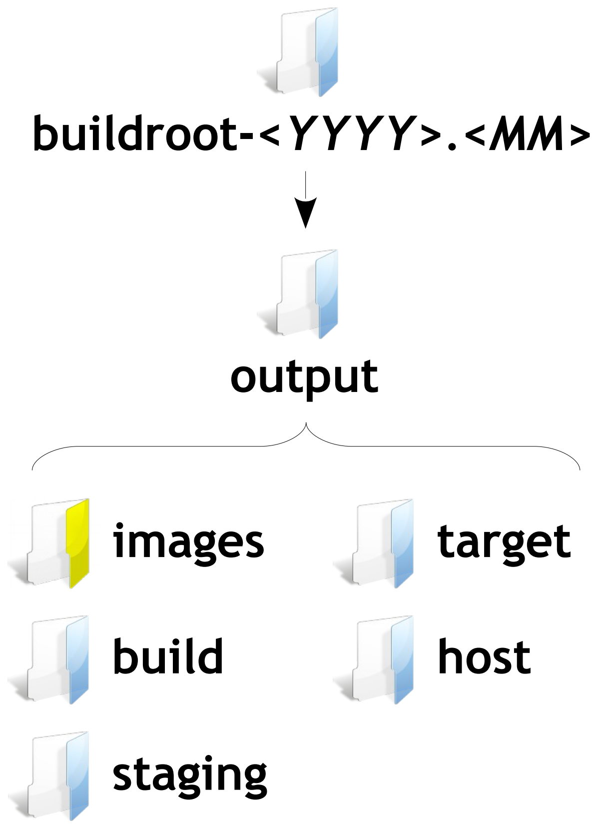
Buildroot - Building - Output



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

Toolchain

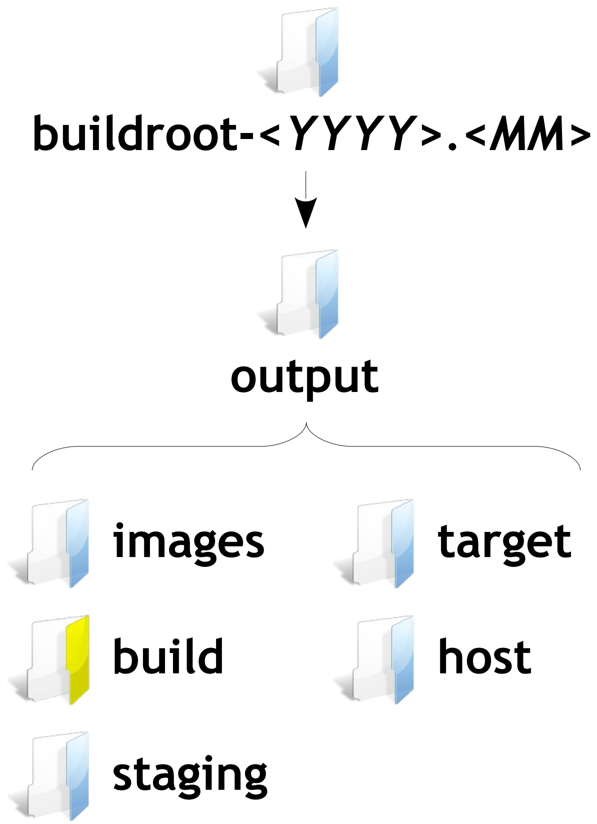
Buildroot - Building - Output



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

Toolchain

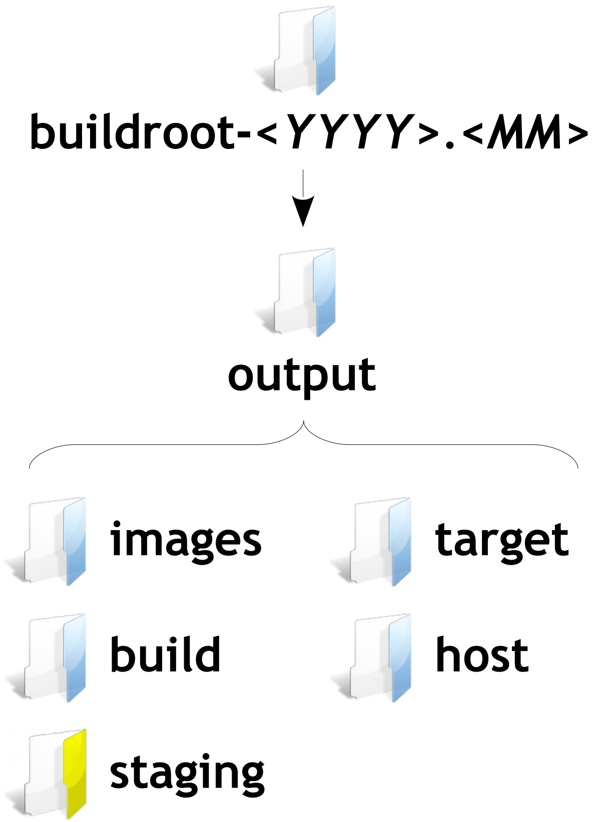
Buildroot - Building - Output



- 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

Toolchain

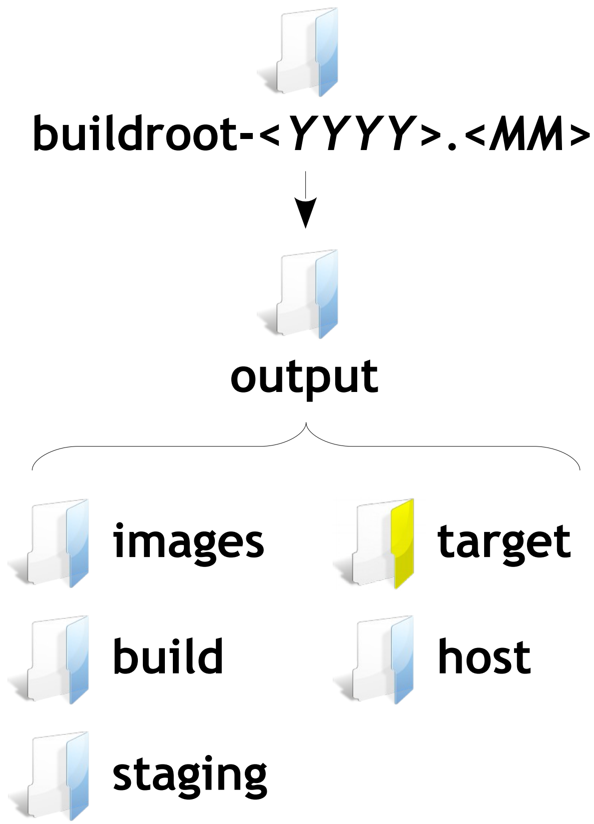
Buildroot - Building - Output



- 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

Toolchain

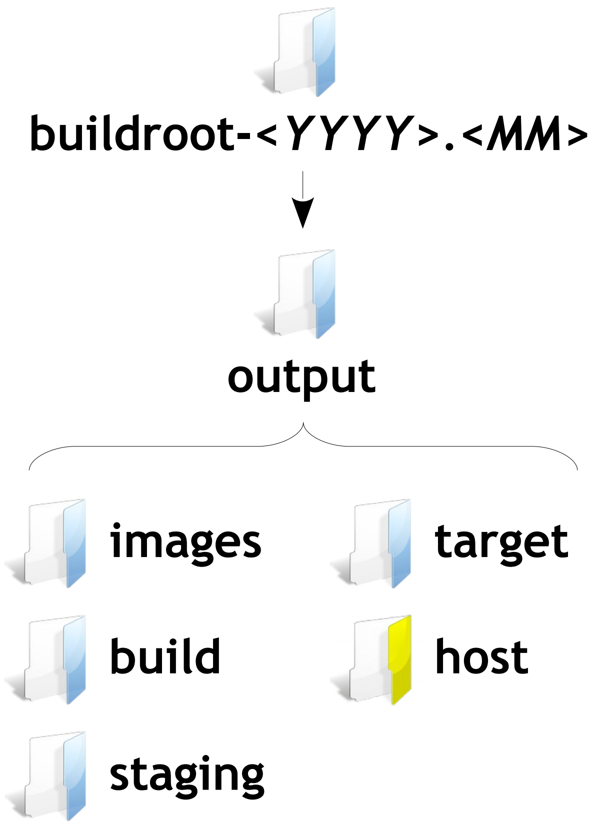
Buildroot - Building - Output



- 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

Toolchain

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

Toolchain

Buildroot - More Infos!!

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

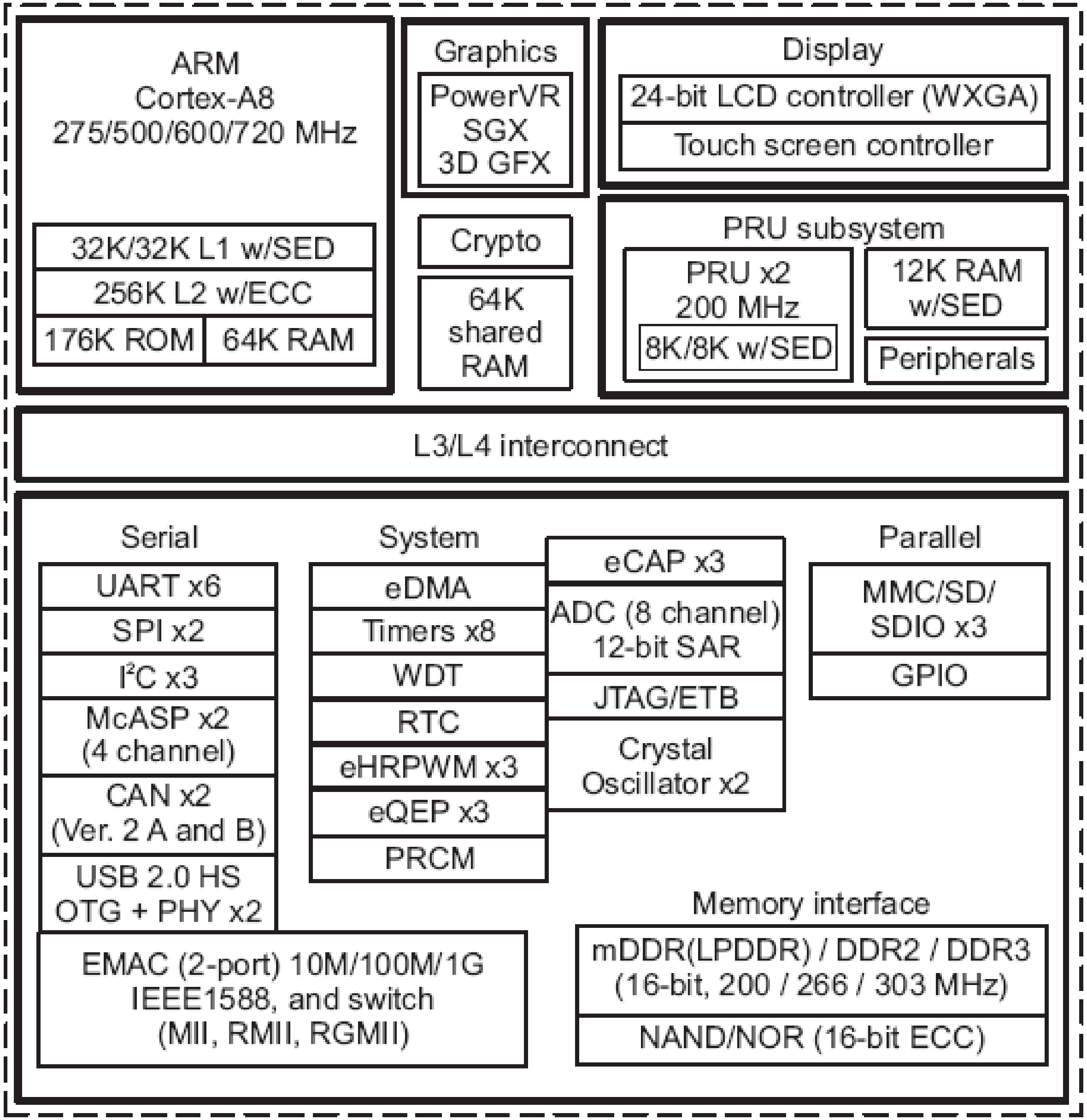
Target Overview

Target Overview
Beagle Bone Black (BBB)

- Know your Target Controller
- Target Architecture
- Target Board

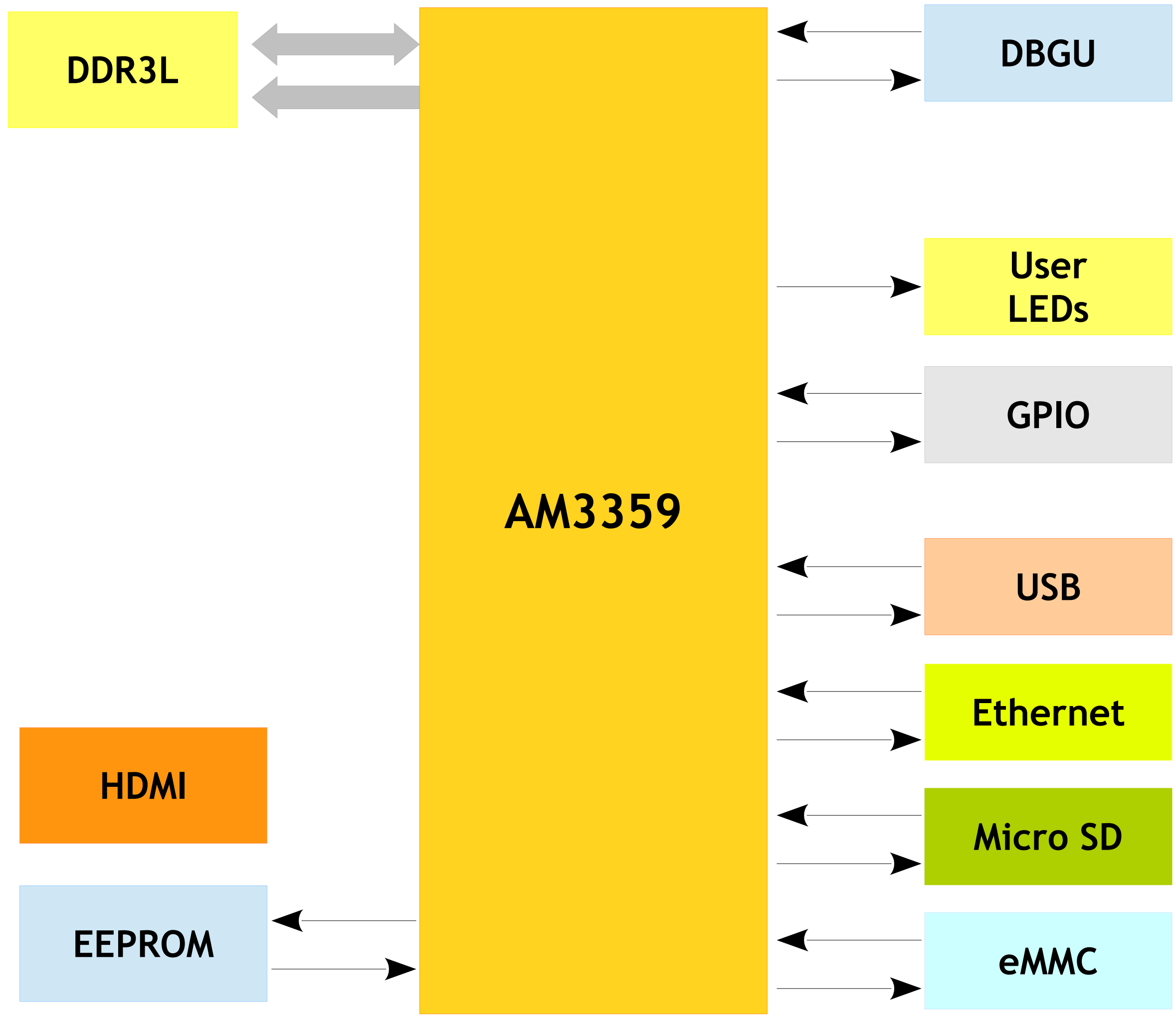
Target Overview

Beagle Bone Black - Target Controller

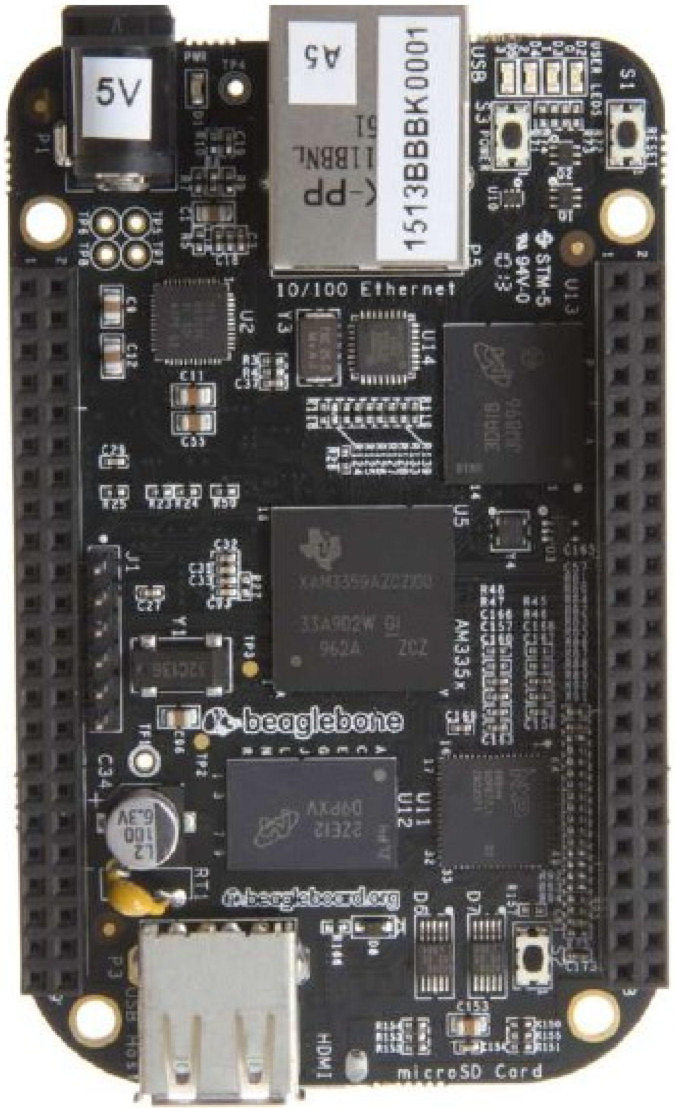


Target Overview

Beagle Bone Black - Target Architecture



Target Overview
Beagle Bone Black - Board



Target Overview

Booting Sequence - Linux as general

System Start-up	BIOS / Boot Monitor
Stage 1 Boot Loader	Master Boot Record
Stage 2 Boot Loader	LILO, GRUB, etc
Kernel	Linux
Init	User Application

Target Overview
Booting Sequence – Beagle Bone Black (BBB)

