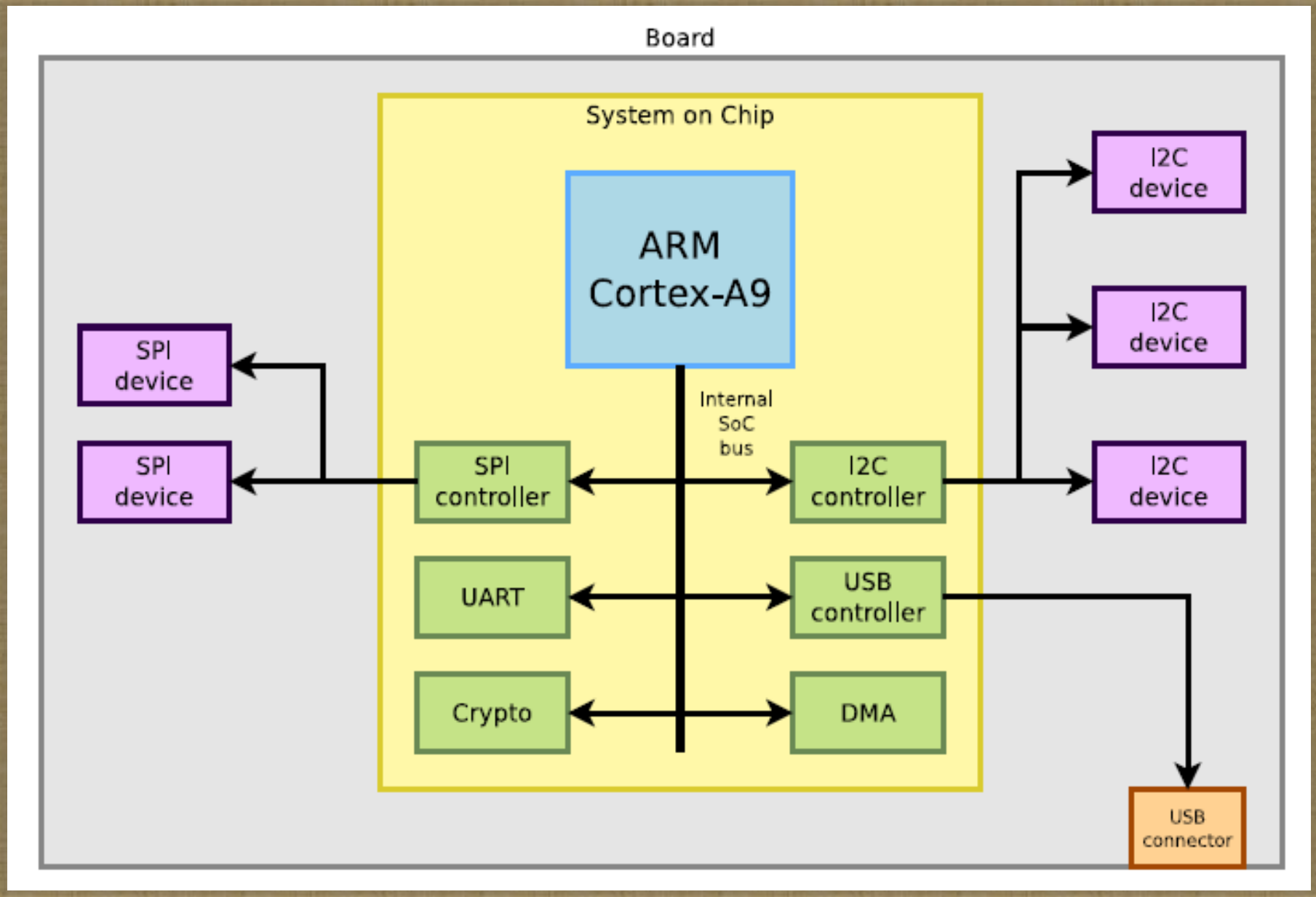
ARM platforms

- **ARM** (the company) designs CPU cores: instruction set, MMU, caches, etc.
  - They don't sell any hardware
- **Silicon vendors** buy the CPU core design from ARM, and around it add a number of *peripherals*, either designed internally or bought from third parties
  - Texas Instruments, Atmel, Marvell, Freescale, Qualcomm, Nvidia, etc.
  - They sell *System-on-chip* or *SoCs*
- **System makers** design an actual board, with one or several processors, and a number of on-board peripherals

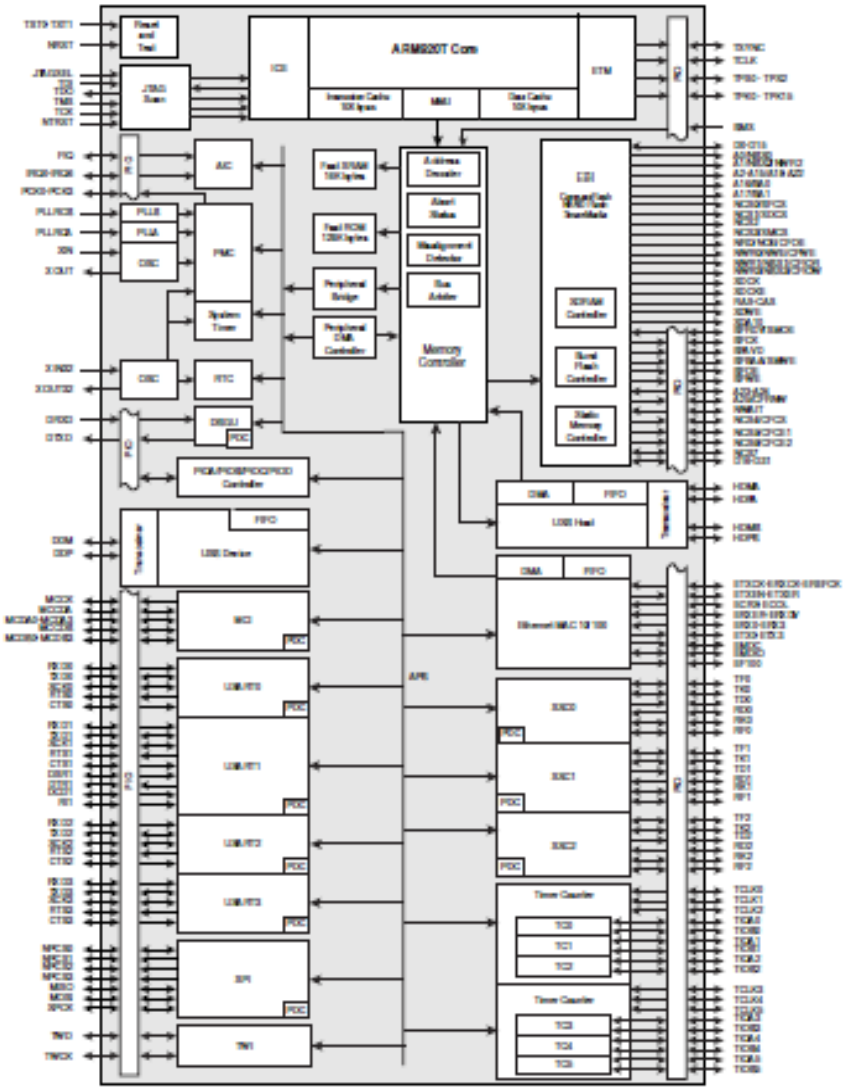
Schematic view of an ARM platform



System on Chip

A System on Chip is typically composed of:

- One or multiple CPU cores
- A bus
- Oscillators and PLL
- Timers
- Memory controller
- Interrupt controller
- Multiple peripherals:
  - UART
  - RTC
  - SPI
  - I2C
  - ADC
  - USB controller
  - ...



ARM cores

The Linux kernel supports a wide range of ARM based architectures, starting with ARMv4T:

ARM family	ARM architecture	ARM Core
ARM7T	ARMv4T	ARM7TDMI ARM720T ARM740T
ARM9T	ARMv4T	ARM9TDMI ARM920T ARM922T
		ARM925T ARM926T ARM940T
ARM9E	ARMv5TE	ARM946E-S
ARM10E	ARMv5TE	ARM1020T ARM1020E ARM1022E
	ARMv5TEJ	ARM1026EJ-S
ARM11	ARMv6Z	ARM1176JZF-S
	ARMv6K	ARM11MPCore
Cortex-M	ARMv7-M	Cortex-M3, Cortex-M4, Cortex-M7
Cortex-A (32-bit)	ARMv7-A	Cortex-A5, Cortex-A7
		Cortex-A8, Cortex-A9, Cortex-A12,
		Cortex-A15, Cortex-A17
Cortex-A (64-bit)	ARMv8-A	Cortex-A53, Cortex-A57, Cortex-A72

ARM cores

Third parties can also license the instruction set and create their own cores:

ARM ISA	Third party core
ARMv4	Faraday FA256, StrongARM SA-110, SA-1100
ARMv5TE	Xscale
ARMv5	Marvell PJ1, Feroceon
ARMv7-A	Broadcom Brahma-B15, Marvell PJ4, PJ4B, Qualcomm Krait, Scorpion
ARMv8-A	Cavium Thunder, Nvidia Denver, Qualcomm Kryo

ARM SoCs vendors

ARM SoC vendors with good mainline kernel support include:

- Allwinner
- Atmel
- Freescale
- Marvell
- Rockchip
- Samsung
- ST Micro
- TI (sitara and OMAP families)
- Xilinx

However, be careful when needing certain features like GPU acceleration.

