Embedded Linux Introduction

Li Zhiguang

June 28, 2013

Rights To Copy

Attribution – ShareAlike 3.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

- **Attribution.** You must give the original author credit.
- **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.
License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

About Me

- Embedded Linux
- Android
- Free Software / Open Source
- Totally work with Linux

The slides

The slides is mainly copied from http://free-electrons.com/training/, please join me to thank the excellent work of Free-electrons.

What is embedded Linux?

- Embedded Linux is the usage of the **Linux kernel** and various **open-source** components in embedded systems
- Android is not embedded Linux

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 locking 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**

Processor and architecture (1)

- The Linux kernel and most other architecture-dependent component 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 and multimedia)
  - 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 busses

- I2C/SPI/CAN/1-wire/SDIO/USB
- Ethernet, Wifi, Bluetooth, CAN, etc.
- IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
- Firewalling, advanced routing, multicast

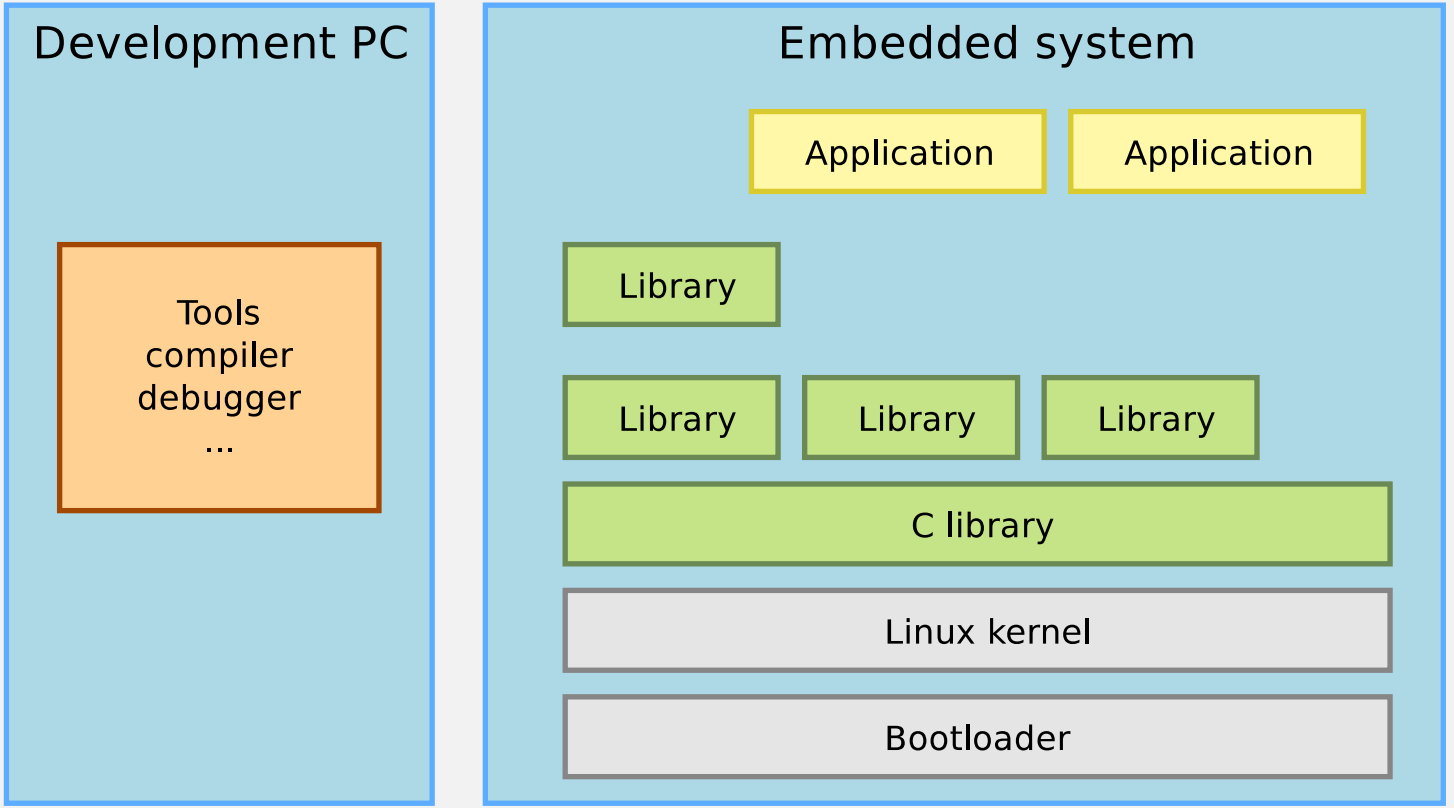
Types of hardware platforms

- **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**, a new trend to make a particular SoC popular and easily available. Those 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.**

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 userspace applications
- **C library** The interface between the kernel and the userspace 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
  - Purpose of our *Kernel Development* training
- **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 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 Free Electrons 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 those 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.