- Controller's Booting Sequence
- Boot Loader Stages

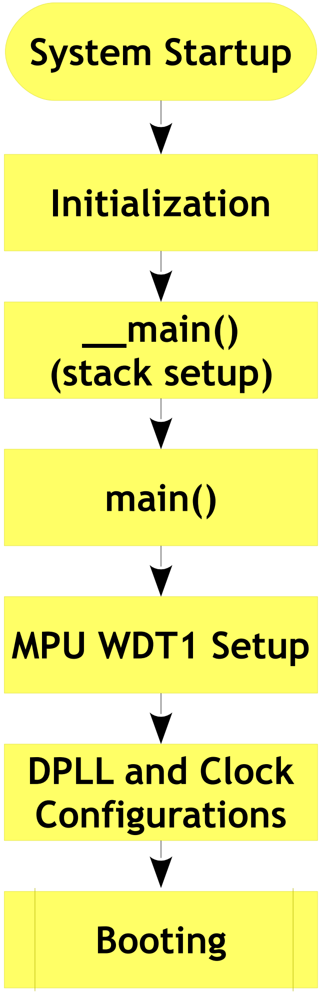
Target Overview

BBB - Booting - Controller's Boot Sequence

- The AM338X controller has many booting options
- The SYSBOOT pins configuration decide the booting sequence
- The ROM Code creates the booting device list based on the the SYSBOOT pins
- The Booting sequence is discussed in the next slide

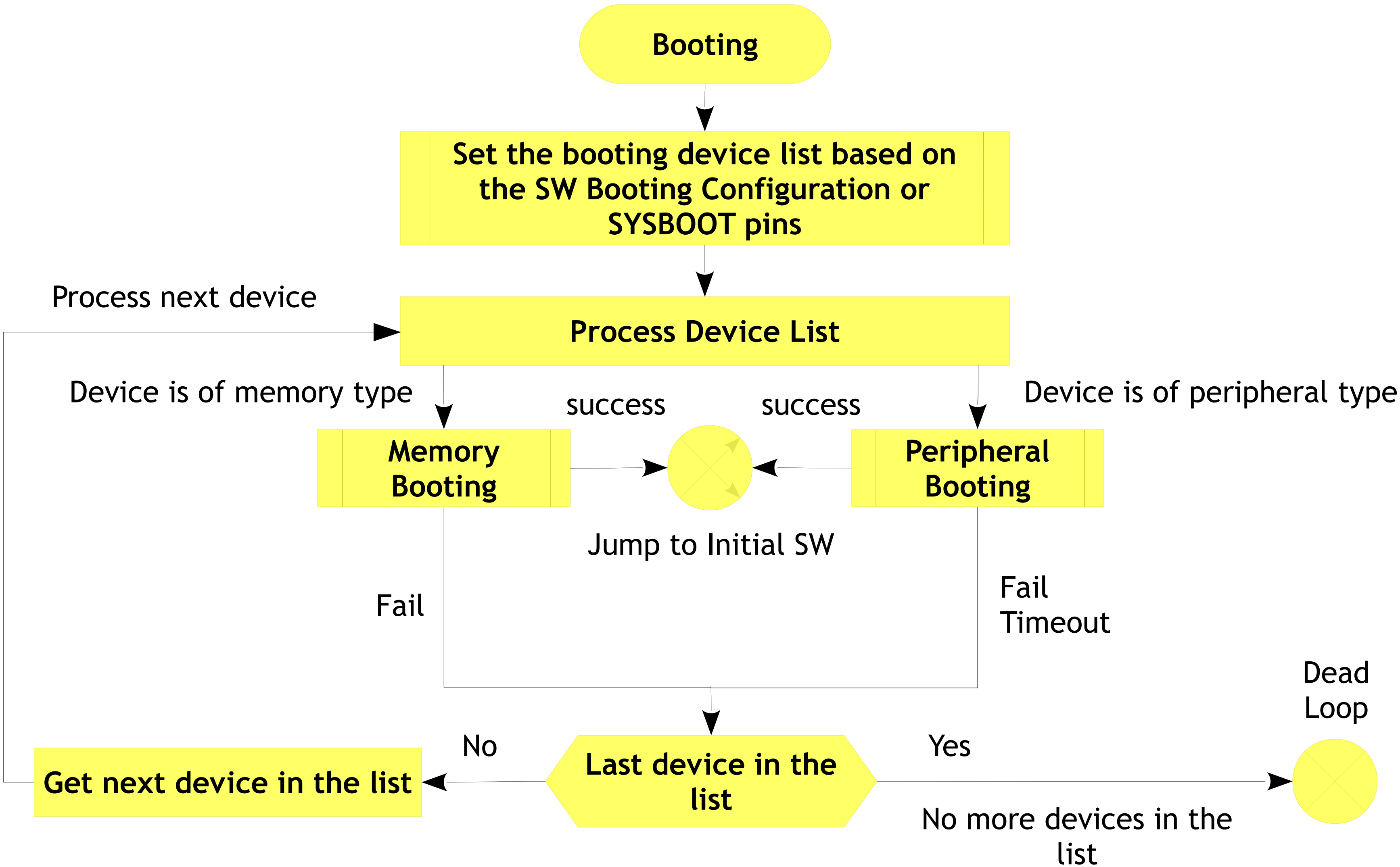
Target Overview

BBB - Booting - Controller's Boot Sequence



Target Overview

BBB - Booting - Controller's Boot Sequence



Target Overview

BBB - Booting - Boot Device List

- On SW2 release
  - MMC1 (on board eMMC)
  - MMC0 (micro SD Card)
  - UART0
  - USB0
- On SW2 pressed
  - SPI0 (SPI EEPROM)
  - MMC0 (micro SD Card)
  - UART0
  - USB0

Target Overview

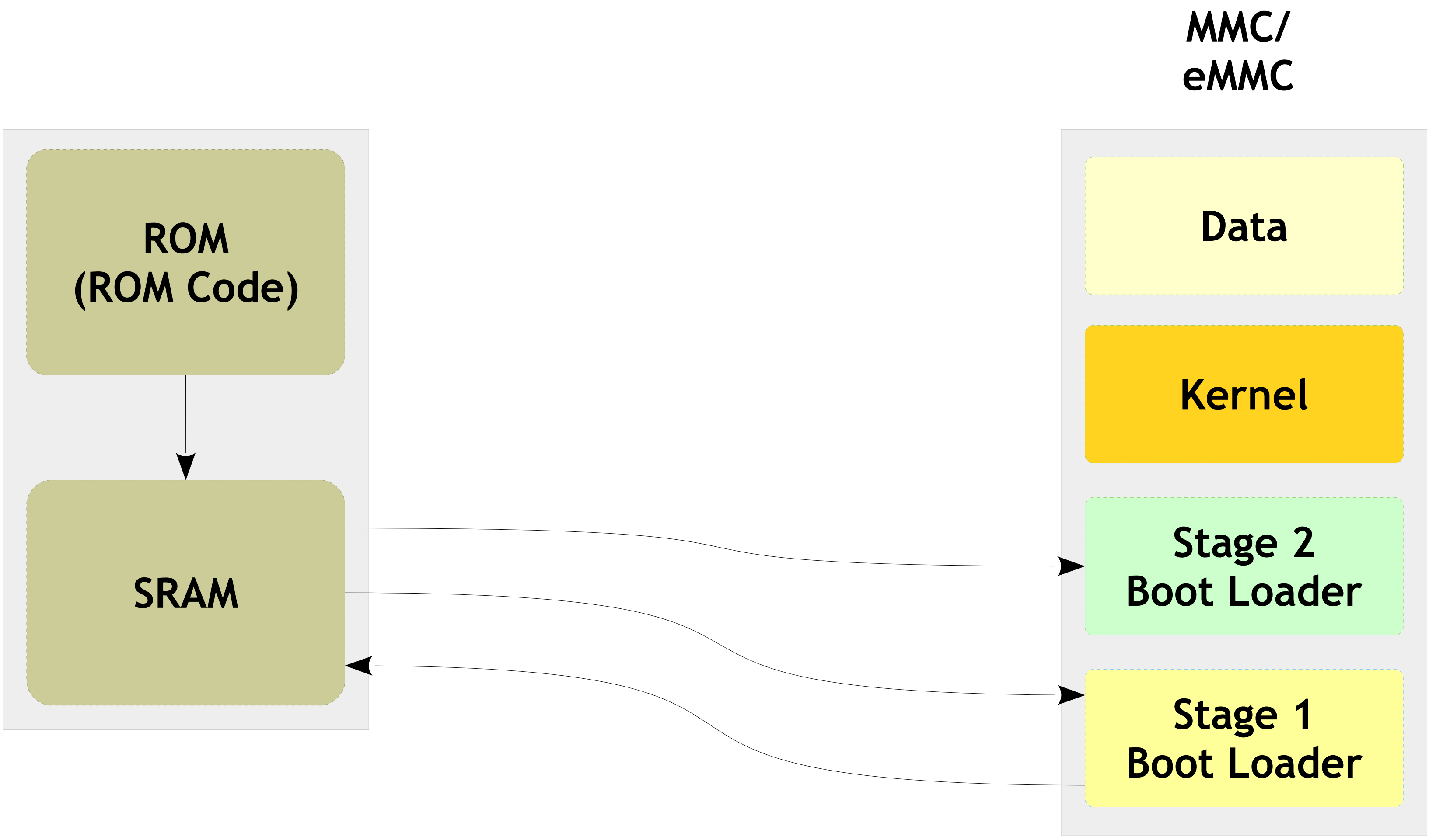
BBB - Booting - Boot Loader Stages

- Stage 1 Boot Loader
- Stage 2 Boot Loader

Target Overview

BBB - Booting Sequence – Stage 1 Boot Loader

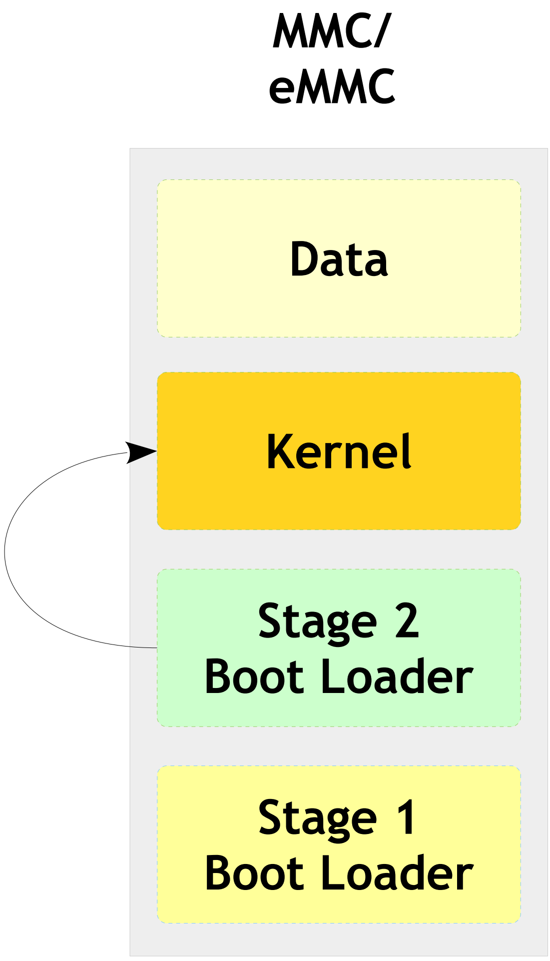
- Pointer to Stage 2 Boot Loader



Target Overview

BBB - Booting Sequence – Stage 2 Boot Loader

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



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

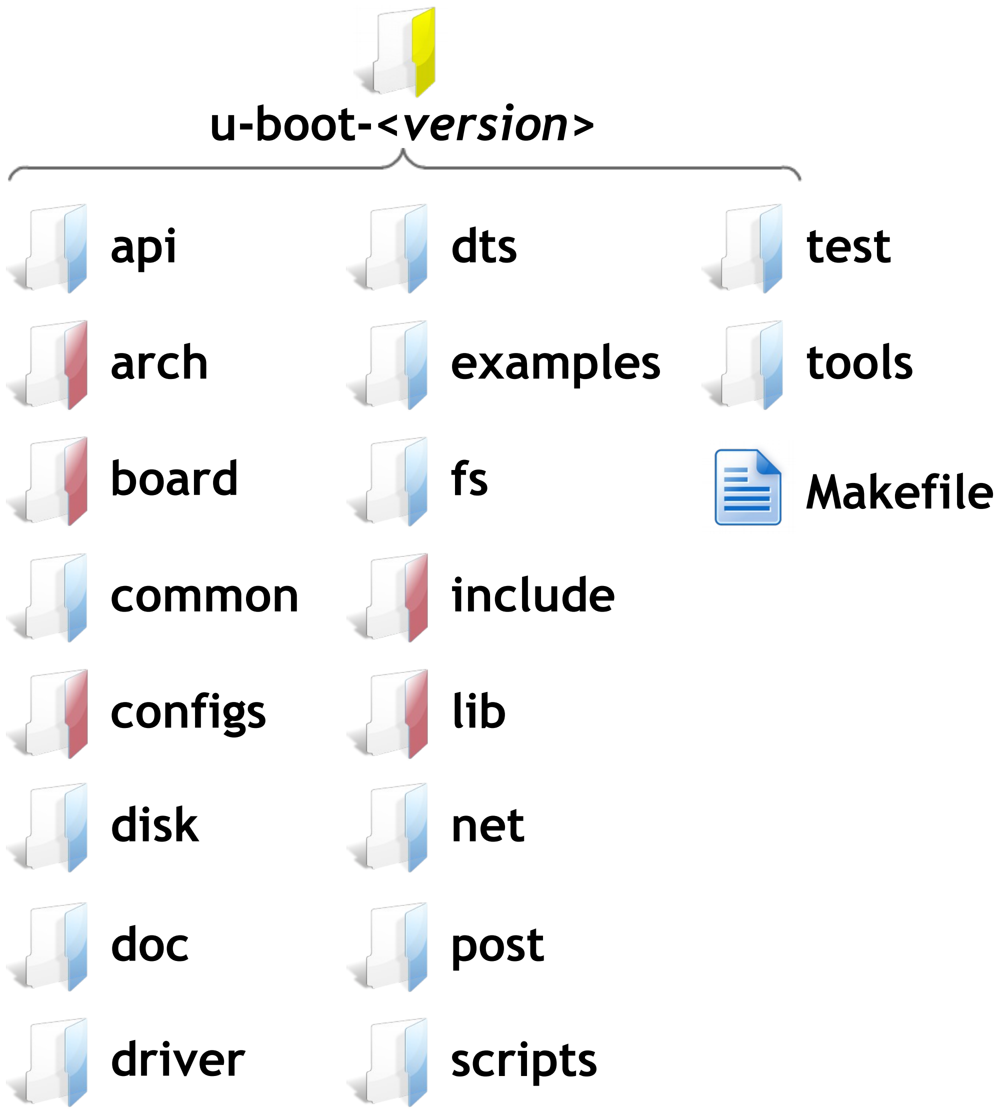
U-Boot

Source 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

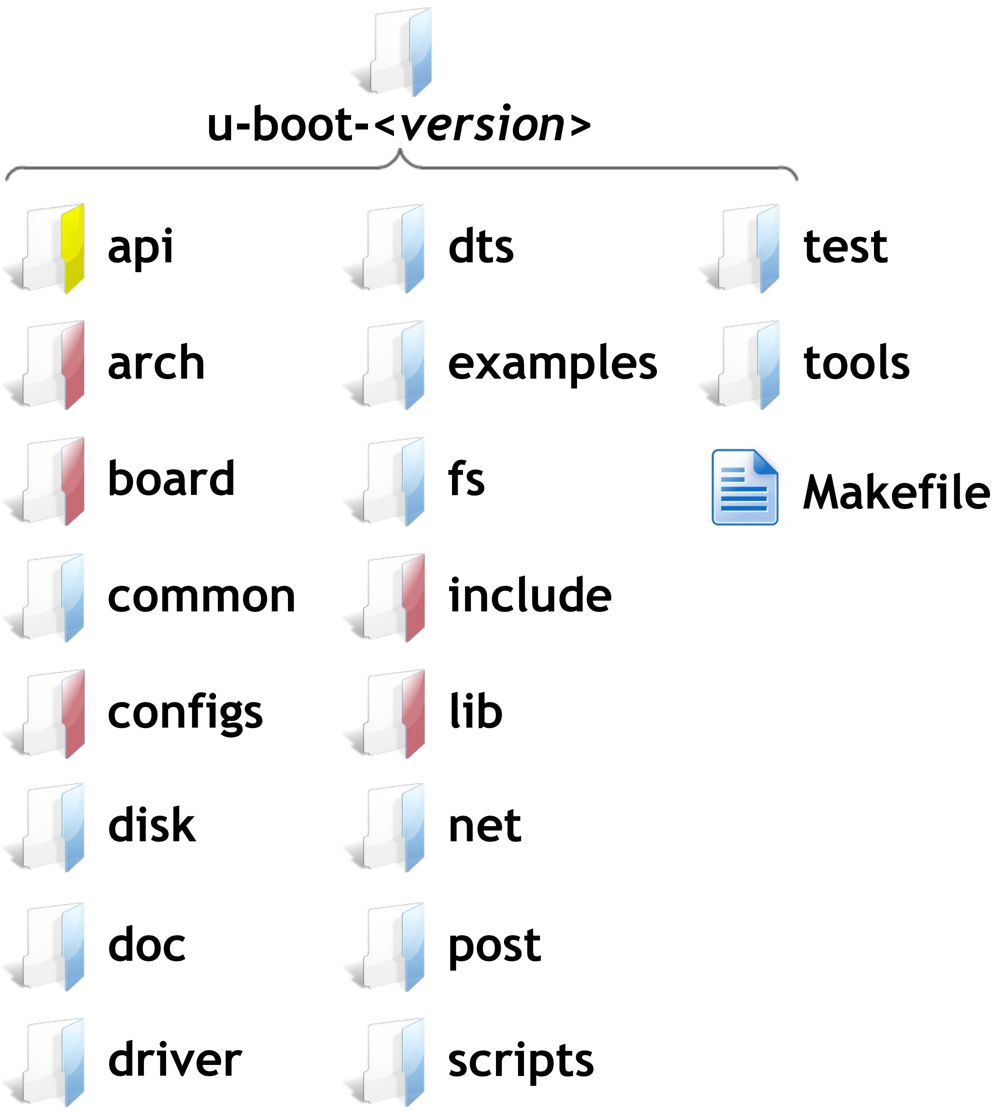
Source Tree



- 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

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

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

ΣMERTXE

U-Boot

Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- 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

u-boot-<*version*>

api	dts	test
arch	examples	tools
board	fs	Makefile
common	include	
configs	lib	
disk	net	
doc	post	
driver	scripts	

