Embedded Linux system development

Endocode AG

Latest update: May 10, 2016.

Document updates and sources:
`http://free-electrons.com/doc/training/embedded-linux`

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@free-electrons.com

Rights to copy

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  `http://kernel.org/`
- Kernel documentation links:
  `Documentation/kmemcheck.txt`
- Links to kernel source files and directories:
  `drivers/input`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

Endocode at a glance

- An employee-owned, software engineering company from Berlin
- We do: Application Engineering, Platform Engineering, Open Source Compliance & Guidance
- Head count: 15 (from 7 countries)
- Recent activities:
  - fleet development `https://github.com/coreos/fleet`
  - MultipathTCP in connman `https://endocode.com/blog/2016/03/22/multipathtcp-support-for-connman/`
  - Immmr to the Cloud `http://coreosfest2016.sched.org/event/6T0b`

Generic course information

Endocode AG

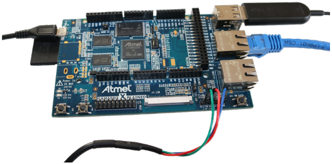
Embedded Linux Experts

Corrections, suggestions, contributions and translations are welcome!

Hardware used in this training session

Using Atmel SAMA5D3 Xplained boards in all practical labs

- SAMA5D36 (Cortex A5) CPU from Atmel
- USB powered!
- 256 MB DDR2 RAM, 256 MB NAND flash
- 2 Ethernet ports (Gigabit + 100 Mbit)
- 2 USB 2.0 host, 1 USB device
- 1 MMC/SD slot
- 3.3 V serial port (like Beaglebone Black)
- Misc: Arduino R3-compatible header, JTAG, buttons, LEDs
- Currently sold at 69 EUR by Mouser (V.A.T. not included)



Board and CPU documentation, design files, software:
`http://www.atmel.com/sama5d3xplained`

Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux / Android with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

Practical lab guidelines

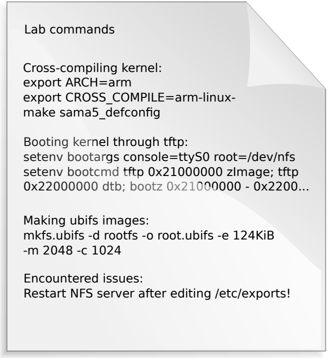
During practical labs...

- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't hesitate to copy and paste commands from the PDF slides and labs.

Advise: write down your commands!

During practical labs, write down all your commands in a text file.

- You can save a lot of time re-using commands in later labs.
- This helps to replay your work if you make significant mistakes.
- You build a reference to remember commands in the long run.
- That's particular useful to keep kernel command line settings that you used earlier.
- Also useful to get help from the instructor, showing the commands that you run.



Lab commands

Cross-compiling kernel:
export ARCH=arm
export CROSS_COMPILE=arm-linux-
make sama5_defconfig

Booting kernel through tftp:
setenv bootargs console=ttyS0 root=/dev/nfs
setenv bootcmd tftp 0x21000000 zImage; tftp 0x22000000 dtb; bootz 0x21000000 - 0x2200...

Making ubifs images:
mkfs.ubifs -d rootfs -o root.ubifs -e 124KiB -m 2048 -c 1024

Encountered issues:
Restart NFS server after editing /etc/exports!

```
gedit ~/lab-history.txt
```

Cooperate!

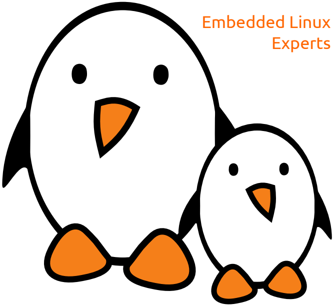
As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

Introduction into Linux

Endocode AG

Corrections, suggestions, contributions and translations are welcome!



What is Linux?

- An Unix-like and mostly POSIX-compliant operating system
- More specific: an operating system kernel
- Linux distribution: kernel bundled with software (GNU)

Short history of Linux

- 1991 started by Linux Torvalds as a hobby project
- 1992 version 0.99 released under GPL
- 1993 first Linux distribution released: Slackware
- 1993 first release of Debian GNU/Linux distribution
- 1995, Linux is more and more popular on server systems
- 2000, Linux is more and more popular on **embedded systems**
- 2008, Linux is more and more popular on mobile devices
- 2010, Linux is more and more popular on phones

Desktop Linux distribution

- **Any good and sufficiently recent Linux desktop distribution** can be used for the development workstation
  - Ubuntu, Debian, Fedora, openSUSE, Red Hat, etc.
- We recommend Ubuntu, as it is a **widely used and easy to use** desktop Linux distribution
- I personally prefer Debian GNU/Linux

Linux filesystem structure (1)

Defined by the Filesystem Hierarchy Standard :
`http://www.pathname.com/fhs/`

/	Toplevel directory (root of the filesystem)
/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

Linux filesystem structure (2)

/root	Home directory of the `root` user
/sbin	Basic system programs
/sys	Mount point of the sysfs virtual filesystem
/tmp	Temporary files
/usr	"Unix system resources"
/usr/bin	Non-basic programs
/usr/lib	Non-basic libraries
/usr/local	Software installed by the sysadmin
/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

Everything is a file

Almost everything in Linux is a file.

Regular files

Directories
: Directories are files listing a set of files

Symbolic links
: File referring to the name of another file

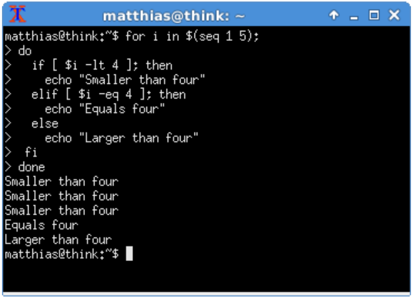
Devices
: Read and Write from devices as from regular files

Pipes
: Cascade programs `cat *.log | grep err`

Sockets
: Inter process communication

Command line interpreters

- Shell: tool to execute user commands
- Commands are typed into a text terminal
- Results are also displayed on the text terminal
- Shells can be scripted (variables, conditionals, iterations...)



Shell

- sh: The Bourne Shell (traditional Unix shell)
- bash: Bourne Again SHell (improved implementation of sh)
- zsh: "extended Bourne Shell"
- csh: C Shell (c-like syntax)
- mksh: MirBSD Korn Shell (fork ksh, now default on Android)

Command line usage

Please download and use this cheat sheet:

http://free-electrons.com/doc/command_memento_de.pdf

Linux root and non-root users

- Linux is a multi-user operating system
  - The **root user is the administrator**, and it can do privileged operations such as: mounting filesystems, configuring the network, creating device files, changing the system configuration, installing or removing software
  - All **other users are unprivileged**, and cannot perform these administrator-level operations
- On an Ubuntu system, it is not possible to log in as `root`, only as a normal user.
- The system has been configured so that the user account created first is allowed to run privileged operations through a program called `sudo`.
  - Example: `sudo mount /dev/sda2 /mnt/disk`

Run commands as another user

Don't work as root all the time. It's dangerous.
Better use `sudo`.

- `sudo -l` Show what commands are allowed
- `sudo less /var/log/syslog` Read a logfile

File ownership and access rights

```
ls -l /bin/bash
-rwxr-xr-x 1 root root 1029624 Nov 13 2014 /bin/bash
```

three types of access rights

- **r** Read access
- **w** Write access
- **x** Execute rights

three types of access levels

- **u** User: owner of the file
- **g** Group: list of users
- **o** Other: all other users

Maintain ownership and permissions

Maintain permissions of files

`chmod -w file` make the file readonly for all

`chmod u+w file` make the file writeable for the owner

`chmod +x file` make the file executeable for all

`chmod go-w file` make the file readonly for group and others

Maintain ownership of files

`chown matthias file` Set the owner of the file to matthias

`chown matthias.dialout file` Set the owner of the file to matthias and the group to dialout

`chgrp dialout file` Set the group of the file to dialout

Software packages

- The distribution mechanism for software in GNU/Linux is different from the one in Windows
- Linux distributions provides a central and coherent way of installing, updating and removing applications and libraries: **packages**
- Packages contains the application or library files, and associated meta-information, such as the version and the dependencies
  - `.deb` on Debian and Ubuntu, `.rpm` on Red Hat, Fedora, openSUSE
- Packages are stored in **repositories**, usually on HTTP or FTP servers
- You should only use packages from official repositories for your distribution, unless strictly required.

Managing software packages (1)

Instructions for Debian based GNU/Linux systems (Debian, Ubuntu...)

- Package repositories are specified in `/etc/apt/sources.list`
- To update package repository lists:
  `sudo apt-get update`
- To find the name of a package to install, the best is to use the search engine on http://packages.debian.org or on http://packages.ubuntu.com. You may also use:
  `apt-cache search <keyword>`

Managing software packages (2)

- To install a given package:
  `sudo apt-get install <package>`
- To remove a given package:
  `sudo apt-get remove <package>`
- To install all available package updates:
  `sudo apt-get dist-upgrade`
- Get information about a package:
  `apt-cache show <package>`
- Graphical interfaces
  - Synaptic for GNOME
  - KPackageKit for KDE

Further details on package management:
`http://www.debian.org/doc/manuals/apt-howto/`

Introduction to Embedded Linux

Endocode AG

Embedded Linux
Experts

Birth of free software

- 1983, Richard Stallman, **GNU project** and the **free software** concept. Beginning of the development of *gcc*, *gdb*, *glibc* and other important tools
- 1991, Linus Torvalds, **Linux kernel project**, a Unix-like operating system kernel. Together with GNU software and many other open-source components: a completely free operating system, GNU/Linux
- 1995, Linux is more and more popular on server systems
- 2000, Linux is more and more popular on **embedded systems**
- 2008, Linux is more and more popular on mobile devices
- 2010, Linux is more and more popular on phones

Free software?

- A program is considered **free** when its license offers to all its users the following **four** freedoms
  - Freedom to run the software for any purpose
  - Freedom to study the software and to change it
  - Freedom to redistribute copies
  - Freedom to distribute copies of modified versions
- These freedoms are granted for both commercial and non-commercial use
- They imply the availability of source code, software can be modified and distributed to customers
- **Good match for embedded systems!**

Embedded Linux is the usage of the **Linux kernel** and various **open-source** components in embedded systems

Advantages of Linux and open-source for embedded systems

Re-using components

- The key advantage of Linux and open-source in embedded systems is the **ability** to re-use components
- The open-source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
- As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.
- Allows to quickly design and develop complicated products, based on existing components.
- No-one should re-develop yet another operating system kernel, TCP/IP stack, USB stack or another graphical toolkit library.
- **Allows to focus on the added value of your product.**

Low cost

- Free software can be duplicated on as many devices as you want, free of charge.
- If your embedded system uses only free software, you can reduce the cost of software licenses to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
- **Allows to have a higher budget for the hardware or to increase the company's skills and knowledge**

Full control

- With open-source, you have the source code for all components in your system
- Allows unlimited modifications, changes, tuning, debugging, optimization, for an unlimited period of time
- Without lock-in or dependency from a third-party vendor
  - To be true, non open-source components must be avoided when the system is designed and developed
- **Allows to have full control over the software part of your system**

Quality

- Many open-source components are widely used, on millions of systems
- Usually higher quality than what an in-house development can produce, or even proprietary vendors
- Of course, not all open-source components are of good quality, but most of the widely-used ones are.
- **Allows to design your system with high-quality components at the foundations**

Eases testing of new features

- Open-source being freely available, it is easy to get a piece of software and evaluate it
- Allows to easily study several options while making a choice
- Much easier than purchasing and demonstration procedures needed with most proprietary products
- **Allows to easily explore new possibilities and solutions**

Community support

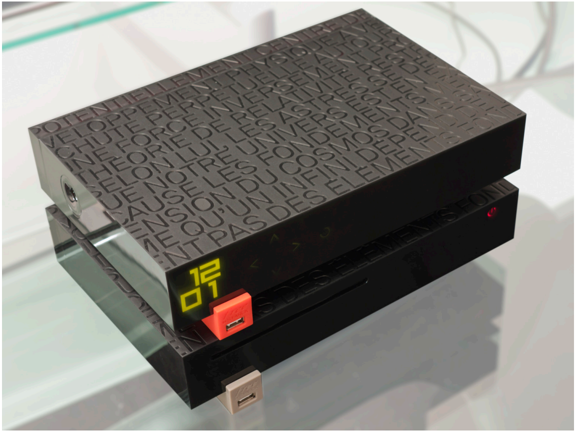
- Open-source software components are developed by communities of developers and users
- This community can provide a high-quality support: you can directly contact the main developers of the component you are using. The likelyhood of getting an answer doesn't depend what company you work for.
- Often better than traditional support, but one needs to understand how the community works to properly use the community support possibilities
- **Allows to speed up the resolution of problems when developing your system**

Taking part into the community

- Possibility of taking part into the development community of some of the components used in the embedded systems: bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.
- Most of the time the open-source components are not the core value of the product: it's the interest of everybody to contribute back.
- For the *engineers*: a very **motivating** way of being recognized outside the company, communication with others in the same field, **opening of new possibilities**, etc.
- For the *managers*: **motivation factor** for engineers, allows the company to be **recognized** in the open-source community and therefore get support more easily and be **more attractive** to open-source developers

A few examples of embedded systems running Linux

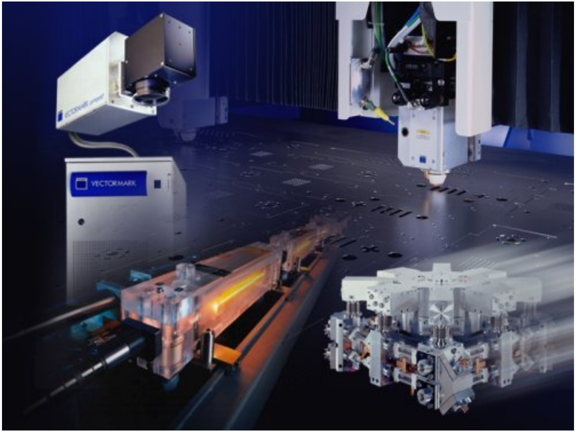
Personal routers



Television

Point of sale terminal

Laser cutting machine



Viticulture machine

Police patrol car

Embedded hardware for Linux systems

Processor and architecture (1)

The Linux kernel and most other architecture-dependent components support a wide range of 32 and 64 bits architectures

- x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
- ARM, with hundreds of different SoC (multimedia, industrial)
- PowerPC (mainly real-time, industrial applications)
- MIPS (mainly networking applications)
- SuperH (mainly set top box and multimedia applications)
- Blackfin (DSP architecture)
- Microblaze (soft-core for Xilinx FPGA)
- Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

Processor and architecture (2)

- Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- Linux is not designed for small microcontrollers.
- Besides the toolchain, the bootloader and the kernel, all other components are generally **architecture-independent**

RAM and storage

- **RAM**: a very basic Linux system can work within 8 MB of RAM, but a more realistic system will usually require at least 32 MB of RAM. Depends on the type and size of applications.
- **Storage**: a very basic Linux system can work within 4 MB of storage, but usually more is needed.
  - Flash storage is supported, both NAND and NOR flash, with specific filesystems
  - Block storage including SD/MMC cards and eMMC is supported
- Not necessarily interesting to be too restrictive on the amount of RAM/storage: having flexibility at this level allows to re-use as many existing components as possible.

Communication

- The Linux kernel has support for many common communication buses
  - I2C
  - SPI
  - CAN
  - 1-wire
  - SDIO
  - USB
- And also extensive networking support
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

Types of hardware platforms

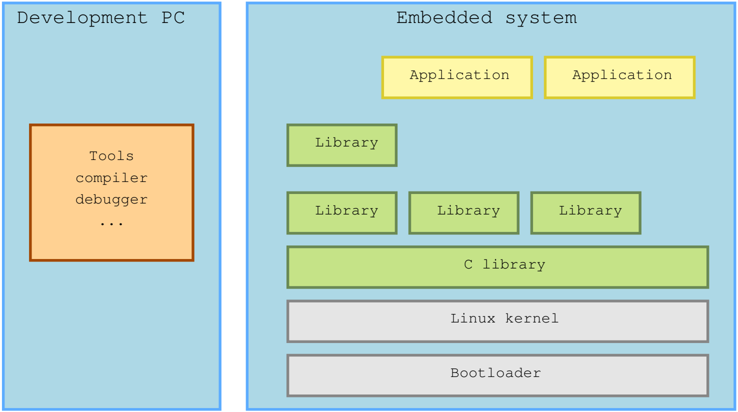
- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module**, a small board with only CPU/RAM/flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Community development platforms**, to make a particular SoC popular and easily available. These are ready-to-use and low cost, but usually have less peripherals than evaluation platforms. To some extent, can also be used for real products.
- **Custom platform**. Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms.