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 have chosen Ubuntu, as it is a **widely used and easy to use** desktop Linux distribution
- The Ubuntu setup on the training laptops has intentionally been left untouched after the normal installation process. Learning embedded Linux is also about learning the tools needed on the development workstation!

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 those 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`

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>`

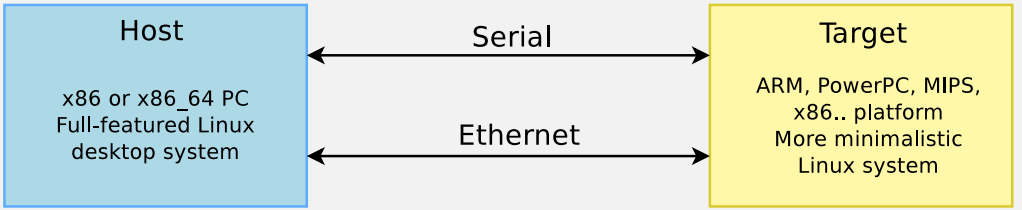
Managing software packages (2)

- 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/

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

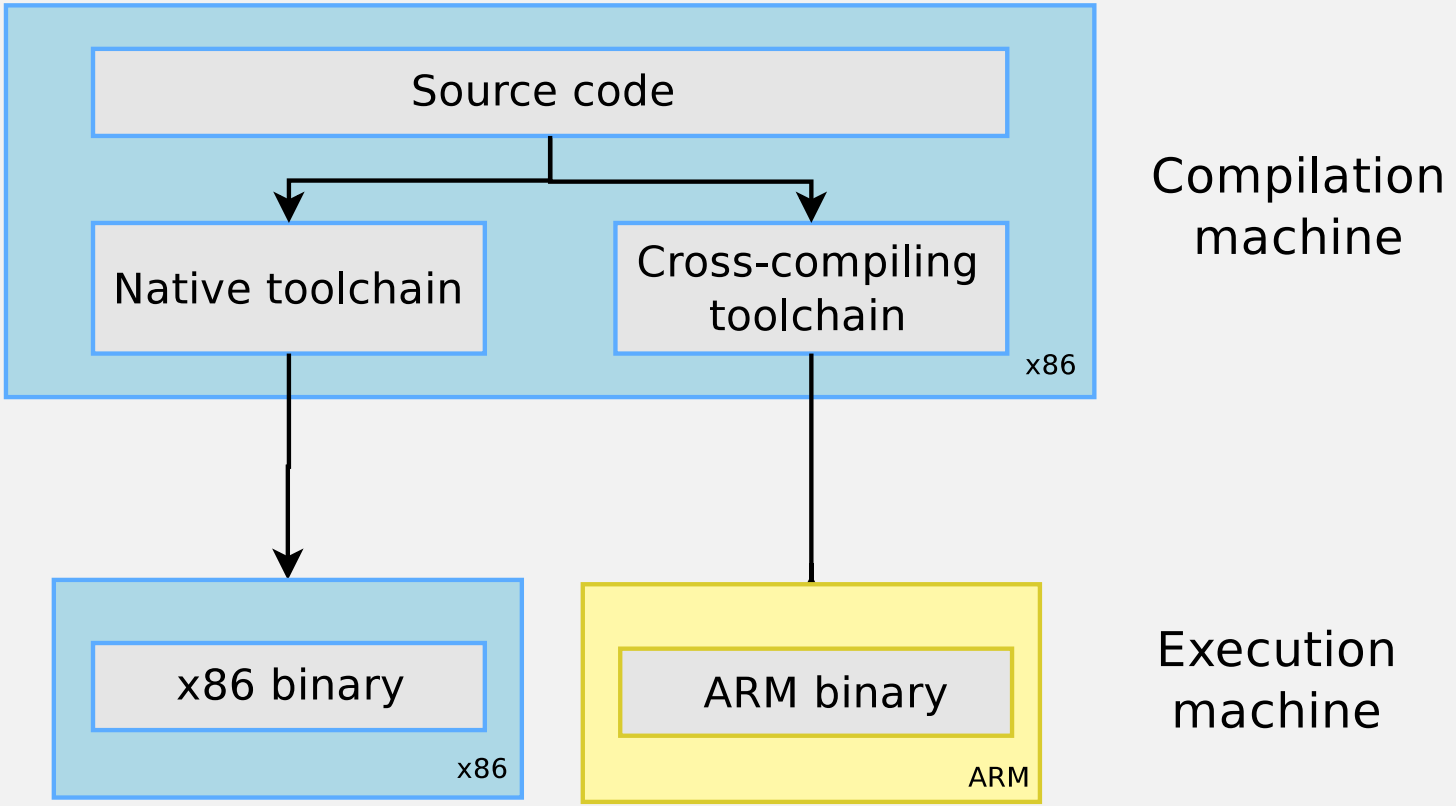
Command line tips

- 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.