- All architecture independent functions
- All the commands

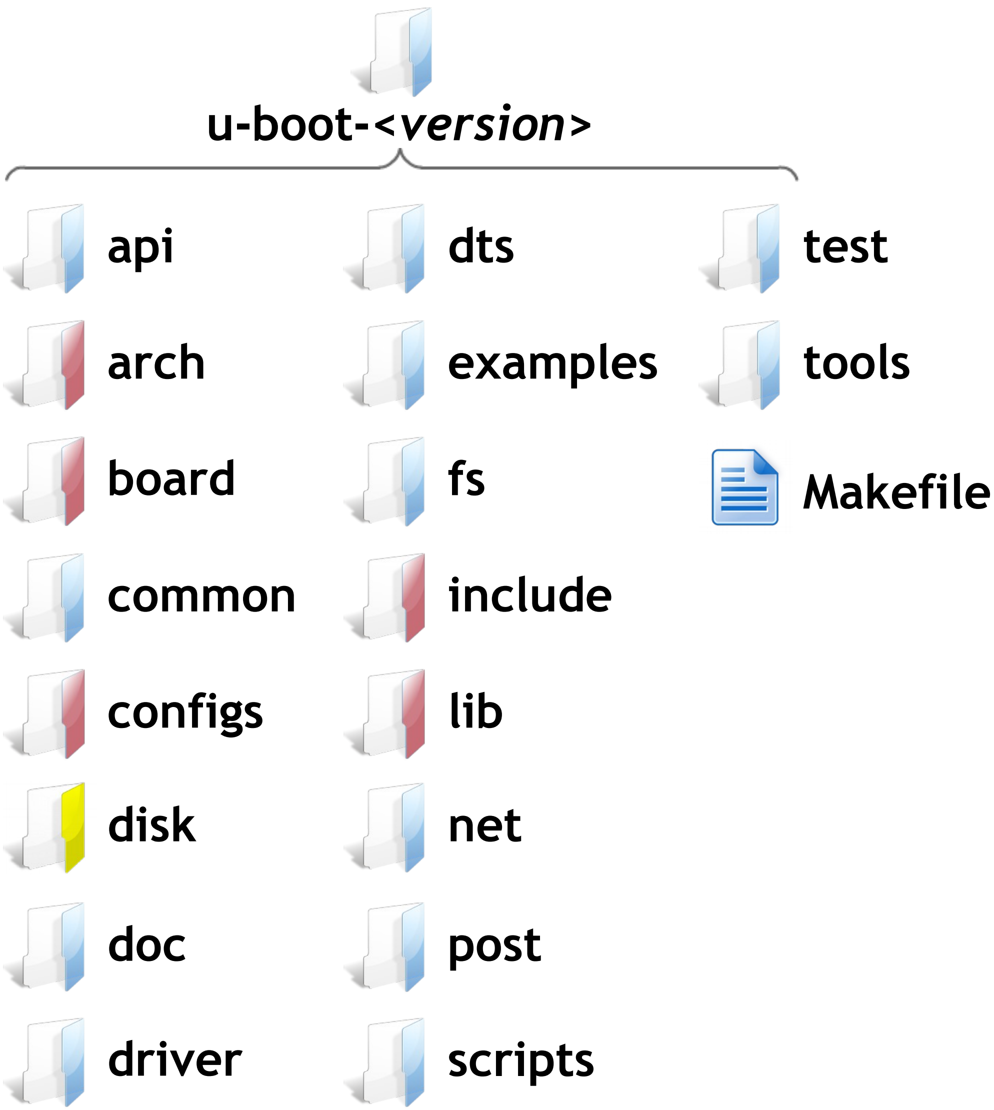
U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Default configuration files for boards

U-Boot
Source Tree



- Partition and device information for disks

U-Boot

Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- You can find all the README files here

U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Various device drivers files

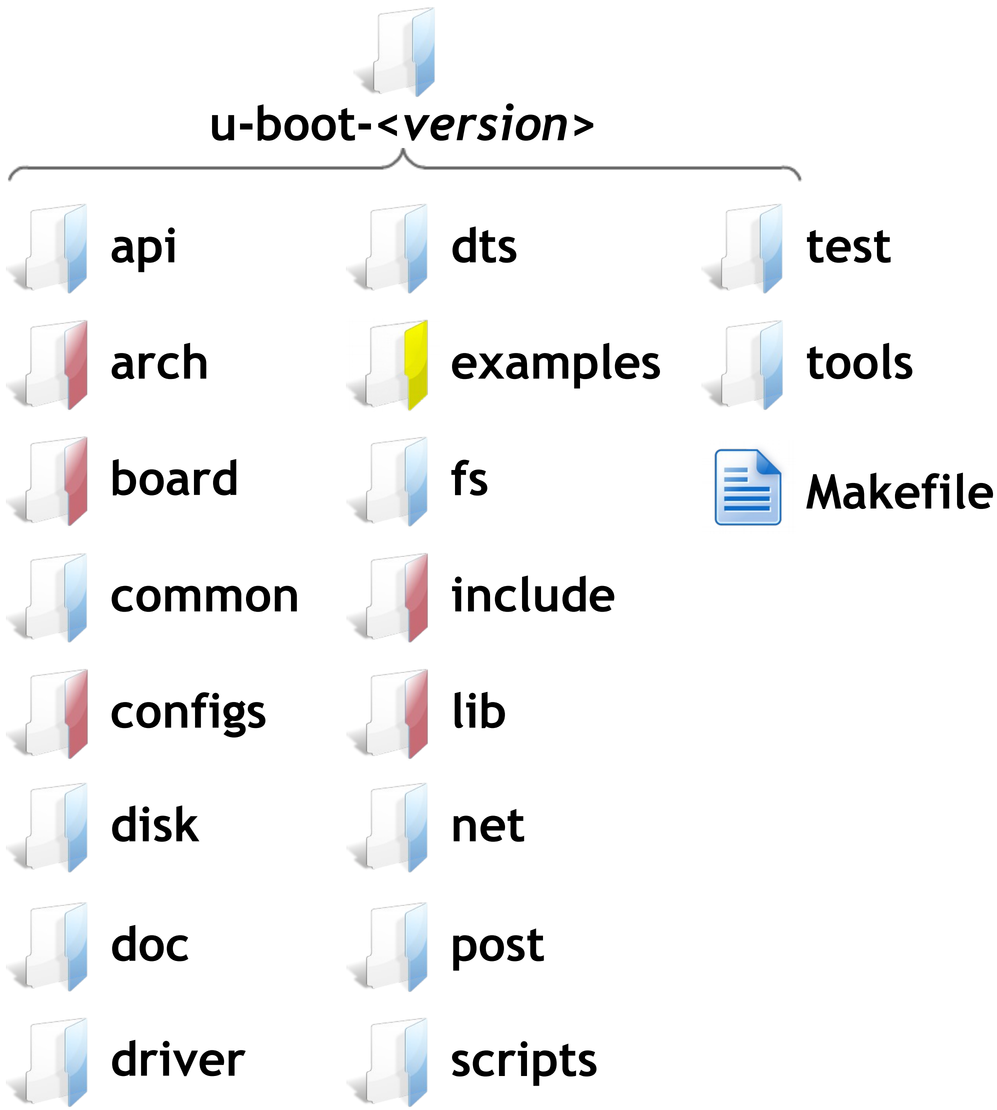
U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

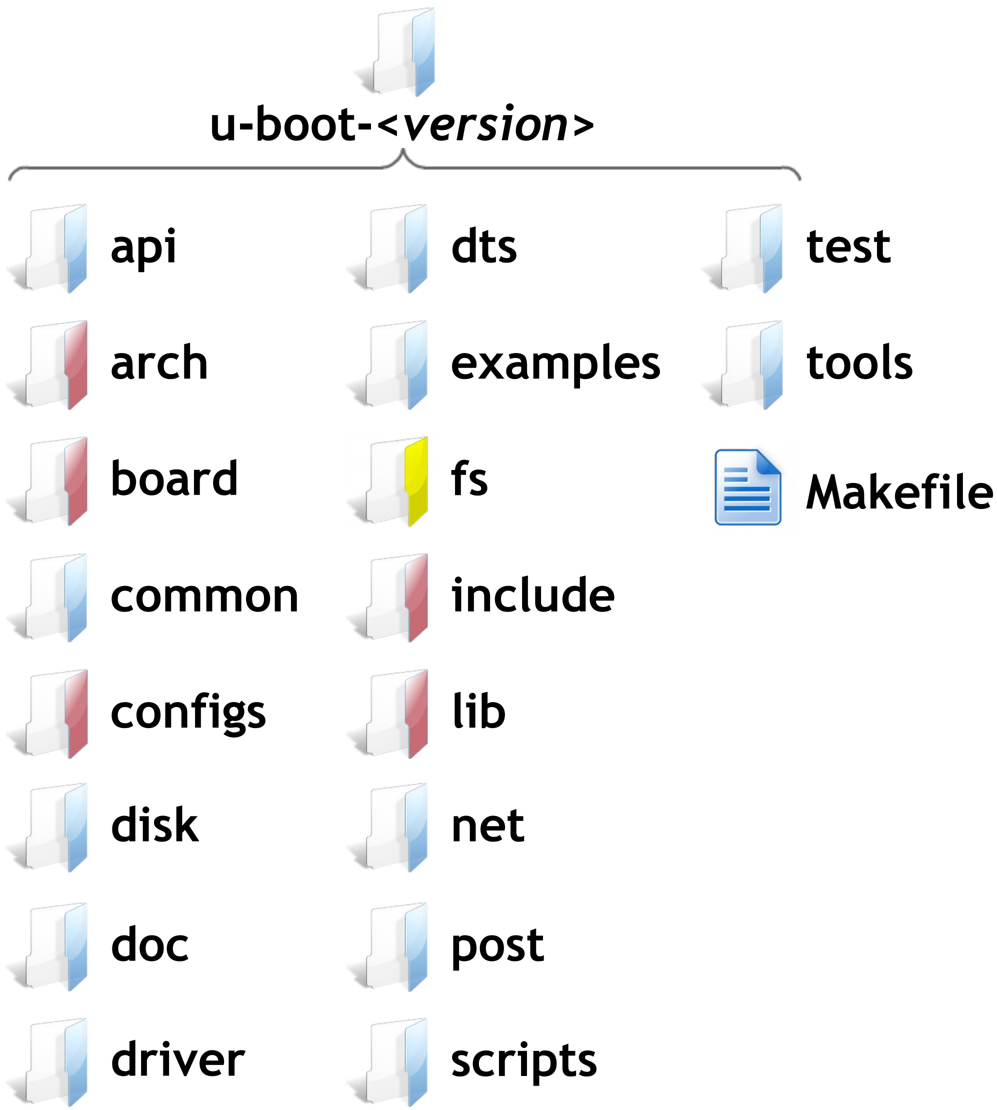
- Contains Makefile for building internal U-Boot fdt

U-Boot
Source Tree



- Example code for standalone application

U-Boot
Source Tree



- File system directories and codes

U-Boot
Source Tree

- u-boot-<*version*>
  - api
  - arch
  - board
  - common
  - configs
  - disk
  - doc
  - driver
  - dts
  - examples
  - fs
  - include
  - lib
  - net
  - post
  - scripts
  - test
  - tools
  - Makefile

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

U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

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

U-Boot

Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Networking related files.

U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Power On Self Test

U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Contains the sources for various helper programs

U-Boot
Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Some unit test codes

U-Boot

Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Various tools directories and files

U-Boot

Source Tree

u-boot-<*version*>

- api
- arch
- board
- common
- configs
- disk
- doc
- driver
- dts
- examples
- fs
- include
- lib
- net
- post
- scripts
- test
- tools
- Makefile

- Top level make file for Uboot build and configuration

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
Responsibility

- Execute from flash (If configured). Do POST
- Relocate to RAM
- Setup console for user interaction
- Setup device driver for kernel (& RFS) image
- Choose the kernel (& RFS) image
- Download the kernel (& RFS) image
- Setup kernel command line arguments
- Jump to kernel start address

Code flow

U-Boot Important Commands

U-Boot
Utilities

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

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
- **`mmc`** : Multim Media Card access

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>`

Embedded Linux Kernel

General Information

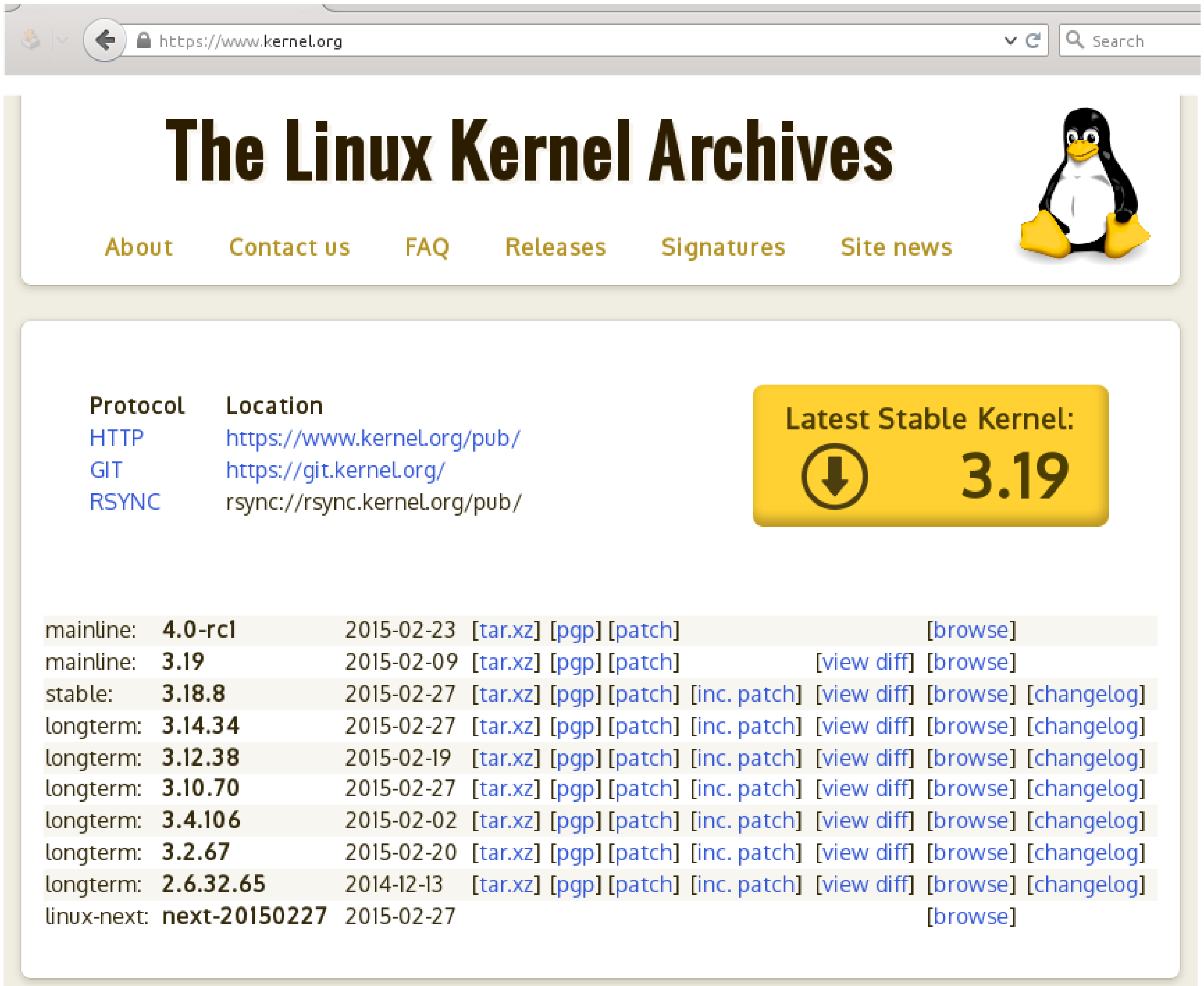
Embedded Linux Kernel

General Information

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

Embedded Linux Kernel

General Information – Where to get?



https://www.kernel.org

Search

The Linux Kernel Archives

About Contact us FAQ Releases Signatures Site news

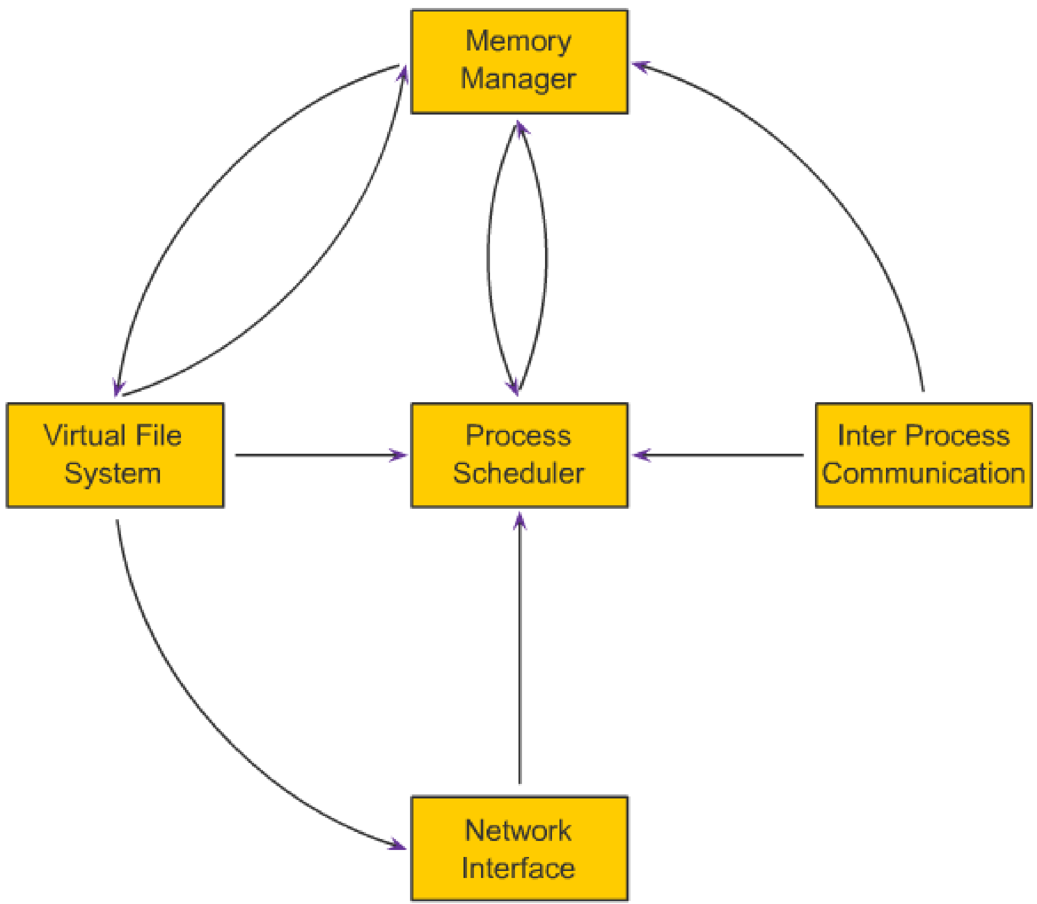
Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Stable Kernel: 3.19

mainline:	4.0-rc1	2015-02-23	[tar.xz]	[pgp]	[patch]			[browse]	
mainline:	3.19	2015-02-09	[tar.xz]	[pgp]	[patch]		[view diff]	[browse]	
stable:	3.18.8	2015-02-27	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.14.34	2015-02-27	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.12.38	2015-02-19	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.10.70	2015-02-27	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.4.106	2015-02-02	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.2.67	2015-02-20	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	2.6.32.65	2014-12-13	[tar.xz]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
linux-next:	next-20150227	2015-02-27						[browse]	