Criteria for choosing the hardware

- Make sure the hardware you plan to use is already supported by the Linux kernel, and has an open-source bootloader, especially the SoC you're targeting.
- Having support in the official versions of the projects (kernel, bootloader) is a lot better: quality is better, and new versions are available.
- Some SoC vendors and/or board vendors do not contribute their changes back to the mainline Linux kernel. Ask them to do so, or use another product if you can. A good measurement is to see the delta between their kernel and the official one.
- **Between properly supported hardware in the official Linux kernel and poorly-supported hardware, there will be huge differences in development time and cost.**

Embedded Linux system architecture

Global architecture



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

Embedded Linux work

Several distinct tasks are needed when deploying embedded Linux in a product:

- **Board Support Package development**
  - A BSP contains a bootloader and kernel with the suitable device drivers for the targeted hardware
- **System integration**
  - Integrate all the components, bootloader, kernel, third-party libraries and applications and in-house applications into a working system
  - Purpose of *this* training
- **Development of applications**
  - Normal Linux applications, but using specifically chosen libraries

Embedded Linux development environment

Endocode AG

Embedded Linux Experts

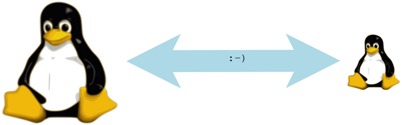
Corrections, suggestions, contributions and translations are welcome!

Embedded Linux solutions

- Two ways to switch to embedded Linux
  - Use **solutions provided and supported by vendors** like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
  - Use **community solutions**. They are completely open, supported by the community.
- In this training sessions, we do not promote a particular vendor, and therefore use community solutions
  - However, knowing the concepts, switching to vendor solutions will be easy

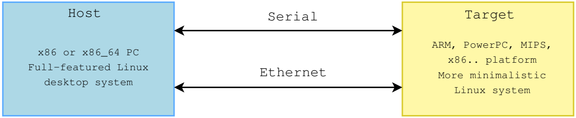
OS for Linux development

- We strongly recommend to use Linux as the desktop operating system to embedded Linux developers, for multiple reasons.
- All community tools are developed and designed to run on Linux. Trying to use them on other operating systems (Windows, Mac OS X) will lead to trouble, and their usage on these systems is generally not supported by community developers.
- As Linux also runs on the embedded device, all the knowledge gained from using Linux on the desktop will apply similarly to the embedded device.



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



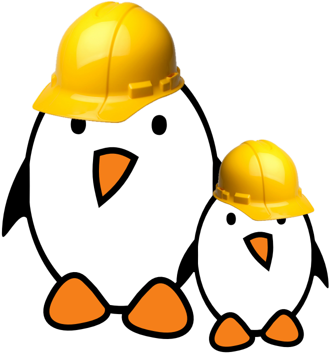
Serial line communication program

- 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

Command line tips

- Using the command line is mandatory for many operations needed for embedded Linux development
- It is a very powerful way of interacting with the system, with which you can save a lot of time.
- Some useful tips
  - You can use several tabs in the Gnome Terminal
  - Remember that you can use relative paths (for example: `../../linux`) in addition to absolute paths (for example: `/home/user`)
  - In a shell, hit `[Control] [r]`, then a keyword, will search through the command history. Hit `[Control] [r]` again to search backwards in the history
  - You can copy/paste paths directly from the file manager to the terminal by drag-and-drop.

Practical lab - Training Setup

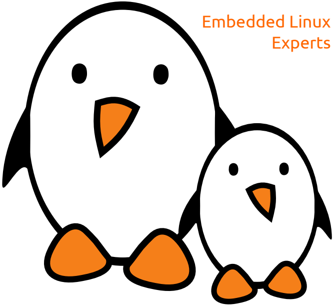


Prepare your lab environment

- Download the lab archive
- Enforce correct permissions

Cross-compiling toolchains

Endocode AG

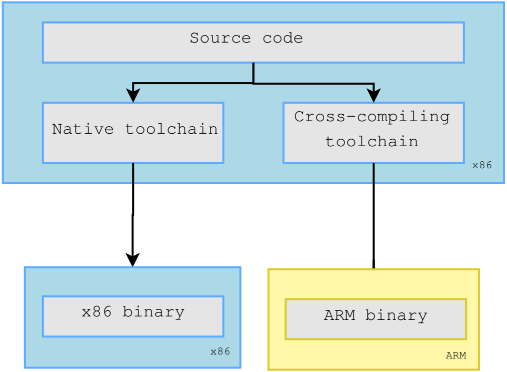


Definition and Components

Definition (1)

- 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 or not interesting to use a native toolchain
  - The target is too restricted in terms of storage and/or memory
  - 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.

Definition (2)



Compilation machine

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

Build | Host | Target

Native build

used to build the normal gcc
of a workstation

Build | Host | Target

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

Build | Host | Target

Cross-native build

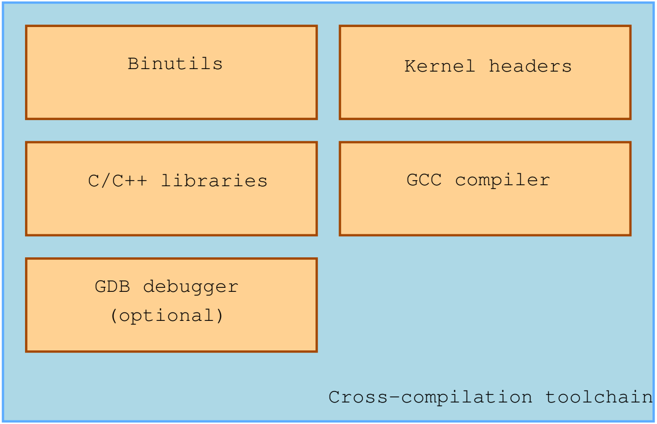
used to build a toolchain that runs on your
arget and generates binaries for the target

Build | Host | Target

Canadian build

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

Components

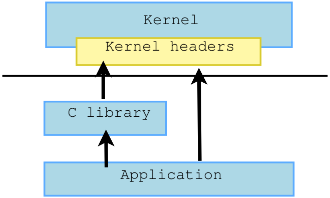

Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger (optional)
Cross-compilation toolchain

Binutils

- **Binutils** is a set of tools to generate and manipulate binaries for a given CPU architecture
  - `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
- `http://www.gnu.org/software/binutils/`
- GPL license

Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in `<linux/...>` and `<asm/...>` and a few other directories corresponding to the ones visible in `include/` in the kernel sources



Kernel headers (2)

- System call numbers, in `<asm/unistd.h>`

```
#define __NR_exit        1
#define __NR_fork        2
#define __NR_read        3
```

- Constant definitions, here in `<asm-generic/fcntl.h>`, included from `<asm/fcntl.h>`, included from `<linux/fcntl.h>`

```
#define O_RDWR 00000002
```

- Data structures, here in `<asm/stat.h>`

```
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    [...]
};
```

Kernel headers (3)

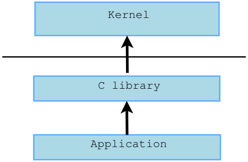
- The kernel to user space ABI is **backward compatible**
  - Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
  - Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don't use the recent features, otherwise they will break
  - Using the latest kernel headers is not necessary, unless access to the new kernel features is needed
- The kernel headers are extracted from the kernel sources using the `headers_install` kernel Makefile target.

GCC

- GNU Compiler Collection, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86_64, IA64, Xtensa, etc.
- `http://gcc.gnu.org/`
- Available under the GPL license, libraries under the LGPL.

C library

- The C library is an essential component of a Linux system
  - Interface between the applications and the kernel
  - Provides the well-known standard C API to ease application development
- Several C libraries are available: *glibc*, *uClibc*, *musl*, *dietlibc*, *newlib*, etc.
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.



C Libraries

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- By default, quite big for small embedded systems: approx 2.5 MB on ARM (version 2.9 - `libc`: 1.5 MB, `libm`: 750 KB)
- But some features not needed in embedded systems can be configured out (merged from the old *eglibc* project).
- `http://www.gnu.org/software/libc/`

uClibc-ng (1)

- `http://uclibc-ng.org/`
- A continuation of the old uClibc project
- License: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface
  - Supports most embedded architectures
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Focus on size rather than performance
  - Small compile time

uClibc-ng (2)

- Most of the applications compile with uClibc-ng. This applies to all applications used in embedded systems.
- Size (arm): 4 times smaller than glibc!
  - uClibc 0.9.30.1: approx. 600 KB (libuClibc: 460 KB, libm: 96KB)
  - glibc 2.9: approx 2.5 MB
- Some features not available or limited: priority-inheritance mutexes, NPTL support is relatively recent, fixed Name Service Switch functionality, etc.
- Used on a large number of production embedded products, including consumer electronic devices

Honey, I shrunk the programs!

- Executable size comparison on ARM, tested with *eglibc* (now *glibc*) 2.15 and *uClibc* 0.9.33.2
- Plain "hello world" program (stripped):

helloworld	static	dynamic
uClibc	18kB	2.5kB
uClibc with Thumb-2	14kB	2.4kB
eglibc with Thumb-2	361kB	2.7kB

- Busybox (stripped):

busybox	static	dynamic
uClibc	750kB	603kB
uClibc with Thumb-2	533kB	439kB
eglibc with Thumb-2	934kB	444kB

musl C library

http://www.musl-libc.org/

- A lightweight, fast and simple library for embedded systems
- Created while uClibc's development was stalled
- In particular, great at making small static executables
- Permissive license (MIT)
- Compare features with other C libraries:
  http://www.etalabs.net/compare_libcs.html
- Supported by build systems such as Buildroot

Other smaller C libraries

- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- Choices:
  - Dietlibc, `http://fefe.de/dietlibc/`. Approximately 70 KB.
  - Newlib, `http://sourceware.org/newlib/`
  - Klibc, `http://www.kernel.org/pub/linux/libs/klibc/`, designed for use in an *initramfs* or *initrd* at boot time.

Toolchain Options

ABI

- When building a toolchain, the ABI used to generate binaries needs to be defined
- ABI, for *Application Binary Interface*, defines the calling conventions (how function arguments are passed, how the return value is passed, how system calls are made) and the organization of structures (alignment, etc.)
- All binaries in a system must be compiled with the same ABI, and the kernel must understand this ABI.
- On ARM, two main ABIs: *OABI* and *EABI*
  - Nowadays everybody uses *EABI*
- On MIPS, several ABIs: *o32, o64, n32, n64*
- `http://en.wikipedia.org/wiki/Application_Binary_Interface`

Floating point support

- Some processors have a floating point unit, some others do not.
  - For example, many ARMv4 and ARMv5 CPUs do not have a floating point unit. Since ARMv7, a VFP unit is mandatory.
- For processors having a floating point unit, the toolchain should generate *hard float* code, in order to use the floating point instructions directly
- For processors without a floating point unit, two solutions
  - Generate *hard float code* and rely on the kernel to emulate the floating point instructions. This is very slow.
  - Generate *soft float code*, so that instead of generating floating point instructions, calls to a user space library are generated
- Decision taken at toolchain configuration time
- Also possible to configure which floating point unit should be used

CPU optimization flags

- A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC)
- However, with the `-march=`, `-mcpu=`, `-mtune=` options, one can select more precisely the target CPU type
  - For example, `-march=armv7 -mcpu=cortex-a8`
- At the toolchain compilation time, values can be chosen. They are used:
  - As the default values for the cross-compiling tools, when no other `-march`, `-mcpu`, `-mtune` options are passed
  - To compile the C library
- Even if the C library has been compiled for armv5t, it doesn't prevent from compiling other programs for armv7

Obtaining a Toolchain

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
- See the *Crosstool-NG* `docs/` directory for details on how toolchains are built.

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 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

Toolchain building utilities (2)

- **Crosstool-ng**
  - Rewrite of the older Crosstool, with a menuconfig-like configuration system
  - Feature-full: supports uClibc, glibc, musl, hard and soft float, many architectures
  - Actively maintained
  - `http://crosstool-ng.org/`

Toolchain building utilities (3)

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/`

Crosstool-NG: installation and usage

- Installation of Crosstool-NG can be done system-wide, or just locally in the source directory. For local installation:

  ```
  ./configure --enable-local
  make
  make install
  ```

- Some sample configurations for various architectures are available in samples, they can be listed using

  ```
  ./ct-ng list-samples
  ```

- To load a sample configuration

  ```
  ./ct-ng <sample-name>
  ```

- To adjust the configuration

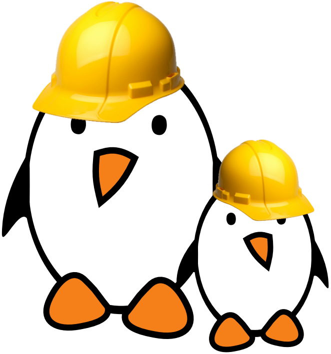
  ```
  ./ct-ng menuconfig
  ```

- To build the toolchain

  ```
  ./ct-ng build
  ```

Toolchain contents

- The cross compilation tool binaries, in `bin/`
  - This directory should be added to your `PATH` to ease usage of the toolchain
- One or several *sysroot*, each containing
  - The C library and related libraries, compiled for the target
  - The C library headers and kernel headers
- There is one *sysroot* for each variant: toolchains can be *multilib* if they have several copies of the C library for different configurations (for example: ARMv4T, ARMv5T, etc.)
  - Old CodeSourcery ARM toolchains were multilib, the sysroots in: `arm-none-linux-gnueabi/libc/`, `arm-none-linux-gnueabi/libc/armv4t/`, `arm-none-linux-gnueabi/libc/thumb2`
  - Crosstool-NG toolchains can be multilib too (still experimental), otherwise the sysroot is in `arm-unknown-linux-uclibcgnueabi/sysroot`



Time to build your toolchain

- Configure Crosstool-NG
- Run it to build your own cross-compiling toolchain

Bootloaders

Endocode AG

Embedded Linux
Experts

Corrections, suggestions, contributions and translations are welcome!

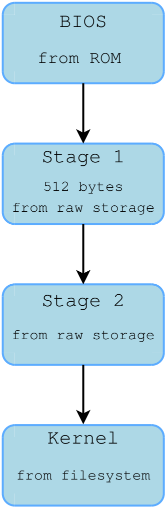
Boot Sequence

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 x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- This program gets executed by the CPU after reset, and is responsible for basic hardware initialization and loading of a small piece of code from non-volatile storage.
  - This piece of code is usually the first 512 bytes of a storage device
- This piece of code is usually a 1st stage bootloader, which will load the full bootloader itself.
- The bootloader can then offer all its features. It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.

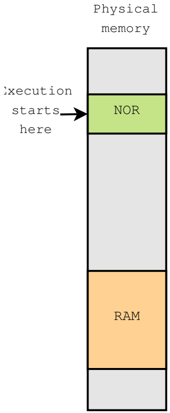


Bootloaders on x86 (2)

- 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.
  - See our dedicated presentation for details:
    http://free-electrons.com/docs/grub/
- 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

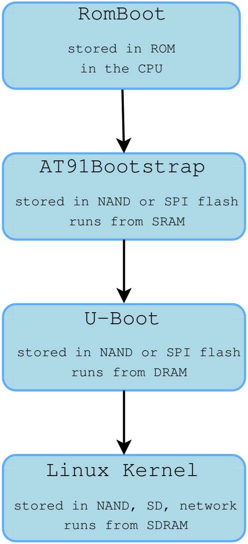
- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader 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

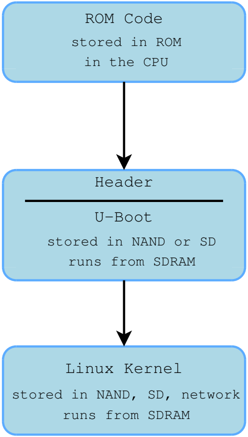
- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, “ROM code” on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, 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 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).

Generic bootloaders for embedded CPUs

- We will focus on the generic part, the main bootloader, offering the most important features.
- 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.

The U-boot bootloader

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 `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 U-Boot features that should be compiled in, etc.
  - It is a simple `.h` file that sets C pre-processor constants. See the `README` file for the documentation of these constants. This file can also be adjusted to add or remove features from U-Boot (commands, etc.).
- Assuming that your board is already supported by U-Boot, there should be one entry corresponding to your board in the `boards.cfg` file.
- Or just look in the `configs/` directory.

U-Boot configuration file excerpt

```
/* CPU configuration */
#define CONFIG_ARMV7 1
#define CONFIG_OMAP 1
#define CONFIG_OMAP34XX 1
#define CONFIG_OMAP3430 1
#define CONFIG_OMAP3_IGEP0020 1
[...]
/* Memory configuration */
#define CONFIG_NR_DRAM_BANKS 2
#define PHYS_SDRAM_1 OMAP34XX_SDRC_CS0
#define PHYS_SDRAM_1_SIZE (32 <<  20)
#define PHYS_SDRAM_2 OMAP34XX_SDRC_CS1
[...]
/* USB configuration */
#define CONFIG_MUSB_UDC 1
#define CONFIG_USB_OMAP3 1
#define CONFIG_TWL4030_USB 1
[...]
```

```
/* Available commands and features */
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_FAT
#define CONFIG_CMD_I2C
#define CONFIG_CMD_MMC
#define CONFIG_CMD_NAND
#define CONFIG_CMD_NET
#define CONFIG_CMD_DHCP
#define CONFIG_CMD_PING
#define CONFIG_CMD_NFS
#define CONFIG_CMD_MTDPARTS
[...]
```

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - `make BOARDNAME_defconfig`
  - Where `BOARDNAME` is the name of the board, as visible in the `boards.cfg` file (first column).
  - New: you can now run `make menuconfig` to further edit 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 2013.04 (May 29 2013 - 10:30:21)

OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0

Die ID #255000029ff800000168580212029011
Net:   smc911x-0
U-Boot #
```

- 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 2013.04 (May 29 2013 - 10:30:21)
```

Important commands (1)

- The exact set of commands depends on the U-Boot configuration
- `help` and `help command`
- `boot`, runs the default boot command, stored in `bootcmd`
- `bootz <address>`, starts a kernel image loaded at the given address in RAM
- `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

Important commands (2)

- `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 commands (1)

- U-Boot can be configured through environment variables, which affect the behavior of the different commands.
- 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 (2)

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'`

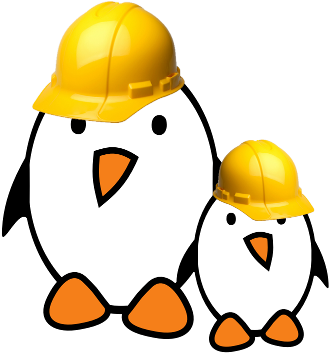
Transferring files to the target

- 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

Practical lab - U-Boot

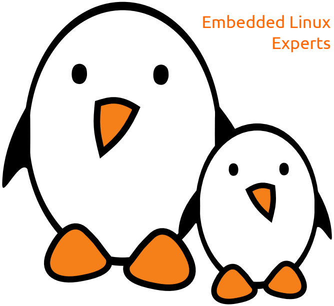


Time to start the practical lab!