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)



Machines in build procedures

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

Different toolchain build procedures

Native build

used to build the normal gcc
of a workstation

Cross build

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

The most common case in embedded development

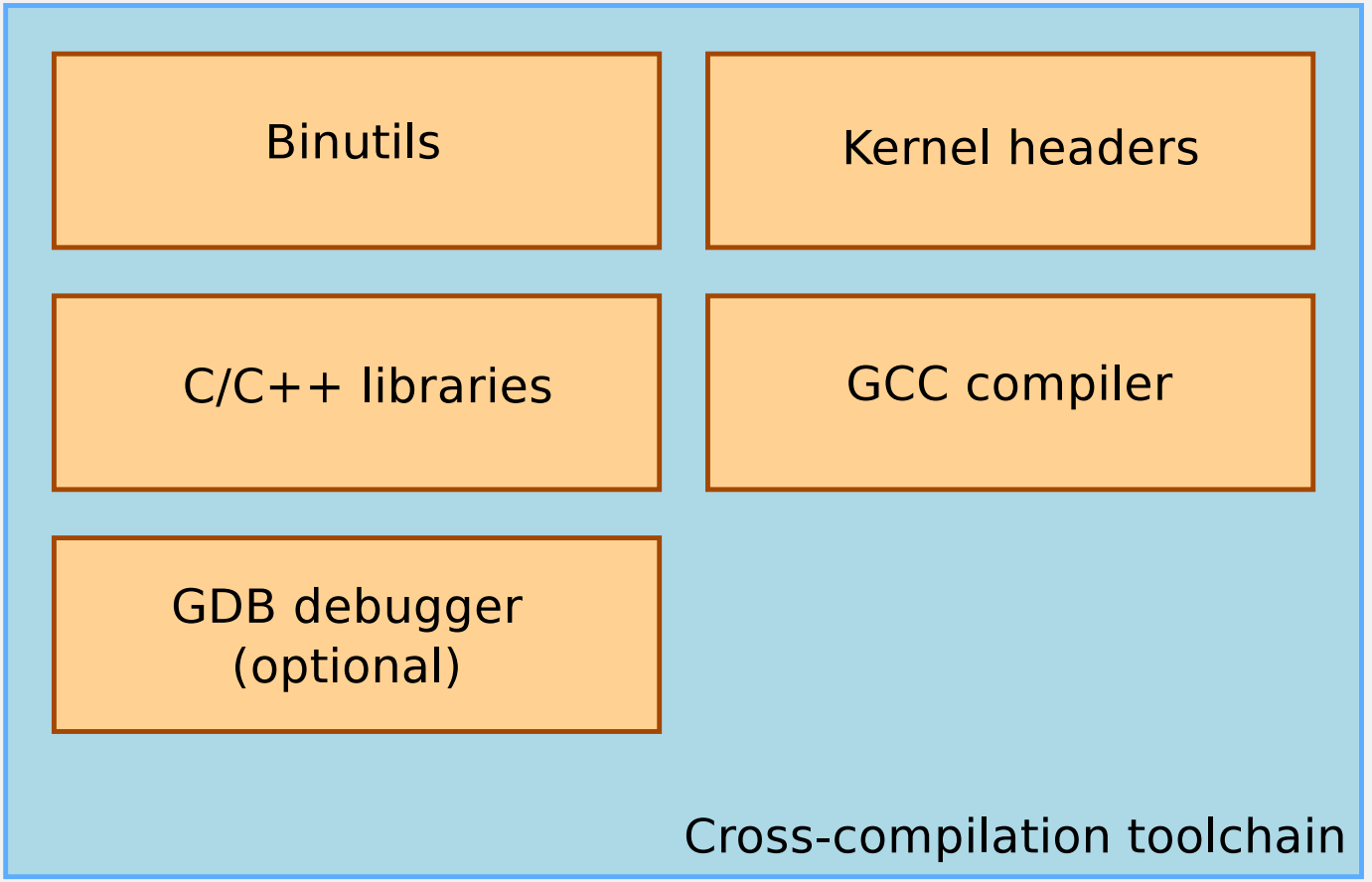
Cross-native build

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

Canadian build

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

Components

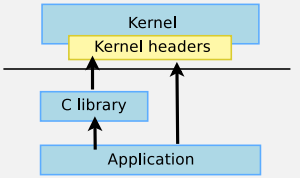

Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger
(optional)
Cross-compilation toolchain

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)

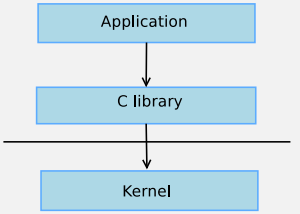
- The kernel-to-userspace ABI is **backward compatible**
  - Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
  - Binaries generated with a toolchain using kernel headers 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*, *eglibc*, *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.

glibc

- 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
- Quite big for small embedded systems: approx 2.5 MB on ARM (version 2.9 - `libc`: 1.5 MB, `libm`: 750 KB)
- http://www.gnu.org/software/libc/

uClibc

- License: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface
  - Works only with Linux/uClinux, works on most embedded architectures
  - No stable ABI, different ABI depending on the library configuration
  - Focus on size rather than performance
  - Small compile time
- http://www.uclibc.org/

uClibc (2)

- Most of the applications compile with uClibc. 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 very new, fixed Name Service Switch functionality, etc.
- Used on a large number of production embedded products, including consumer electronic devices
- Actively maintained, large developer and user base
- Supported and used by MontaVista, TimeSys and Wind River.