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

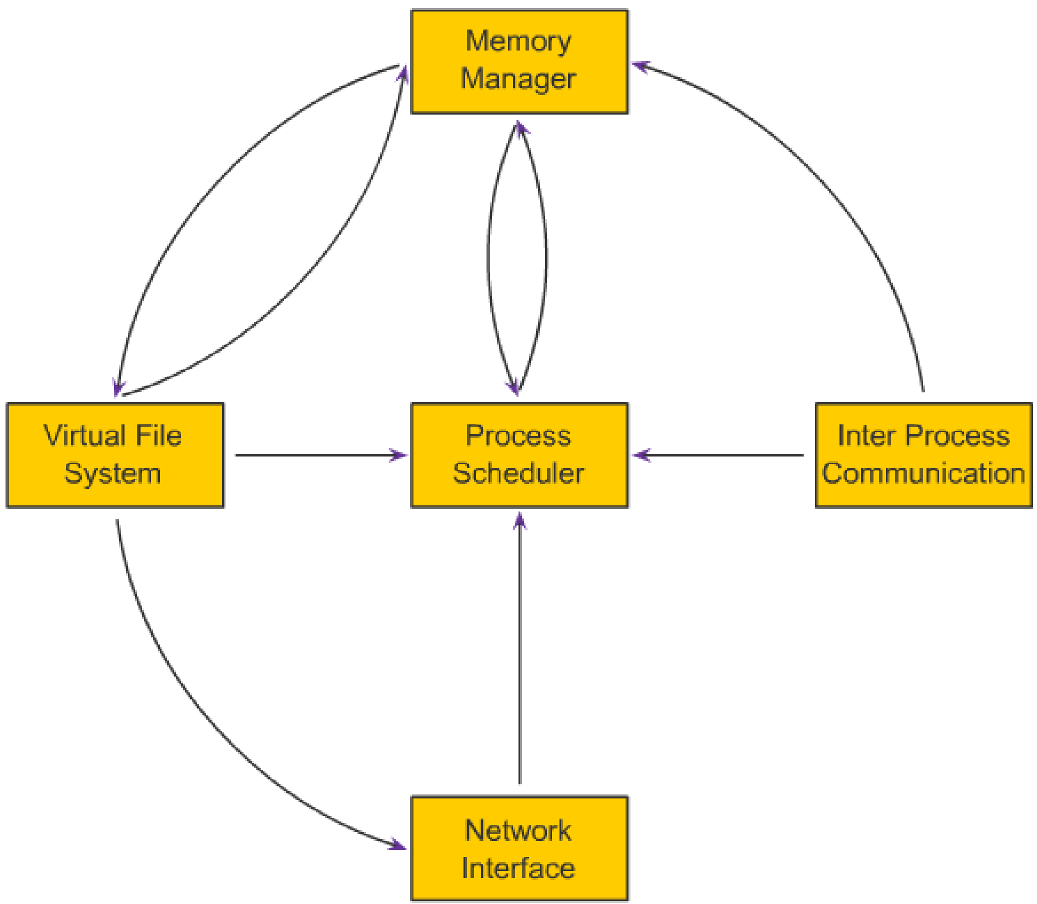
Embedded Linux Kernel
General Information – Kernel Subsystem



- 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

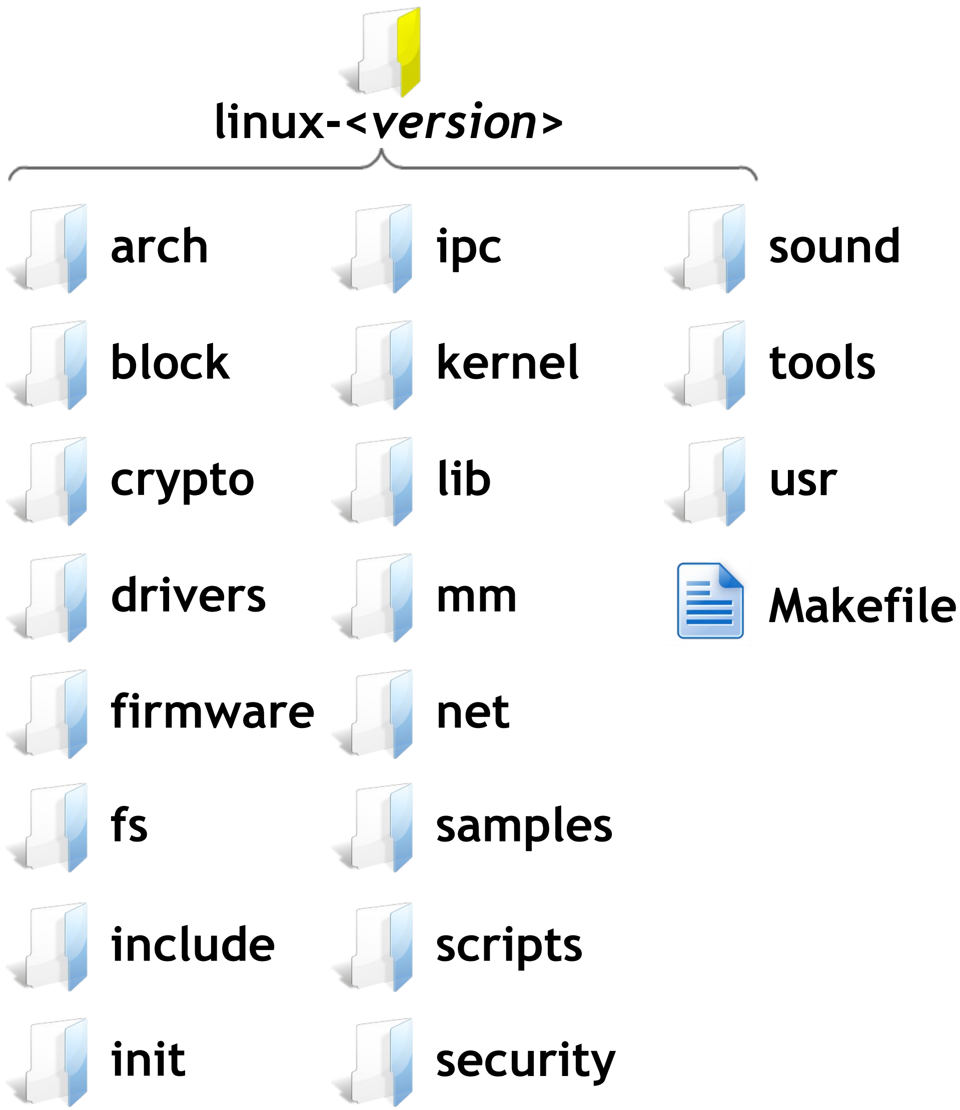
Embedded Linux Kernel

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

linux-<*version*>

- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- tools
- usr
- Makefile

- 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

linux-<*version*>

arch
block
crypto
drivers
firmware
fs
include
init
ipc
kernel
lib
mm
net
samples
scripts
security
sound
tools
usr
Makefile

- Contains core block layer files

EMERTXE

Embedded Linux Kernel

General Information – Source Code Browsing

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

- Cryptographic API for use by kernel itself

Embedded Linux Kernel

General Information – Source Code Browsing

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

- 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

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

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

Embedded Linux Kernel

General Information – Source Code Browsing

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

- 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

linux-<*version*>

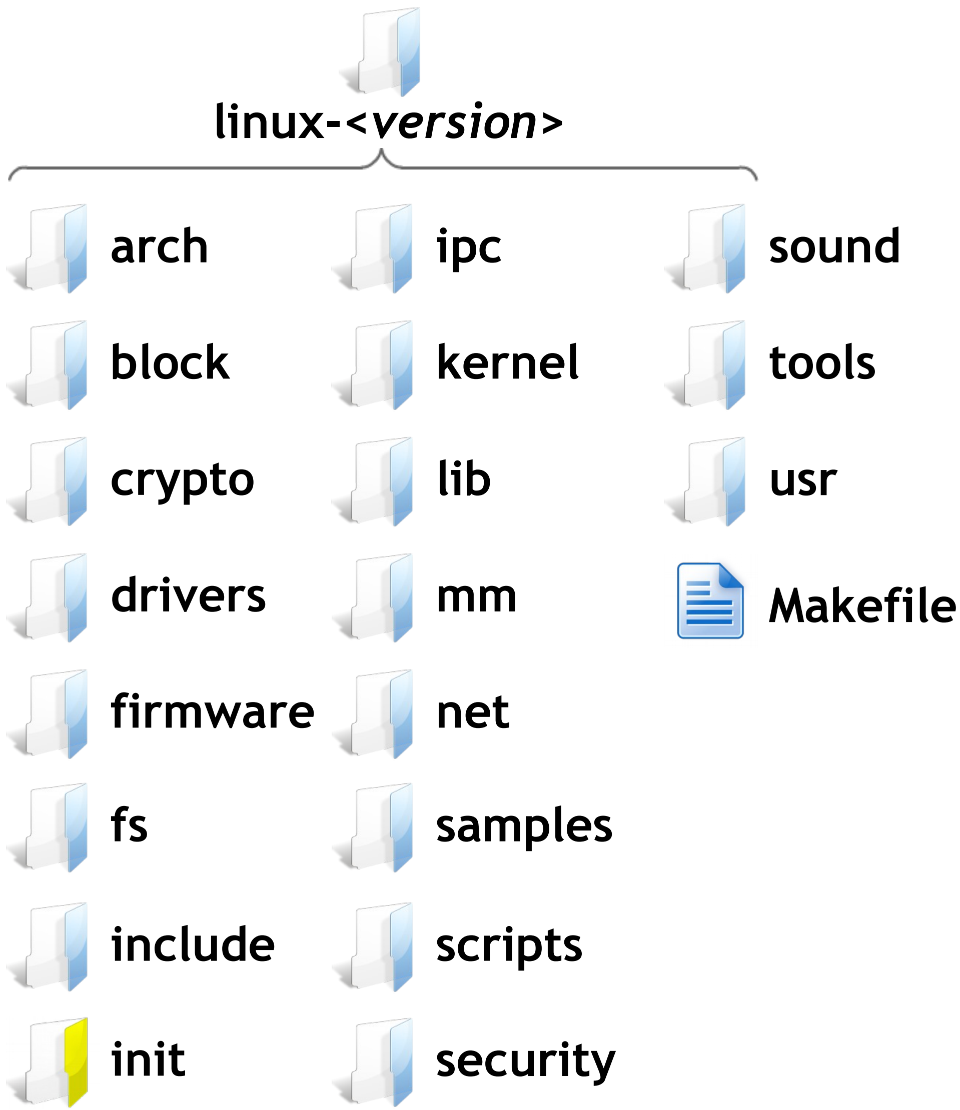
- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- tools
- usr
- Makefile

- 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

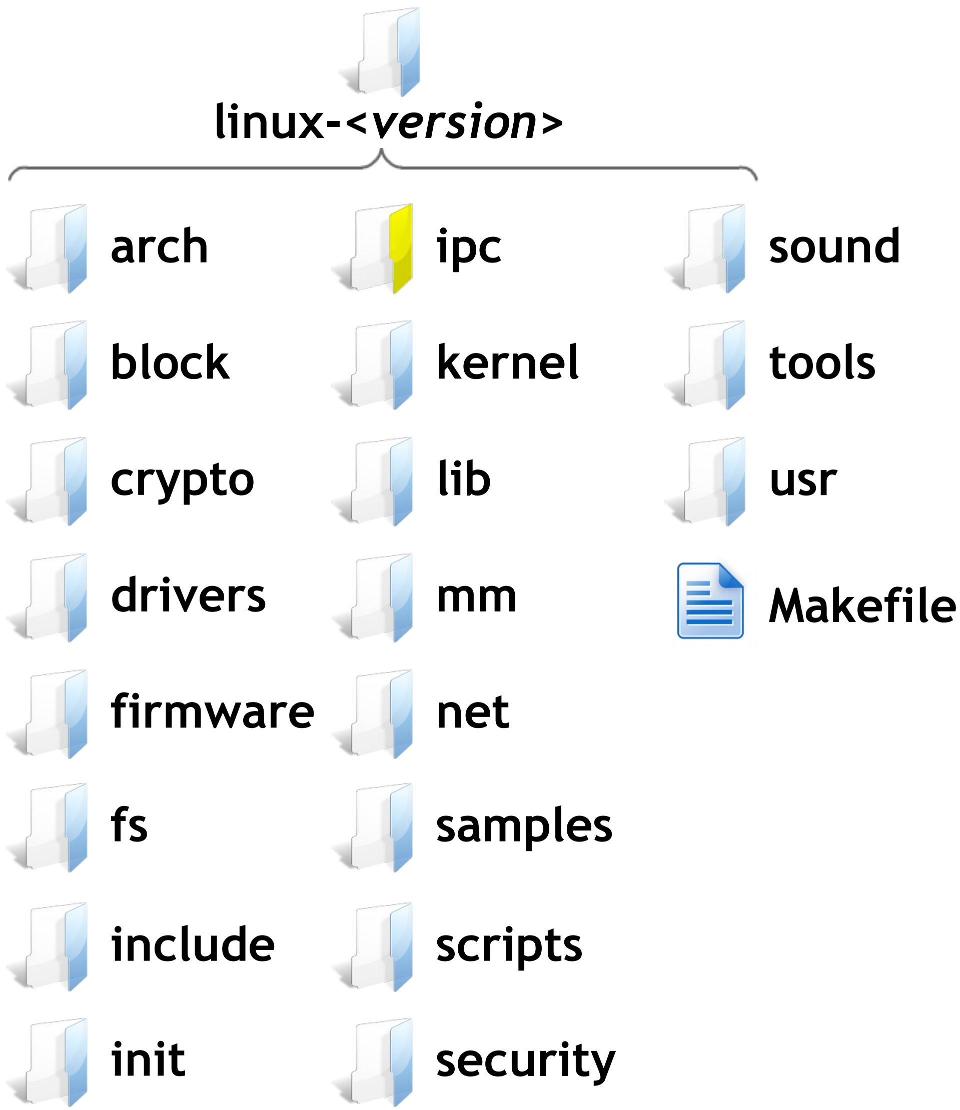
General Information – Source Code Browsing



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

Embedded Linux Kernel

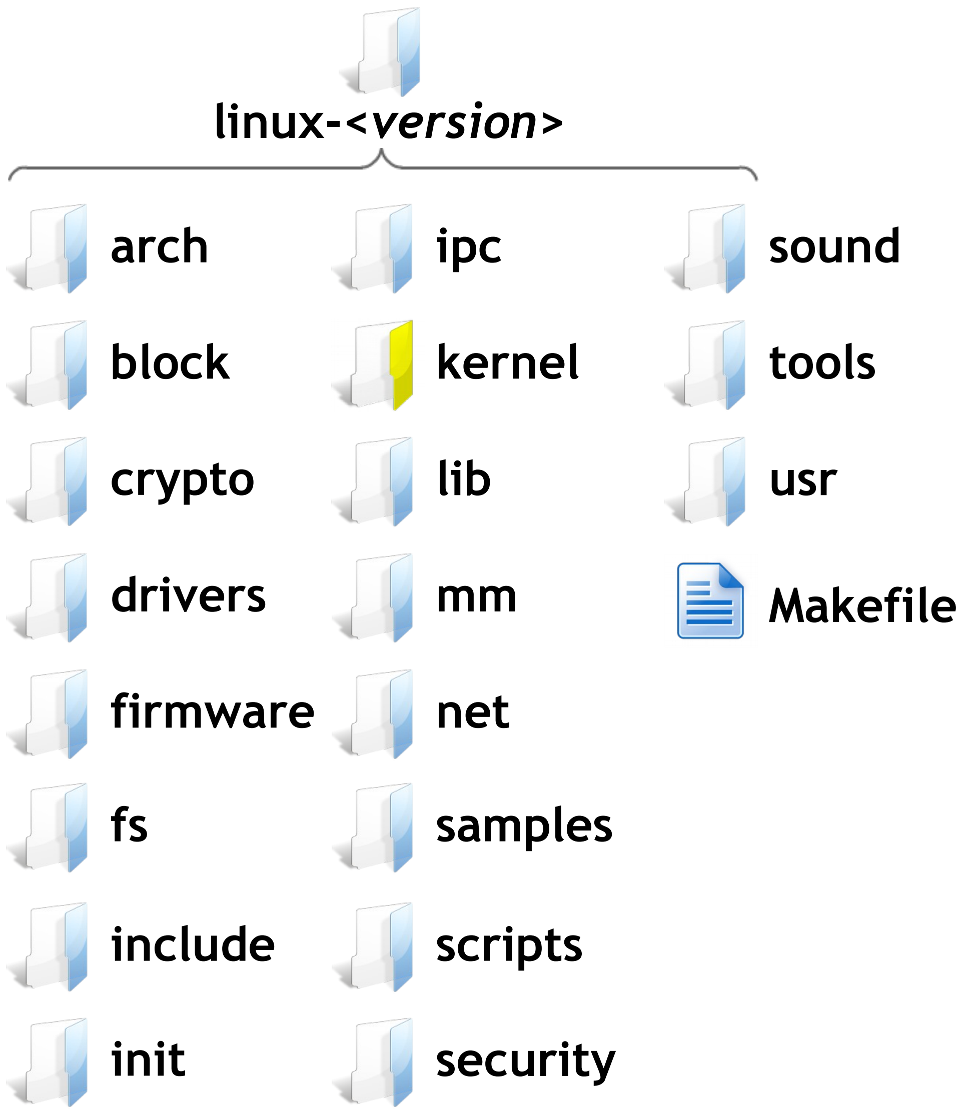
General Information – Source Code Browsing