- Communicate with the board using a serial console
- Configure, build and install *U-Boot*
- Learn *U-Boot* commands
- Set up *TFTP* communication with the board

Linux kernel introduction

Endocode AG



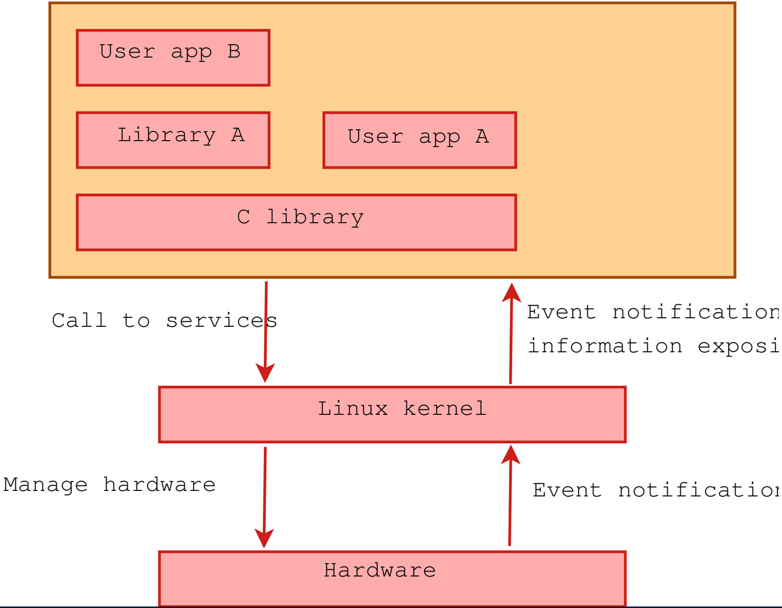
Linux features

- 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 in the system



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.

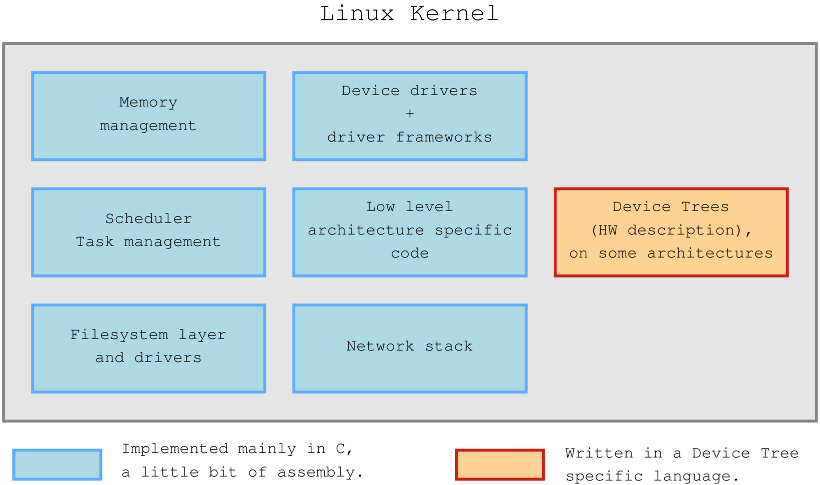
System calls

- The main interface between the kernel and user space is the set of system calls
- About 300 system calls that provide the main kernel services
  - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

Pseudo filesystems

- Linux makes system and kernel information available in user space through **pseudo filesystems**, sometimes also called **virtual filesystems**
- Pseudo filesystems allow applications to see directories and files that do not exist on any real storage: they are created and updated on the fly by the kernel
- The two most important pseudo filesystems are
  - `proc`, usually mounted on `/proc`:
    Operating system related information (processes, memory management parameters...)
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a set of devices and buses. Information about these devices.

Inside the Linux kernel



Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.

Supported hardware architectures

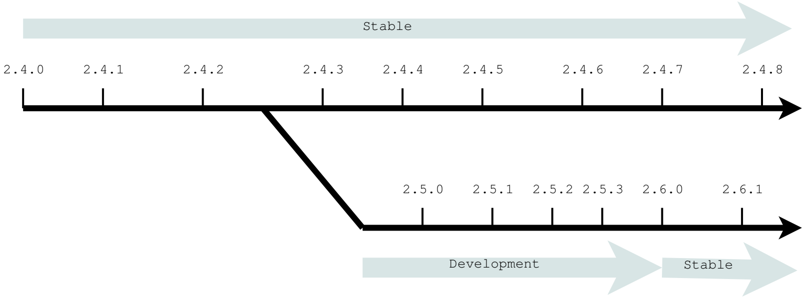
- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples: `arm`, `avr32`, `blackfin`, `c6x`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `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>/`

Linux versioning scheme and development process

Until 2.6 (1)

- One stable major branch every 2 or 3 years
  - Identified by an even middle number
  - Examples: `1.0.x, 2.0.x, 2.2.x, 2.4.x`
- One development branch to integrate new functionalities and major changes
  - Identified by an odd middle number
  - Examples: `2.1.x, 2.3.x, 2.5.x`
  - After some time, a development version becomes the new base version for the stable branch
- Minor releases once in while: `2.2.23, 2.5.12`, etc.

Until 2.6 (2)


Stable
2.4.0
2.4.1
2.4.2
2.4.3
2.4.4
2.4.5
2.4.6
2.4.7
2.4.8
2.5.0
2.5.1
2.5.2
2.5.3
2.6.0
2.6.1
Development
Stable

Changes since Linux 2.6

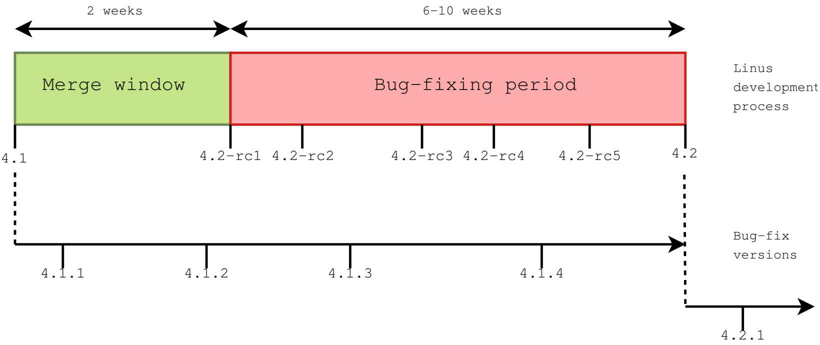
- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make disruptive changes to existing subsystems.
- Since then, there has been no need to create a new development branch massively breaking compatibility with the stable branch.
- Thanks to this, **more features are released to users at a faster pace**.

Versions since 2.6.0

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- Linux `4.0` was released in April 2015
- This is only a change to the numbering scheme
  - Official kernel versions are now named `x.y` (`3.0, 3.1, 3.2, ..., 3.19, 4.0, 4.1`, etc.)
  - Stabilized versions are named `x.y.z` (`3.0.2, 4.2.7`, etc.)
  - It effectively only removes a digit compared to the previous numbering scheme

New development model

Using merge and bug fixing windows



New development model - Details

- After the release of a `4.x` version (for example), a two-weeks merge window opens, during which major additions are merged.
- The merge window is closed by the release of test version `4.(x+1)-rc1`
- The bug fixing period opens, for 6 to 10 weeks.
- At regular intervals during the bug fixing period, `4.(x+1)-rcY` test versions are released.
- When considered sufficiently stable, kernel `4.(x+1)` is released, and the process starts again.

More stability for the kernel source tree

- Issue: bug and security fixes only released for most recent stable kernel versions.
- Some people need to have a recent kernel, but with long term support for security updates.
- You could get long term support from a commercial embedded Linux provider.
- You could reuse sources for the kernel used in Ubuntu Long Term Support releases (5 years of free security updates).
- The `http://kernel.org` front page shows which versions will be supported for some time (up to 2 or 3 years), and which ones won't be supported any more ("EOL: End Of Life")

mainline:	**4.4-rc4**	2015-12-06
stable:	**4.3.2**	2015-12-10
stable:	**4.2.7**	2015-12-09
longterm:	**4.1.14**	2015-12-09
longterm:	**3.18.24**	2015-10-31
longterm:	**3.14.58**	2015-12-09
longterm:	**3.12.51**	2015-11-25
longterm:	**3.10.94**	2015-12-09
longterm:	**3.4.110**	2015-10-22
longterm:	**3.2.74**	2015-11-27
longterm:	**2.6.32.69**	2015-12-05
linux-next:	**next-20151211**	2015-12-11

What's new in each Linux release?

- The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Jul 13 11:29:17 2011 +0200

    at91: at91-ohci: support overcurrent notification

    Several USB power switches (AIC1526 or MIC2026) have a digital output
    that is used to notify that an overcurrent situation is taking
    place. This digital outputs are typically connected to GPIO inputs of
    the processor and can be used to be notified of these overcurrent
    situations.

    Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data
    structure so that boards can tell the AT91 OHCI driver which pins are
    used for the overcurrent notification, and an overcurrent_supported
    boolean to tell the driver whether overcurrent is supported or not.

    The code has been largely borrowed from ohci-da8xx.c and
    ohci-s3c2410.c.

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
    Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>
```

  - Very difficult to find out the key changes and to get the global picture out of individual changes.
- Fortunately, there are some useful resources available
  - `http://wiki.kernelnewbies.org/LinuxChanges` (4.2 and 4.3 are missing)
  - `http://lwn.net`
  - `http://www.heise.de/open/kernel-log-3007.html`, for German readers

Linux kernel sources

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 (1)

- Linux 3.10 sources:
  Raw size: 573 MB (43,000 files, approx 15,800,000 lines)
  `gzip` compressed tar archive: 105 MB
  `bzip2` compressed tar archive: 83 MB (better)
  `xz` compressed tar archive: 69 MB (best)
- 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!

Linux kernel size (2)

As of kernel version 3.10.

- `drivers/`: 49.4%
- `arch/`: 21.9%
- `fs/`: 6.0%
- `include/`: 4.7%
- `sound/`: 4.4%
- `Documentation/`: 4.0%
- `net/`: 3.9%
- `firmware/`: 1.0%
- `kernel/`: 1.0%
- `tools/`: 0.9%
- `scripts/`: 0.5%
- `mm/`: 0.5%
- `crypto/`: 0.4%
- `security/`: 0.4%
- `lib/`: 0.4%
- `block/`: 0.2%
- ...

Getting Linux sources

- **Full tarballs**
  - Contain the complete kernel sources: long to download and uncompress, but must be done at least once
  - Example:

    `http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.10.9.tar.xz`
  - Extract command:

    `tar xf linux-3.10.9.tar.xz`
- **Incremental patches between versions**
  - It assumes you already have a base version and you apply the correct patches in the right order. Quick to download and apply
  - Examples:
    `http://www.kernel.org/pub/linux/kernel/v3.x/patch-3.10.xz` (3.9 to 3.10)
    `http://www.kernel.org/pub/linux/kernel/v3.x/patch-3.10.9.xz` (3.10 to 3.10.9)
- All previous kernel versions are available in `http://kernel.org/pub/linux/kernel/`

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:

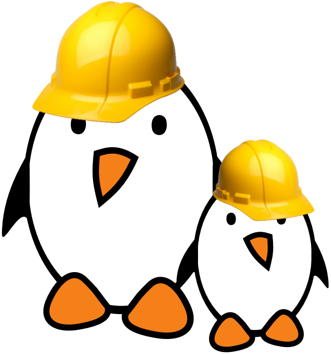
- `patch -p<n> < diff_file`
- `cat diff_file | patch -p<n>`
- `xzcat diff_file.xz | patch -p<n>`
- `bzcat diff_file.bz2 | patch -p<n>`
- `zcat diff_file.gz | patch -p<n>`
- Notes:
  - `n`: number of directory levels to skip in the file paths
  - You can reverse apply a patch with the `-R` option
  -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

Practical lab - Kernel sources



Time to start the practical lab!

- Get the Linux kernel sources
- Apply patches

Kernel configuration

Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main `Makefile`, present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the `make` tool, which parses the Makefile
  - through various **targets**, defining which action should be done (configuration, compilation, installation, etc.). Run `make help` to see all available targets.
- Example
  - `cd linux-3.6.x/`
  - `make <target>`

Kernel configuration (1)

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled
- The set of options depends
  - On your hardware (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

Kernel configuration (2)

- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `key=value` style
- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:
  - `make xconfig`, `make gconfig` (graphical)
  - `make menuconfig`, `make nconfig` (text)
  - You can switch from one to another, they all load/save the same `.config` file, and show the same set of options
- To modify a kernel in a GNU/Linux distribution: the configuration files are usually released in `/boot/`, together with kernel images: `/boot/config-3.2.0-31-generic`

Kernel or module?

- The **kernel image** is a **single file**, resulting from the linking of all object files that correspond to features enabled in the configuration
  - This is the file that gets loaded in memory by the bootloader
  - All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists
- Some features (device drivers, filesystems, etc.) can however be compiled as **modules**
  - These are *plugins* that can be loaded/unloaded dynamically to add/remove features to the kernel
  - Each **module is stored as a separate file in the filesystem**, and therefore access to a filesystem is mandatory to use modules
  - This is not possible in the early boot procedure of the kernel, because no filesystem is available

Kernel option types

There are different types of options

- `bool` options, they are either
  - *true* (to include the feature in the kernel) or
  - *false* (to exclude the feature from the kernel)
- `tristate` options, they are either
  - *true* (to include the feature in the kernel image) or
  - *module* (to include the feature as a kernel module) or
  - *false* (to exclude the feature)
- `int` options, to specify integer values
- `hex` options, to specify hexadecimal values
- `string` options, to specify string values

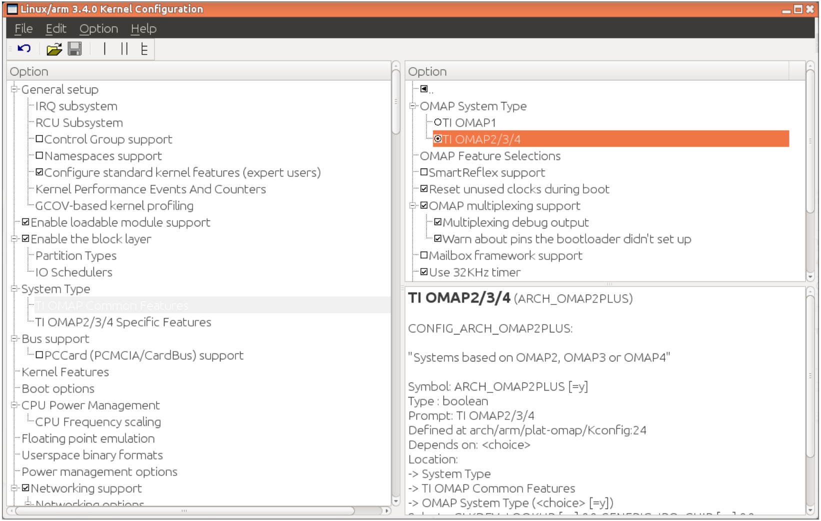
Kernel option dependencies

- There are dependencies between kernel options
- For example, enabling a network driver requires the network stack to be enabled
- Two types of dependencies
  - `depends on` dependencies. In this case, option A that depends on option B is not visible until option B is enabled
  - `select` dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
- `make xconfig` allows to see all options, even the ones that cannot be selected because of missing dependencies. In this case, they are displayed in gray.

`make xconfig`

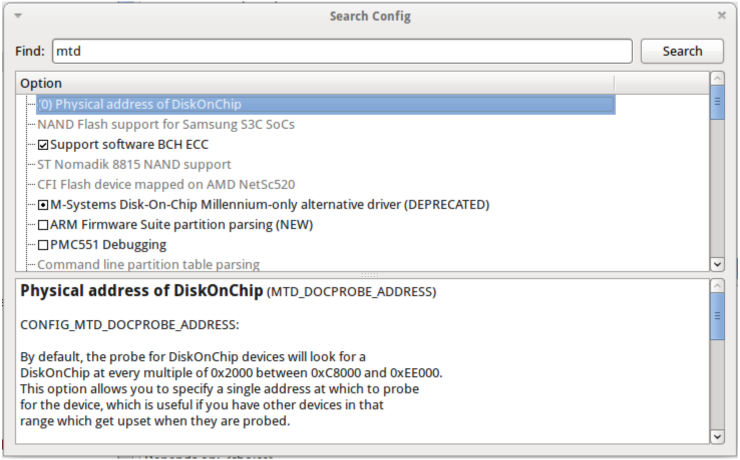
- The most common graphical interface to configure the kernel.
- Make sure you read
  `help -> introduction: useful options!`
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt4-dev g++`

make xconfig screenshot



make xconfig search interface

Looks for a keyword in the parameter name. Allows to select or unselect found parameters.



Kernel configuration options

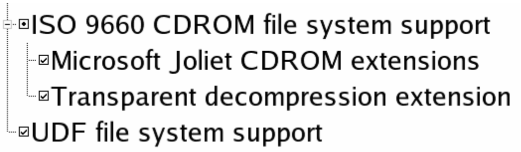
Compiled as a module (separate file)

CONFIG_ISO9660_FS=m

Driver options

CONFIG_JOLIET=y

CONFIG_ZISOFS=y



Compiled statically into the kernel

CONFIG_UDF_FS=y

Corresponding .config file excerpt

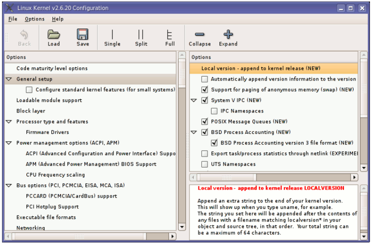
Options are grouped by sections and are prefixed with `CONFIG_`.