Community boards

A good way to create a prototype is to use a community board which is usually inexpensive and has expansion headers:

Examples include:

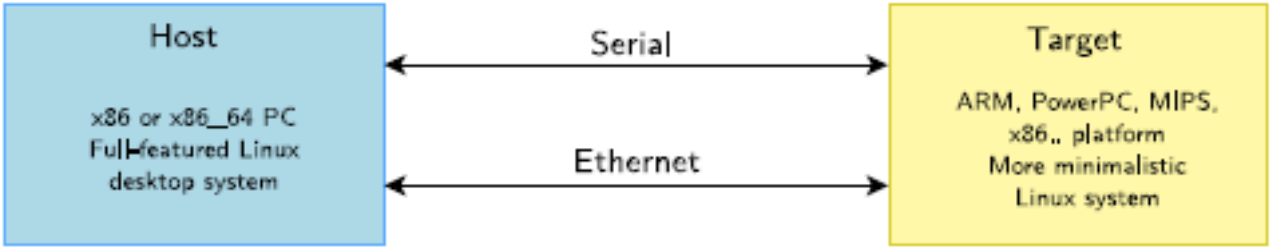
- BeagleBone Black
- Sama5dx Xplained
- OLinuXino boards

Software components

- Cross-compilation toolchain
  - Compiler that runs on the development machine, but generates code for the target
- Bootloader
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- Linux Kernel
  - Contains the process and memory management, network stack, device drivers and provides services to user space applications
- C library
  - The interface between the kernel and the user space applications
- Libraries and applications
  - Third-party or in-house

Host vs. target

- When doing embedded development, there is always a split between
  - The *host*, the development workstation, which is typically a powerful PC
  - The *target*, which is the embedded system under development
- They are connected by various means: almost always a serial line for debugging purposes, frequently an Ethernet connection, sometimes a JTAG interface for low-level debugging

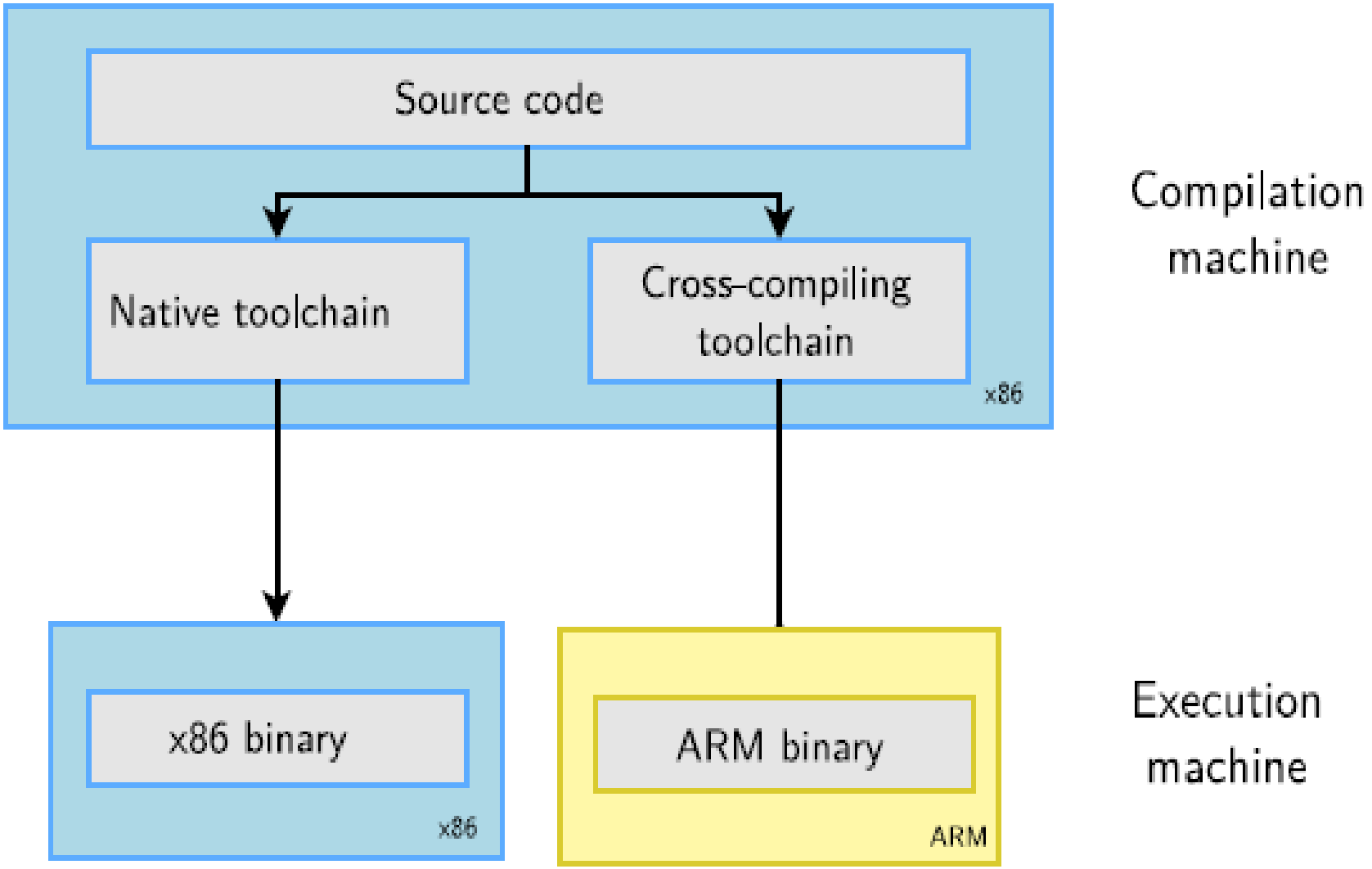


Serial line communication program

- An essential tool for embedded development is a serial line communication program, like HyperTerminal in Windows.
- There are multiple options available in Linux: Minicom, Picocom, Gtkterm, Putty, etc.
- In this training session, we recommend using the simplest of them: `picocom`
  - Installation with `sudo apt-get install picocom`
  - Run with `picocom -b BAUD_RATE /dev/SERIAL_DEVICE`
  - Exit with `Control-A Control-X`
- `SERIAL_DEVICE` is typically
  - `ttyUSBx` for USB to serial converters
  - `ttySx` for real serial ports

Cross-compiling toolchains

- The usual development tools available on a GNU/Linux workstation is a **native toolchain**
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible o not interesting to use a native toolchain
  - The target is too restricted in terms of storage and/or memor
  - The target is very slow compared to your workstation
  - You may not want to install all development tools on your target.
- Therefore, **cross-compiling toolchains** are generally used. They run on your workstation but generate code for your target.


Source code
Native toolchain
Cross-compiling toolchain
x86
Compilation machine
x86 binary
x86
ARM binary
ARM
Execution machine

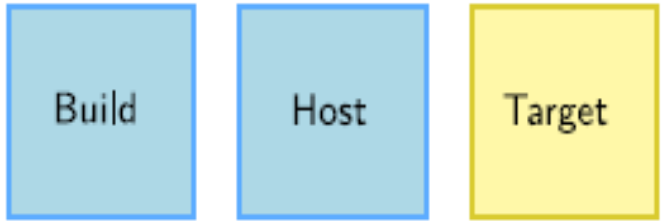
Machines in build procedures

- Three machines must be distinguished when discussing toolchain creation
  - The **build** machine, where the toolchain is built.
  - The **host** machine, where the toolchain will be executed.
  - The **target** machine, where the binaries created by the toolchain are executed.
- Four common build types are possible for toolchains

Different toolchain build procedures

Native build

used to build the normal gcc of a workstation



Cross build

used to build a toolchain that runs on your workstation but generates binaries for the target

The most common case in embedded development

Cross-native build

used to build a toolchain that runs on your target and generates binaries for the target

Canadian build

used to build on architecture A a toolchain that runs on architecture B and generates binaries for architecture C

Building a toolchain manually

Building a cross-compiling toolchain by yourself is a difficult and painful task! Can take days or weeks!

- Lots of details to learn: many components to build, complicated configuration
- Lots of decisions to make (such as C library version, ABI, floating point mechanisms, component versions)
- Need kernel headers and C library sources
- Need to be familiar with current gcc issues and patches on your platform
- Useful to be familiar with building and configuring tools

Get a pre-compiled toolchain

- Solution that many people choose
  - Advantage: it is the simplest and most convenient solution
  - Drawback: you can't fine tune the toolchain to your needs
- Make sure the toolchain you find meets your requirements: CPU, endianness, C library, component versions, ABI, soft float or hard float, etc.
- Possible choices
  - Toolchains packaged by your distribution
    Ubuntu examples:
    ```
    sudo apt-get install gcc-arm-linux-gnueabi
    sudo apt-get install gcc-arm-linux-gnueabihf
    ```
  - Sourcery CodeBench toolchains, now only supporting MIPS, NIOS-II, AMD64, Hexagon. Old versions with ARM support still available through build systems (Buildroot...)
  - Toolchain provided by your hardware vendor.

Toolchain building utilities

Another solution is to use utilities that **automate the process of building the toolchain**

- Same advantage as the pre-compiled toolchains: you don't need to mess up with all the details of the build process
- But also offers more flexibility in terms of toolchain configuration, component version selection, etc.
- They also usually contain several patches that fix known issues with the different components on some architectures
- Multiple tools with identical principle: shell scripts or Makefile that automatically fetch, extract, configure, compile and install the different components

Many root filesystem build systems also allow the construction of a cross-compiling toolchain

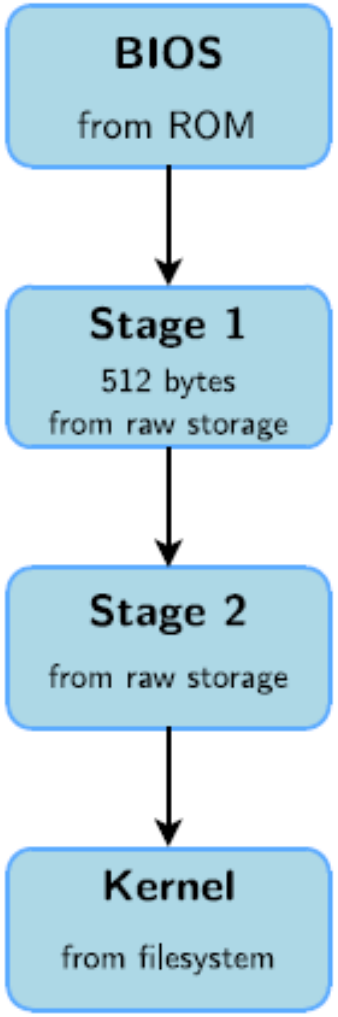
- **Buildroot**
  - Makefile-based. Can build (e)glibc, uClibc and musl based toolchains, for a wide range of architectures.
  - `http://www.buildroot.net`
- **PTXdist**
  - Makefile-based, uClibc or glibc, maintained mainly by *Pengutronix*
  - `http://pengutronix.de/software/ptxdist/`
- **OpenEmbedded / Yocto**
  - A featureful, but more complicated build system
  - `http://www.openembedded.org/`
  - `https://www.yoctoproject.org/`

Bootloaders

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

Bootloaders on BIOS-based x86

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- On old BIOS-based x86 platforms: the BIOS is responsible for basic hardware initialization and loading of a very small piece of code from non-volatile storage.
- This piece of code is typically a 1st stage bootloader, which will load the full bootloader itself.
- It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.
- This sequence is different for modern EFI-based systems.