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

Embedded Linux Kernel

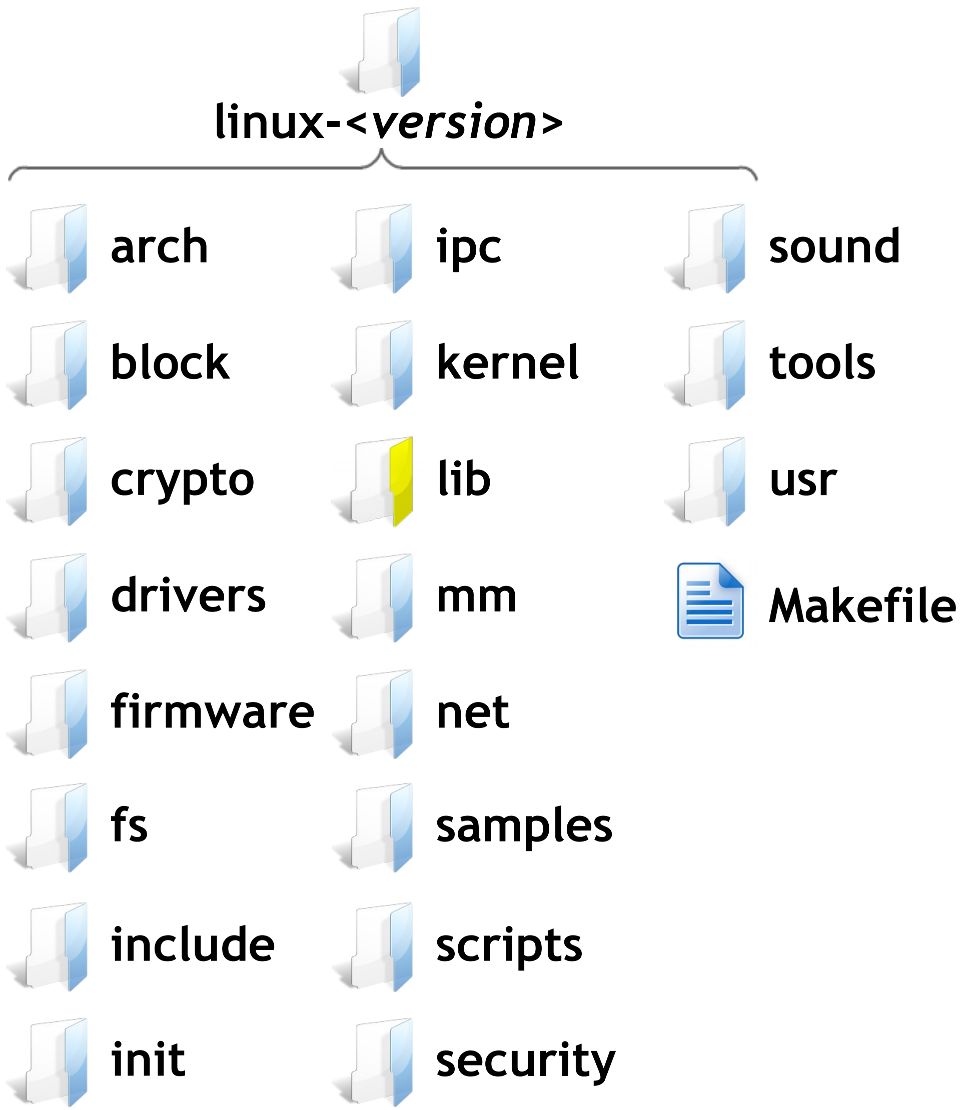
General Information – Source Code Browsing



- 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

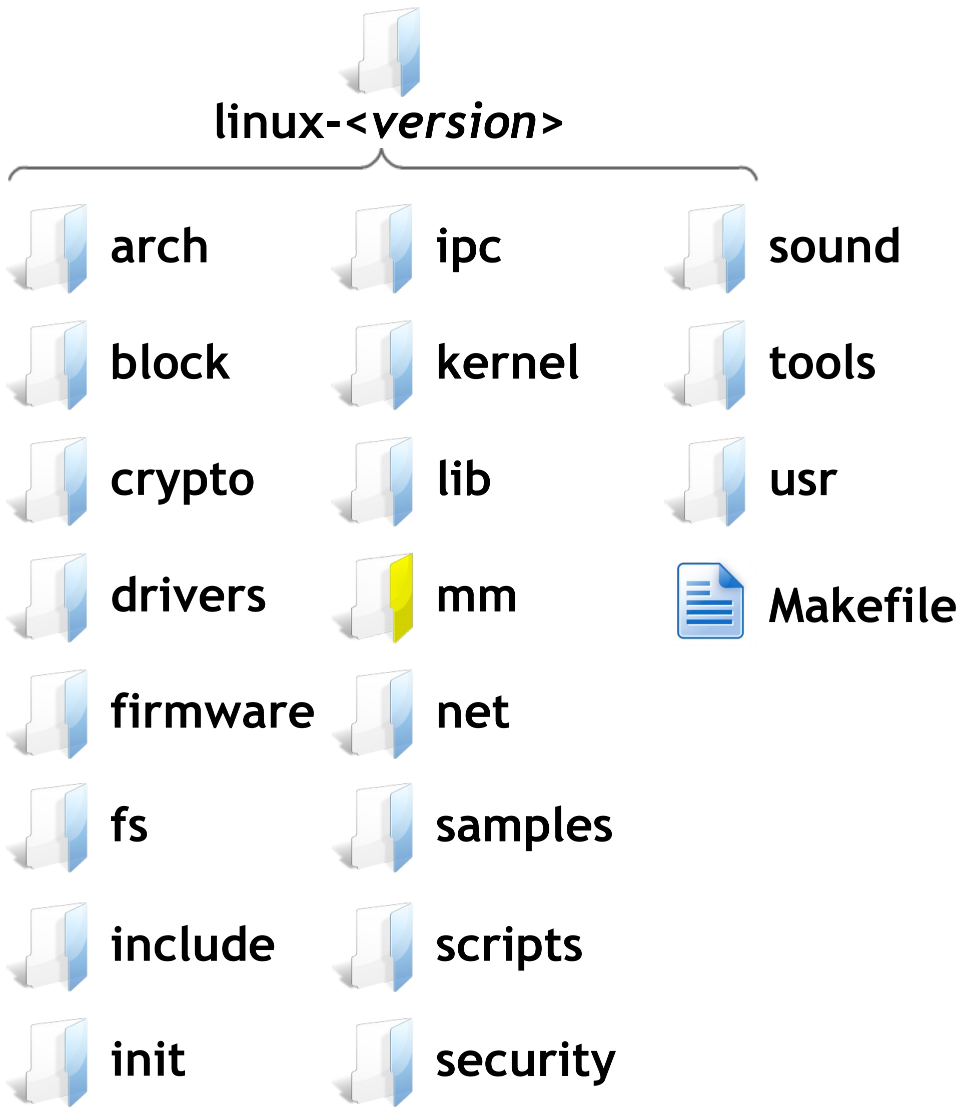
General Information – Source Code Browsing



- 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

EMERTXE

linux-<*version*>

- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- tools
- usr
- Makefile

- The kernels networking code

Embedded Linux Kernel

General Information – Source Code Browsing

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

- Some sample programs

Embedded Linux Kernel

General Information – Source Code Browsing

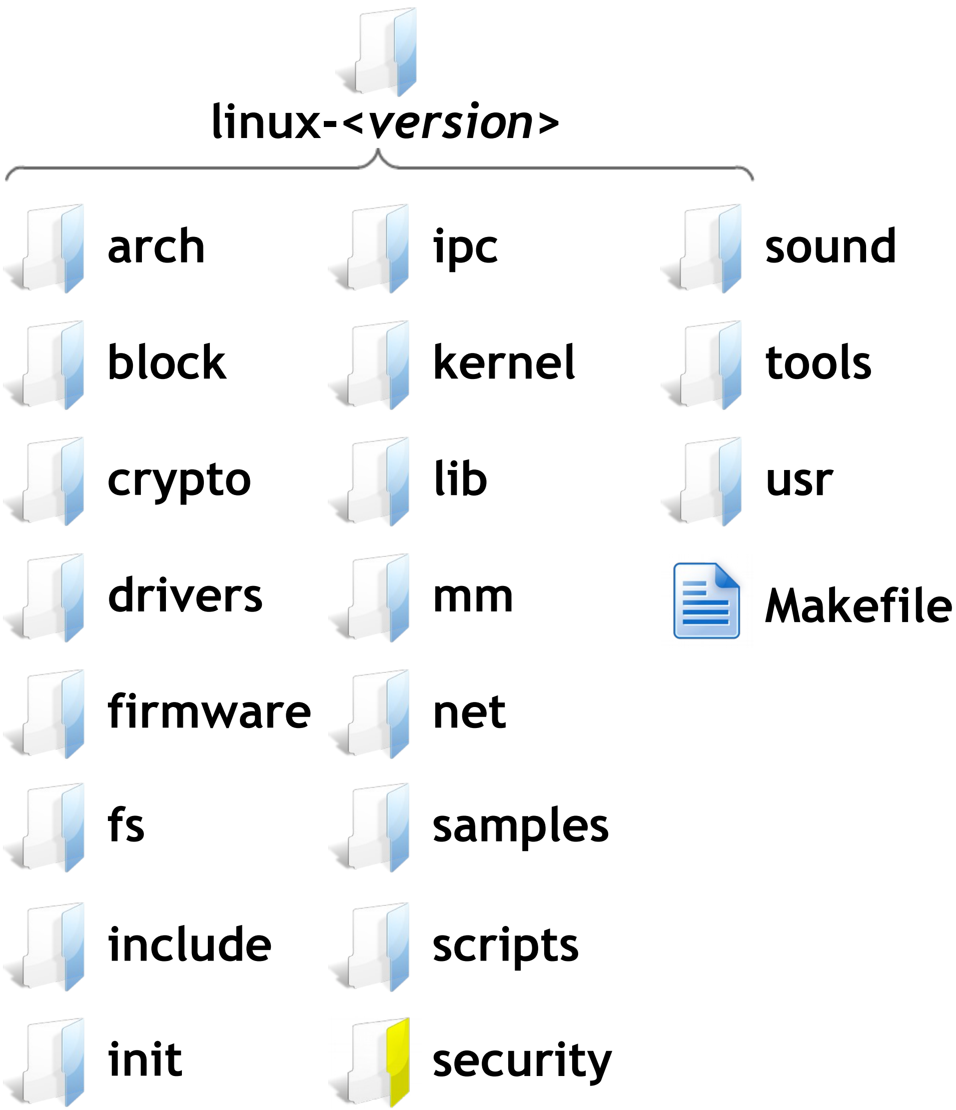
linux-<*version*>

- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- tools
- usr
- Makefile

- 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

linux-<*version*>

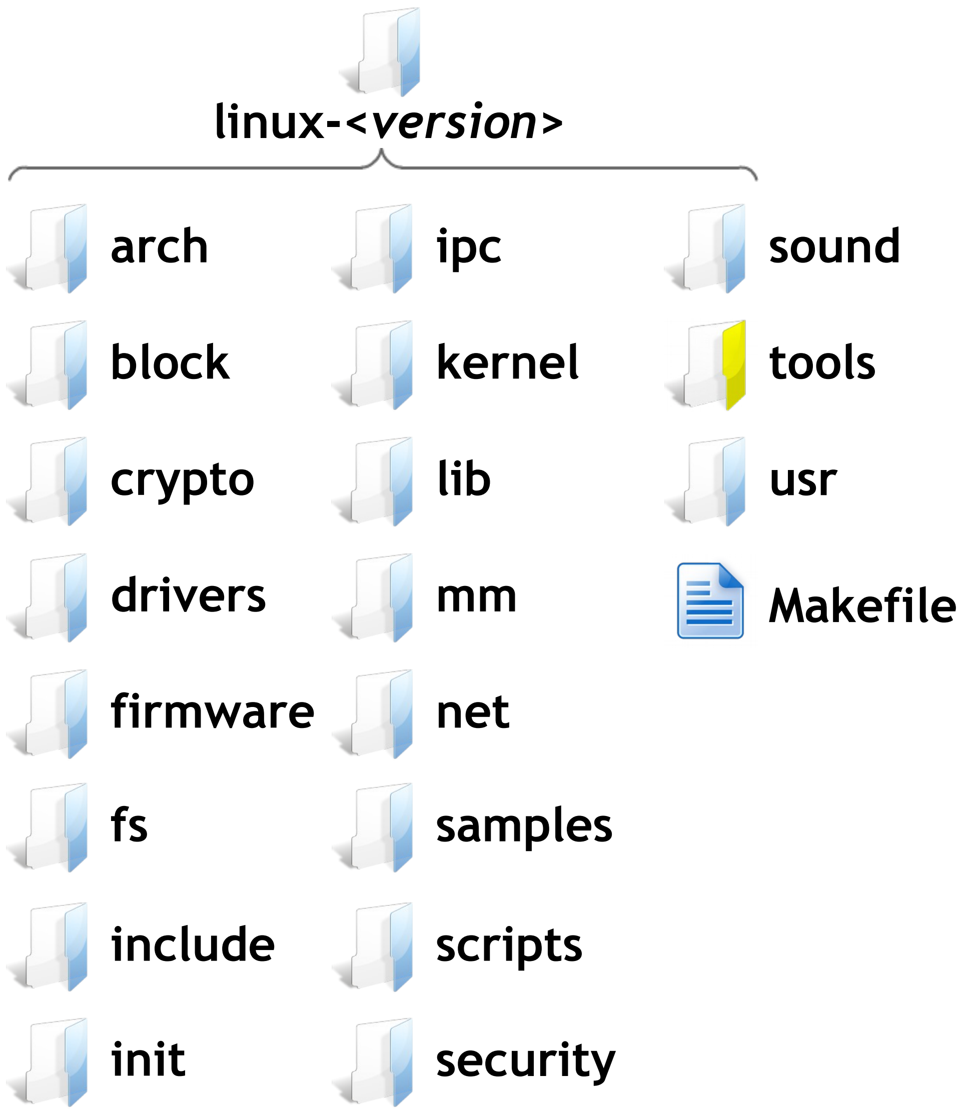
- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- tools
- usr
- Makefile

- Contains all the sound card drivers

EMERTXE

Embedded Linux Kernel

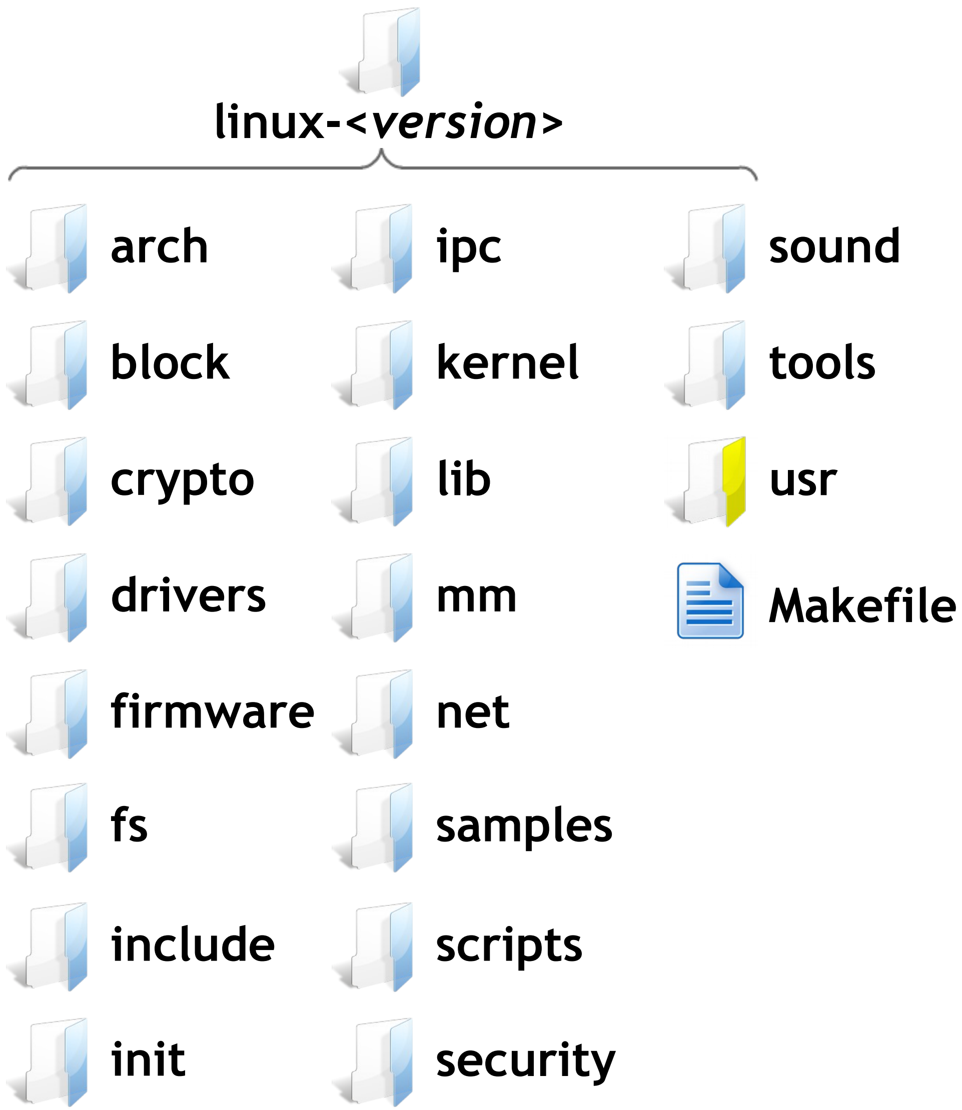
General Information – Source Code Browsing



- 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

linux-<*version*>

arch	ipc	sound
block	kernel	tools
crypto	lib	usr
drivers	mm	Makefile
firmware	net	
fs	samples	
include	scripts	
init	security	