```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
CONFIG_UDF_NLS=y

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

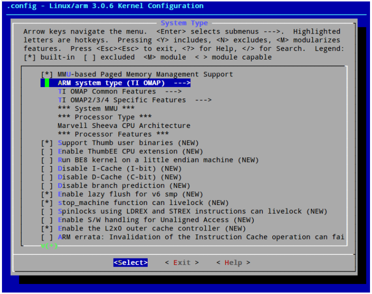
`make gconfig`

- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`



`make menuconfig`

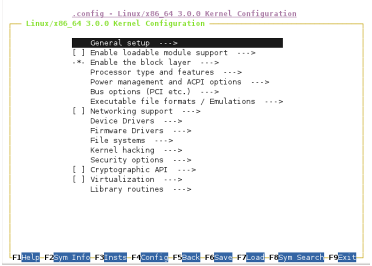
- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot…
- Required Debian packages: `libncurses-dev`



make nconfig

`make nconfig`

- A newer, similar text interface
- More user friendly (for example, easier to access help information).
- Required Debian packages: `libncurses-dev`



`make oldconfig`

- Needed very often!
- Useful to upgrade a `.config` file from an earlier kernel release
- Issues warnings for configuration parameters that no longer exist in the new kernel.
- Asks for values for new parameters (while `xconfig` and `menuconfig` silently set default values for new parameters).

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!

Undoing configuration changes

A frequent problem:

- After changing several kernel configuration settings, your kernel no longer works.
- If you don't remember all the changes you made, you can get back to your previous configuration:
  `$ cp .config.old .config`
- All the configuration interfaces of the kernel (`xconfig`, `menuconfig`, `oldconfig`...) keep this `.config.old` backup copy.

Configuration per architecture

- The set of configuration options is architecture dependent
  - Some configuration options are very architecture-specific
  - Most of the configuration options (global kernel options, network subsystem, filesystems, most of the device drivers) are visible in all architectures.
- By default, the kernel build system assumes that the kernel is being built for the host architecture, i.e. native compilation
- The architecture is not defined inside the configuration, but at a higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Compiling and installing the kernel for the host system

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

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 (1)

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 (2)

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.)

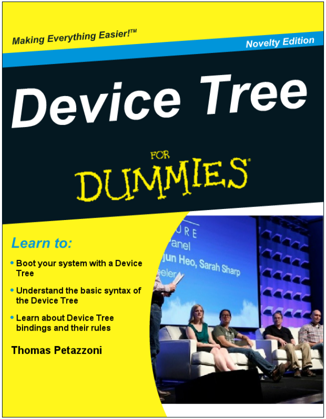
Device Tree

- Many embedded architectures have a lot of non-discoverable hardware.
- Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ARM, PowerPC, OpenRISC, ARC, Microblaze are examples of architectures using the Device Tree.
- A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, passed at boot time to the kernel.
  - There is one different Device Tree for each board/platform supported by the kernel, available in `arch/arm/boot/dts/<board>.dtb`.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Customize your board device tree!

Often needed for embedded board users:

- To describe external devices attached to non-discoverable busses (such as I2C) and configure them.
- To configure pin muxing: choosing what SoC signals are made available on the board external connectors.
- To configure some system parameters: flash partitions, kernel command line (other ways exist)
- Useful reference: Device Tree for Dummies, Thomas Petazzoni (Apr. 2014): `http://j.mp/1jQU6NR`



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.

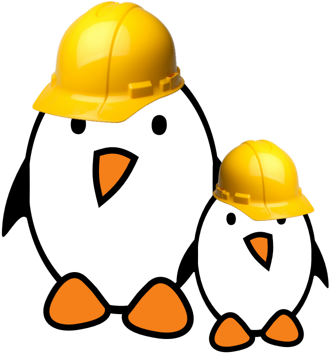
Booting with U-Boot

- 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*

Kernel command line

- In addition to the compile time configuration, the kernel behaviour can be adjusted with no recompilation using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root filesystem (covered later)
  - `console=` for the destination of kernel messages
  - Many more exist. The most important ones are documented in `Documentation/kernel-parameters.txt` in kernel sources.
- This kernel command line is either
  - Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel
  - Built into the kernel, using the `CONFIG_CMDLINE` option.

Practical lab - Kernel cross-compiling



- Set up the cross-compiling environment
- Configure and cross-compile the kernel for an `arm` platform
- On this platform, interact with the bootloader and boot your kernel

Using kernel modules

Advantages of modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the `root` user can load and unload modules.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the `usb-storage` module depends on the `scsi_mod`, `libusual` and `usbcore` modules.
- Dependencies are described both in `/lib/modules/<kernel-version>/modules.dep` and in `/lib/modules/<kernel-version>/modules.dep.bin`
  These files are generated when you run `make modules_install`.

Kernel log

When a new module is loaded, related information is available in the kernel log.

- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the `dmesg` command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel parameter, or completely disabled with the `quiet` parameter).
- Note that you can write to the kernel log from user space too:
  `echo "<n>Debug info" > /dev/kmsg`

Module utilities (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`
  Gets information about a module: parameters, license, description and dependencies.
  Very useful before deciding to load a module or not.
- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.

Understanding module loading issues

- When loading a module fails, `insmod` often doesn't give you enough details!
- Details are often available in the kernel log.
- Example:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Module utilities (2)

- `sudo modprobe <module_name>`
  Most common usage of `modprobe`: tries to load all the modules the given module depends on, and then this module. Lots of other options are available. `modprobe` automatically looks in `/lib/modules/<version>/` for the object file corresponding to the given module name.
- `lsmod`
  Displays the list of loaded modules
  Compare its output with the contents of `/proc/modules`!

Module utilities (3)

- `sudo rmmod <module_name>`
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- `sudo modprobe -r <module_name>`
  Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules

- Find available parameters:
  `modinfo snd-intel8x0m`
- Through `insmod`:
  `sudo insmod ./snd-intel8x0m.ko index=-2`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options snd-intel8x0m index=-2`
- Through the kernel command line, when the driver is built statically into the kernel:
  `snd-intel8x0m.index=-2`
  - `snd-intel8x0m` is the *driver name*
  - `index` is the *driver parameter name*
  - `-2` is the *driver parameter value*

Check module parameter values

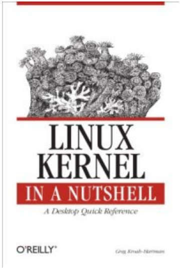
How to find the current values for the parameters of a loaded module?

- Check `/sys/module/<name>/parameters`.
- There is one file per parameter, containing the parameter value.

Useful reading

Linux Kernel in a Nutshell, Dec 2006

- By Greg Kroah-Hartman, O'Reilly
  `http://www.kroah.com/lkn/`
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  `http://free-electrons.com/community/kernel/lkn/`



Linux Root Filesystem

Endocode AG

Embedded Linux Experts

Principle and solutions

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=`

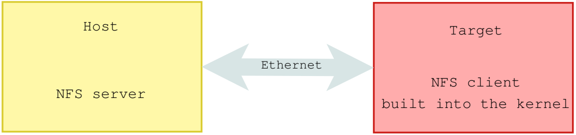
Mounting rootfs from storage devices

- 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)

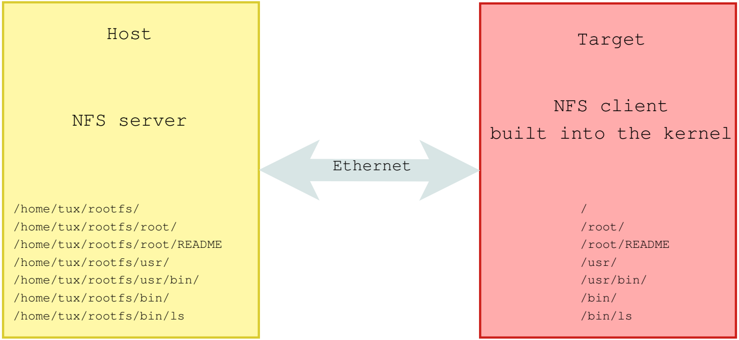
On the development workstation side, a NFS server is needed

- 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)

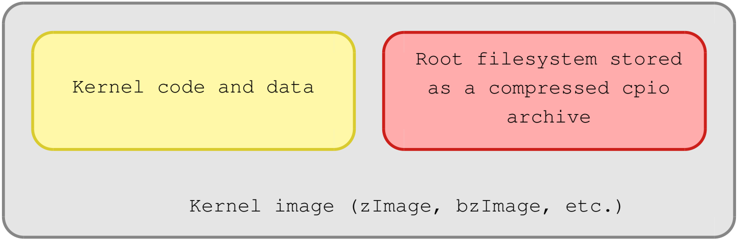
Mounting rootfs over the network (4)



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)



rootfs in memory: initramfs (3)

- 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`

Contents

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

Important directories (1)

/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

Important directories (2)

/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`

Device Files

Devices

- One of the kernel important role is to **allow applications to access hardware devices**
- In the Linux kernel, most devices are presented to user space applications through two different abstractions
  - **Character** device
  - **Block** device
- Internally, the kernel identifies each device by a triplet of information
  - **Type** (character or block)
  - **Major** (typically the category of device)
  - **Minor** (typically the identifier of the device)

Types of devices

- Block devices
  - A device composed of fixed-sized blocks, that can be read and written to store data
  - Used for hard disks, USB keys, SD cards, etc.
- Character devices
  - Originally, an infinite stream of bytes, with no beginning, no end, no size. The pure example: a serial port.
  - Used for serial ports, terminals, but also sound cards, video acquisition devices, frame buffers
  - Most of the devices that are not block devices are represented as character devices by the Linux kernel

ENDO CODE

Devices: everything is a file

- A very important Unix design decision was to represent most *system objects* as files
- It allows applications to manipulate all *system objects* with the normal file API (`open`, `read`, `write`, `close`, etc.)
- So, devices had to be represented as files to the applications
- This is done through a special artifact called a **device file**
- It is a special type of file, that associates a file name visible to user space applications to the triplet *(type, major, minor)* that the kernel understands
- All *device files* are by convention stored in the `/dev` directory

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a serial port

```c
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Creating device files

- Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needed root privileges
  - Coherency between device files and devices handled by the kernel was left to the system developer
- The `devtmpfs` virtual filesystem can be mounted on `/dev` and contains all the devices known to the kernel. The `CONFIG_DEVTMPFS_MOUNT` kernel configuration option makes the kernel mount it automatically at boot time, except when booting on an initramfs.

Pseudo Filesystems

proc virtual filesystem

- The `proc` virtual filesystem exists since the beginning of Linux
- It allows
  - The kernel to expose statistics about running processes in the system
  - The user to adjust at runtime various system parameters about process management, memory management, etc.
- The `proc` filesystem is used by many standard user space applications, and they expect it to be mounted in `/proc`
- Applications such as `ps` or `top` would not work without the `proc` filesystem
- Command to mount `/proc`:
  `mount -t proc nodev /proc`
- `Documentation/filesystems/proc.txt` in the kernel sources
- `man proc`

proc contents

- One directory for each running process in the system
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - It contains details about the files opened by the process, the CPU and memory usage, etc.
- `/proc/interrupts`, `/proc/devices`, `/proc/iomem`, `/proc/ioports` contain general device-related information
- `/proc/cmdline` contains the kernel command line
- `/proc/sys` contains many files that can be written to to adjust kernel parameters
  - They are called *sysctl*. See `Documentation/sysctl/` in kernel sources.
  - Example
    `echo 3 > /proc/sys/vm/drop_caches`

sysfs filesystem

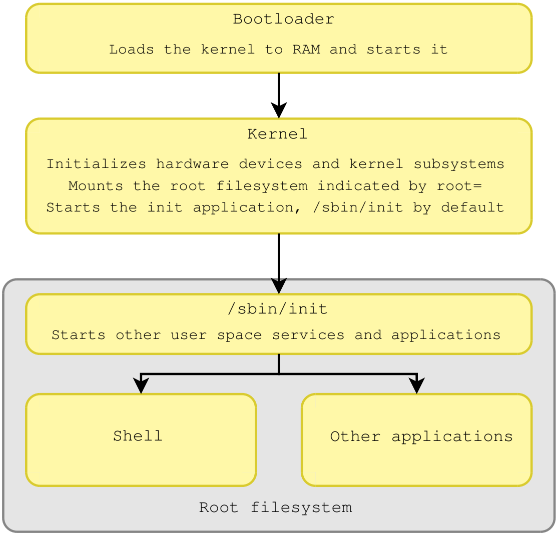
- The `sysfs` filesystem is a feature integrated in the 2.6 Linux kernel
- It allows to represent in user space the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various user space applications that need to list and query the available hardware, for example `udev` or `mdev`.
- All applications using sysfs expect it to be mounted in the `/sys` directory
- Command to mount `/sys`:
  `mount -t sysfs nodev /sys`
- ```
  $ ls /sys/
  block bus class dev devices firmware
  fs kernel module power
  ```
  ```

Minimal filesystem

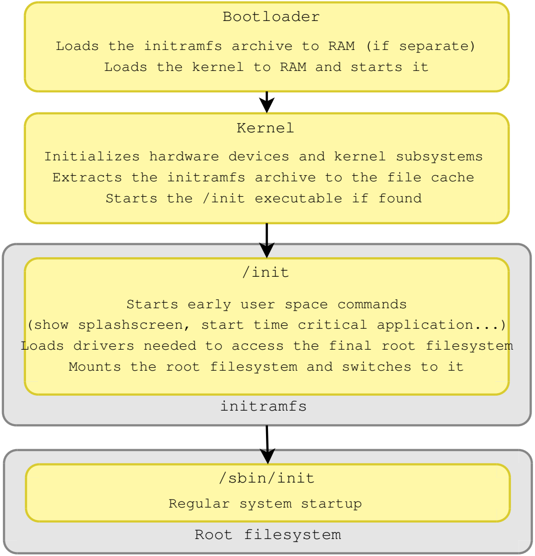
Basic applications

- In order to work, a Linux system needs at least a few applications
- An `init` application, which is the first user space application started by the kernel after mounting the root filesystem
  - The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
  - In the case of an initramfs, it will only look for `/init`. Another path can be supplied by the `rdinit` kernel argument.
  - If none of them are found, the kernel panics and the boot process is stopped.
  - The `init` application is responsible for starting all other user space applications and services
- A shell, to implement scripts, automate tasks, and allow a user to interact with the system
- Basic Unix applications, to copy files, move files, list files (commands like `mv`, `cp`, `mkdir`, `cat`, etc.)
- These basic components have to be integrated into the root filesystem to make it usable

Overall booting process



Overall booting process with initramfs



Busybox

Endocode AG

Embedded Linux Experts

Why Busybox?

- A Linux system needs a basic set of programs to work
  - An init program
  - A shell
  - Various basic utilities for file manipulation and system configuration
- In normal Linux systems, these programs are provided by different projects
  - `coreutils`, `bash`, `grep`, `sed`, `tar`, `wget`, `modutils`, etc. are all different projects
  - A lot of different components to integrate
  - Components not designed with embedded systems constraints in mind: they are not very configurable and have a wide range of features
- Busybox is an alternative solution, extremely common on embedded systems

General purpose toolbox: BusyBox

- Rewrite of many useful Unix command line utilities
  - Integrated into a single project, which makes it easy to work with
  - Designed with embedded systems in mind: highly configurable, no unnecessary features
- All the utilities are compiled into a single executable, `/bin/busybox`
  - Symbolic links to `/bin/busybox` are created for each application integrated into Busybox
- For a fairly featureful configuration, less than 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc).
- `http://www.busybox.net/`

BusyBox commands!