Bootloaders on x86

- GRUB, Grand Unified Bootloader, the most powerful one.
  `http://www.gnu.org/software/grub/`
  - Can read many filesystem formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.

- Syslinux, for network and removable media booting (USB key, CD-ROM)
  `http://www.kernel.org/pub/linux/utils/boot/syslinux/`

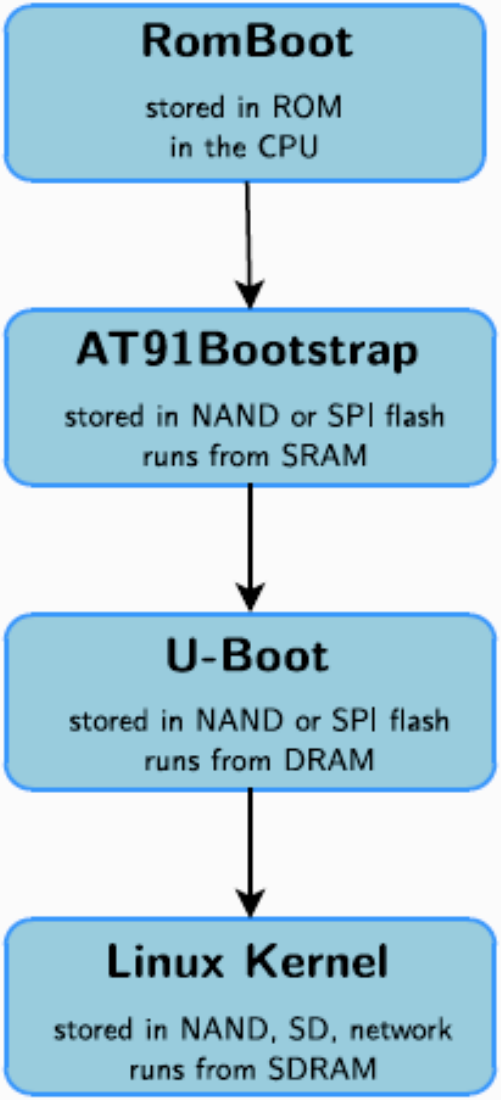
Booting on embedded CPUs: case 1

- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows random access, which NAND doesn't allow
- **Not very common anymore** (unpractical, and requires NOR flash)

Booting on embedded CPUs: case 2

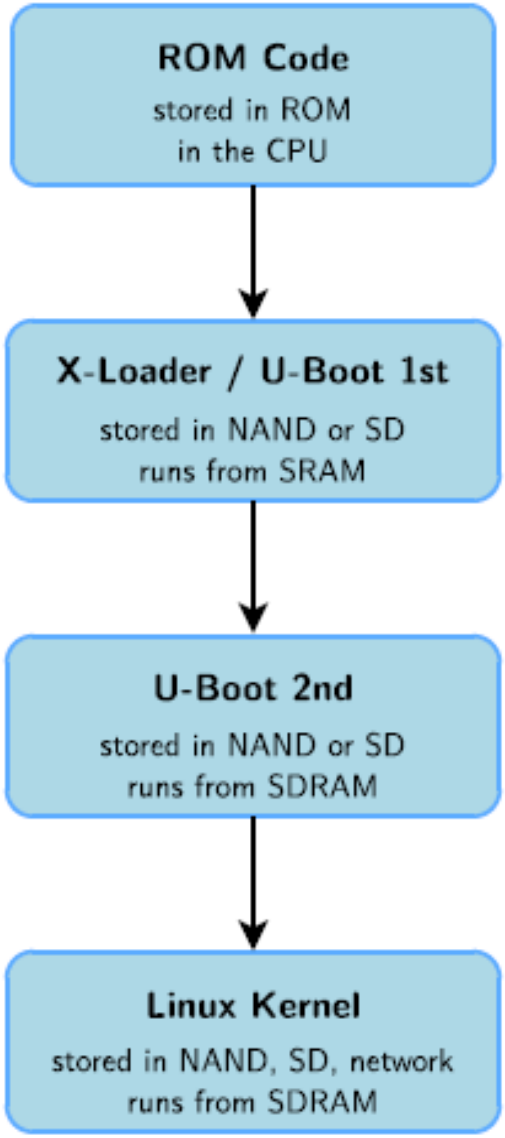
- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, "ROM code" on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash, UART (transmitting data over the serial line), etc.
- The first stage bootloader is
  - Limited in size due to hardware constraints (SRAM size)
  - Provided either by the CPU vendor or through community projects
- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into RAM

Booting on ARM Atmel AT91



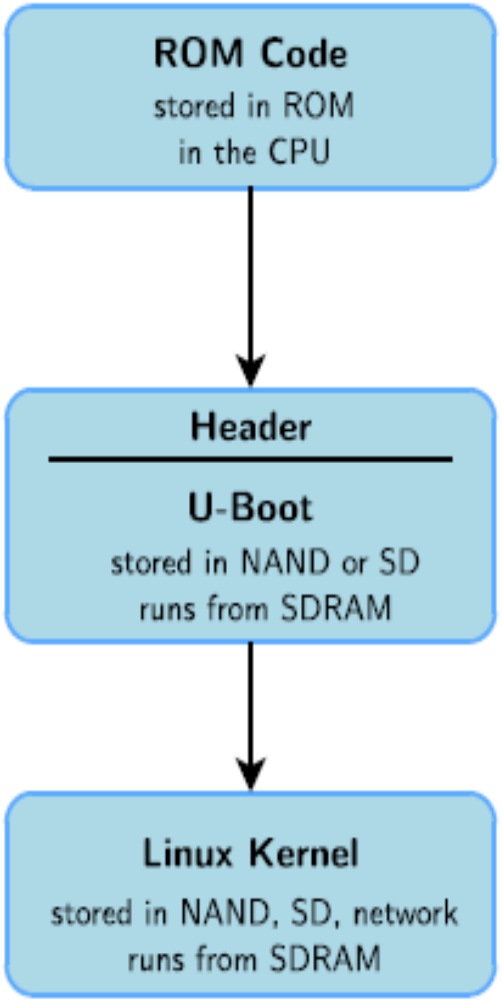
- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- **AT91Bootstrap**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Booting on ARM TI OMAP3



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.
- **X-Loader** or **U-Boot**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.bin` or `u-boot.img`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Booting on Marvell SoC



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.kwb`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Porting the Bootloader

1st stage

The main goal of the first stage bootloader is to configure the RAM controller. Then it needs to be able to load the second stage bootloader from storage (NAND flash, SPI flash, NOR flash, MMC/eMMC) to RAM. The main porting steps are:

- Finding the proper RAM timings and settings them from the first stage.
- Configuring the storage IP
- Copying the second stage to RAM

Usually, the driver for the storage IP is already present in your first stage bootloader.

2nd stage

- The second stage bootloader has to load the Linux kernel from storage to RAM.
- Depending on the kernel version, it will also set the `ATAGS` or load the Device Tree.
- It may also load an `initramfs` to be used as the `root filesystem`.
- That is also a good place to implement base board or board variant detection if necessary.
- During development, the second stage bootloader also provides more debugging utilities like reading and writing to memory or Ethernet access.

The main porting steps are:

- Configuring the storage IP
- Copying the Linux kernel from storage to RAM
- Optional: copying the Device Tree to RAM
- Optional: implement boot scripts
- Optional: implement base board/board variant detection
- Optional: implement debug tools

Generic bootloaders for embedded CPUs

- There are several open-source generic bootloaders. Here are the most popular ones:
  - **U-Boot**, the universal bootloader by Denx
    The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays. We will study it in detail.
    `http://www.denx.de/wiki/U-Boot`
  - **Barebox**, a new architecture-neutral bootloader, written as a successor of U-Boot. Better design, better code, active development, but doesn't yet have as much hardware support as U-Boot.
    `http://www.barebox.org`
- There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - RedBoot, Yaboot, PMON, etc.

U-Boot

U-Boot is a typical free software project

- License: GPLv2 (same as Linux)
- Freely available at `http://www.denx.de/wiki/U-Boot`
- Documentation available at `http://www.denx.de/wiki/U-Boot/Documentation`
- The latest development source code is available in a Git repository: `http://git.denx.de/?p=u-boot.git;a=summary`
- Development and discussions happen around an open mailing-list `http://lists.denx.de/pipermail/u-boot/`
- Since the end of 2008, it follows a fixed-interval release schedule. Every three months, a new version is released. Versions are named `YYYY.MM`.

U-Boot configuration

- Get the source code from the website, and uncompress it
- The `configs/` directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc.
- Note: U-Boot is migrating from board configuration defined in header files (`include/configs/`) to *defconfig* like in the Linux kernel (`configs/`)
  - Not all boards have been converted to the new configuration system.
  - Older U-Boot releases provided by hardware vendors may not yet use this new configuration system.