- 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

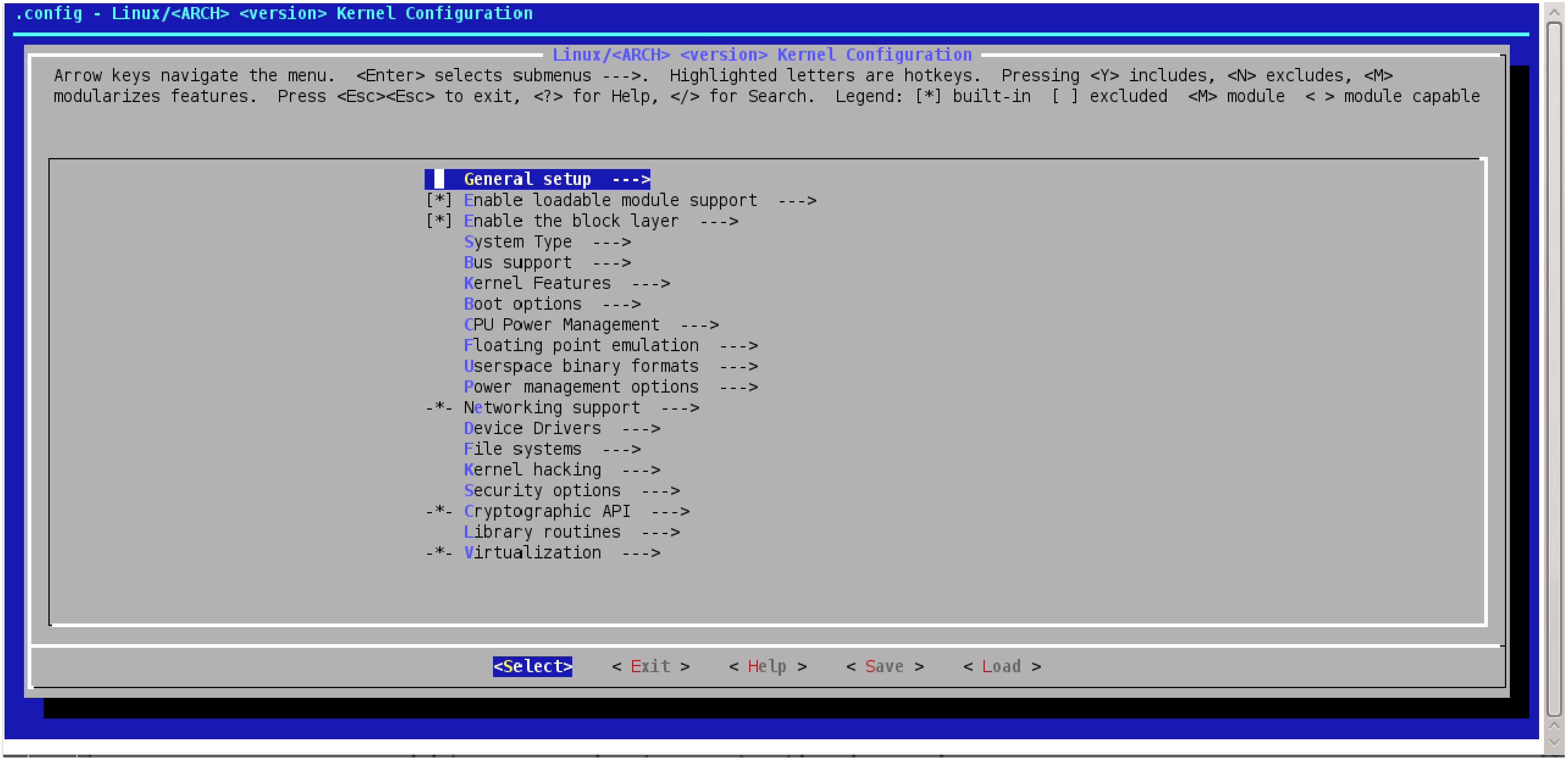
Configuration – make config

```
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

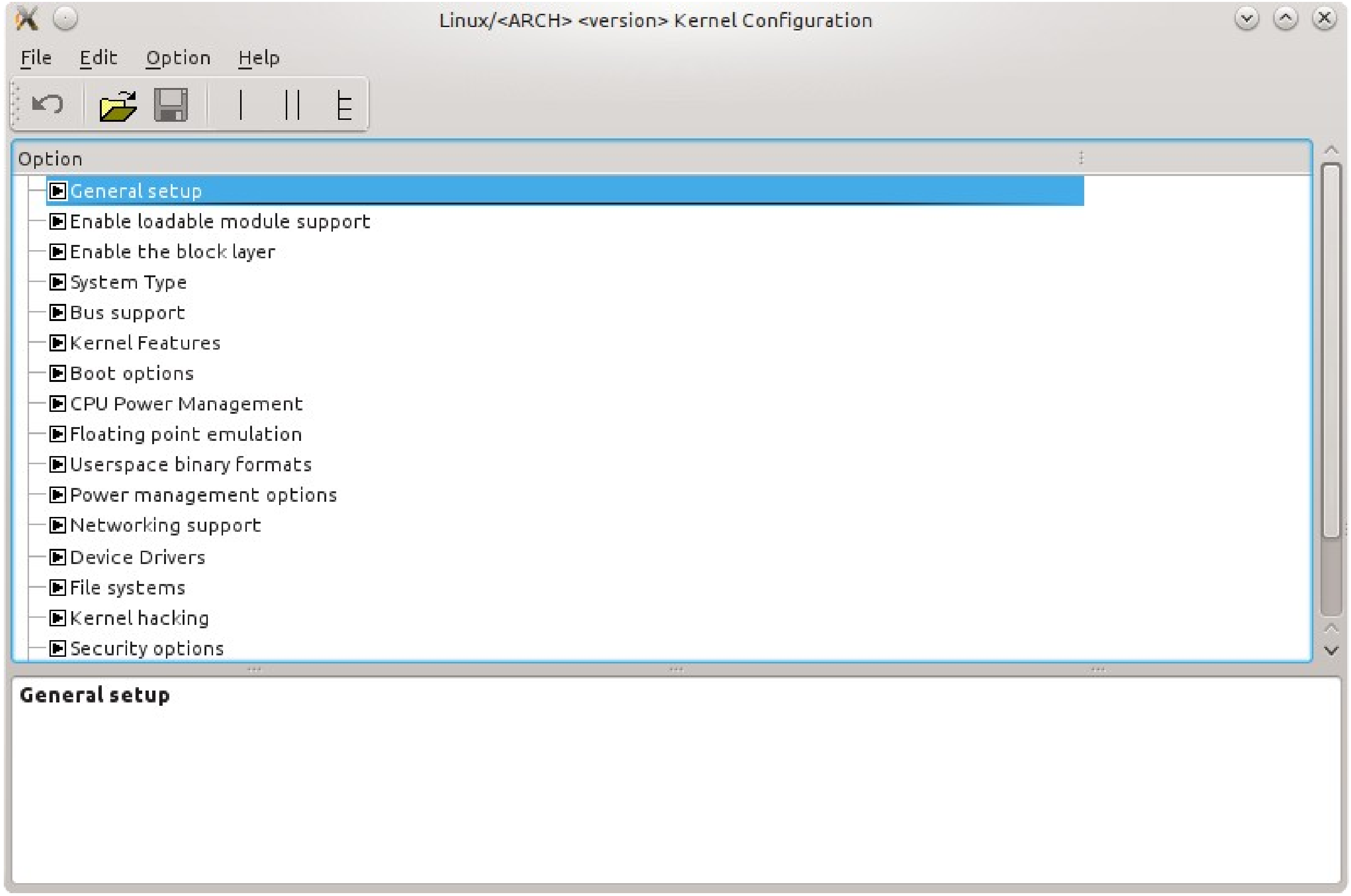
Embedded Linux Kernel

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

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 ARCH=<arch_name> 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 ARCH=<arch_name> <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 configs 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

```
$ export ARCH=<arch name>

$ export CROSS_COMPILE=<Path of Cross Compiler>
```

- Now you can type

```
$ make uImage LOADADDR=0x80008000 dtbs
```

- Can use the below command if you have multicore CPU

```
$ make -j
```

Embedded Linux Kernel
Building - Compilation

- 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
- Once the compilation is done you will get the kernel image in the following location **`arch/<arch>/boot`**
- To know some more details details on compilation you continue to next slide else **SKIP** to deploy the built image

- `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 vmlinux 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
80008000 -e 80008000 -n "Embedded Linux" -d
linux.bin uImage
```

- 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, you may type

```
$ cp arch/<arch>/boot/uImage /var/lib/tftpboot/

$ cp arch/<arch>/boot/dts/<required>.dtb
var/lib/tftpboot/
```

- Copy the kernel image to the target board as mentioned below

```
U-boot> tftp 0x80008000 uImage

U-boot> tftp 0x81D00000 <required>.dtb
```

Embedded Linux Kernel

Deploy

- Once the image is transferred you can boot the image as

  `U-boot> bootm`

- Your kernel should be loaded and executed now :)

File Systems

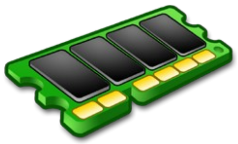
File Systems
Introduction

- 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

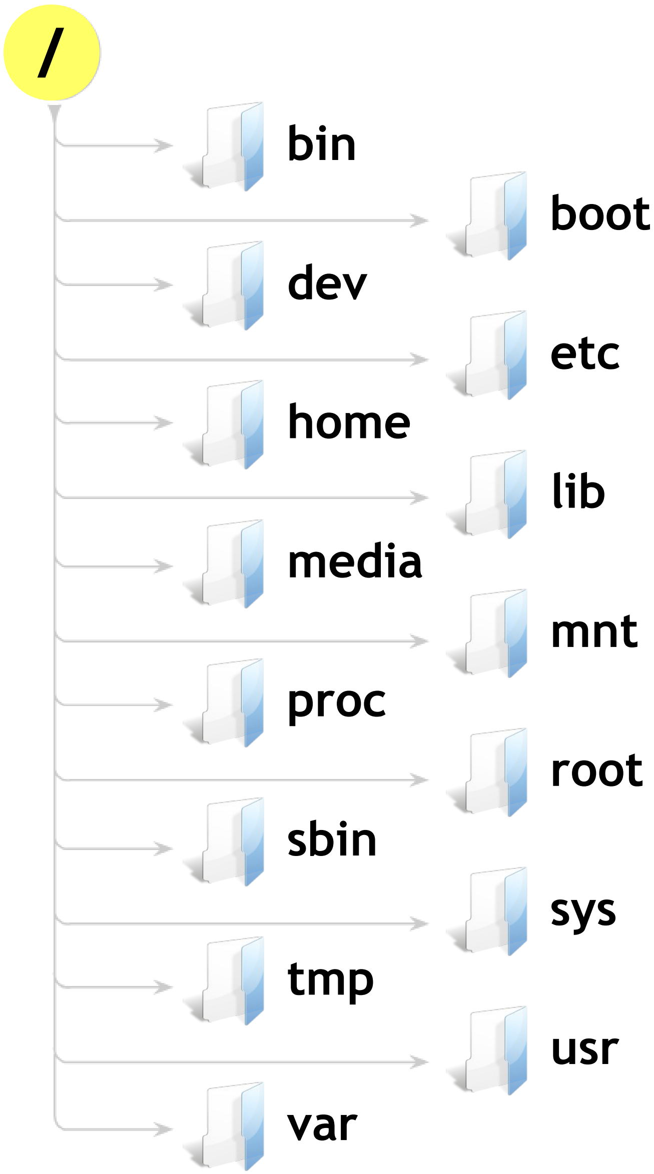
File System Contents

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

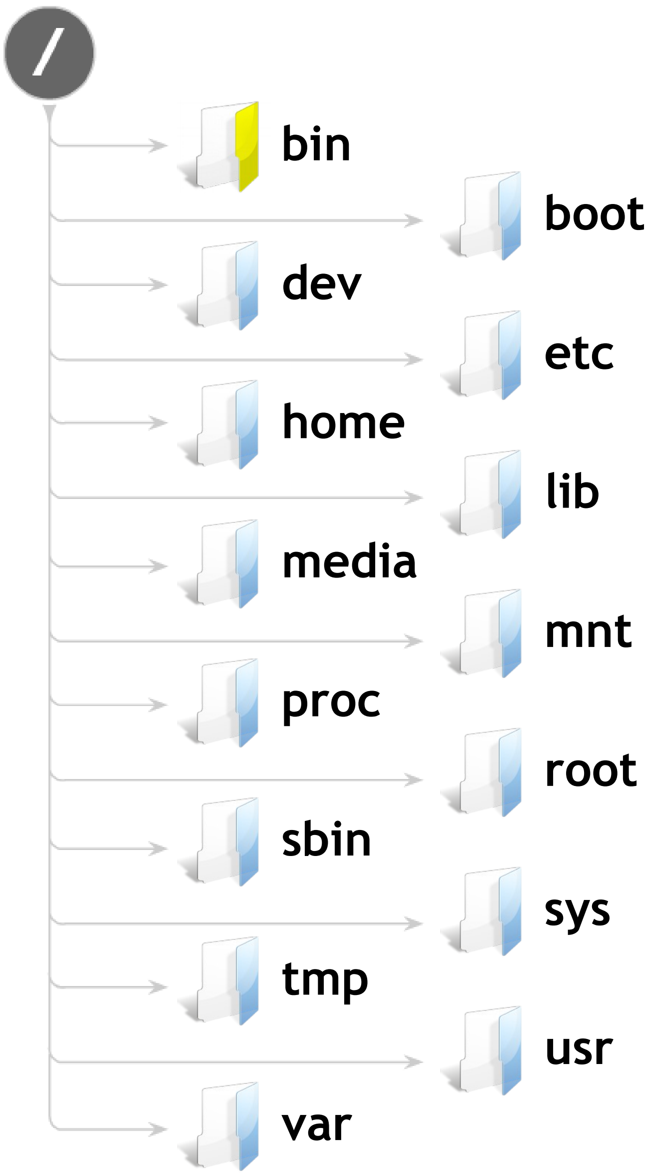
Linux Directory Structure



- 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

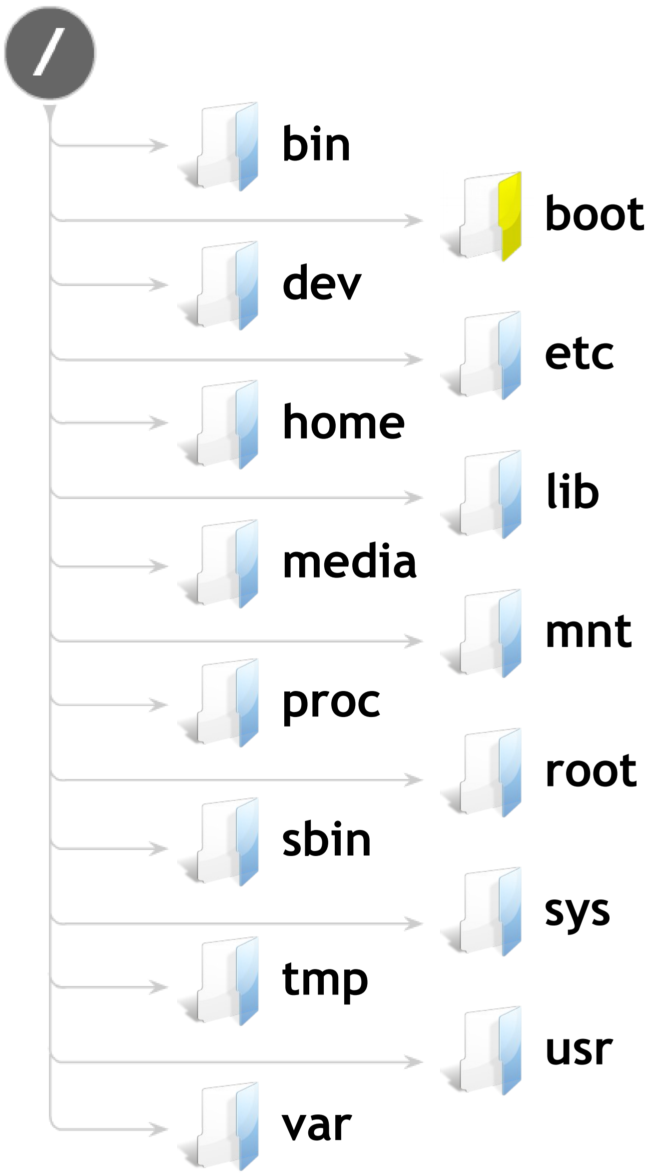
Linux Directory Structure



- 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

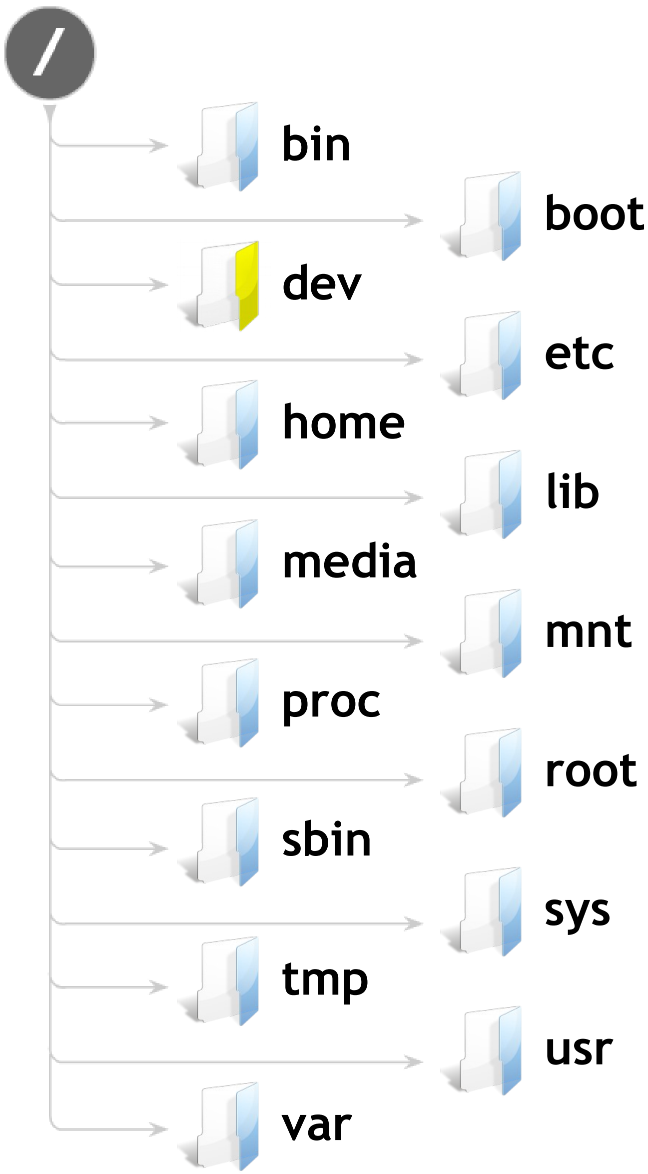
Linux Directory Structure



- 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)

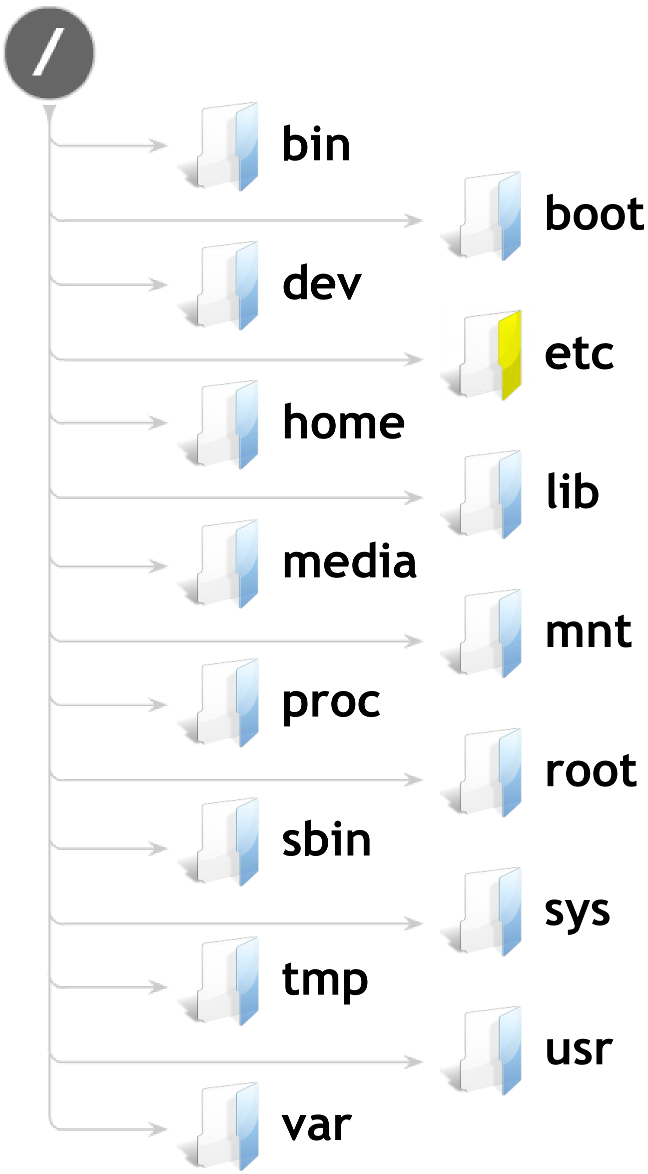
File System Contents

Linux Directory Structure



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

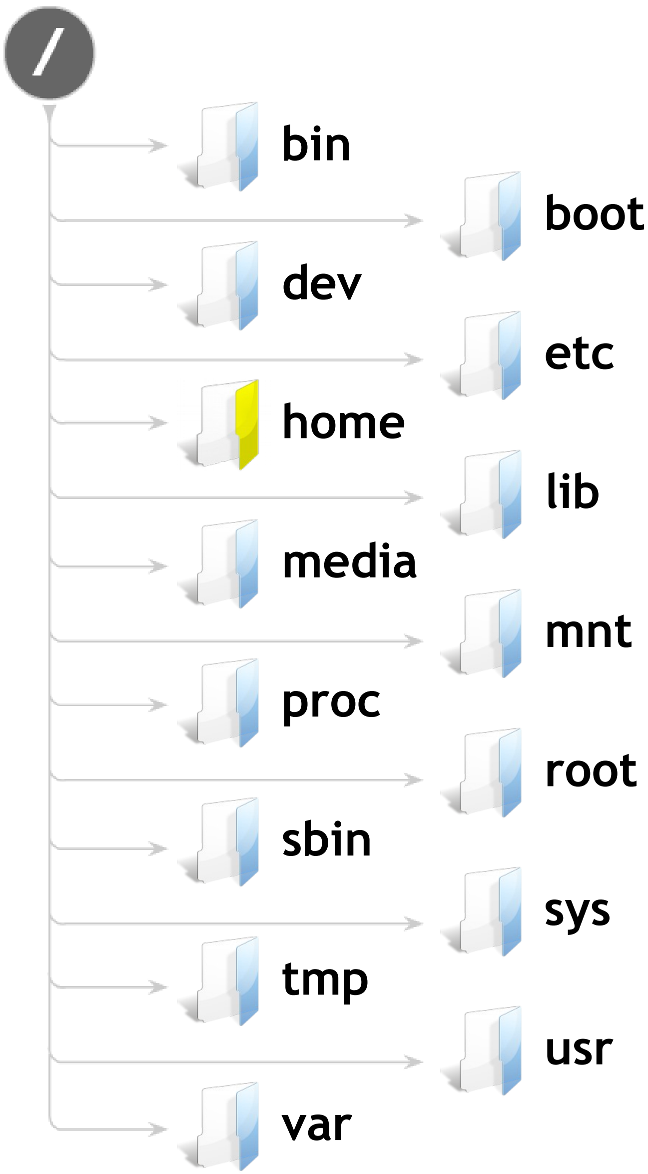
File System Contents
Linux Directory Structure