Commands available in BusyBox 1.13

[, [[, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, bbconfig, bbsh, brctl, bunzip2, busybox, bzcat, bzip2, cal, cat, catv, chat, chattr, chcon, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devfsd, df, dhcprelay, diff, dirname, dmesg, dnsd, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dumpleases, e2fsck, echo, ed, egrep, eject, env, envdir, envuidgid, ether_wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fetchmail, fgrep, find, findfs, fold, free, freeramdisk, fsck, fsck_minix, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifup, inetd, init, inotifyd, insmod, install, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, lash, last, length, less, linux32, linux64, linuxrc, ln, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lzmacat, makedevs, man, matchpathcon, md5sum, mdev, mesg, microcom, mkdir, mke2fs, mkfifo, mkfs_minix, mknod, mkswap, mktemp, modprobe, more, mount, mountpoint, msh, mt, mv, nameif, nc, netstat, nice, nmeter, nohup, nslookup, od, openvt, parse, passwd, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, poweroff, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, renice, reset, resize, restorecon, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run_parts, runcon, runlevel, runsv, runsvdir, rx, script, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfiles, setfont, setkeycodes, setlogcons, setsebool, setsid, setuidgid, sh, sha1sum, showkey, slattach, sleep, softlimit, sort, split, start_stop_daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, top, touch, tr, traceroute, true, tty, ttysize, tune2fs, udhcpc, udhcpd, udpsvd, umount, uname, uncompress, unexpand, uniq, unix2dos, unlzma, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip

Applet highlight: Busybox init

- Busybox provides an implementation of an `init` program
- Simpler than the init implementation found on desktop/server systems: no runlevels are implemented
- A single configuration file: `/etc/inittab`
  - Each line has the form `<id>::<action>:<process>`
- Allows to run services at startup, and to make sure that certain services are always running on the system
- See `examples/inittab` in Busybox for details on the configuration

Applet highlight - BusyBox vi

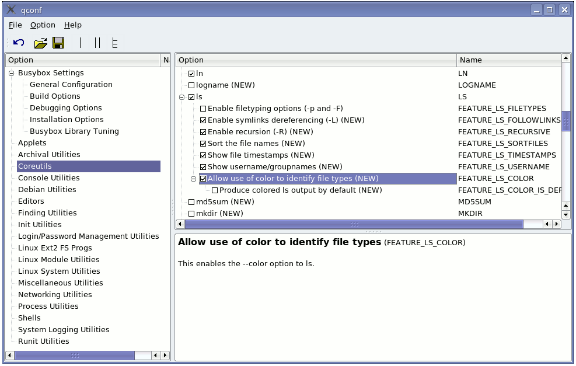
- If you are using BusyBox, adding `vi` support only adds 20K. (built with shared libraries, using uClibc).
- You can select which exact features to compile in.
- Users hardly realize that they are using a lightweight `vi` version!
- Tip: you can learn `vi` on the desktop, by running the `vimtutor` command.

Configuring BusyBox

- Get the latest stable sources from `http://busybox.net`
- Configure BusyBox (creates a `.config` file):
  - `make defconfig`
    Good to begin with BusyBox.
    Configures BusyBox with all options for regular users.
  - `make allnoconfig`
    Unselects all options. Good to configure only what you need.
- `make xconfig` (graphical, needs the `libqt3-mt-dev` package) or `make menuconfig` (text)
  Same configuration interfaces as the ones used by the Linux kernel (though older versions are used).

You can choose:

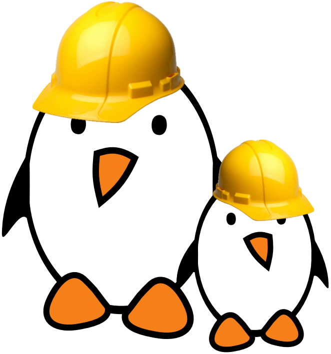
- the commands to compile,
- and even the command options and features that you need!



Compiling BusyBox

- Set the cross-compiler prefix in the configuration interface:
  `BusyBox Settings -> Build Options -> Cross Compiler prefix`
  Example: `arm-linux-`
- Set the installation directory in the configuration interface:
  `BusyBox Settings -> Installation Options -> BusyBox installation prefix`
- Add the cross-compiler path to the PATH environment variable:
  `export PATH=/usr/xtools/arm-unknown-linux-uclibcgnueabi/bin:$PATH`
- Compile BusyBox:
  `make`
- Install it (this creates a Unix directory structure symbolic links to the `busybox` executable):
  `make install`

Practical lab - A tiny embedded system



- Make Linux boot on a directory on your workstation, shared by NFS
- Create and configure a minimalistic Linux embedded system
- Install and use BusyBox
- System startup with `/sbin/init`
- Set up a simple web interface
- Use shared libraries

Flash filesystems

Endocode AG

Embedded Linux
Experts

Corrections, suggestions, contributions and translations are welcome!

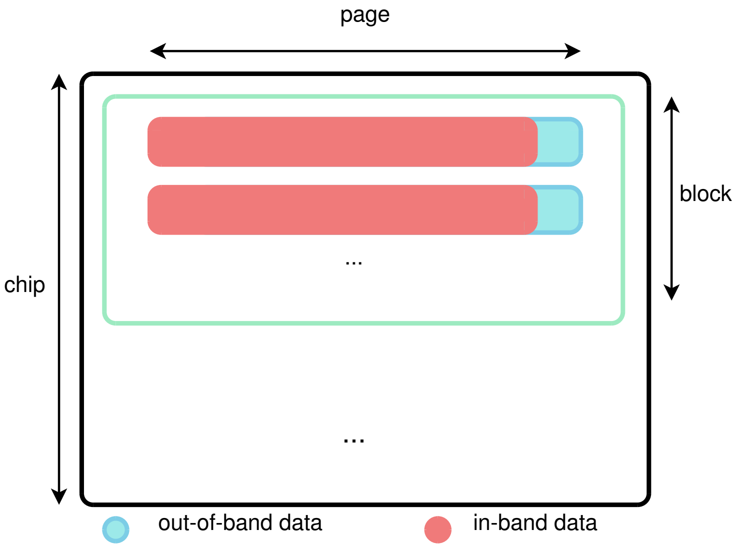
Block devices vs flash devices: reminder

- Block devices:
  - Allow for random data access using fixed size blocks
  - Do not require special care when writing on the media
  - Block size is relatively small (minimum 512 bytes, can be increased for performance reasons)
  - Considered as reliable (if the storage media is not, some hardware or software parts are supposed to make it reliable)
- Flash devices:
  - Allow for random data access too
  - Require special care before writing on the media (erasing the region you are about to write on)
  - Erase, write and read operation might not use the same block size
  - Reliability depends on the flash technology

NAND flash chips: how they work ?

- Encode bits with voltage levels
- Start with all bits set to 1
- Programming implies changing some bits from 1 to 0
- Restoring bits to 1 is done via the ERASE operation
- Programming and erasing is not done on a per bit or per byte basis
- Organization
  - Page: minimum unit for PROGRAM operation
  - Block: minimum unit for ERASE operation

NAND flash storage: organization



NAND flash storage: constraints

- Reliability
  - Far less reliable than NOR flash
  - Reliability depends on the NAND flash technology (SLC, MLC)
  - Require additional mechanisms to recover from bit flips: ECC (Error Correcting Code)
  - ECC information stored in the OOB (Out-of-band area)
- Lifetime
  - Short lifetime compared to other storage media
  - Lifetime depends on the NAND flash technology (SLC, MLC): between 1000000 and 1000 erase cycles per block
  - Wear leveling mechanisms are required
  - Bad block detection/handling required too
- Despite the number of constraints brought by NAND they are widely used in embedded systems for several reasons:
  - Cheaper than other flash technologies
  - Provide high capacity storage
  - Provide good performance (both in read and write access)

ENDO CODE

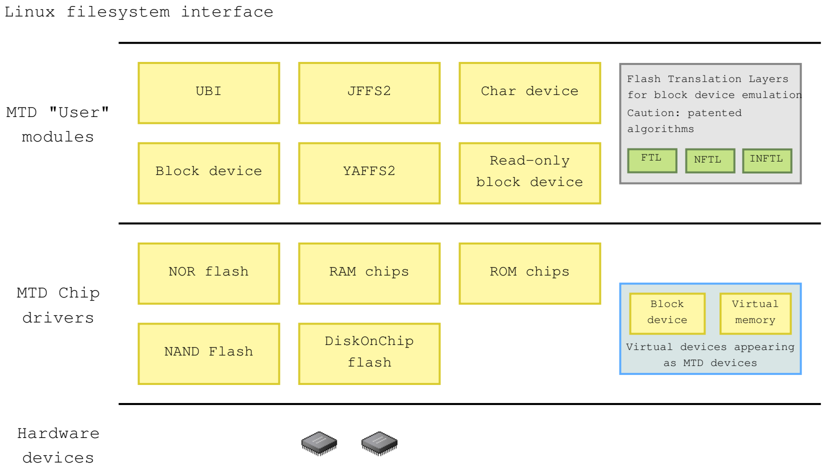
NAND flash: ECC

- ECC partly addresses the reliability problem on NAND flash
- Operates on blocks (usually 512 or 1024 bytes)
- ECC data are stored in the OOB area
- Three algorithms:
  - Hamming: can fixup a single bit per block
  - Reed-Solomon: can fixup several bits per block
  - BCH: can fixup several bits per block
- BCH and Reed-Solomon strength depends on the size allocated for ECC data, which in turn depends on the OOB size
- NAND manufacturers specify the required ECC strength in their datasheets: ignoring these requirements might compromise data integrity

The MTD subsystem (1)

- MTD stands for *Memory Technology Devices*
- Generic subsystem dealing with all types of storage media that are not fitting in the block subsystem
- Supported media types: RAM, ROM, NOR flash, NAND flash, Dataflash
- Independent of the communication interface (drivers available for parallel, SPI, direct memory mapping, ...)
- Abstract storage media characteristics and provide a simple API to access MTD devices
- MTD device characteristics exposed to users:
  - `erasesize`: minimum erase size unit
  - `writesize`: minimum write size unit
  - `oobsize`: extra size to store metadata or ECC data
  - `size`: device size
  - `flags`: information about device type and capabilities
- Various kind of MTD users: file-systems, block device emulation layers, user space interfaces...

The MTD subsystem (2)



MTD partitioning

- MTD devices are usually partitioned
  - It allows to use different areas of the flash for different purposes: read-only filesystem, read-write filesystem, backup areas, bootloader area, kernel area, etc.
- Unlike block devices, which contains their own partition table, the partitioning of MTD devices is described externally (don't want to put it in a flash sector which could become bad)
  - Specified in the board Device Tree
  - Hard-coded into the kernel code (if no Device Tree)
  - Specified through the kernel command line
- Each partition becomes a separate MTD device
  - Different from block device labeling (`hda3`, `sda2`)
  - `/dev/mtd1` is either the second partition of the first flash device, or the first partition of the second flash device
  - Note that the master MTD device (the device those partitions belongs to) is not exposed in `/dev`

Linux: definition of MTD partitions

The Device Tree is the standard place to define MTD partitions for platforms with Device Tree support.

Example from `arch/arm/boot/dts/omap3-igep.dtsi`:

```
        nand@0,0 {
                linux,mtd-name= "micron,mt29c4g96maz";
[...]
                partition@0 {
                        label = "SPL";
                        reg = <0 0x100000>;
                };
                partition@0x80000 {
                        label = "U-Boot";
                        reg = <0x100000 0x180000>;
                };
[...]
                partition@0x780000 {
                        label = "Filesystem";
                        reg = <0x680000 0x1f980000>;
                };
```

U-Boot: defining MTD partitions (1)

- U-Boot also provides a way to define MTD partitions on flash devices
- Named partitions are easier to use, and much less error prone than using offsets.
- U-Boot partition definitions can also be used by Linux too, eliminating the risk of mismatches between Linux and U-Boot.
- Use flash specific commands (detailed soon), and pass partition names instead of numerical offsets
- Example: `nand erase.part <partname>`

U-Boot: defining MTD partitions (2)

- Example:

```
setenv mtdids nand0=omap2-nand.0
setenv mtdparts mtdparts=omap2-nand.0:512k(X-Loader)ro,1536k(U-Boot)ro,512k(Env),4m(Kernel),-(RootFS)
```

- This defines 5 partitions in the `omap2-nand.0` device:
  - `1st stage bootloader` (512 KiB, read-only)
  - `U-Boot` (1536 KiB, read-only)
  - `U-Boot environment` (512 KiB)
  - `Kernel` (4 MiB)
  - `Root filesystem` (Remaining space)
- Partition sizes must be multiple of the erase block size. You can use sizes in hexadecimal too. Remember the below sizes: `0x20000` = 128k, `0x100000` = 1m, `0x1000000` = 16m
- `ro` lists the partition as read only
- `-` is used to use all the remaining space.

U-Boot: defining MTD partitions (3)

Details about the two environment variables needed by U-Boot:

- `mtdids` attaches an *mtdid* to a flash device.
  `setenv mtdids <devid>=<mtdid>[,<devid>=<mtdid>]`
  - `devid`: device identifier retrieved with `nand info` or `flinfo`
  - `mtdid`: mtd identifier (should match Linux MTD device name). It is displayed when booting the Linux kernel:

```
NAND device: Manufacturer ID: 0x2c, Chip ID: 0xbc (Micron NAND 512MiB 1,8V 16-bit)
Creating 5 MTD partitions on "omap2-nand.0":
0x000000000000-0x000000080000 : "X-Loader"
0x000000080000-0x000000200000 : "U-Boot"
0x000000200000-0x000000280000 : "Environment"
0x000000280000-0x000000580000 : "Kernel"
0x000000580000-0x000020000000 : "File System"
```

- `mtdparts` defines partitions for the different devices
  `setenv mtdparts <mtdid>:<partition>[,partition]`
  `partition` format: `<size>[@offset](<name>)[ro]`

Use the `mtdparts` command to setup the configuration specified by the `mtdids` and `mtdparts` variables

U-Boot: sharing partition definitions with Linux

Linux understands U-Boot's `mtdparts` partition definitions.
Here is a recommended way to pass them from U-Boot to Linux:

- Define a `bootargs_base` environment variable:
  `setenv bootargs_base console=ttyS0 root=....`
- Define the final kernel command line (`bootargs`) through the `bootcmd` environment variable:
  ```
  setenv bootcmd 'setenv bootargs ${bootargs_base}
  ${mtdparts}; <rest of bootcmd>'
  ```

U-Boot: manipulating NAND devices

U-Boot provides a set of commands to manipulate NAND devices, grouped under the `nand` command

- `nand info`
  Show available NAND devices and characteristics
- `nand device [dev]`
  Select or display the active NAND device
- `nand read[.option] <addr> <offset|partname> <size>`
  Read data from NAND
- `nand write[.option] <addr> <offset|partname> <size>`
  Write data on NAND
  - Use `nand write.trimffs` to avoid writing empty pages (those filled with `0xff`)
- `nand erase <offset> <size>`
  Erase a NAND region
- `nand erase.part <partname>`
  Erase a NAND partition
- More commands for debugging purposes

U-Boot: manipulating NOR devices (1)

- U-Boot provides a set of commands to manipulate NOR devices
- Memory mapped NOR devices
  - `flinfo [devid]`
    Display information of all NOR devices or a specific one if `devid` is provided
  - `cp.[bwl] <src> <target> <count>`
    Read/write data from/to the NOR device
  - `erase <start> <end>` or `erase <start> +<len>`
    Erase a memory region
  - `erase bank <bankid>`
    Erase a memory bank
  - `erase all`
    Erase all banks
  - `protect on|off <range-description>`
    Protect a memory range

U-Boot: manipulating NOR devices (2)

- SPI NOR devices
  - Grouped under the `sf` command
  - `sf probe [[bus:]cs] [hz] [mode]`
    Probe a NOR device on
  - `sf read|write <addr> <offset> <len>`
    Read/write data from/to a SPI NOR
  - `sf erase <offset> +<len>`
    Erase a memory region
  - `sf update <addr> <offset> <len>`
    Erase + write operation

Linux: MTD devices interface with user space

- MTD devices are visible in `/proc/mtd`
- The user space only see MTD partitions, not the flash device under those partitions
- The **mtdchar** driver creates a character device for each MTD device/partition of the system
  - Usually named `/dev/mtdX` or `/dev/mtdXro`
  - Provide `ioctl()` to erase and manage the flash
  - Used by the *mtd-utils* utilities

Linux: user space flash management tools

- `mtd-utils` is a set of utilities to manipulate MTD devices
  - `mtdinfo` to get detailed information about an MTD device
  - `flash_erase` to partially or completely erase a given MTD device
  - `flashcp` to write to NOR flash
  - `nandwrite` to write to NAND flash
  - Flash filesystem image creation tools: `mkfs.jffs2`, `mkfs.ubifs`, `ubinize`, etc.
- Usually available as the `mtd-utils` package in your distribution
- Most commands now also available in BusyBox
- See http://www.linux-mtd.infradead.org/

Flash wear leveling (1)

- Wear leveling consists in distributing erases over the whole flash device to avoid quickly reaching the maximum number of erase cycles on blocks that are written really often
- Can be done in:
  - the filesystem layer (JFFS2, YAFFS2, ...)
  - an intermediate layer dedicated to wear leveling (UBI)
- The wear leveling implementation is what makes your flash lifetime good or not

Flash wear leveling (2)

Flash users should also take the limited lifetime of flash devices into account by taking additional precautions

- Do not use your flash storage as swap area (rare in embedded systems anyway)
- Mount your filesystems as read-only, or use read-only filesystems (SquashFS), whenever possible.
- Keep volatile files in RAM (tmpfs)
- Don't use the `sync` mount option (commits writes immediately). Use the `fsync()` system call for per-file synchronization.