U-Boot configuration file

CHIP_defconfig

```
CONFIG_ARM=y
CONFIG_ARCH_SUNXI=y
CONFIG_MACH_SUN5I=y
CONFIG_DRAM_TIMINGS_DDR3_800E_1066G_1333J=y
# CONFIG_MMC is not set
CONFIG_USB0_VBUS_PIN="PB10"
CONFIG_VIDEO_COMPOSITE=y
CONFIG_DEFAULT_DEVICE_TREE="sun5i-r8-chip"
CONFIG_SPL=y
CONFIG_SYS_EXTRA_OPTIONS="CONS_INDEX=2"
# CONFIG_CMD_IMLS is not set
CONFIG_CMD_DFU=y
CONFIG_CMD_USB_MASS_STORAGE=y
CONFIG_AXP_ALDO3_VOLT=3300
CONFIG_AXP_ALDO4_VOLT=3300
CONFIG_USB_MUSB_GADGET=y
CONFIG_USB_GADGET=y
CONFIG_USB_GADGET_DOWNLOAD=y
CONFIG_G_DNL_MANUFACTURER="Allwinner Technology"
CONFIG_G_DNL_VENDOR_NUM=0x1f3a
CONFIG_G_DNL_PRODUCT_NUM=0x1010
CONFIG_USB_EHCI_HCD=y
```

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - `make BOARDNAME_defconfig`
  - Where `BOARDNAME` is the name of a configuration, as visible in the `configs/` directory.
  - You can then run `make menuconfig` to further customize U-Boot's configuration!
- Make sure that the cross-compiler is available in `PATH`
- Compile U-Boot, by specifying the cross-compiler prefix. Example, if your cross-compiler executable is `arm-linux-gcc`:
  `make CROSS_COMPILE=arm-linux-`
- The main result is a `u-boot.bin` file, which is the U-Boot image. Depending on your specific platform, there may be other specialized images: `u-boot.img`, `u-boot.kwb`, `MLO`, etc.

Installing U-Boot

U-Boot must usually be installed in flash memory to be executed by the hardware. Depending on the hardware, the installation of U-Boot is done in a different way:

- The CPU provides some kind of specific boot monitor with which you can communicate through serial port or USB using a specific protocol
- The CPU boots first on removable media (MMC) before booting from fixed media (NAND). In this case, boot from MMC to reflash a new version
- U-Boot is already installed, and can be used to flash a new version of U-Boot. However, be careful: if the new version of U-Boot doesn't work, the board is unusable
- The board provides a JTAG interface, which allows to write to the flash memory remotely, without any system running on the board. It also allows to rescue a board if the bootloader doesn't work.

U-boot prompt

- Connect the target to the host through a serial console.
- Power-up the board. On the serial console, you will see something like:

```
U-Boot 2016.05 (May 17 2016 - 12:41:15 -0400)

CPU: SAMA5D36
Crystal frequency:       12 MHz
CPU clock        :      528 MHz
Master clock     :      132 MHz
DRAM:  256 MiB
NAND:  256 MiB
MMC:   mci: 0

In:    serial
Out:   serial
Err:   serial
Net:   gmac0

Hit any key to stop autoboot:  0
=>
```

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the `help` command.

Information commands

Flash information (NOR and SPI flash)

```
U-Boot> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (RO) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C0041FFF (RO) U-Boot
```

NAND flash information

```
U-Boot> nand info
Device 0: nand0, sector size 128 KiB
  Page size      2048 b
  OOB size         64 b
  Erase size   131072 b
```

Version details

```
U-Boot> version
U-Boot 2016.05 (May 17 2016 - 12:41:15 -0400)
```

Important commands

- The exact set of commands depends on the U-Boot configuration
- `help` and `help command`
- `ext2load`, loads a file from an ext2 filesystem to RAM
  - And also `ext2ls` to list files, `ext2info` for information
- `fatload`, loads a file from a FAT filesystem to RAM
  - And also `fatls` and `fatinfo`
- `tftp`, loads a file from the network to RAM
- `ping`, to test the network
- `boot`, runs the default boot command, stored in `bootcmd`
- `bootz <address>`, starts a kernel image loaded at the given address in RAM

Important commands

- `loadb`, `loads`, `loady`, load a file from the serial line to RAM
- `usb`, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- `mmc`, to initialize and control the MMC subsystem, used for SD and microSD cards
- `nand`, to erase, read and write contents to NAND flash
- `erase`, `protect`, `cp`, to erase, modify protection and write to NOR flash
- `md`, displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- `mm`, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

Environment variables: principle

- U-Boot can be configured through environment variables
  - Some specific environment variables affect the behavior of the different commands
  - Custom environment variables can be added, and used in scripts
- Environment variables are loaded from flash to RAM at U-Boot startup, can be modified and saved back to flash for persistence
- There is a dedicated location in flash (or in MMC storage) to store the U-Boot environment, defined in the board configuration file

Environment variables commands

Commands to manipulate environment variables:

- `printenv`
  Shows all variables
- `printenv <variable-name>`
  Shows the value of a variable
- `setenv <variable-name> <variable-value>`
  Changes the value of a variable, only in RAM
- `editenv <variable-name>`
  Edits the value of a variable, only in RAM
- `saveenv`
  Saves the current state of the environment to flash

Environment variables commands - Example

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial
u-boot # printenv serverip
serverip=10.0.0.1
u-boot # setenv serverip 10.0.0.100
u-boot # saveenv
```

Important U-Boot env variables

- `bootcmd`, contains the command that U-Boot will automatically execute at boot time after a configurable delay (`bootdelay`), if the process is not interrupted
- `bootargs`, contains the arguments passed to the Linux kernel, covered later
- `serverip`, the IP address of the server that U-Boot will contact for network related commands
- `ipaddr`, the IP address that U-Boot will use
- `netmask`, the network mask to contact the server
- `ethaddr`, the MAC address, can only be set once
- `autostart`, if yes, U-Boot starts automatically an image that has been loaded into memory
- `filesize`, the size of the latest copy to memory (from `tftp`, `fatload`, `nand read`...)

Scripts in environment variables

- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the `;` operator
  - Tests can be done using
    `if command ; then ... ; else ... ; fi`
  - Scripts are executed using `run <variable-name>`
  - You can reference other variables using `${variable-name}`
- Example
  - `setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 zImage; then run mmc-do-boot; fi; fi'`

Transferring files to the target

- U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.
- Files must be exchanged between the target and the development workstation. This is possible:
  - Through the network if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. This is the fastest and most efficient solution.
  - Through a USB key, if U-Boot supports the USB controller of your platform
  - Through a SD or microSD card, if U-Boot supports the MMC controller of your platform
  - Through the serial port

TFTP

- Network transfer from the development workstation to U-Boot on the target takes place through TFTP
  - *Trivial File Transfer Protocol*
  - Somewhat similar to FTP, but without authentication and over UDP
- A TFTP server is needed on the development workstation
  - `sudo apt-get install tftpd-hpa`
  - All files in `/var/lib/tftpboot` are then visible through TFTP
  - A TFTP client is available in the `tftp-hpa` package, for testing
- A TFTP client is integrated into U-Boot
  - Configure the `ipaddr` and `serverip` environment variables
  - Use `tftp <address> <filename>` to load a file

Linux kernel introduction

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

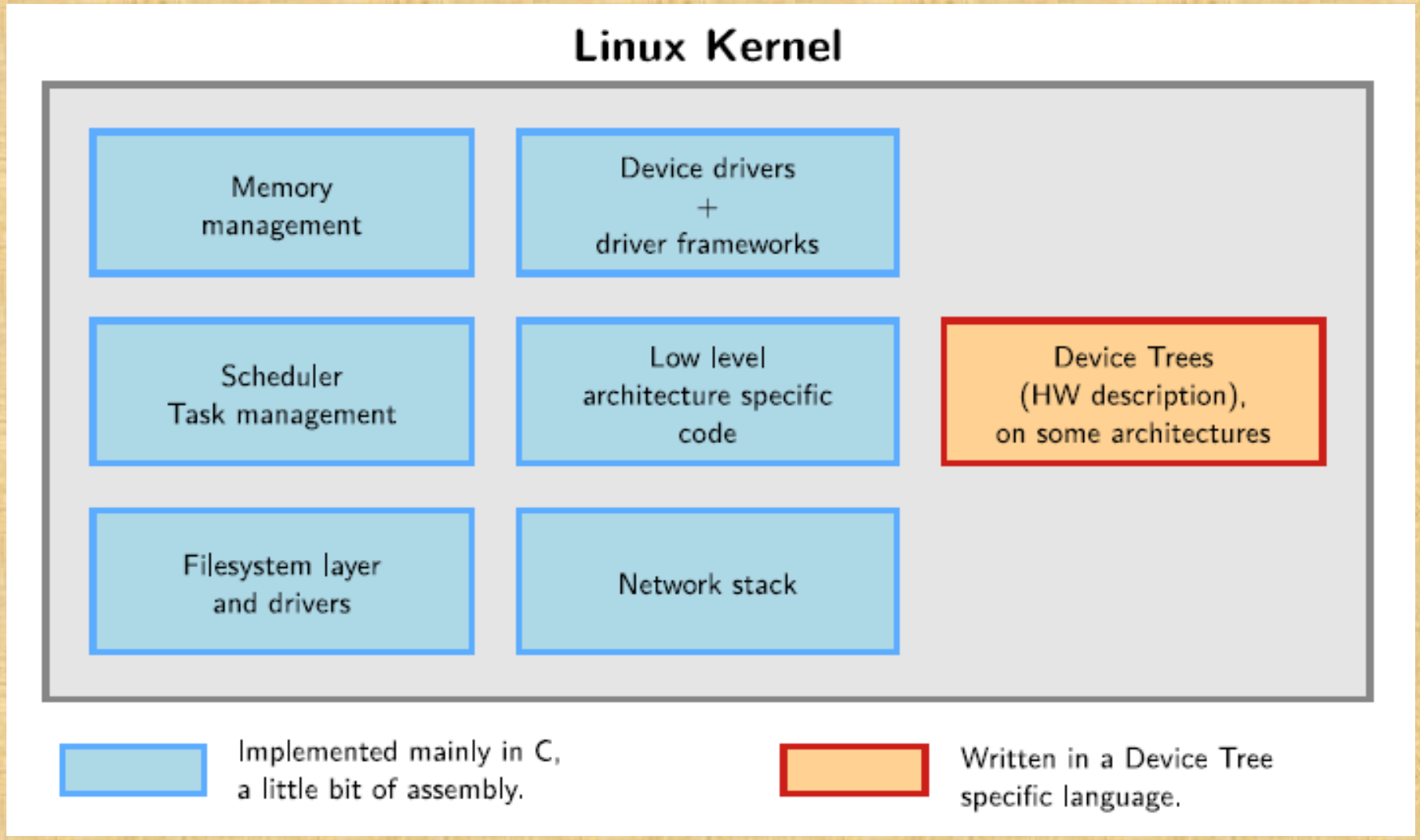
Linux kernel key features

- Portability and hardware support. Runs on most architectures.
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security. It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

Linux kernel main roles

- **Manage all the hardware resources**: CPU, memory, I/O.
- Provide a **set of portable, architecture and hardware independent APIs** to allow user space applications and libraries to use the hardware resources.
- **Handle concurrent accesses and usage** of hardware resources from different applications.
  - Example: a single network interface is used by multiple user space applications through various network connections. The kernel is responsible to ``multiplex'' the hardware resource.

Inside the Linux kernel



Supported hardware architectures

- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples: `arm`, `avr32`, `blackfin`, `c6x`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `alpha`, `arm64`, `ia64`, `tile`
- 32/64 bit architectures
  Examples: `powerpc`, `x86`, `sh`, `sparc`
- Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

Location of kernel sources

- The official versions of the Linux kernel, as released by Linus Torvalds, are available at `http://www.kernel.org`
  - These versions follow the development model of the kernel
  - However, they may not contain the latest development from a specific area yet. Some features in development might not be ready for mainline inclusion yet.
- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Useful only when mainline hasn't caught up yet.
- Many kernel sub-communities maintain their own kernel, with usually newer but less stable features
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - No official releases, only development trees are available.

Getting Linux sources

- The kernel sources are available from `http://kernel.org/pub/linux/kernel` as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- However, more and more people use the `git` version control system. Absolutely needed for kernel development!
  - Fetch the entire kernel sources and history
    `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
  - Create a branch that starts at a specific stable version
    `git checkout -b <name-of-branch> v3.11`
  - Web interface available at `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/`.
  - Read more about Git at `http://git-scm.com/`

Linux kernel size

- Linux 4.6 sources:
  Raw size: 730 MB (53,600 files, approx 21,400,000 lines)
  `gzip` compressed tar archive: 130 MB
  `xz` compressed tar archive: 85 MB
- Minimum Linux 3.17 compiled kernel size, booting on the ARM Versatile board (hard drive on PCI, ext2 filesystem, ELF executable support, framebuffer console and input devices): 876 KB (compressed), 2.3 MB (raw)
- Why are these sources so big?
  Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...
- The Linux core (scheduler, memory management...) is pretty small!

As of kernel version 4.6 (in lines).

- drivers/: 57.0%
- arch/: 16.3%
- fs/: 5.5%
- sound/: 4.4%
- net/: 4.3%
- include/: 3.5%
- Documentation/: 2.8%
- tools/: 1.3%
- kernel/: 1.2%
- firmware/: 0.6%
- lib/: 0.5%
- mm/: 0.5%
- scripts/: 0.4%
- crypto/: 0.4%
- security/: 0.3%
- block/: 0.1%
- ...

Patch

- A patch is the difference between two source trees
  - Computed with the `diff` tool, or with more elaborate version control systems
- They are very common in the open-source community
- Excerpt from a patch:

```
diff -Nru a/Makefile b/Makefile
--- a/Makefile 2005-03-04 09:27:15 -08:00
+++ b/Makefile 2005-03-04 09:27:15 -08:00
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
-EXTRAVERSION =
+EXTRAVERSION = .1
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

Contents of a patch

- One section per modified file, starting with a header

```
diff -Nru a/Makefile b/Makefile
--- a/Makefile 2005-03-04 09:27:15 -08:00
+++ b/Makefile 2005-03-04 09:27:15 -08:00
```

- One sub-section per modified part of the file, starting with header with the affected line numbers

```
@@ -1,7 +1,7 @@
```

- Three lines of context before the change

```
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
```

- The change itself

```
-EXTRAVERSION =
+EXTRAVERSION = .1
```

- Three lines of context after the change

```
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

Using the patch command

The `patch` command:

- Takes the patch contents on its standard input
- Applies the modifications described by the patch into the current directory

`patch` usage examples:

- `patch -p<n> < diff_file`
- `cat diff_file | patch -p<n>`
- `xzcat diff_file.xz | patch -p<n>`
- `bzcat diff_file.bz2 | patch -p<n>`
- `zcat diff_file.gz | patch -p<n>`
- Notes:
  - `n`: number of directory levels to skip in the file paths
  - You can reverse apply a patch with the `-R` option
  - You can test a patch with `--dry-run` option

Applying a Linux patch

- Two types of Linux patches:
  - Either to be applied to the previous stable version (from `3.<x-1>` to `3.x`)
  - Or implementing fixes to the current stable version (from `3.x` to `3.x.y`)
- Can be downloaded in `gzip`, `bzip2` or `xz` (much smaller) compressed files.
- Always produced for `n=1`
  (that's what everybody does... do it too!)
- Need to run the `patch` command inside the kernel source directory
- Linux patch command line example:

```
cd linux-3.9
xzcat ../patch-3.10.xz | patch -p1
xzcat ../patch-3.10.9.xz | patch -p1
cd ..; mv linux-3.9 linux-3.10.9
```

Kernel compilation

- `make`
  - in the main kernel source directory
  - Remember to run multiple jobs in parallel if you have multiple CPU cores. Example: `make -j 4`
  - No need to run as root!
- Generates
  - `vmlinux`, the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted
  - `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
    - `bzImage` for x86, `zImage` for ARM, `vmImage.gz` for Blackfin, etc.
  - `arch/<arch>/boot/dts/*.dtb`, compiled Device Tree files (on some architectures)
  - All kernel modules, spread over the kernel source tree, as `.ko` files.

Kernel installation

- `make install`
  - Does the installation for the host system by default, so needs to be run as root. Generally not used when compiling for an embedded system, as it installs files on the development workstation.
- Installs
  - `/boot/vmlinuz-<version>`
    Compressed kernel image. Same as the one in `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Stores kernel symbol addresses
  - `/boot/config-<version>`
    Kernel configuration for this version
- Typically re-runs the bootloader configuration utility to take the new kernel into account.

Module installation

- `make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs all modules in `/lib/modules/<version>/`
  - `kernel/`
    Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias`
    Module aliases for module loading utilities. Example line:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`, `modules.dep.bin` (binary hashed)
    Module dependencies
  - `modules.symbols`, `modules.symbols.bin` (binary hashed)
    Tells which module a given symbol belongs to.

Kernel cleanup targets

- Clean-up generated files (to force re-compilation):
  `make clean`
- Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your `.config` file!
  `make mrproper`
- Also remove editor backup and patch reject files (mainly to generate patches):
  `make distclean`

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- Much easier as development tools for your GNU/Linux workstation are much easier to find.
- To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:
  `mips-linux-gcc`, the prefix is `mips-linux-`
  `arm-linux-gnueabi-gcc`, the prefix is `arm-linux-gnueabi-`

Specifying cross-compilation

The CPU architecture and cross-compiler prefix are defined through the `ARCH` and `CROSS_COMPILE` variables in the toplevel Makefile.

- `ARCH` is the name of the architecture. It is defined by the name of the subdirectory in `arch/` in the kernel sources
  - Example: `arm` if you want to compile a kernel for the `arm` architecture.
- `CROSS_COMPILE` is the prefix of the cross compilation tools
  - Example: `arm-linux-` if your compiler is `arm-linux-gcc`

Specifying cross-compilation

Two solutions to define `ARCH` and `CROSS_COMPILE`:

- Pass `ARCH` and `CROSS_COMPILE` on the `make` command line:
  `make ARCH=arm CROSS_COMPILE=arm-linux- ...`
  Drawback: it is easy to forget to pass these variables when you run any `make` command, causing your build and configuration to be screwed up.
- Define `ARCH` and `CROSS_COMPILE` as environment variables:
  ```
  export ARCH=arm
  export CROSS_COMPILE=arm-linux-
  ```
  Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your `~/.bashrc` file to make them permanent and visible from any terminal.

Predefined configuration files

- Default configuration files available, per board or per-CPU family
  - They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files
  - This is the most common way of configuring a kernel for embedded platforms
- Run `make help` to find if one is available for your platform
- To load a default configuration file, just run `make acme_defconfig`
  - This will overwrite your existing `.config` file!
- To create your own default configuration file
  - `make savedefconfig`, to create a minimal configuration file
  - `mv defconfig arch/<arch>/configs/myown_defconfig`

Configuring the kernel

- After loading a default configuration file, you can adjust the configuration to your needs with the normal `xconfig`, `gconfig` or `menuconfig` interfaces
- As the architecture is different from your host architecture
  - Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - Many options will be identical (filesystems, network protocols, architecture-independent drivers, etc.)

Building and installing the kernel

- Run `make`
- Copy the final kernel image to the target storage
  - can be `zImage`, `vmlinux`, `bzImage` in `arch/<arch>/boot`
  - copying the Device Tree Blob might be necessary as well, they are available in `arch/<arch>/boot/dts`
- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behaviour in `arch/<arch>/boot/install.sh`
- `make modules_install` is used even in embedded development, as it installs many modules and description files
  - `make INSTALL_MOD_PATH=<dir>/ modules_install`
  - The `INSTALL_MOD_PATH` variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

Booting with U-Boot

- Recent versions of U-Boot can boot the `zImage` binary.
- Older versions require a special kernel image format: `uImage`
  - `uImage` is generated from `zImage` using the `mkimage` tool. It is done automatically by the kernel `make uImage` target.
  - On some ARM platforms, `make uImage` requires passing a `LOADADDR` environment variable, which indicates at which physical memory address the kernel will be executed.
- In addition to the kernel image, U-Boot can also pass a *Device Tree Blob* to the kernel.
- The typical boot process is therefore:
  1. Load `zImage` or `uImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `bootz X - Y` (`zImage` case), or `bootm X - Y` (`uImage` case)
     The `-` in the middle indicates no *initramfs*

Filesystems

- Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- In Unix systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- Filesystems are **mounted** in a specific location in this hierarchy of directories
  - When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflects the contents of the storage device
  - When the filesystem is unmounted, the *mount point* is empty again.
- This allows applications to access files and directories easily, regardless of their exact storage location

Filesystems (2)

- Create a mount point, which is just a directory