- 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

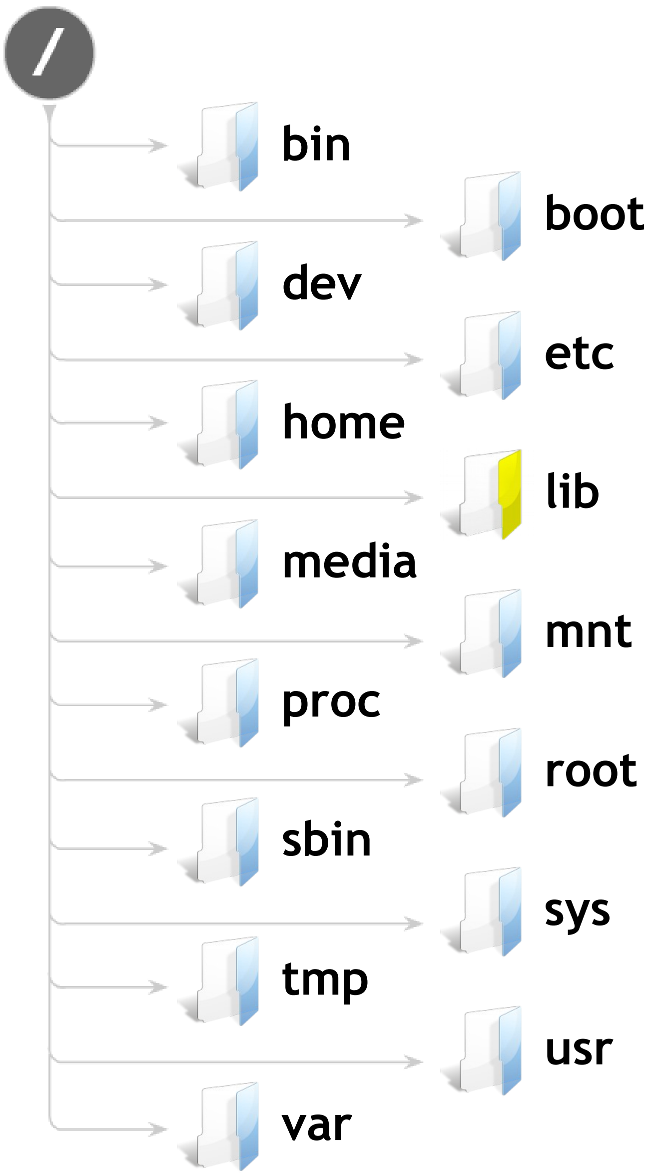
Linux Directory Structure



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

File System Contents

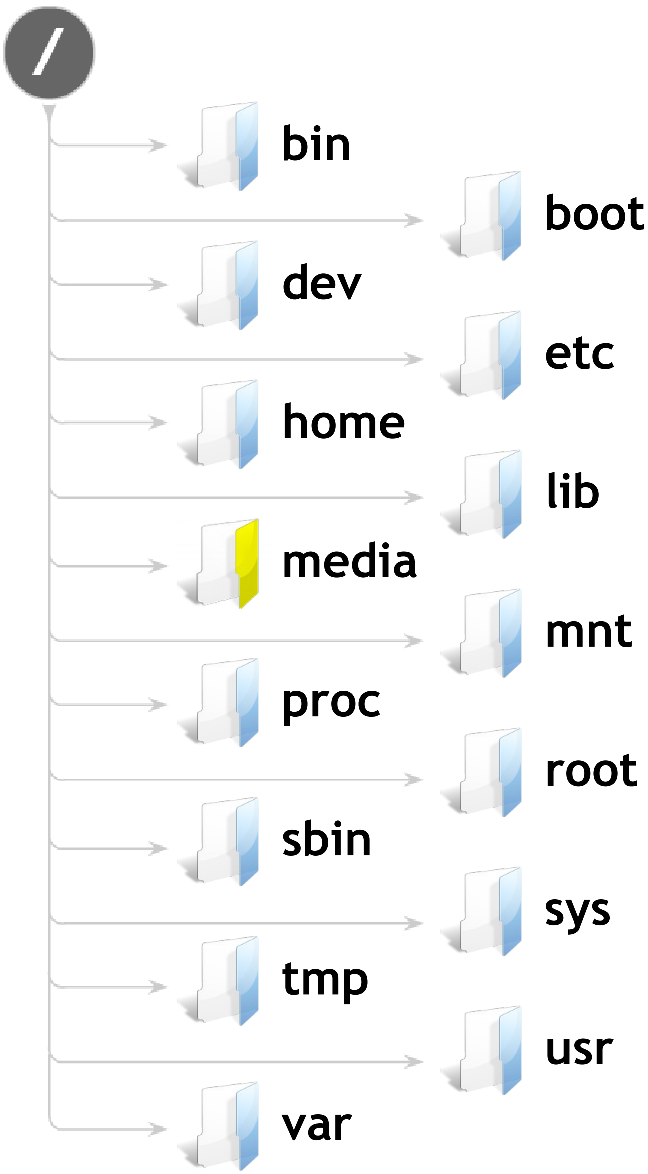
Linux Directory Structure



- 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

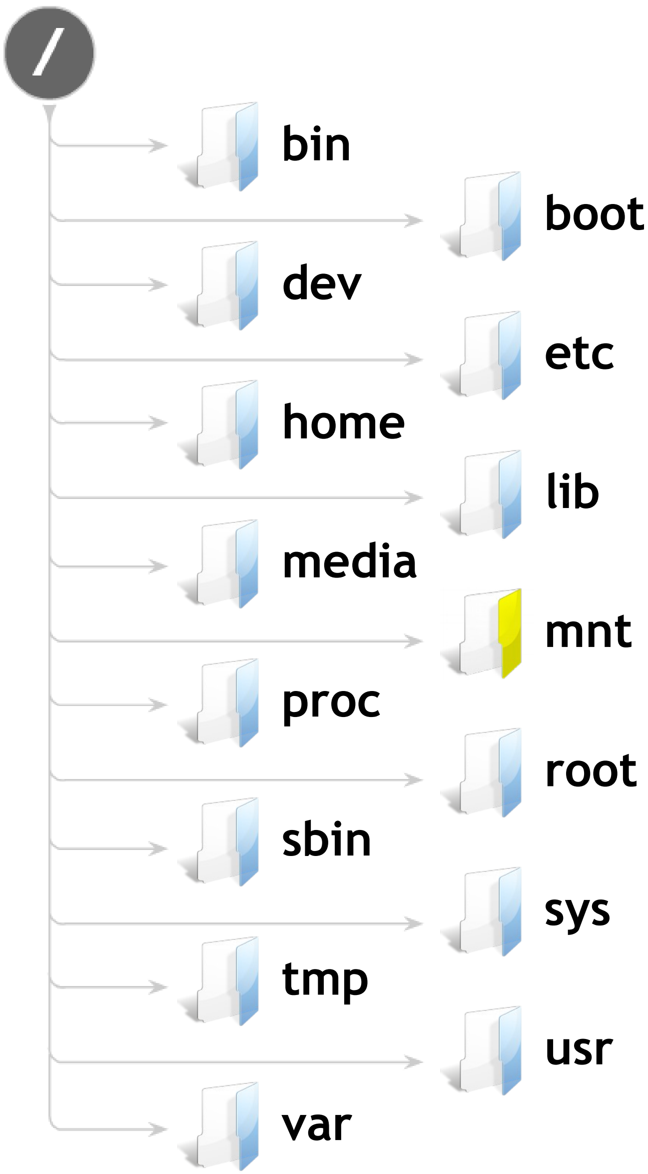
Linux Directory Structure



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

File System Contents

Linux Directory Structure

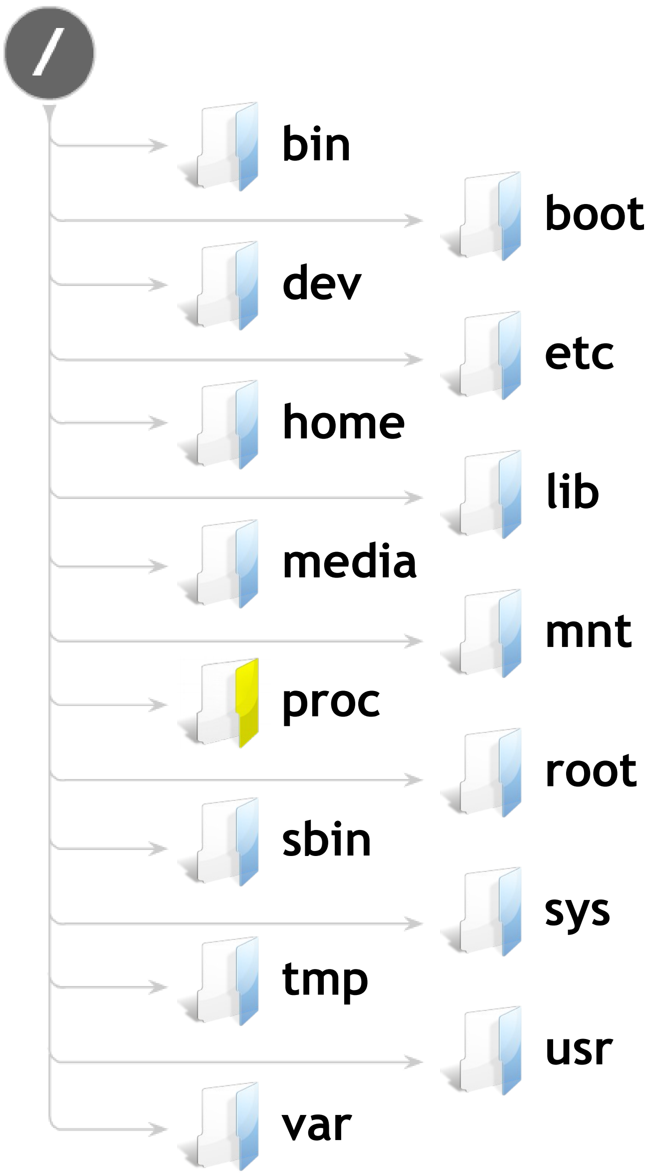


- Temporary mount directory where sysadmins can mount file systems.

EMERTXE

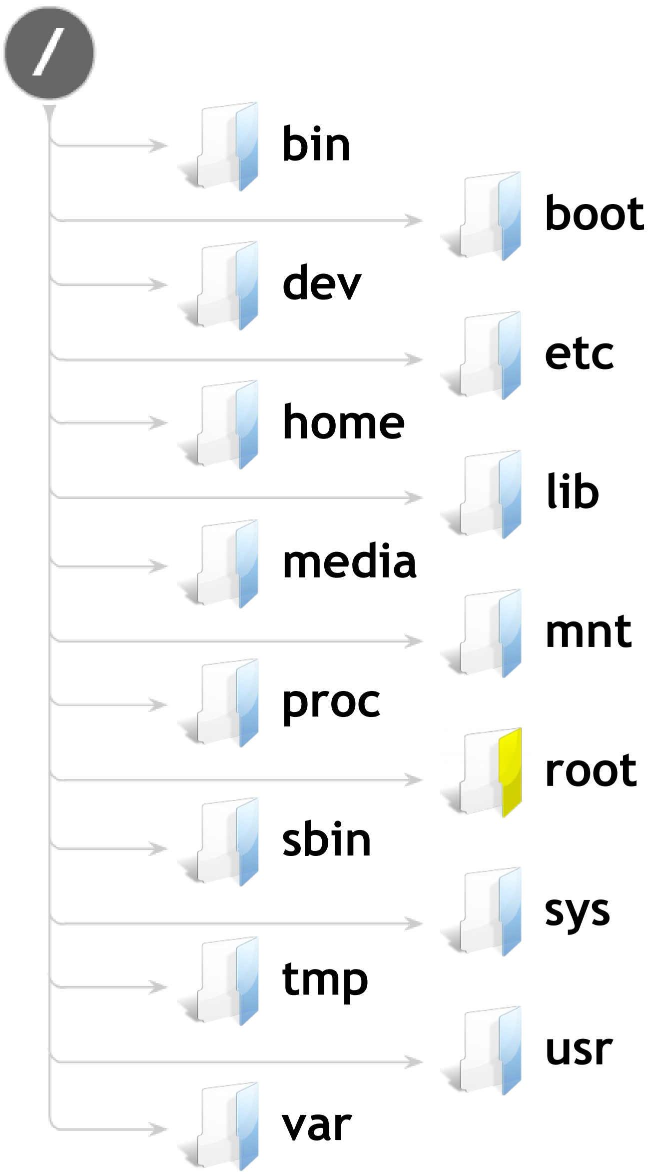
File System Contents

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

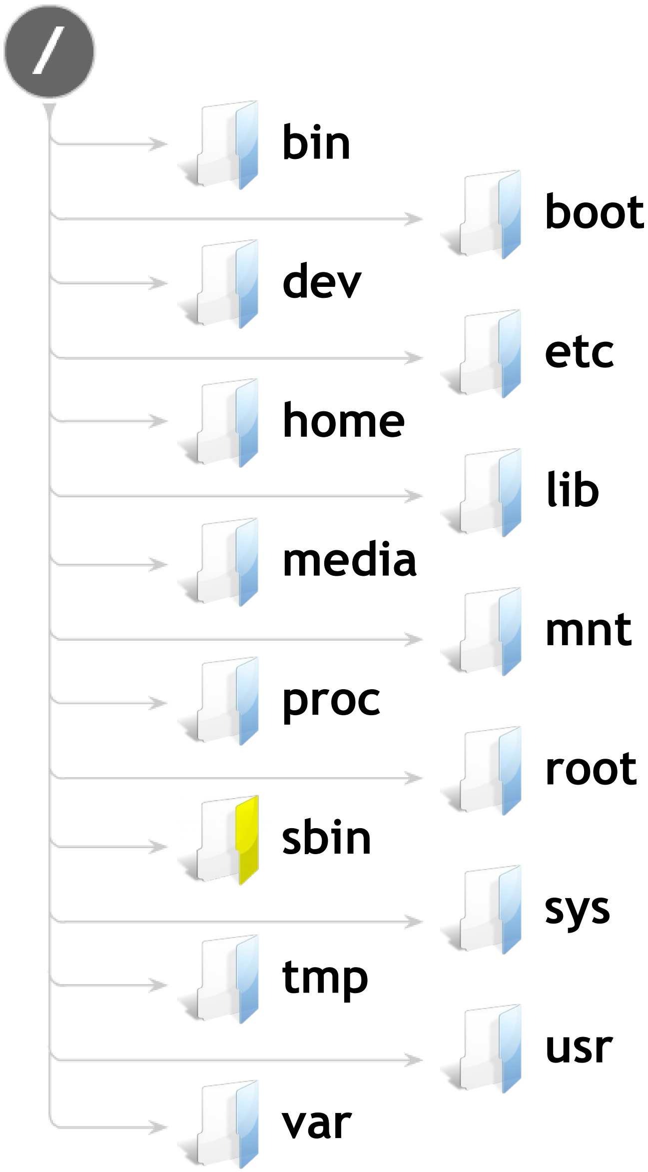
File System Contents
Linux Directory Structure