Flash file-systems

- 'Standard' file systems are meant to work on block devices
- Specific file systems have been developed to deal flash constraints
- These file systems are relying on the MTD layer to access flash chips
- There are several legacy flash filesystems which might be useful for specific usage: JFFS2, YAFFS2.
- Nowadays, UBI/UBIFS is the de facto standard for medium to large capacity NANDs (above 128MB)

Legacy flash filesystems: JFFS2

- Supports on the fly compression
- Wear leveling, power failure resistant
- Available in the official Linux kernel
- Boot time depends on the filesystem size: doesn't scale well for large partitions.
- `http://www.linux-mtd.infradead.org/doc/jffs2.html`

Standard file API

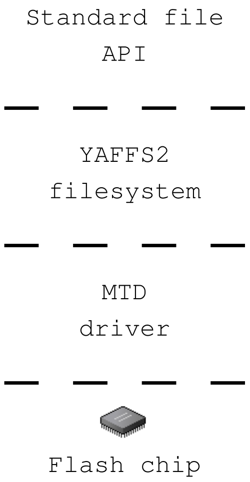
JFFS2 filesystem

MTD driver

Flash chip

Legacy flash filesystems: YAFFS2

- Mainly supports NAND flash
- No compression
- Wear leveling, power failure resistant
- Fast boot time
- Not part of the official Linux kernel: code only available separately
  (Dual GPL / Proprietary license for non Linux operating systems)
- http://www.yaffs.net/



UBI/UBIFS

- Aimed at replacing JFFS2 by addressing its limitations
- Design choices:
  - Split the wear leveling and filesystem layers
  - Add some flexibility
  - Focus on scalability, performance and reliability
- Drawback: introduces noticeable overhead, especially when used on small devices or partitions

Standard file API

UBIFS filesystem

UBI

MTD driver

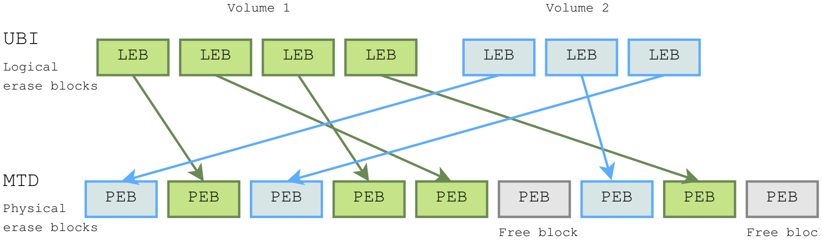
Flash chip

UBI (1)

Unsorted Block Images

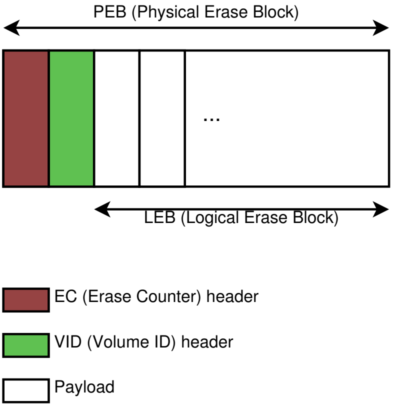
- http://www.linux-mtd.infradead.org/doc/ubi.html
- Volume management system on top of MTD devices (similar to what LVM provides for block devices)
- Allows to create multiple logical volumes and spread writes across all physical blocks
- Takes care of managing the erase blocks and wear leveling. Makes filesystems easier to implement
- Wear leveling can operate on the whole storage, not only on individual partitions (strong advantage)
- Volumes can be dynamically resized or, on the opposite, can be read-only (static)

UBI (2)


Volume 1
Volume 2
UBI
Logical
erase blocks
LEB
LEB
LEB
LEB
LEB
LEB
LEB
MTD
Physical
erase blocks
PEB
PEB
PEB
PEB
PEB
PEB
PEB
PEB
PEB
Free block

UBI: internals

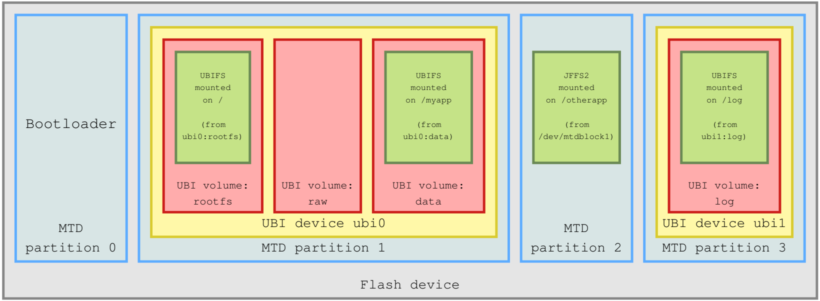
- UBI is storing its metadata in-band
- In each MTD erase block
  - One page is reserved to count the number of erase cycles
  - Another page is reserved to attach the erase block to a UBI volume
  - The remaining pages are used to store payload data
- If the device supports subpage write, the EC and VID headers can be stored on the same page



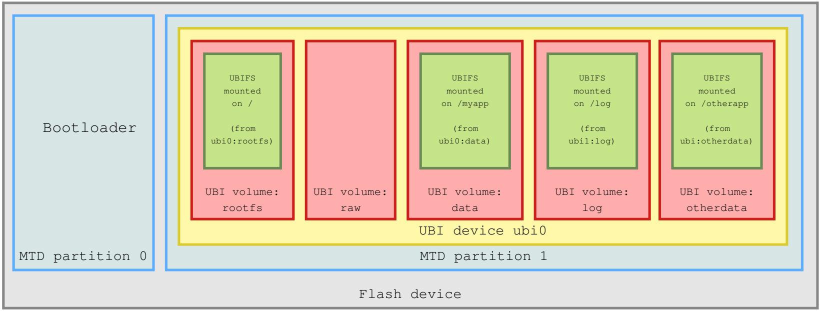
UBI: good practice

- UBI is responsible for distributing writes all over the flash device: the more space you assign to a partition attached to the UBI layer the more efficient the wear leveling will be
- If you need partitioning, use UBI volumes not MTD partitions
- Some partitions will still have to be MTD partitions: e.g. the bootloaders and bootloader environments
- If you need extra MTD partitions, try to group them at the end or the beginning of the flash device

UBI layout: bad example



UBI layout: good example



Unsorted Block Images File System

- http://www.linux-mtd.infradead.org/doc/ubifs.html
- The filesystem part of the UBI/UBIFS couple
- Works on top of UBI volumes
- Journaling file system providing better performance than JFFS2 and addressing its scalability issues
- See this paper for more technical details about UBIFS internals http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf

Linux: UBI host tools (1)

- `ubinize` is the only host tool for the UBI layer
- Creates a UBI image to be flashed on an MTD partition
- Takes the following arguments:
  - `-o <output-file-path>`
    Path to the output image file
  - `-p <peb-size>`
    The PEB size (MTD erase block size)
  - `-m <min-io-size>`
    The minimum write unit size (e.g. MTD write size)
  - `-s <subpage-size>`
    Subpage size, only needed if both your flash and your flash controller are supporting subpage writes
  - The last argument is a path to a UBI image description file
- Example:

```
ubinize -o ubi.img -p 16KiB -m 512 -s 256 cfg.ini
```

Linux: UBI host tools (2)

- ubinize config file can contain several sections
- Each section is describing a UBI volume
- Example:

```
[kernel-volume]
mode=ubi
image=zImage
vol_id=1
vol_type=static
vol_name=kernel
```

```
[rootfs-volume]
mode=ubi
image=rootfs.squashfs
vol_id=2
vol_type=static
vol_name=rootfs
```

```
[data-volume]
mode=ubi
image=data.ubifs
vol_id=3
vol_size=30MiB
vol_type=dynamic
vol_name=data
vol_flags=autoresize
```

U-Boot: UBI tools

Grouped under the `ubi` command

- `ubi part <part-name>`
  Attach an MTD partition to the UBI layer
- `ubi info [layout]`
  Display UBI device information
  (or volume information if the `layout` string is passed
- `ubi check <vol-name>`
  Check if a volume exists
- `ubi readvol <dest-addr> <vol-name> [<size>]`
  Read volume contents
- U-Boot also provides tools to update the UBI device contents
- Using them is highly discouraged (the U-Boot UBI implementation is not entirely stable, and using commands that do not touch the UBI metadata is safer)
  - `ubi createvol <vol-name> [<size>] [<type>]`
  - `ubi removevol <vol-name>`
  - `ubi writevol <src-addr> <vol-name> <size>`

Linux: UBI target tools (1)

- Tools used on the target to dynamically create and modify UBI elements
- UBI device management:
  - `ubiformat <MTD-device-id>`
    Format an MTD partition and preserve Erase Counter information if any
  - `ubiattach -m <MTD-device-id> /dev/ubi_ctrl`
    Attach an MTD partition/device to the UBI layer, and create a UBI device
  - `ubidetach -m <MTD-device-id> /dev/ubi_ctrl`
    Detach an MTD partition/device from the UBI layer, and remove the associated UBI device

Linux: UBI target tools (2)

UBI volume management:

- `ubimkvol /dev/ubi<UBI-device-id> -N <name> -s <size>`
  Create a new volume. Use `-m` in place of `-s <size>` if you want to assign all the remaining space to this volume.
- `ubirmvol /dev/ubi<UBI-device-id> -N <name>`
  Delete a UBI volume
- `ubiupdatevol /dev/ubi<UBI-device-id>_<UBI-vol-id> [-s <size>] <vol-image-file>`
  Update volume contents
- `ubirsvol /dev/ubi<UBI-device-id> -N <name> -s <size>`
  Resize a UBI volume
- `ubirename /dev/ubi<UBI-device-id>_<UBI-vol-id> <old-name> <new-size>`
  Rename a UBI volume

Linux tools: BusyBox UBI limitations

Beware that the implementation of UBI commands in BusyBox is still incomplete. For example:

- `ubirsvol` doesn't support `-N <name>`. You have to use specify the volume to resize by its id (`-n num`):
  `ubirsvol /dev/ubi0 -n 4 -s 64 MiB`
- Same constraint for `ubirmvol`:
  `ubirmvol /dev/ubi0 -n 4`

Linux: UBIFS host tools

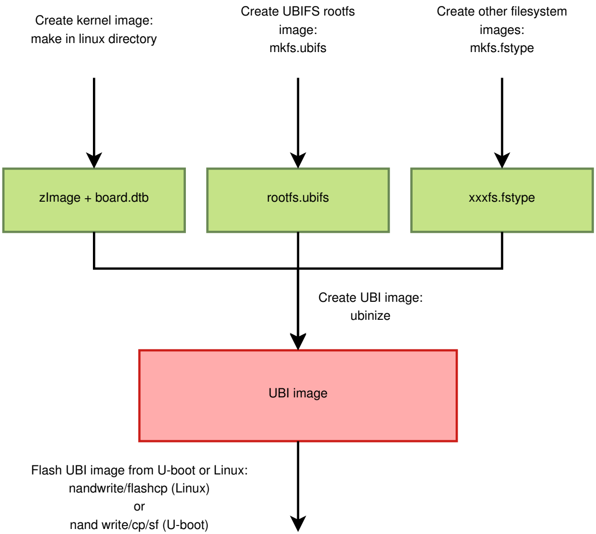
UBIFS filesystems images can be created using `mkfs.ubifs`

- `mkfs.ubifs -m 4096 -e 258048 -c 1000 -r rootfs/ ubifs.img`
    - `-m 4096`, minimal I/O size (see `/sys/class/mtd/mtdx/writesize`).
    - `-e 258048`, logical erase block size (smaller than PEB size, can be found in the kernel log after running `ubiattach`)
    - `-c 1000`, maximum number of logical erase blocks. Details: `http://linux-mtd.infradead.org/faq/ubifs.html#L_max_leb_cnt`
- Once created
    - Can be written to a UBI volume from the target using `ubiupdatevol`
    - Or, can be included in a UBI image (using `ubinize` on the host)

Linux: UBIFS target tools

- No specific tools are required to manipulate a UBIFS filesystem
- Mounting a UBIFS filesystem is done with `mount`:
  `mount -t ubifs <ubi-device-id>:<volume-name> <mount-point>`
- Example:
  `mount -t ubifs ubi0:data /data`

Linux: UBI image creation workflow



Linux: flashing UBI/UBIFS howtos

- See http://free-electrons.com/blog/creating-flashing-ubi-ubifs-images/ for details about creating UBI and UBIFS images.
- See http://www.linux-mtd.infradead.org/doc/ubi.html#L_flasher_algo for what is required when flashing UBI images.

Linux: Using a UBIFS filesystem as root filesystem

- You just have to pass the following information on the kernel command line:
  - `ubi.mtd=1`
    Attach `/dev/mtd1` to the UBI layer and create `ubi0`
  - `rootfstype=ubifs root=ubi0:rootfs`
    Mount the `rootfs` volume on ubi0 as a UBIFS filesystem
- Example: `rootfstype=ubifs ubi.mtd=1 root=ubi0:rootfs`

Summary: how to boot on a UBIFS filesystem

In U-Boot:

- Define partitions:

  ```
  setenv mtdids ...
  setenv mtdparts ...
  ```

- Define the base Linux kernel bootargs, specifying booting on UBIFS, the UBI volume used as root filesystem, and the MTD partition attached to UBI. Example:

  ```
  setenv base_bootargs console=ttyS0 rootfstype=ubifs
  root=ubi0:rootfs ubi.mtd=2 ...
  ```

- Define the boot command sequence, loading the U-Boot partition definitions, loading kernel and DTB images from UBI partitions, and adding `mtdparts` to the kernel command line. Example:

  ```
  setenv bootcmd 'mtdparts; ubi part UBI; ubi readvol
  0x81000000 kernel; ubi readvol 0x82000000 dtb;
  setenv bootargs ${bootargs_base} ${mtdparts}; bootz
  0x81000000 - 0x82000000'
  ```

Linux: Block emulation layers

- Sometimes we need block devices to re-use existing block filesystems
- Particularly useful for read-only block filesystems like squashfs
- Linux provides two block emulation layers:
  - `mtdblock`: block devices emulated on top of MTD devices
  - `ubiblock`: block devices emulated on top of UBI volumes
- For read access, using emulated block devices is exactly the same as using regular block devices
- Even if supported through the `mtdblock` emulation layer, writing on emulated block devices is highly discouraged
  - the emulation layer does not properly deal with wear leveling and data retention issues occurring on the flash media

Linux: mtdblock

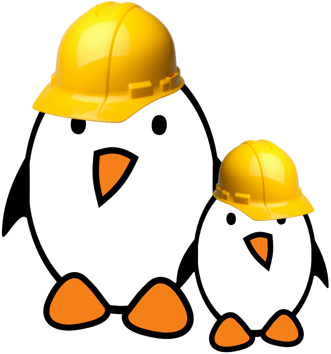
- The `mtdblock` layer creates a block device for each MTD device of the system
- Usually named `/dev/mtdblockX`.
- Allows read/write block-level access. But bad blocks are not handled, and no wear leveling is done for writes.
- For historical reasons JFFS2 filesystems require a block device to be mounted
- **Do not write on mtdblock devices**

Linux: ubiblock

- Preferred over `mtdblock` if UBI is available (UBI accounts for data retention and wear leveling issues, while MTD does not)
- The `ubiblock` layer creates **read-only** block devices on demand
- The user specifies which static volumes (s)he would like to attach to `ubiblock`
  - Through the kernel command line: by passing `ubi.block=<ubi-dev-id>,<volume-name>`
  - Using the `ubiblock` utility provided by `mtd-utils`: `ubiblock --create <ubi-volume-dev-file>`
- Usually named `/dev/ubiblockX_Y`, where X is the UBI device id and Y is the UBI volume id

Useful reading

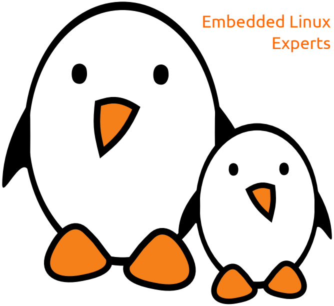
- Managing flash storage with Linux: http://free-electrons.com/blog/managing-flash-storage-with-linux/
- Documentation on the linux-mtd website: http://www.linux-mtd.infradead.org/



- Creating partitions in your internal flash storage
- Creating a UBI image with several volumes and flashing it from U-Boot
- Manipulating UBI volumes from Linux

Embedded Linux system development

Endocode AG



Contents

Leveraging open-source components in an Embedded Linux system

Third party libraries and applications

- One of the advantages of embedded Linux is the wide range of third-party libraries and applications that one can leverage in its product
  - They are freely available, freely distributable, and thanks to their open-source nature, they can be analyzed and modified according to the needs of the project
- However, efficiently re-using these components is not always easy. One must:
  - Find these components
  - Choose the most appropriate ones
  - Cross-compile them
  - Integrate them in the embedded system and with the other applications

Find existing components

- Free Software Directory
  `http://directory.fsf.org`
- Look at other embedded Linux products, and see what their components are
- Look at the list of software packaged by embedded Linux build systems
  - These are typically chosen for their suitability to embedded systems
- Ask the community or Google
- This presentation will also feature a list of components for common needs

Choosing components

Not all free software components are necessarily good to re-use. One must pay attention to:

- **Vitality** of the developer and user communities. This vitality ensures long-term maintenance of the component, and relatively good support. It can be measured by looking at the mailing-list traffic and the version control system activity.
- **Quality** of the component. Typically, if a component is already available through embedded build systems, and has a dynamic user community, it probably means that the quality is relatively good.
- **License**. The license of the component must match your licensing constraints. For example, GPL libraries cannot be used in proprietary applications.
- **Technical requirements**. Of course, the component must match your technical requirements. But don't forget that you can improve the existing components if a feature is missing!