Honey, I shrunk the programs!

- Executable size comparison on ARM, tested with *glibc* 2.9 and *uClibc* 0.9.30.1
- Plain ``hello world'' program (stripped)
  - With shared libraries: 5.6 KB with *glibc*, 5.4 KB with *uClibc*
  - With static libraries: 472 KB with *glibc*, 18 KB with *uClibc*
- Busybox (stripped)
  - With shared libraries: 245 KB with *glibc*, 231 KB with *uClibc*
  - With static libraries: 843 KB with *glibc*, 311 KB with *uClibc*

eglibc

- *Embedded glibc*, under the LGPL
- Variant of the GNU C Library (GLIBC) designed to work well on embedded systems

- Strives to be source and binary compatible with GLIBC
- eglibc's goals include reduced footprint, configurable components, better support for cross-compilation and cross-testing.
- Can be built without support for NIS, locales, IPv6, and many other features.
- Supported by a consortium, with Freescale, MIPS, MontaVista and Wind River as members.
- The Debian distribution has switched to eglibc too, http://blog.aurel32.net/?p=47
- http://www.eglibc.org

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 need specially written programs and applications
- Choices:
  - Dietlibc, http://www.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.

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 userspace 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

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
- Determine what toolchain you need: CPU, endianism, C library, component versions, ABI, soft float or hard float, etc.
- Check whether the available toolchains match your requirements.
- Possible choices
  - Sourcery CodeBench toolchains
  - Linaro toolchains
  - More references at http://elinux.org/Toolchains

Sourcery CodeBench

- *CodeSourcery* was a a company with an extended expertise on free software toolchains: gcc, gdb, binutils and glibc. It has been bought by *Mentor Graphics*, which continues to provide similar services and products
- They sell toolchains with support, but they also provide a "*Lite*" version, which is free and usable for commercial products
- They have toolchains available for
  - ARM
  - MIPS
  - PowerPC
  - SuperH
  - x86

- Be sure to use the GNU/Linux versions. The EABI versions are for bare-metal development (no operating system)

Linaro toolchains

- Linaro contributes to improving mainline gcc on ARM, in particular by hiring CodeSourcery developers.

- For people who can't wait for the next releases of gcc, Linaro releases modified sources of stable releases of gcc, with these optimizations for ARM (mainly for recent Cortex A CPUs).
- As any gcc release, these sources can be used by build tools to build their own binary toolchains (Buildroot, OpenEmbedded...) This allows to support glibc, uClibc and eglibc.
- https://wiki.linaro.org/WorkingGroups/ToolChain
- Binary packages are available for Ubuntu users, https://launchpad.net/~linaro-maintainers/+archive/toolchain

Installing and using a pre-compiled toolchain

- Follow the installation procedure proposed by the vendor
- Usually, it is simply a matter of extracting a tarball wherever you want.
- Then, add the path to toolchain binaries in your `PATH`:
  `export PATH=/path/to/toolchain/bin/:$PATH`
- Finally, compile your applications
  `PREFIX-gcc -o foobar foobar.c`
- `PREFIX` depends on the toolchain configuration, and allows to distinguish cross-compilation tools from native compilation utilities

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, eglibc, hard and soft float, many architectures
  - Actively maintained
  - http://crosstool-ng.org/

Toolchain building utilities (3)

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

- **Buildroot**
  - Makefile-based, has a Crosstool-NG back-end, maintained by the community
  - http://www.buildroot.net
- **PTXdist**
  - Makefile-based, uClibc or glibc, maintained mainly by *Pengutronix*
  - http://www.pengutronix.de/software/ptxdist/index_en.html
- **OpenEmbedded**

- The feature-full, but more complicated building system
- http://www.openembedded.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 can 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.)

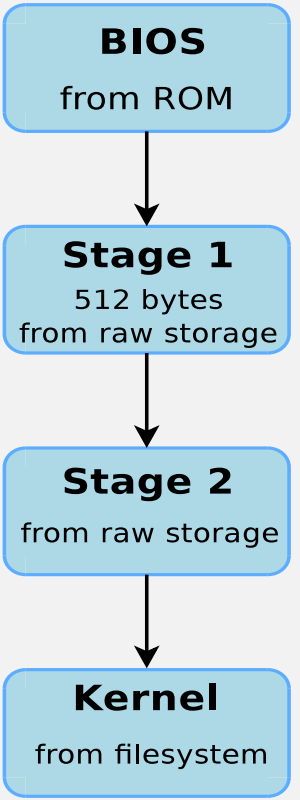
- CodeSourcery ARM toolchain are multilib, the sysroots are in `arm-none-linux-gnueabi/libc/`, `arm-none-linux-gnueabi/libc/armv4t` `arm-none-linux-gnueabi/libc/thumb2`
- Crosstool-NG toolchains are never multilib, the sysroot is in `arm-unknown-linux-uclibcgnueabi/sysroot`

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.