```
$ mkdir /mnt/usbkey
```

- It is empty

```
$ ls /mnt/usbkey
$
```

- Mount a storage device in this mount point

```
$ mount -t vfat /dev/sda1 /mnt/usbkey
$
```

- You can access the contents of the USB key

```
$ ls /mnt/usbkey
docs prog.c picture.png movie.avi
$
```

mount / umount

- `mount` allows to mount filesystems
  - `mount -t type device mountpoint`
  - `type` is the type of filesystem
  - `device` is the storage device, or network location to mount
  - `mountpoint` is the directory where files of the storage device or network location will be accessible
  - `mount` with no arguments shows the currently mounted filesystems
- `umount` allows to unmount filesystems
  - This is needed before rebooting, or before unplugging a USB key, because the Linux kernel caches writes in memory to increase performance. `umount` makes sure that these writes are committed to the storage.

Root filesystem

- A particular filesystem is mounted at the root of the hierarchy, identified by `/`
- This filesystem is called the **root filesystem**
- As `mount` and `umount` are programs, they are files inside a filesystem.
  - They are not accessible before mounting at least one filesystem.
- As the root filesystem is the first mounted filesystem, it cannot be mounted with the normal `mount` command
- It is mounted directly by the kernel, according to the `root=` kernel option
- When no root filesystem is available, the kernel panics

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Location of the root filesystem

- It can be mounted from different locations
  - From the partition of a hard disk
  - From the partition of a USB key
  - From the partition of an SD card
  - From the partition of a NAND flash chip or similar type of storage device
  - From the network, using the NFS protocol
  - From memory, using a pre-loaded filesystem (by the bootloader)
  - etc.
- It is up to the system designer to choose the configuration for the system, and configure the kernel behaviour with `root=`

Mounting rootfs from storage devices

- Partitions of a hard disk or USB key
  - `root=/dev/sdXY`, where `X` is a letter indicating the device, and `Y` a number indicating the partition
  - `/dev/sdb2` is the second partition of the second disk drive (either USB key or ATA hard drive)
- Partitions of an SD card
  - `root=/dev/mmcblkXpY`, where `X` is a number indicating the device and `Y` a number indicating the partition
  - `/dev/mmcblk0p2` is the second partition of the first device
- Partitions of flash storage
  - `root=/dev/mtdblockX`, where `X` is the partition number
  - `/dev/mtdblock3` is the fourth partition of a NAND flash chip (if only one NAND flash chip is present)

Mounting rootfs over the network (1)

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- Makes it very easy to update files on the root filesystem, without rebooting. Much faster than through the serial port.
- Can have a big root filesystem even if you don't have support for internal or external storage yet.
- The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).

Mounting rootfs over the network (2)

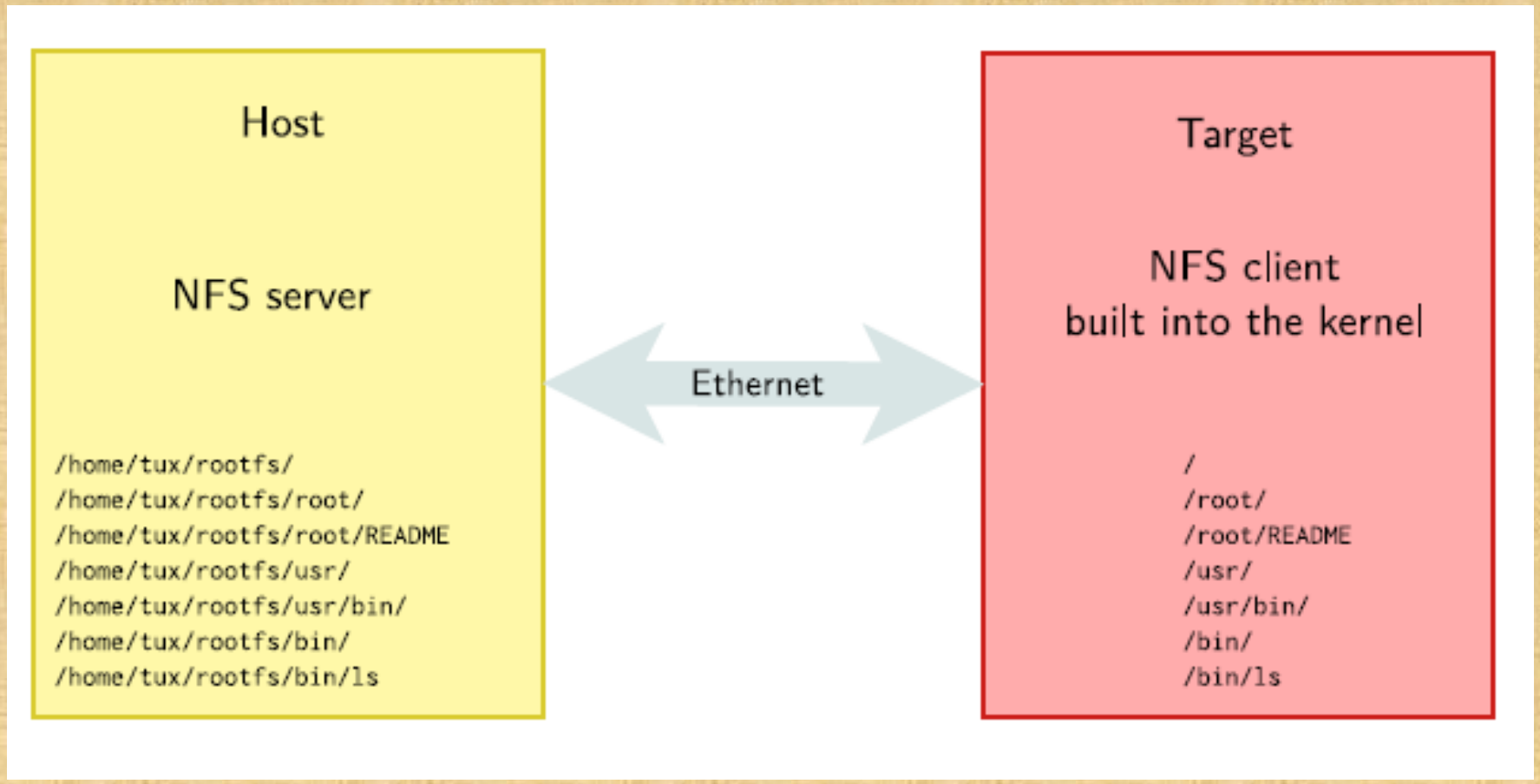
On the development workstation side, a NFS server is needed

- Install an NFS server (example: Debian, Ubuntu)
  `sudo apt-get install nfs-kernel-server`