Licenses (1)

- All software that are under a free software license give four freedoms to all users
  - Freedom to use
  - Freedom to study
  - Freedom to copy
  - Freedom to modify and distribute modified copies
- See `http://www.gnu.org/philosophy/free-sw.html` for a definition of Free Software
- Open Source software, as per the definition of the Open Source Initiative, are technically similar to Free Software in terms of freedoms
- See `http://www.opensource.org/docs/osd` for the definition of Open Source Software

Licenses (2)

- Free Software licenses fall in two main categories
  - The copyleft licenses
  - The non-copyleft licenses
- The concept of *copyleft* is to ask for reciprocity in the freedoms given to a user.
- The result is that when you receive a software under a copyleft free software license and distribute modified versions of it, you must do so under the same license
  - Same freedoms to the new users
  - It's an incentive to contribute back your changes instead of keeping them secret
- Non-copyleft licenses have no such requirements, and modified versions can be kept proprietary, but they still require attribution

GPL

- **GNU General Public License**
- Covers around 55% of the free software projects
  - Including the Linux kernel, Busybox and many applications
- Is a copyleft license
  - Requires derivative works to be released under the same license
  - Programs linked with a library released under the GPL must also be released under the GPL
- Some programs covered by version 2 (Linux kernel, Busybox and others)
- More and more programs covered by version 3, released in 2007
  - Major change for the embedded market: the requirement that the user must be able to **run** the modified versions on the device, if the device is a *consumer* device

GPL: redistribution

- No obligation when the software is not distributed
  - You can keep your modifications secret until the product delivery
- It is then authorized to distribute binary versions, if one of the following conditions is met:
  - Convey the binary with a copy of the source on a physical medium
  - Convey the binary with a written offer valid for 3 years that indicates how to fetch the source code
  - Convey the binary with the network address of a location where the source code can be found
  - See section 6. of the GPL license
- In all cases, the attribution and the license must be preserved
  - See section 4. and 5.

LGPL

- **GNU Lesser General Public License**
- Covers around 10% of the free software projects
- A copyleft license
  - Modified versions must be released under the same license
  - But, programs linked against a library under the LGPL do not need to be released under the LGPL and can be kept proprietary.
  - However, the user must keep the ability to update the library independently from the program. Dynamic linking is the easiest solution. Statically linked executables are only possible if the developer provides a way to relink with an update (with source code or linkable object files).
- Used instead of the GPL for most of the libraries, including the C libraries
  - Some exceptions: MySQL, or Qt $<=$ 4.4
- Also available in two versions, v2 and v3

Licensing: examples

- You make modifications to the Linux kernel (to add drivers or adapt to your board), to Busybox, U-Boot or other GPL software
  - You must release the modified versions under the same license, and be ready to distribute the source code to your customers
- You make modifications to the C library or any other LGPL library
  - You must release the modified versions under the same license
- You create an application that relies on LGPL libraries
  - You can keep your application proprietary, but you must link dynamically with the LGPL libraries
- You make modifications to a non-copyleft licensed software
  - You can keep your modifications proprietary, but you must still credit the authors

Non-copyleft licenses

- A large family of non-copyleft licenses that are relatively similar in their requirements
- A few examples
  - Apache license (around 4%)
  - BSD license (around 6%)
  - MIT license (around 4%)
  - X11 license
  - Artistic license (around 9 %)

BSD license

```
Copyright (c) <year>, <copyright holder>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the <organization> nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

[...]
```

Is this free software?

- Most of the free software projects are covered by 10 well-known licenses, so it is fairly easy for the majority of project to get a good understanding of the license
- Otherwise, read the license text
- Check Free Software Foundation's opinion `http://www.fsf.org/licensing/licenses/`
- Check Open Source Initiative's opinion `http://www.opensource.org/licenses`

Respect free software licenses

- Free Software is not public domain software, the distributors have obligations due to the licenses
  - **Before** using a free software component, make sure the license matches your project constraints
  - Make sure to keep a complete list of the free software packages you use, the original version numbers you used, and to keep your modifications and adaptations well-separated from the original version.
  - Buildroot and Yocto Project can generate this list for you!
  - Conform to the license requirements before shipping the product to the customers.
- Free Software licenses have been enforced successfully in courts. Organizations which can help:
  - Software Freedom Law Center, `http://www.softwarefreedom.org/`
  - Software Freedom Conservancy, `http://sfconservancy.org/`
- Ask your legal department!

Keeping changes separate (1)

- When integrating existing open-source components in your project, it is sometimes needed to make modifications to them
  - Better integration, reduced footprint, bug fixes, new features, etc.
- Instead of mixing these changes, it is much better to keep them separate from the original component version
  - If the component needs to be upgraded, easier to know what modifications were made to the component
  - If support from the community is requested, important to know how different the component we're using is from the upstream version
  - Makes contributing the changes back to the community possible
- It is even better to keep the various changes made on a given component separate
  - Easier to review and to update to newer versions

Keeping changes separate (2)

- The simplest solution is to use Quilt
  - Quilt is a tool that allows to maintain a stack of patches over source code
  - Makes it easy to add, remove modifications from a patch, to add and remove patches from stack and to update them
  - The stack of patches can be integrated into your version control system
  - `https://savannah.nongnu.org/projects/quilt/`
- Another solution is to use a version control system
  - Import the original component version into your version control system
  - Maintain your changes in a separate branch

Tools for the target device: Networking

ssh server and client: Dropbear

`http://matt.ucc.asn.au/dropbear/dropbear.html`

- Very small memory footprint ssh server for embedded systems
- Satisfies most needs. Both client and server!
- Size: 110 KB, statically compiled with uClibc on i386. (OpenSSH client and server: approx 1200 KB, dynamically compiled with glibc on i386)
- Useful to:
  - Get a remote console on the target device
  - Copy files to and from the target device (`scp` or `rsync -e ssh`).
- An alternative to OpenSSH, used on desktop and server systems.

Benefits of a web server interface

Many network enabled devices can just have a network interface

- Examples: modems / routers, IP cameras, printers...
- No need to develop drivers and applications for computers connected to the device. No need to support multiple operating systems!
- Just need to develop static or dynamic HTML pages (possibly with powerful client-side JavaScript).
  Easy way of providing access to device information and parameters.
- Reduced hardware costs (no LCD, very little storage space needed)

Web servers

- *BusyBox http server*: `http://busybox.net`
  - Tiny: only adds 9 K to BusyBox (dynamically linked with glibc on i386, with all features enabled.)
  - Sufficient features for many devices with a web interface, including CGI, http authentication and script support (like PHP, with a separate interpreter).
  - License: GPL
- Other possibilities: lightweight servers like *Boa*, *thttpd*, *lighttpd*, *nginx*, etc
- Some products are using *Node.js*, which is lightweight enough to be used.

Network utilities (1)

- **avahi** is an implementation of Multicast DNS Service Discovery, that allows programs to publish and discover services on a local network
- **bind**, a DNS server
- **iptables**, the user space tools associated to the Linux firewall, Netfilter
- **iw and wireless tools**, the user space tools associated to Wireless devices
- **netsnmp**, implementation of the SNMP protocol
- **openntpd**, implementation of the Network Time Protocol, for clock synchronization
- **openssl**, a toolkit for SSL and TLS connections

Network utilities (2)

- **pppd**, implementation of the Point to Point Protocol, used for dial-up connections
- **samba**, implements the SMB and CIFS protocols, used by Windows to share files and printers
- **coherence**, a UPnP/DLNA implementation
- **vsftpd**, proftpd, FTP servers

Tools for the target device: System utilities

System utilities

- **dbus**, an inter-application object-oriented communication bus
- **gpsd**, a daemon to interpret and share GPS data
- **libraw1394**, raw access to Firewire devices
- **libusb**, a user space library for accessing USB devices without writing an in-kernel driver
- Utilities for kernel subsystems: **i2c-tools** for I2C, **input-tools** for input, **mtd-utils** for MTD devices, **usbutils** for USB devices

Tools for the target device: Language interpreters

Language interpreters

- Interpreters for the most common scripting languages are available. Useful for
  - Application development
  - Web services development
  - Scripting
- Languages supported
  - Lua
  - Python
  - Perl
  - Ruby
  - TCL
  - PHP

Tools for the target device: Audio, video and multimedia

Audio, video and multimedia

- **GStreamer**, a multimedia framework
  - Allows to decode/encode a wide variety of codecs.
  - Supports hardware encoders and decoders through plugins, proprietary/specific plugins are often provided by SoC vendors.
- **alsa-lib**, the user space tools associated to the ALSA sound kernel subsystem
- Directly using encoding and decoding libraries, if you decide not to use GStreamer:
  libavcodec, libogg, libtheora, libvpx, flac, libvorbis, libmad, libsndfile, speex, etc.

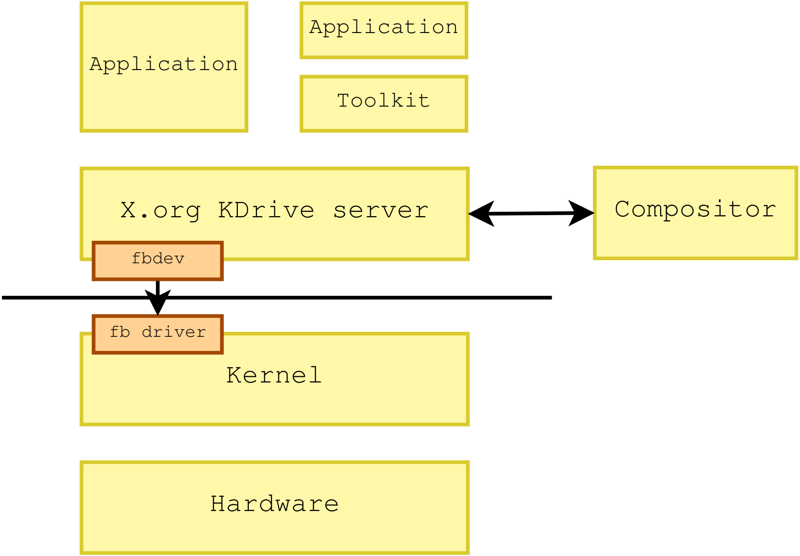
Tools for the target device: Graphical toolkits

Graphical toolkits: “Low-level” solutions and layers

X.org - KDrive

- Stand-alone simplified version of the X server, for embedded systems
  - Formerly know as Tiny-X
  - Kdrive is integrated in the official X.org server
- Works on top of the Linux frame buffer, thanks to the Xfbdev variant of the server
- Real X server
  - Fully supports the X11 protocol: drawing, input event handling, etc.
  - Allows to use any existing X11 application or library
- Actively developed and maintained.
- X11 license
- `http://www.x.org`

Kdrive: architecture



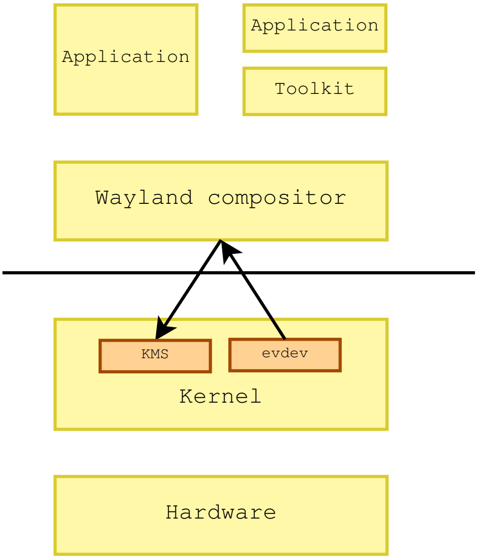
Kdrive: usage

- Can be directly programmed using Xlib / XCB
  - Low-level graphic library, rarely used
- Or, usually used with a toolkit on top of it
  - Gtk
  - Qt
  - Enlightment Foundation Libraries
  - Others: Fltk, WxEmbedded, etc

Wayland

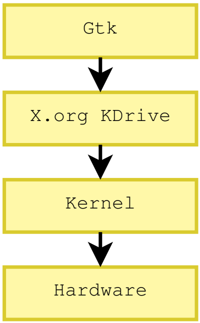
- Intended to be a simpler replacement for X
- *Wayland is a protocol for a compositor to talk to its clients as well as a C library implementation of that protocol.*
- Weston: a minimal and fast reference implementation of a Wayland compositor, and is suitable for many embedded and mobile use cases.
- Not fully deployed yet. However, the ports of Gtk and Qt to Wayland are complete.
- `http://wayland.freedesktop.org/`

Wayland: architecture



Graphical toolkits: “High-level” solutions

- The famous toolkit, providing widget-based high-level APIs to develop graphical applications
- Standard API in C, but bindings exist for various languages: C++, Python, etc.
- Works on top of X.org.
- No windowing system, a lightweight window manager needed to run several applications. Possible solution: Matchbox.
- License: LGPL
- http://www.gtk.org



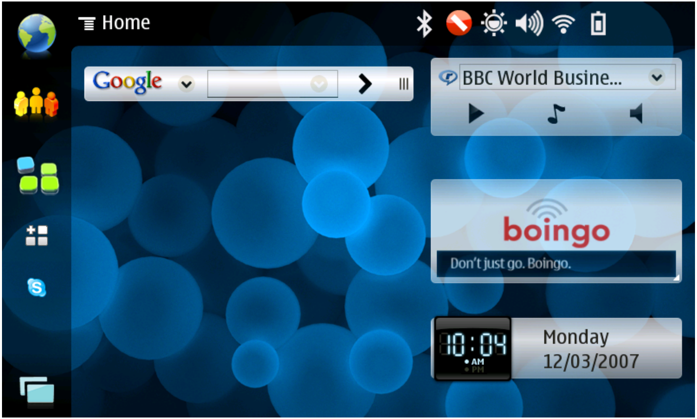
Gtk stack components

- **Glib**, core infrastructure
  - Object-oriented infrastructure GObject
  - Event loop, threads, asynchronous queues, plug-ins, memory allocation, I/O channels, string utilities, timers, date and time, internationalization, simple XML parser, regular expressions
  - Data types: memory slices and chunks, linked lists, arrays, trees, hash tables, etc.
- **Pango**, internationalization of text handling
- **ATK**, accessibility toolkit
- **Cairo**, vector graphics library
- **Gtk+**, the widget library itself
- *The Gtk stack is a complete framework to develop applications*

Gtk examples (1)

Openmoko phone interface

Gtk examples (2)



Maemo tablet / phone interface

GTK is losing traction, however: Mer, the descendent of Maemo, is now implemented in EFL (see next slides).

Qt (1)

- The other famous toolkit, providing widget-based high-level APIs to develop graphical applications
- Implemented in C++
  - the C++ library is required on the target system
  - standard API in C++, but with bindings for other languages
- Works either on top of
  - Framebuffer
  - X11
  - Wayland

Qt (2)

- Qt is more than just a graphical toolkit, it also offers a complete development framework: data structures, threads, network, databases, XML, etc.
- See our presentation *Qt for non graphical applications* presentation at ELCE 2011 (Thomas Petazzoni): `http://j.mp/W4PK85`
- Qt Embedded has an integrated windowing system, allowing several applications to share the same screen
- Very well documented
- Since version 4.5, available under the LGPL, allowing proprietary applications

Qt on the Dash Express navigation system

Qt on the Netflix player by Roku

Other less frequent solutions

- Enlightenment Foundation Libraries (EFL)
  - Very powerful. Supported by Samsung, Intel and Free.fr.
  - Work on top of X or Wayland.
  - `http://www.enlightenment.org/p.php?p=about/efl`

Tools for the target device: Databases

Lightweight database - SQLite

http://www.sqlite.org

- SQLite is a small C library that implements a self-contained, embeddable, lightweight, zero-configuration SQL database engine
- The database engine of choice for embedded Linux systems
  - Can be used as a normal library
  - Can be directly embedded into a application, even a proprietary one since SQLite is released in the public domain

Tools for the target device: Web browsers

WebKit

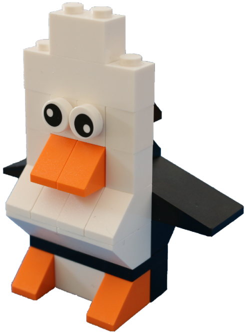
`http://webkit.org/`

- Web browser engine. Application framework that can be used to develop web browsers.
- License: portions in LGPL and others in BSD. Proprietary applications allowed.
- Used by many web browsers: Safari, iPhone and Android default browsers ... Google Chrome now uses a fork of its WebCore component). Used by e-mail clients too to render HTML: `http://trac.webkit.org/wiki/Applications%20using%20WebKit`
- Multiple graphical back-ends: Qt4, GTK, EFL...
- You could use it to create your custom browser.