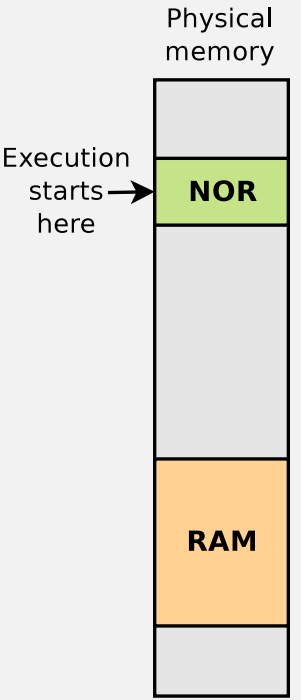


Bootloaders on x86 (2)

- 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

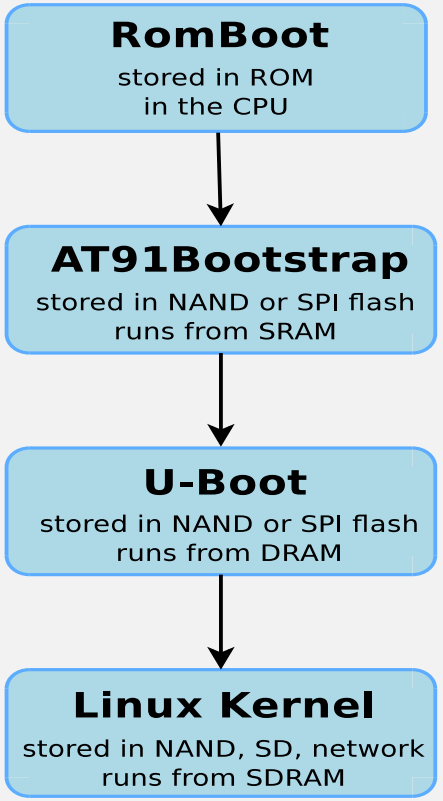
- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader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

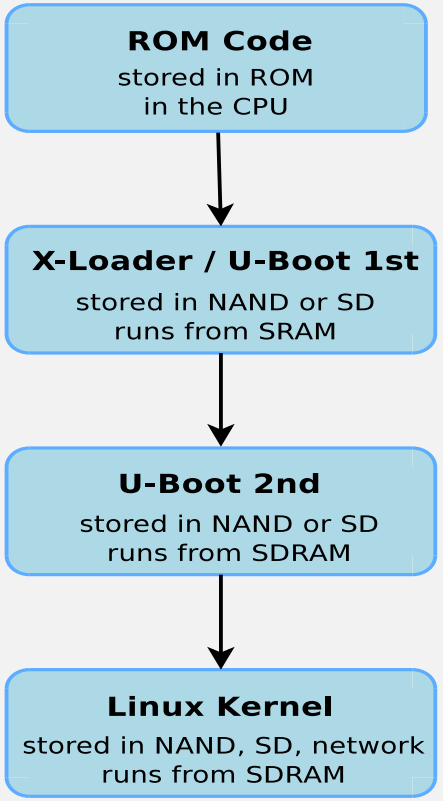
- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, "ROM code" on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC,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

Booting on ARM Atmel AT91



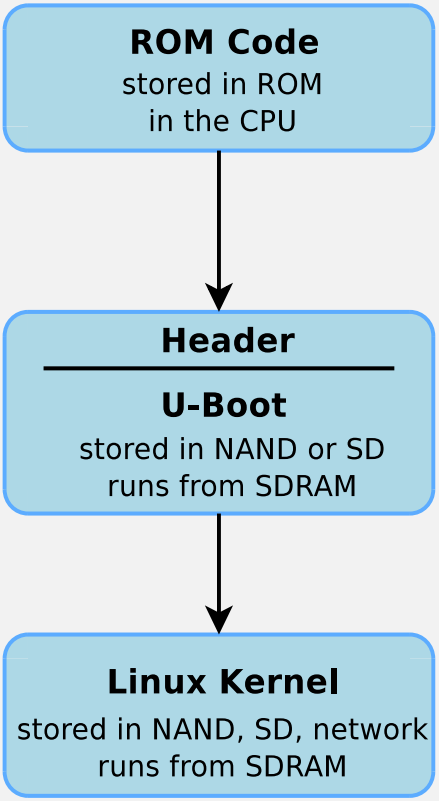
- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- **AT91Bootstrap**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads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

Booting on ARM OMAP3



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

Booting on Marvell SoC



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

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.