- Root user's home directory

File System Contents

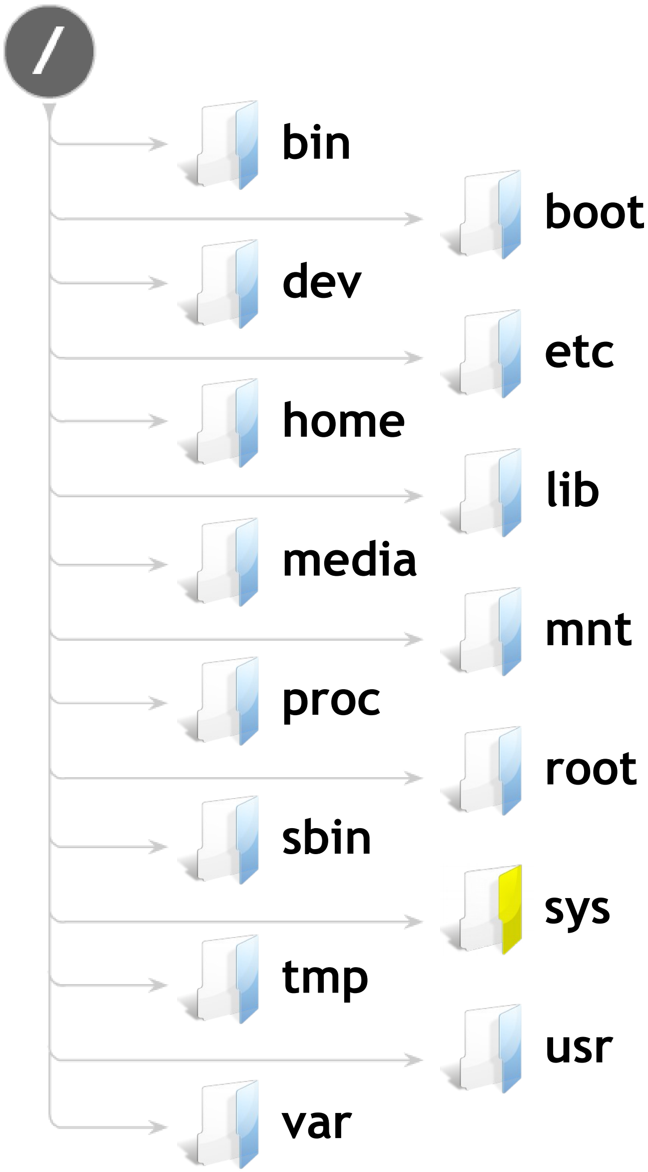
Linux Directory Structure



- 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

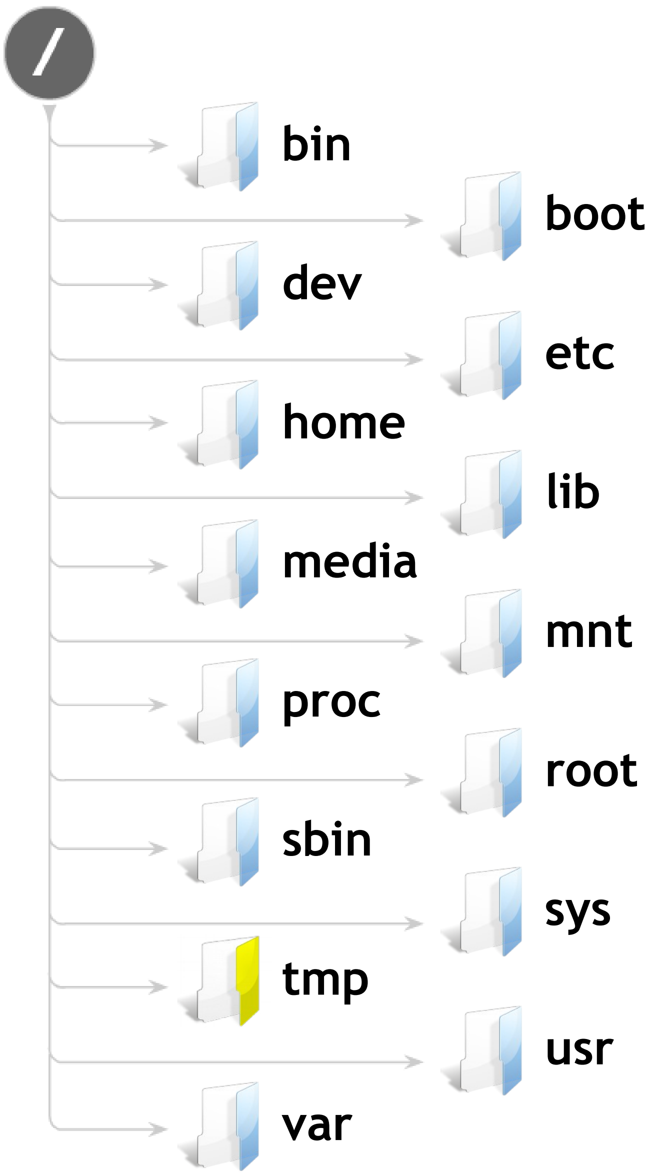
Linux Directory Structure



- Mount point of the sysfs virtual file system

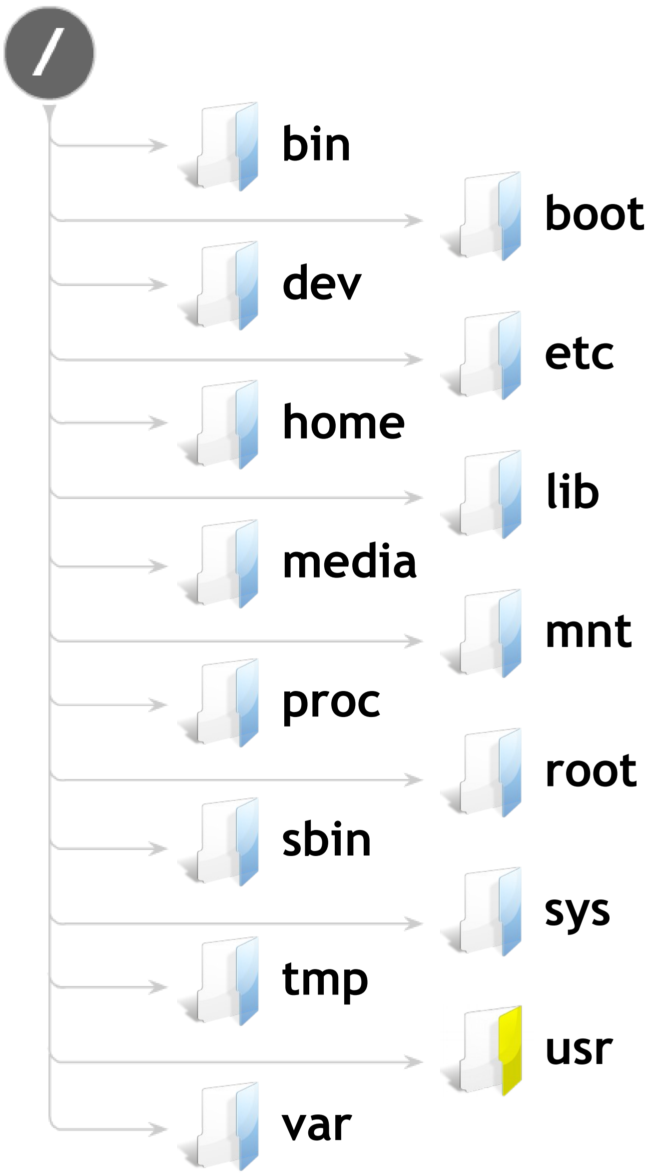
File System Contents

Linux Directory Structure



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

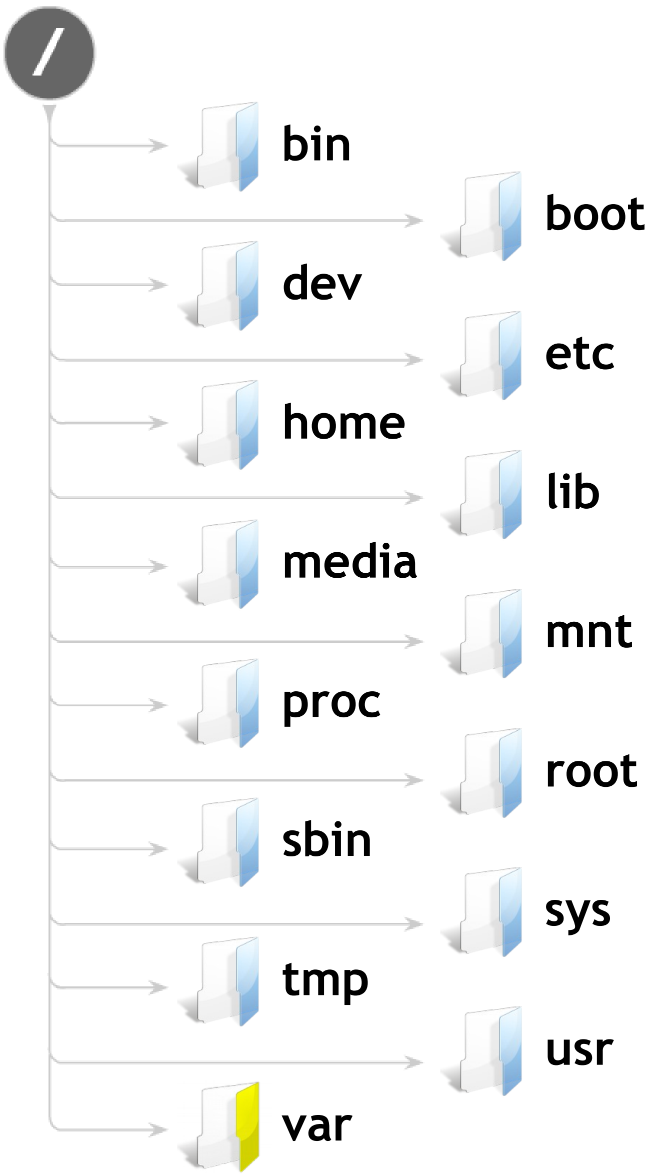
File System Contents
Linux Directory Structure



- 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
    - JFFS2
    - YAFFS

File System Formats

Variants

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

File System Partitions
Linux System

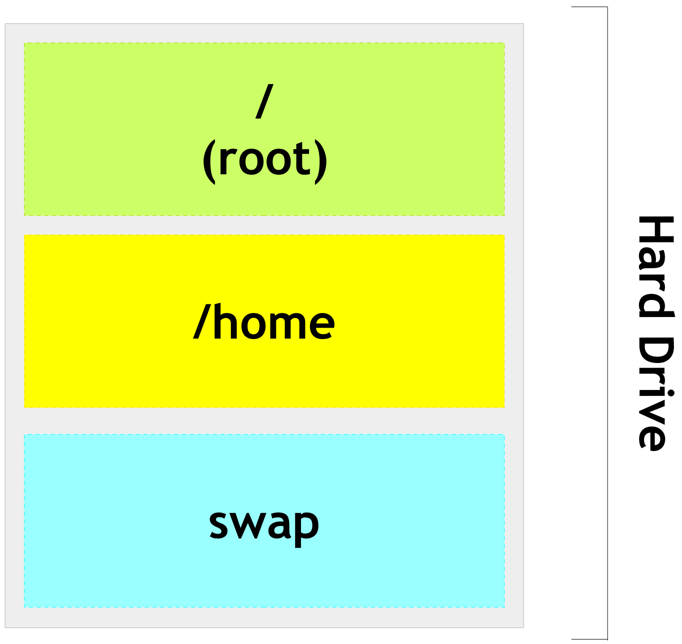
- Two major partitions
  - Data partition – Linux data partition
    - A root partition and one or more data partition
  - Swap partition - Expansion of physical memory, extra memory on storage (like virtual RAM)
    - Rarely used in embedded system because of the nature of the storage devices used

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

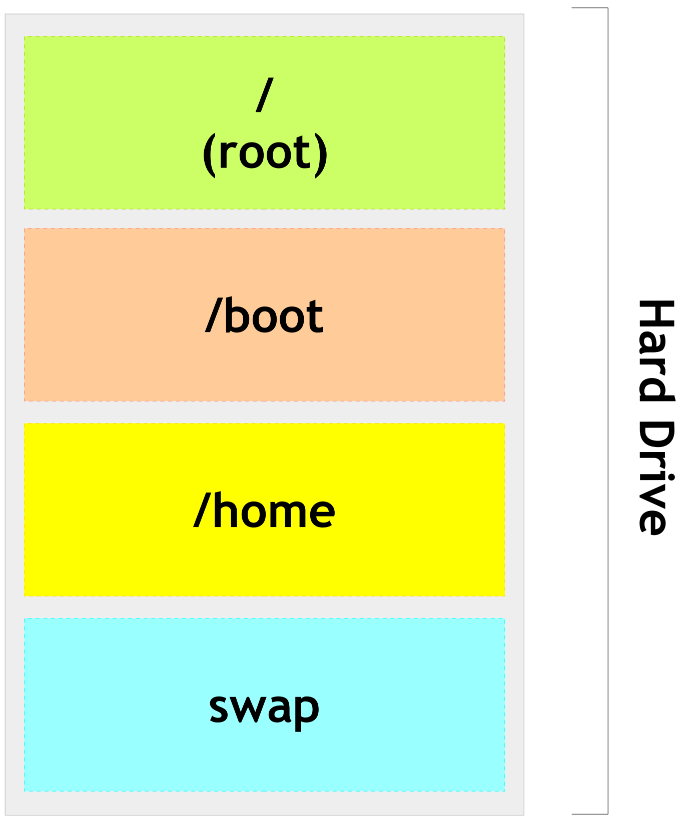
- One possible Desktop scheme is shown here
- Simplest and standard one to have
- Partition for OS which get mounted as / (root)
- Data partition mounted as /home
- Augment to RAM, referred and mounted as swap



File System Partitions

Schemes - Desktop – Example 2

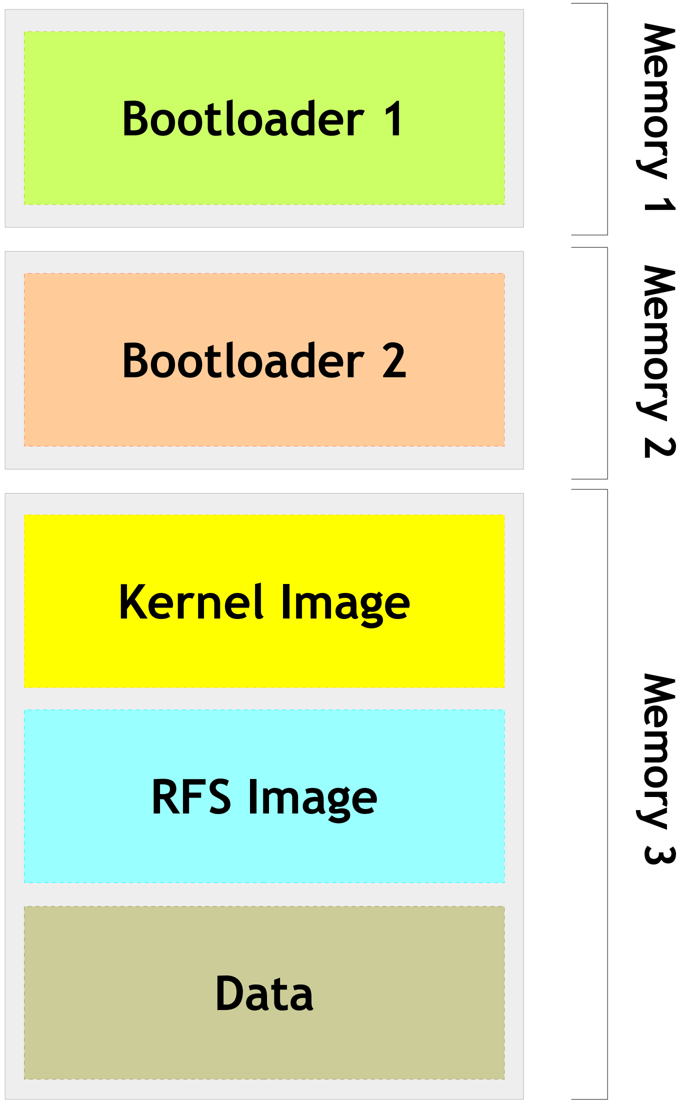
- Similar to previous one with a separate boot partition which will contain the boot images



File System Partitions

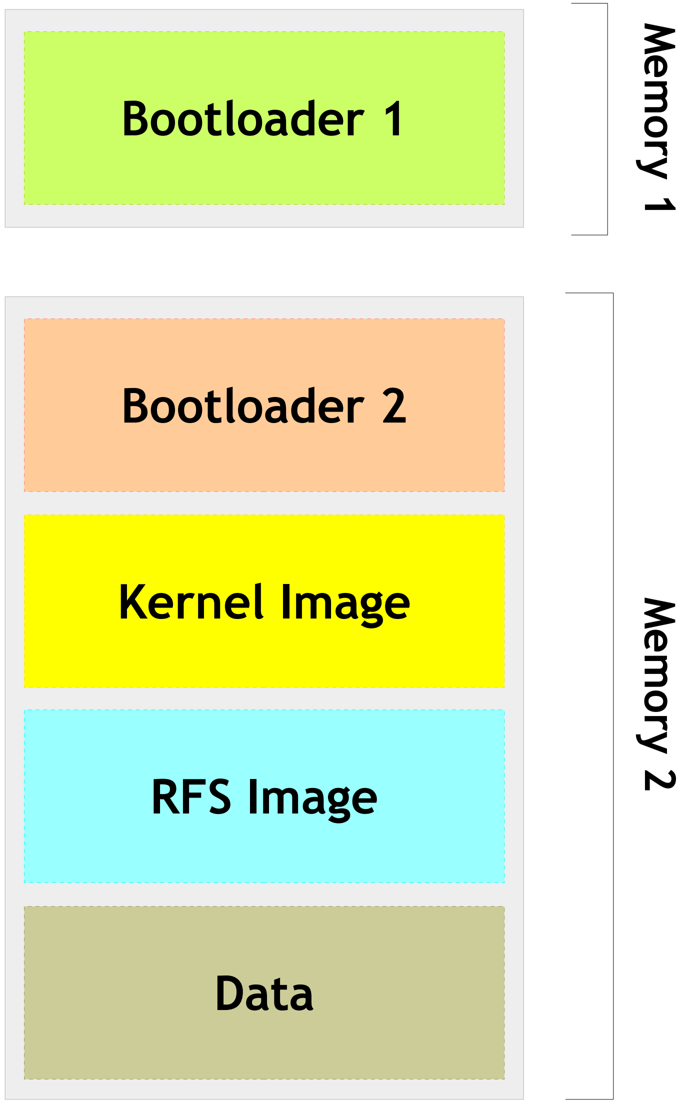
Schemes - Embedded – Example 1

- 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



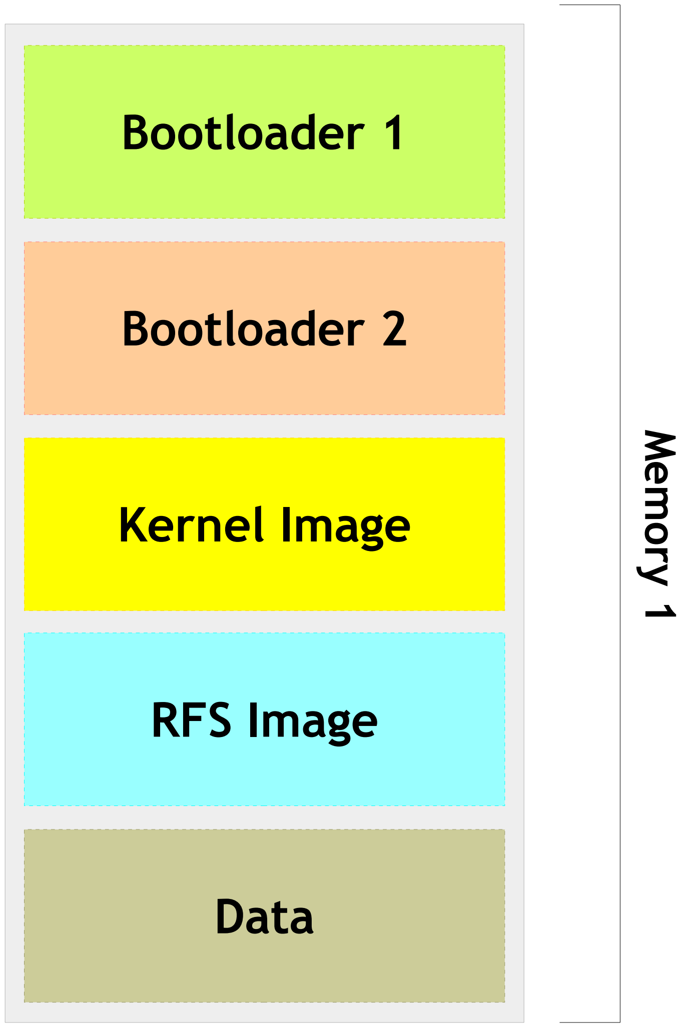
File System Partitions
Schemes - Embedded – Example 2

- 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



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 Builtroot is an alternative solution which we have already built!
- You will find it a output/target directory

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

Thank You