- Add the exported directory to your `/etc/exports` file:
  `/home/tux/rootfs 192.168.1.111(rw,no_root_squash,no_subtree_check)`
  - `192.168.1.111` is the client IP address
  - `rw,no_root_squash,no_subtree_check` are the NFS server options for this directory export.
- Start or restart your NFS server (example: Debian, Ubuntu)
  `sudo /etc/init.d/nfs-kernel-server restart`

Mounting rootfs over the network (3)

- On the target system
- The kernel must be compiled with
  - `CONFIG_NFS_FS=y` (NFS support)
  - `CONFIG_IP_PNP=y` (configure IP at boot time)
  - `CONFIG_ROOT_NFS=y` (support for NFS as rootfs)
- The kernel must be booted with the following parameters:
  - `root=/dev/nfs` (we want rootfs over NFS)
  - `ip=192.168.1.111` (target IP address)
  - `nfsroot=192.168.1.110:/home/tux/rootfs/` (NFS server details)

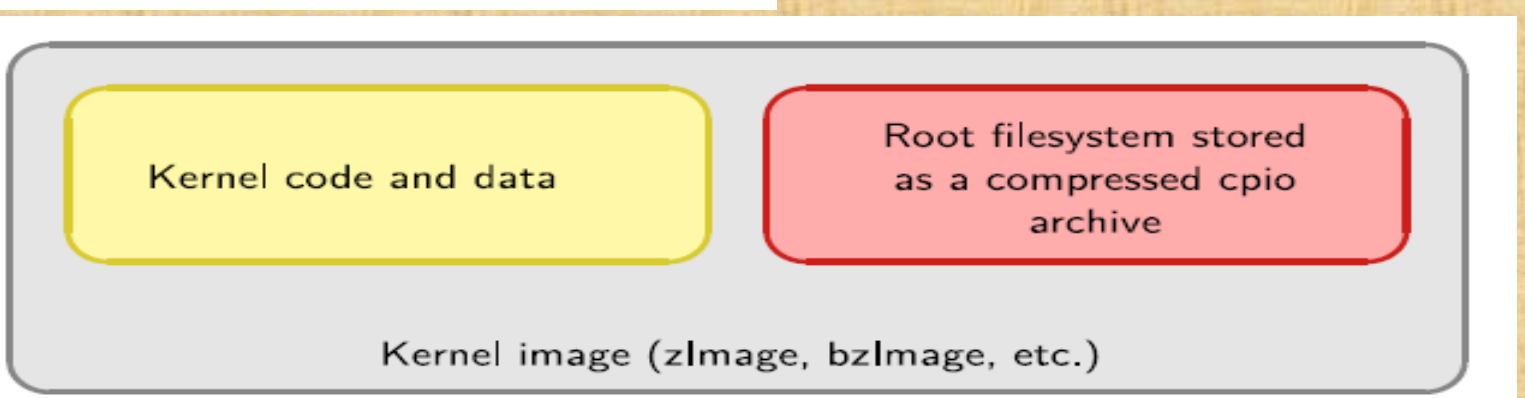
Mounting rootfs over the network (4)



rootfs in memory: initramfs (1)

- It is also possible to have the root filesystem integrated into the kernel image
- It is therefore loaded into memory together with the kernel
- This mechanism is called **initramfs**
  - It integrates a compressed archive of the filesystem into the kernel image
  - Variant: the compressed archive can also be loaded separately by the bootloader.
- It is useful for two cases
  - Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
  - As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.

rootfs in memory: initramfs (2)



- The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
  - Can be the path to a directory containing the root filesystem contents
  - Can be the path to a cpio archive
  - Can be a text file describing the contents of the initramfs (see documentation for details)
- The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- Details (in kernel sources):
  `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

Root filesystem organization

- The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- `http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs`
- Most Linux systems conform to this specification
  - Applications expect this organization
  - It makes it easier for developers and users as the filesystem organization is similar in all systems

Filesystems

The Linux kernel supports a wide-range of filesystem types:

- **ext2, ext3, ext4** are the default filesystem types for Linux. They are usable on block devices.
- **jffs2, ubifs** are the filesystems usable on flash devices (NAND, NOR, SPI flashes). Note that SD/MMC cards or USB keys are not flash devices, but block devices.
- **squashfs** is a read-only highly-compressed filesystem, appropriate for all system files that never change.
- **vfat, nfts**, the Windows-world filesystems, are also supported for compatibility
- **nfs, cifs** are the two most important network filesystems supported

Important directories

/bin	Basic programs
/boot	Kernel image (only when the kernel is loaded from a filesystem, not common on non-x86 architectures)
/dev	Device files (covered later)
/etc	System-wide configuration
/home	Directory for the users home directories
/lib	Basic libraries
/media	Mount points for removable media
/mnt	Mount points for static media
/proc	Mount point for the proc virtual filesystem

/root Home directory of the root user