U-Boot

U-Boot is a typical free software project

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

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_config`
  - Where `BOARDNAME` is the name of the board, as visible in the `boards.cfg` file (first column).
- 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 #255000029ff800001685802120290 11
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`
- `bootm <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

- 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 to store U-Boot environment, defined in the board configuration file
- Commands to manipulate environment variables:
  - `printenv`, shows all variables
  - `printenv <variable-name>`, shows the value of one variable
  - `setenv <variable-name> <variable-value>`, changes the value of a variable, only in RAM
  - `saveenv`, saves the current state of the environment to flash

Environment variables commands (2)

```
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=10.0.0.2
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, 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
- `bootdelay`, the delay in seconds before which U-Boot runs `bootcmd`
- `autostart`, if yes, U-Boot starts automatically an image that has been loaded into memory

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 'mmc init 0; if fatload mmc 0 80000000
    boot.ini; then source; else if fatload mmc 0 80000000
    uImage;       then run mmc-bootargs; bootm; 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 support 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

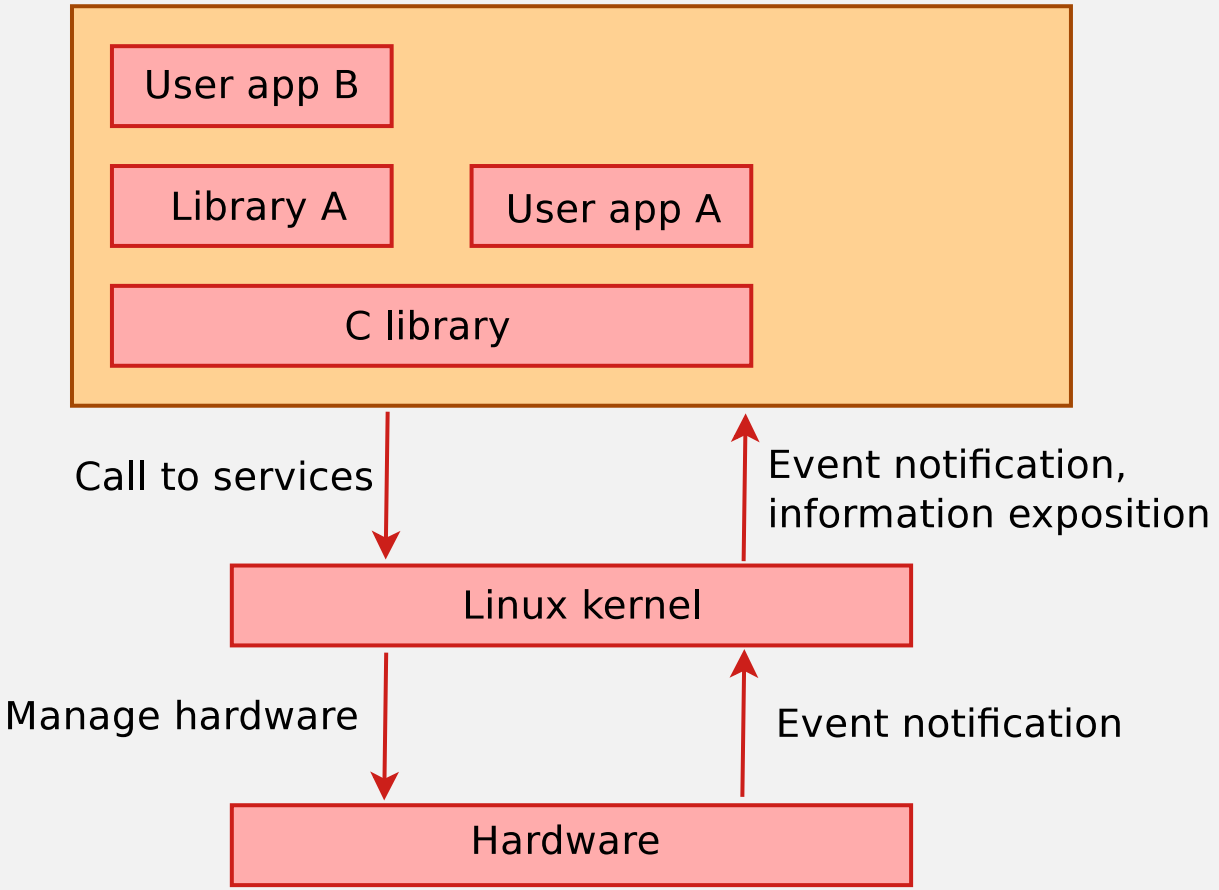
TFTP

- Network transfer from the development workstation and 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

U-boot mkimage

- The kernel image that U-Boot loads and boots must be prepared, so that a U-Boot specific header is added in front of the image
  - This header gives details such as the image size, the expected load address, the compression type, etc.
- This is done with a tool that comes in U-Boot, `mkimage`
- Debian / Ubuntu: just install the `u-boot-tools` package.
- Or, compile it by yourself: simply configure U-Boot for any board of any architecture and compile it. Then install `mkimage`:
  `cp tools/mkimage /usr/local/bin/`
- The special target `uImage` of the kernel Makefile can then be used to generate a kernel image suitable for U-Boot.

Linux kernel in the system



History

- 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, hundreds of people contribute to each kernel release, individuals or companies big and small.