System building

System building: goal and solutions

- Goal
  - Integrate all the software components, both third-party and in-house, into a working root filesystem
  - It involves the download, extraction, configuration, compilation and installation of all components, and possibly fixing issues and adapting configuration files
- Several solutions
  - Manually
  - System building tools
  - Distributions or ready-made filesystems



Penguin picture: http://bit.ly/1PwDklz

System building: manually

- Manually building a target system involves downloading, configuring, compiling and installing all the components of the system.
- All the libraries and dependencies must be configured, compiled and installed in the right order.
- Sometimes, the build system used by libraries or applications is not very cross-compile friendly, so some adaptations are necessary.
- There is no infrastructure to reproduce the build from scratch, which might cause problems if one component needs to be changed, if somebody else takes over the project, etc.

System building: manually (2)

- Manual system building is not recommended for production projects
- However, using automated tools often requires the developer to dig into specific issues
- Having a basic understanding of how a system can be built manually is therefore very useful to fix issues encountered with automated tools
  - We will first study manual system building, and during a practical lab, create a system using this method
  - Then, we will study the automated tools available, and use one of them during a lab

System foundations

- A basic root file system needs at least
  - A traditional directory hierarchy, with `/bin`, `/etc`, `/lib`, `/root`, `/usr/bin`, `/usr/lib`, `/usr/share`, `/usr/sbin`, `/var`, `/sbin`
  - A set of basic utilities, providing at least the `init` program, a shell and other traditional Unix command line tools. This is usually provided by *Busybox*
  - The C library and the related libraries (thread, math, etc.) installed in `/lib`
  - A few configuration files, such as `/etc/inittab`, and initialization scripts in `/etc/init.d`
- On top of this foundation common to most embedded Linux system, we can add third-party or in-house components

Target and build spaces

- The system foundation, Busybox and C library, are the core of the target root filesystem
- However, when building other components, one must distinguish two directories
  - The *target* space, which contains the target root filesystem, everything that is needed for **execution** of the application
  - The *build* space, which will contain a lot more files than the *target* space, since it is used to keep everything needed to **compile** libraries and applications. So we must keep the headers, documentation, and other configuration files

Build systems

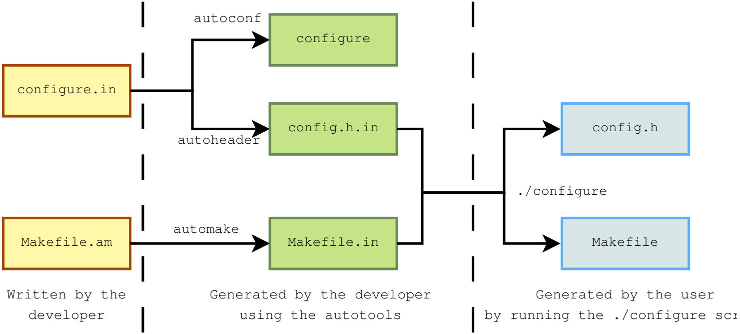
Each open-source component comes with a mechanism to configure, compile and install it

- A basic `Makefile`
  - Need to read the `Makefile` to understand how it works and how to tweak it for cross-compilation
- A build system based on the *Autotools*
  - As this is the most common build system, we will study it in details
- CMake, `http://www.cmake.org/`
  - Newer and simpler than the *autotools*. Used by large projects such as KDE or Second Life
- Scons, `http://www.scons.org/`
- Waf, `http://code.google.com/p/waf/`
- Other manual build systems

Autotools and friends

- A family of tools, which associated together form a complete and extensible build system
  - **autoconf** is used to handle the configuration of the software package
  - **automake** is used to generate the Makefiles needed to build the software package
  - **pkgconfig** is used to ease compilation against already installed shared libraries
  - **libtool** is used to handle the generation of shared libraries in a system-independent way
- Most of these tools are old and relatively complicated to use, but they are used by a majority of free software packages today. One must have a basic understanding of what they do and how they work.

automake / autoconf / autoheader



automake / autoconf

- Files written by the developer
  - `configure.in` describes the configuration options and the checks done at configure time
  - `Makefile.am` describes how the software should be built
- The `configure` script and the `Makefile.in` files are generated by `autoconf` and `automake` respectively.
  - They should never be modified directly
  - They are usually shipped pre-generated in the software package, because there are several versions of `autoconf` and `automake`, and they are not completely compatible
- The `Makefile` files are generated at configure time, before compiling
  - They are never shipped in the software package.

Configuring and compiling: native case

- The traditional steps to configure and compile an autotools based package are
  - Configuration of the package
    `./configure`
  - Compilation of the package
    `make`
  - Installation of the package
    `make install`
- Additional arguments can be passed to the `./configure` script to adjust the component configuration.
- Only the `make install` needs to be done as root if the installation should take place system-wide

Configuring and compiling: cross case (1)

- For cross-compilation, things are a little bit more complicated.
- At least some of the environment variables `AR`, `AS`, `LD`, `NM`, `CC`, `GCC`, `CPP`, `CXX`, `STRIP`, `OBJCOPY` must be defined to point to the proper cross-compilation tools. The host tuple is also by default used as prefix.
- `configure` script arguments:
  - `--host`: mandatory but a bit confusing. Corresponds to the *target* platform the code will run on. Example: `--host=arm-linux`
  - `--build`: build system. Automatically detected.
  - `--target` is only for tools generating code.
- It is recommended to pass the `--prefix` argument. It defines from which location the software will run in the target environment. Usually, `/usr` is fine.

Configuring and compiling: cross case (2)

- If one simply runs `make install`, the software will be installed in the directory passed as `--prefix`. For cross-compiling, one must pass the `DESTDIR` argument to specify where the software must be installed.
- Making the distinction between the prefix (as passed with `--prefix` at configure time) and the destination directory (as passed with `DESTDIR` at installation time) is very important.
- Example:

```
export PATH=/usr/local/arm-linux/bin:$PATH
export CC=arm-linux-gcc
export STRIP=arm-linux-strip
./configure --host=arm-linux --prefix=/usr
make
make DESTDIR=$HOME/work/rootfs install
```

Installation (1)

- The autotools based software packages provide both a `install` and `install-strip` make targets, used to install the software, either stripped or unstripped.
- For applications, the software is usually installed in `<prefix>/bin`, with configuration files in `<prefix>/etc` and data in `<prefix>/share/<application>/`
- The case of libraries is a little more complicated:
  - In `<prefix>/lib`, the library itself (a `.so.<version>`), a few symbolic links, and the libtool description file (a `.la` file)
  - The *pkgconfig* description file in `<prefix>/lib/pkgconfig`
  - Include files in `<prefix>/include/`
  - Sometimes a `<libname>-config` program in `<prefix>/bin`
  - Documentation in `<prefix>/share/man` or `<prefix>/share/doc/`

Installation (2)

Contents of `usr/lib` after installation of *libpng* and *zlib*

- *libpng* libtool description files
  `./lib/libpng12.la`
  `./lib/libpng.la -> libpng12.la`
- *libpng* static version
  `./lib/libpng12.a`
  `./lib/libpng.a -> libpng12.a`
- *libpng* dynamic version
  `./lib/libpng.so.3.32.0`
  `./lib/libpng12.so.0.32.0`
  `./lib/libpng12.so.0 -> libpng12.so.0.32.0`
  `./lib/libpng12.so -> libpng12.so.0.32.0`
  `./lib/libpng.so -> libpng12.so`
  `./lib/libpng.so.3 -> libpng.so.3.32.0`
- *libpng* pkg-config description files
  `./lib/pkgconfig/libpng12.pc`
  `./lib/pkgconfig/libpng.pc -> libpng12.pc`
- *zlib* dynamic version
  `./lib/libz.so.1.2.3`
  `./lib/libz.so -> libz.so.1.2.3`
  `./lib/libz.so.1 -> libz.so.1.2.3`

Installation in the build and target spaces

- From all these files, everything except documentation is necessary to build an application that relies on libpng.
  - These files will go into the *build space*
- However, only the library `.so` binaries in `<prefix>/lib` and some symbolic links are needed to execute the application on the target.
  - Only these files will go in the *target space*
- The build space must be kept in order to build other applications or recompile existing applications.

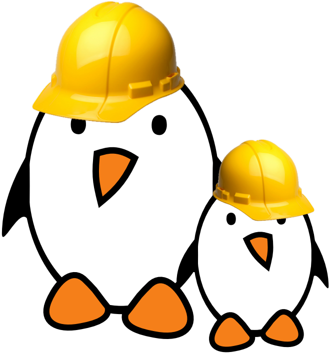
pkg-config

- `pkg-config` is a tool that allows to query a small database to get information on how to compile programs that depend on libraries
- The database is made of `.pc` files, installed by default in `<prefix>/lib/pkgconfig/`.
- `pkg-config` is used by the `configure` script to get the library configurations
- It can also be used manually to compile an application: `arm-linux-gcc -o test test.c $(pkg-config --libs --cflags thelib)`
- By default, `pkg-config` looks in `/usr/lib/pkgconfig` for the `*.pc` files, and assumes that the paths in these files are correct.
- `PKG_CONFIG_PATH` allows to set another location for the `*.pc` files and `PKG_CONFIG_SYSROOT_DIR` to prepend a prefix to the paths mentioned in the `.pc` files.

Let's find the libraries

- When compiling an application or a library that relies on other libraries, the build process by default looks in `/usr/lib` for libraries and `/usr/include` for headers.
- The first thing to do is to set the `CFLAGS` and `LDFLAGS` environment variables:
  ```
  export CFLAGS=-I/my/build/space/usr/include/
  export LDFLAGS=-L/my/build/space/usr/lib
  ```
- The libtool files (`.la` files) must be modified because they include the absolute paths of the libraries:
  ```
  - libdir='/usr/lib'
  + libdir='/my/build/space/usr/lib'
  ```
- The `PKG_CONFIG_PATH` environment variable must be set to the location of the .pc files and the `PKG_CONFIG_SYSROOT_DIR` variable must be set to the build space directory.

Practical lab - Manual cross-compiling



- Manually cross-compiling applications and libraries
- Learning about common techniques and issues.

System building tools: principle

- Different tools are available to automate the process of building a target system, including the kernel, and sometimes the toolchain.
- They automatically download, configure, compile and install all the components in the right order, sometimes after applying patches to fix cross-compiling issues.
- They already contain a large number of packages, that should fit your main requirements, and are easily extensible.
- The build becomes reproducible, which allows to easily change the configuration of some components, upgrade them, fix bugs, etc.

Available system building tools

Large choice of tools

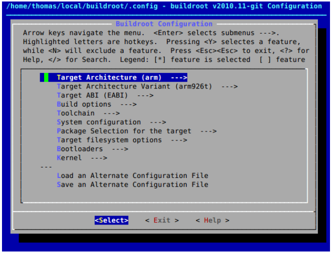
- **Buildroot**, developed by the community http://www.buildroot.net See our dedicated course and training materials: http://free-electrons.com/training/buildroot/
- **PTXdist**, developed by Pengutronix http://pengutronix.de/software/ptxdist/
- **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project http://www.openwrt.org
- **LTIB**. Good support for Freescale boards, but small community http://ltib.org/
- **OpenEmbedded**, more flexible but also far more complicated http://www.openembedded.org, its industrialized version **Yocto Project** and vendor-specific derivatives such as **Arago**.
  See our dedicated course and training materials: http://free-electrons.com/training/yocto/.
- Vendor specific tools (silicon vendor or embedded Linux vendor)

Buildroot (1)

- Allows to build a toolchain, a root filesystem image with many applications and libraries, a bootloader and a kernel image
  - Or any combination of the previous items
- Supports building uClibc, glibc and musl toolchains, either built by Buildroot, or external
- Over 1200+ applications or libraries integrated, from basic utilities to more elaborate software stacks: X.org, GStreamer, Qt, Gtk, WebKit, Python, PHP, etc.
- Good for small to medium embedded systems, with a fixed set of features
  - No support for generating packages (`.deb` or `.ipk`)
  - Needs complete rebuild for most configuration changes.
- Active community, releases published every 3 months.

Buildroot (2)

- Configuration takes place through a `*config` interface similar to the kernel
  `make menuconfig`
- Allows to define
  - Architecture and specific CPU
  - Toolchain configuration
  - Set of applications and libraries to integrate
  - Filesystem images to generate
  - Kernel and bootloader configuration
- Build by just running
  `make`



Buildroot: adding a new package (1)

- A package allows to integrate a user application or library to Buildroot
- Each package has its own directory (such as `package/gqview`). This directory contains:
  - A `Config.in` file (mandatory), describing the configuration options for the package. At least one is needed to enable the package. This file must be sourced from `package/Config.in`
  - A `gqview.mk` file (mandatory), describing how the package is built.
  - Patches (optional). Each file of the form `gqview-*.patch` will be applied as a patch.

- For a simple package with a single configuration option to enable/disable it, the `Config.in` file looks like:

```
config BR2_PACKAGE_GQVIEW
        bool "gqview"
        depends on BR2_PACKAGE_LIBGTK2
        help
          GQview is an image viewer for Unix operating systems

          http://prdownloads.sourceforge.net/gqview
```

- It must be sourced from `package/Config.in`:

```
source "package/gqview/Config.in"
```

Buildroot: adding new package (3)

- Create the gqview.mk file to describe the build steps

```
GQVIEW_VERSION = 2.1.5
GQVIEW_SOURCE = gqview-$(GQVIEW_VERSION).tar.gz
GQVIEW_SITE = http://prdownloads.sourceforge.net/gqview
GQVIEW_DEPENDENCIES = host-pkgconf libgtk2
GQVIEW_CONF_ENV = LIBS="-lm"

$(eval $(autotools-package))
```

- The package directory and the prefix of all variables must be identical to the suffix of the main configuration option `BR2_PACKAGE_GQVIEW`
- The `autotools-package` infrastructure knows how to build autotools packages. A more generic `generic-package` infrastructure is available for packages not using the autotools as their build system.

OpenEmbedded / Yocto Project

- The most versatile and powerful embedded Linux build system
  - A collection of recipes (`.bb` files)
  - A tool that processes the recipes: `bitbake`
- Integrates 2000+ application and libraries, is highly configurable, can generate binary packages to make the system customizable, supports multiple versions/variants of the same package, no need for full rebuild when the configuration is changed.
- Configuration takes place by editing various configuration files
- Good for larger embedded Linux systems, or people looking for more configurability and extensibility
- Drawbacks: very steep learning curve, very long first build.

Distributions - Debian

Debian GNU/Linux, `http://www.debian.org`

- Provides the easiest environment for quickly building prototypes and developing applications. Countless runtime and development packages available.
- But probably too costly to maintain and unnecessarily big for production systems.
- Available on ARM (`armel`, `armhf`, `arm64`), MIPS and PowerPC architectures
- Software is compiled natively by default.

Distributions - Others

Fedora

- `http://fedoraproject.org/wiki/Architectures/ARM`
- Supported on various recent ARM boards (such as Beaglebone Black). Pidora supports Raspberry Pi too.
- Supports QEMU emulated ARM boards too (Versatile Express board)
- Shipping the same version as for desktops!

Ubuntu

- Had some releases for ARM mobile multimedia devices, but stopped at version 12.04. Now focusing on ARM servers only.

Embedded distributions

Distributions designed for specific types of devices

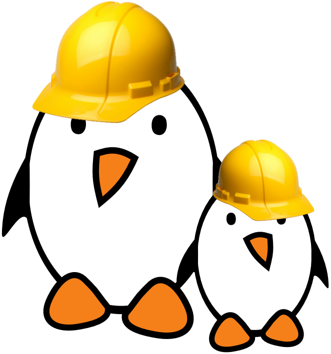
- **Android**: `http://www.android.com/`
  Google's distribution for phones and tablet PCs. Except the Linux kernel, very different user space than other Linux distributions. Very successful, lots of applications available (many proprietary).
- **Ångström**:
  `http://www.angstrom-distribution.org/`
  Produces nightly built images for a nice list of ARM and x86 systems (see `http://dominion.thruhere.net/angstrom/nightlies/`

ENDO CODE

Application frameworks

Not real distributions you can download. Instead, they implement middleware running on top of the Linux kernel and allowing to develop applications.

- **Mer**: http://merproject.org/
  Fork from the Meego project.
  Targeting mobile devices.
  Supports x86, ARM and MIPS.
  See http://en.wikipedia.org/wiki/Mer_(software_distribution)
- **Tizen**: https://www.tizen.org/
  Targeting smartphones, tablets, netbooks, smart TVs and In Vehicle Infotainment devices.
  Supported by big phone manufacturers and operators
  HTML5 base application framework.
  See http://en.wikipedia.org/wiki/Tizen



- Rebuild the same system, this time with Buildroot.
- See how easier it gets!