/sbin Basic system programs

/sys Mount point of the sysfs virtual filesystem

/tmp Temporary files

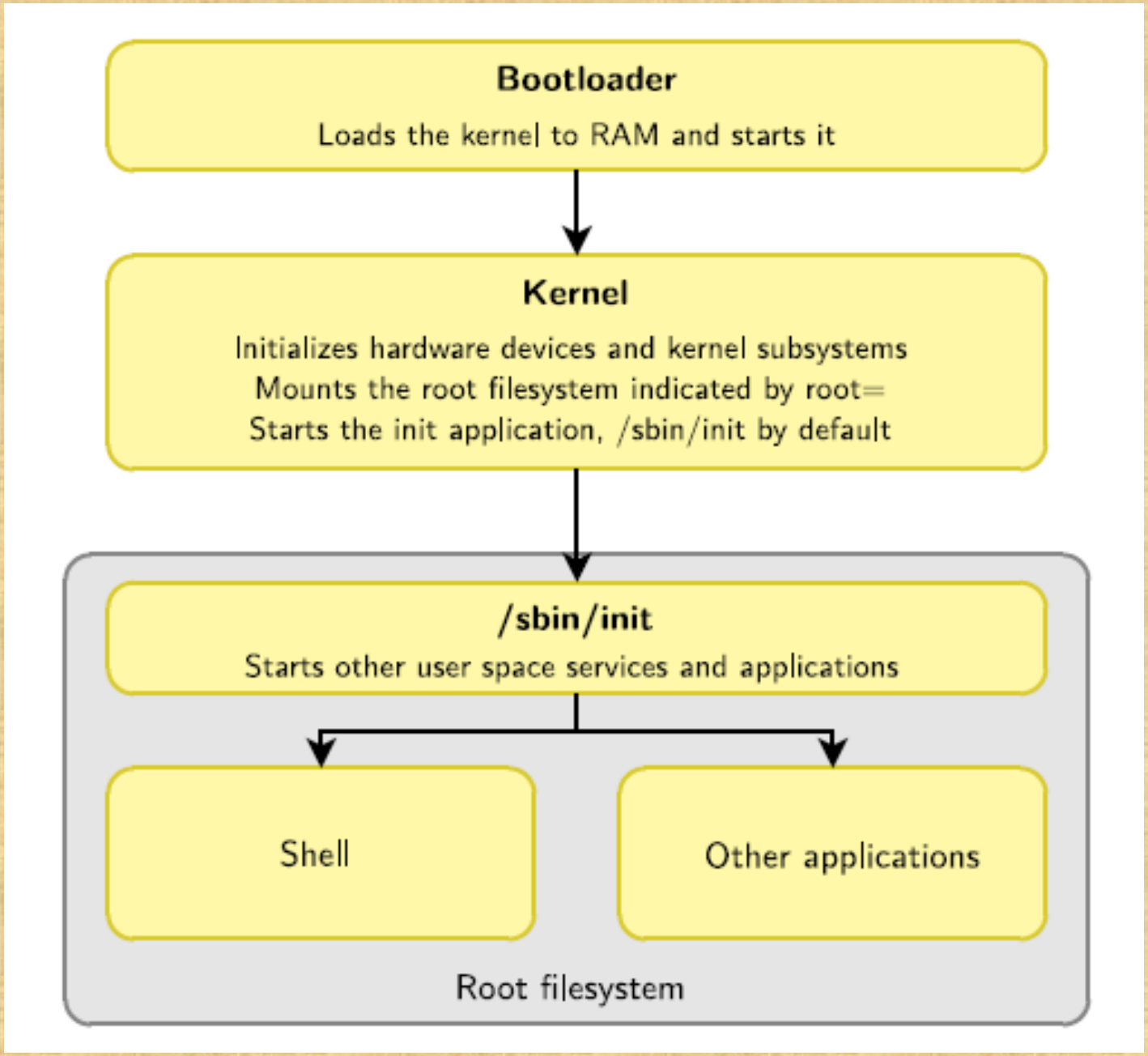
/usr
- /usr/bin Non-basic programs
- /usr/lib Non-basic libraries
- /usr/sbin Non-basic system programs

/var Variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files

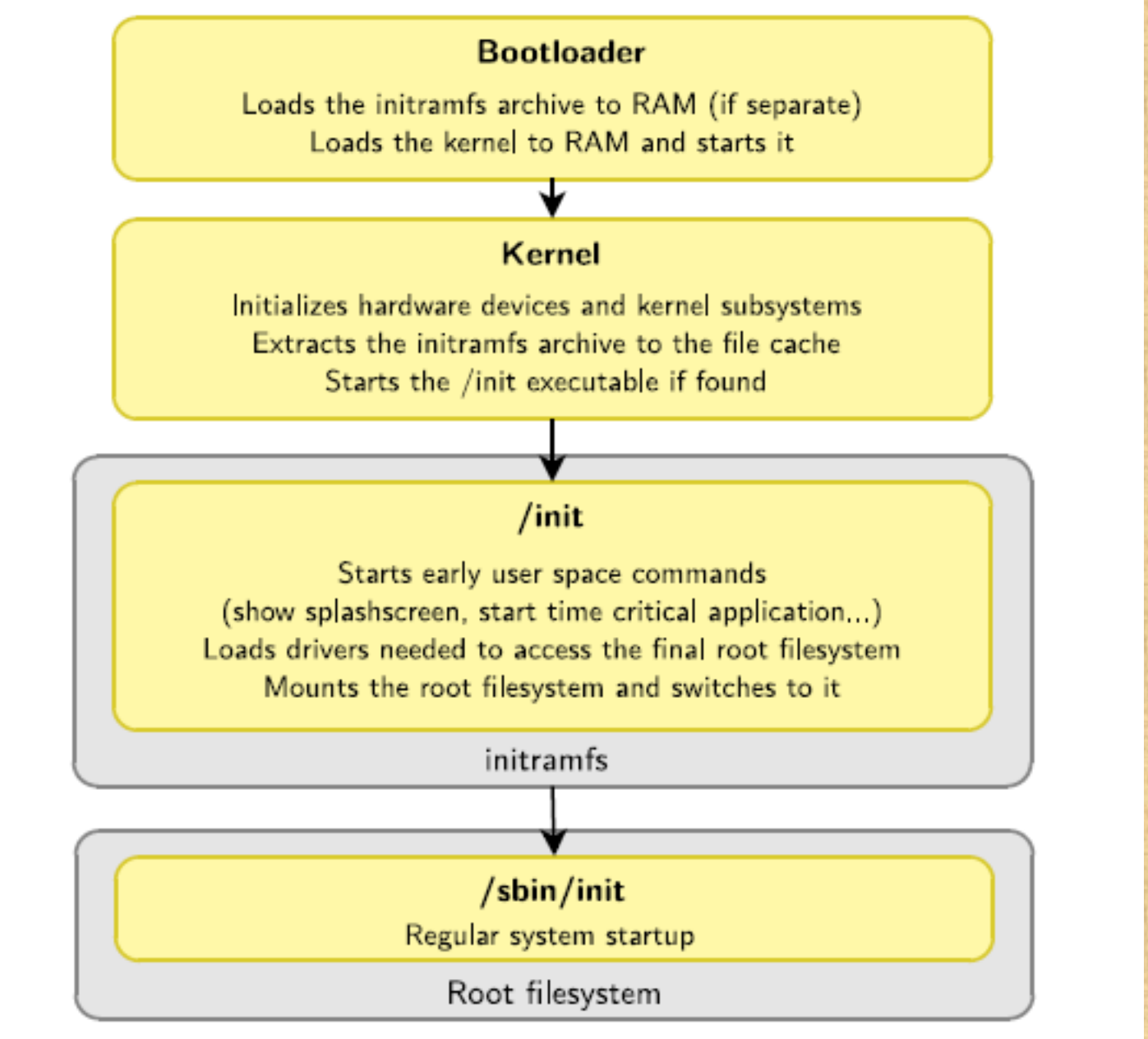
Separation of programs and libraries

- Basic programs are installed in `/bin` and `/sbin` and basic libraries in `/lib`
- All other programs are installed in `/usr/bin` and `/usr/sbin` and all other libraries in `/usr/lib`
- In the past, on Unix systems, `/usr` was very often mounted over the network, through NFS
- In order to allow the system to boot when the network was down, some binaries and libraries are stored in `/bin`, `/sbin` and `/lib`
- `/bin` and `/sbin` contain programs like `ls`, `ifconfig`, `cp`, `bash`, etc.
- `/lib` contains the C library and sometimes a few other basic libraries
- All other programs and libraries are in `/usr`

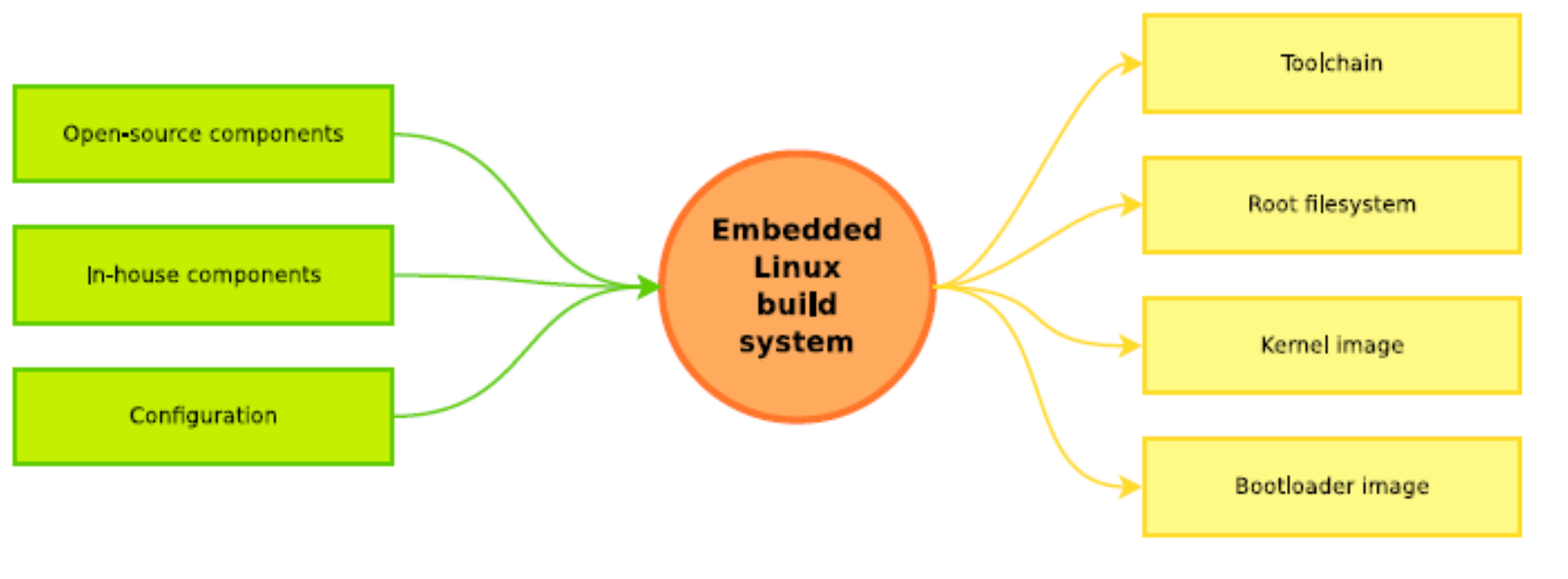
Overall booting process



Overall booting process with initramfs



Embedded Linux build system: principle



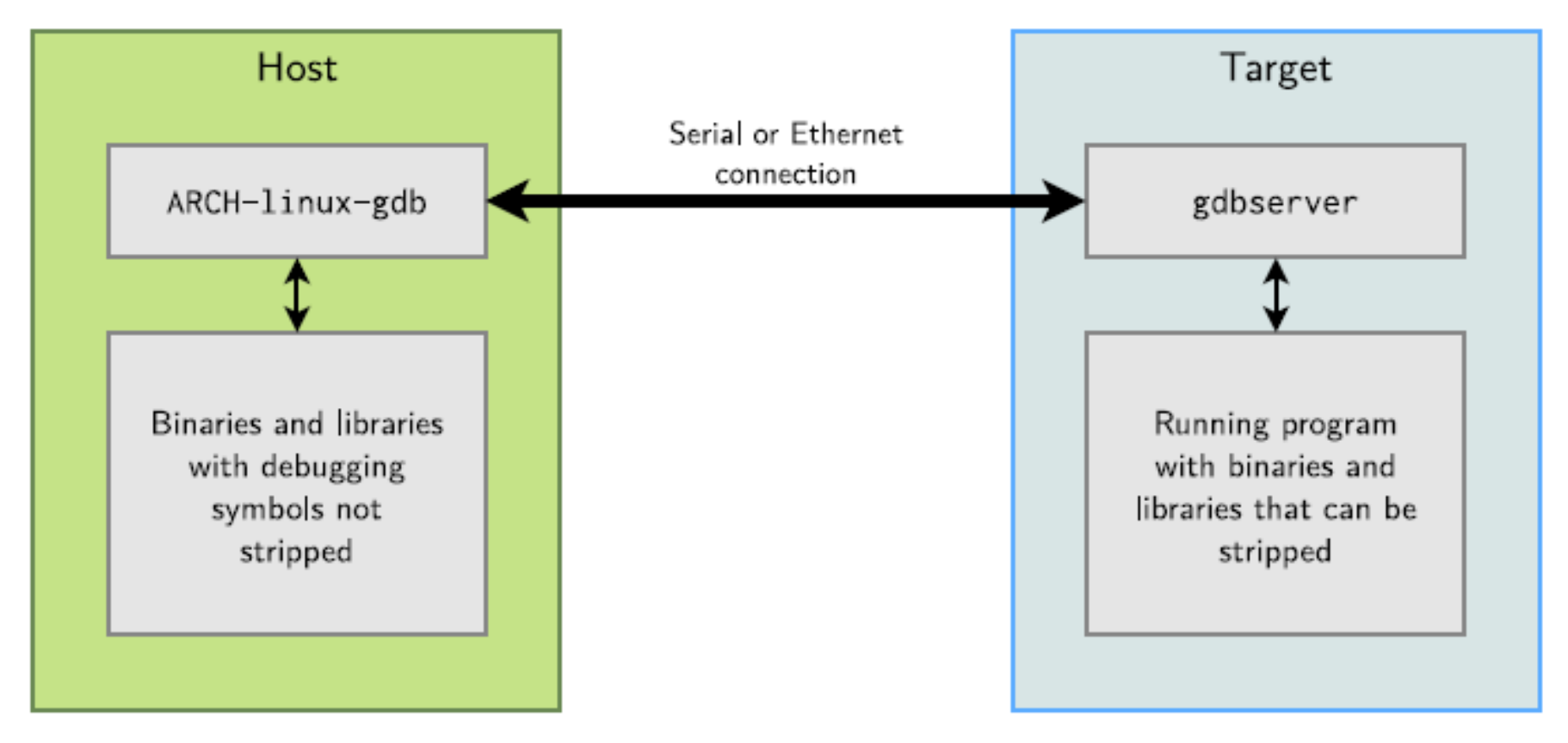
Some common open-source build systems:

- Buildroot, `http://www.buildroot.org`
- OpenEmbedded, `http://www.openembedded.org`
- Yocto, `http://www.yoctoproject.org`
- and others: PTXdist, OpenBricks, OpenWRT, etc.

Remote debugging

- In a non-embedded environment, debugging takes place using gdb or one of its front-ends.
- gdb has direct access to the binary and libraries compiled with debugging symbols.
- However, in an embedded context, the target platform environment is often too limited to allow direct debugging with gdb (2.4 MB on x86).
- Remote debugging is preferred
  - gdb is used on the development workstation, offering all its features.
  - gdbserver is used on the target system (only 100 KB on arm).

Remote debugging: architecture



Remote debugging: usage

- On the target, run a program through `gdbserver`.
  Program execution will not start immediately.
  `gdbserver localhost:<port> <executable> <args>`
  `gdbserver /dev/ttyS0 <executable> <args>`
- Otherwise, attach `gdbserver` to an already running program:
  `gdbserver --attach localhost:<port> <pid>`
- Then, on the host, run the `ARCH-linux-gdb` program, and use the following `gdb` commands:
  - To connect to the target:
    `gdb> target remote <ip-addr>:<port>` (networking)
    `gdb> target remote /dev/ttyS0` (serial link)
  - To tell `gdb` where shared libraries are:
    `gdb> set sysroot <library-path>` (without `lib/`)

Post mortem analysis

- When an application crashes due to a *segmentation fault* and the application was not under control of a debugger, we get no information about the crash
- Fortunately, Linux can generate a *core* file that contains the image of the application memory at the moment of the crash, and gdb can use this *core* file to let us analyze the state of the crashed application
- On the target
  - Use `ulimit -c unlimited` to enable the generation of a *core* file when a crash occurs
- On the host
  - After the crash, transfer the *core* file from the target to the host, and run
    `ARCH-linux-gdb -c core-file application-binary`

Thank you