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..

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.

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`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `alpha`, `arm64`, `ia64`, `sparc64`, `tile`
- 32/64 bit architectures
  Examples: `powerpc`, `x86`, `sh`
- Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

System calls

- The main interface between the kernel and userspace 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 userspace applications usually never make a system call directly but rather use the corresponding C library function

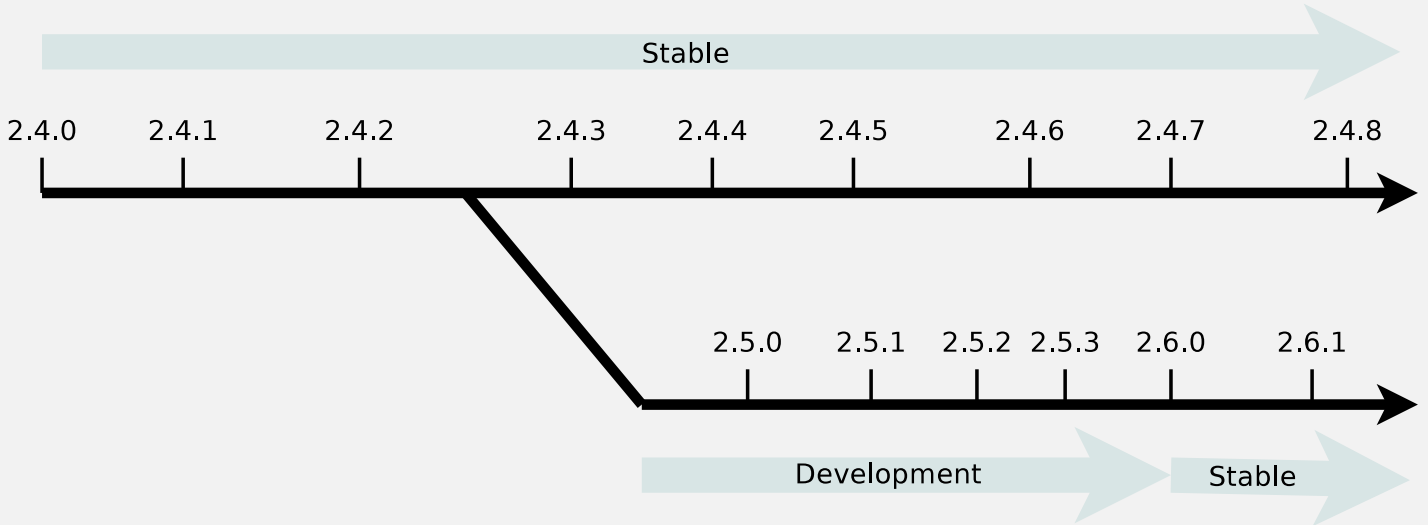
Virtual filesystems

- Linux makes system and kernel information available in user-space through virtual filesystems.
- Virtual filesystems allow applications to see directories and files that do not exist on any real storage: they are created on the fly by the kernel
- The two most important virtual 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.

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 (1)

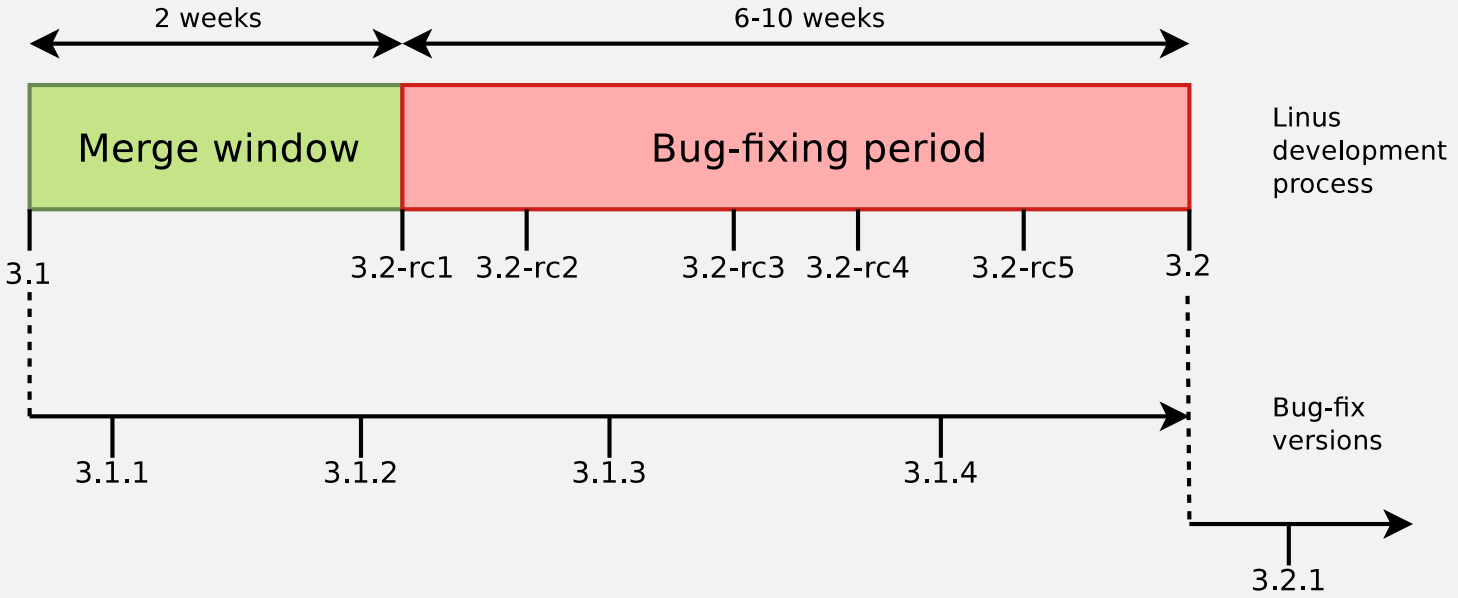
- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- So far, there was no need to create a new development branch (such as `2.7`), which would massively break compatibility with the stable branch.
- Thanks to this, **more features are released to users at a faster pace**.

Changes since Linux 2.6 (2)

Since 2.6.14, the kernel developers agreed on the following development model:

- After the release of a `2.6.x` version, a two-weeks merge window opens, during which major additions are merged.
- The merge window is closed by the release of test version `2.6.(x+1)-rc1`
- The bug fixing period opens, for 6 to 10 weeks.
- At regular intervals during the bug fixing period, `2.6.(x+1)-rcY` test versions are released.
- When considered sufficiently stable, kernel `2.6.(x+1)` is released, and the process starts again.

Merge and bug fixing windows



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.

mainline:	**3.9-rc7**	2013-04-15
stable:	**3.8.8**	2013-04-17
stable:	**3.7.10 [EOL]**	2013-02-27
stable:	**3.6.11 [EOL]**	2012-12-17
longterm:	**3.4.41**	2013-04-17
longterm:	**3.2.43**	2013-04-10
longterm:	**3.0.74**	2013-04-17
longterm:	**2.6.34.14**	2013-01-16
linux-next:	**next-20130419**	2013-04-19

- 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")

New 3.x branch

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- There is no change to the development model, only a change to the numbering scheme
  - Official kernel versions will be named `3.x` (`3.0`, `3.1`, `3.2`, etc.)
  - Stabilized versions will be named `3.x.y` (`3.0.2`, `3.4.3`, etc.)
  - It effectively only removes a digit compared to the previous numbering scheme

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 those
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_sup
    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-electr
    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
  - http://lwn.net
  - http://linuxfr.org, for French readers

Location of kernel sources

- The official version of the Linux kernel, as released by Linus Torvalds is available at http://www.kernel.org
  - This version follows the well-defined development model of the kernel
  - However, it may not contain the latest development from a specific area, due to the organization of the development model and because features in development might not be ready for mainline inclusion
- 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.)
  - They generally don't release official versions, only development trees are available

Linux kernel size (1)

- Linux 3.1 sources:
  Raw size: 434 MB (39,400 files, approx 14,800,000 lines)
  `gzip` compressed tar archive: 93 MB
  `bzip2` compressed tar archive: 74 MB (better)
  `xz` compressed tar archive: 62 MB (best)
- Minimum Linux 2.6.29 compiled kernel size with `CONFIG_EMBEDDED`, for a kernel that boots a QEMU PC (IDE hard drive, ext2 filesystem, ELF executable support):
  532 KB (compressed), 1325 KB (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 3.2.

- `drivers/`: 53.65%
- `arch/`: 20.78%
- `fs/`: 6.88%
- `sound/`: 5.04%
- `net/`: 4.33%
- `include/`: 3.80%
- `firmware/`: 1.46%
- `kernel/`: 1.10%

- `tools/`: 0.56%
- `mm/`: 0.53%
- `scripts/`: 0.44%
- `security/`: 0.40%
- `crypto/`: 0.38%
- `lib/`: 0.30%
- `block/`: 0.13%
- `ipc/`: 0.04%
- `virt/`: 0.03%
- `init/`: 0.03%

- `samples/`: 0.02%
- `usr/`: 0%

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/v3.0/linux-3.1.3.tar.xz
  - Extract command:
    `tar Jxf linux-3.1.3.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/v3.0/patch-3.1.xz

(3.0 to 3.1)
http://www.kernel.org/pub/linux/kernel/v3.0/patch-3.1.3.xz
(3.1 to 3.1.3)

- 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

Linux patches...

- Always applied to the `x.y.<z-1>` version
  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-3.0
xzcat ../patch-3.1.xz | patch -p1
xzcat ../patch-3.1.3.xz | patch -p1
cd ..; mv linux-3.0 linux-3.1.3
```

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**
  - Those 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
  - `string` options, to specify string values

Kernel option dependencies

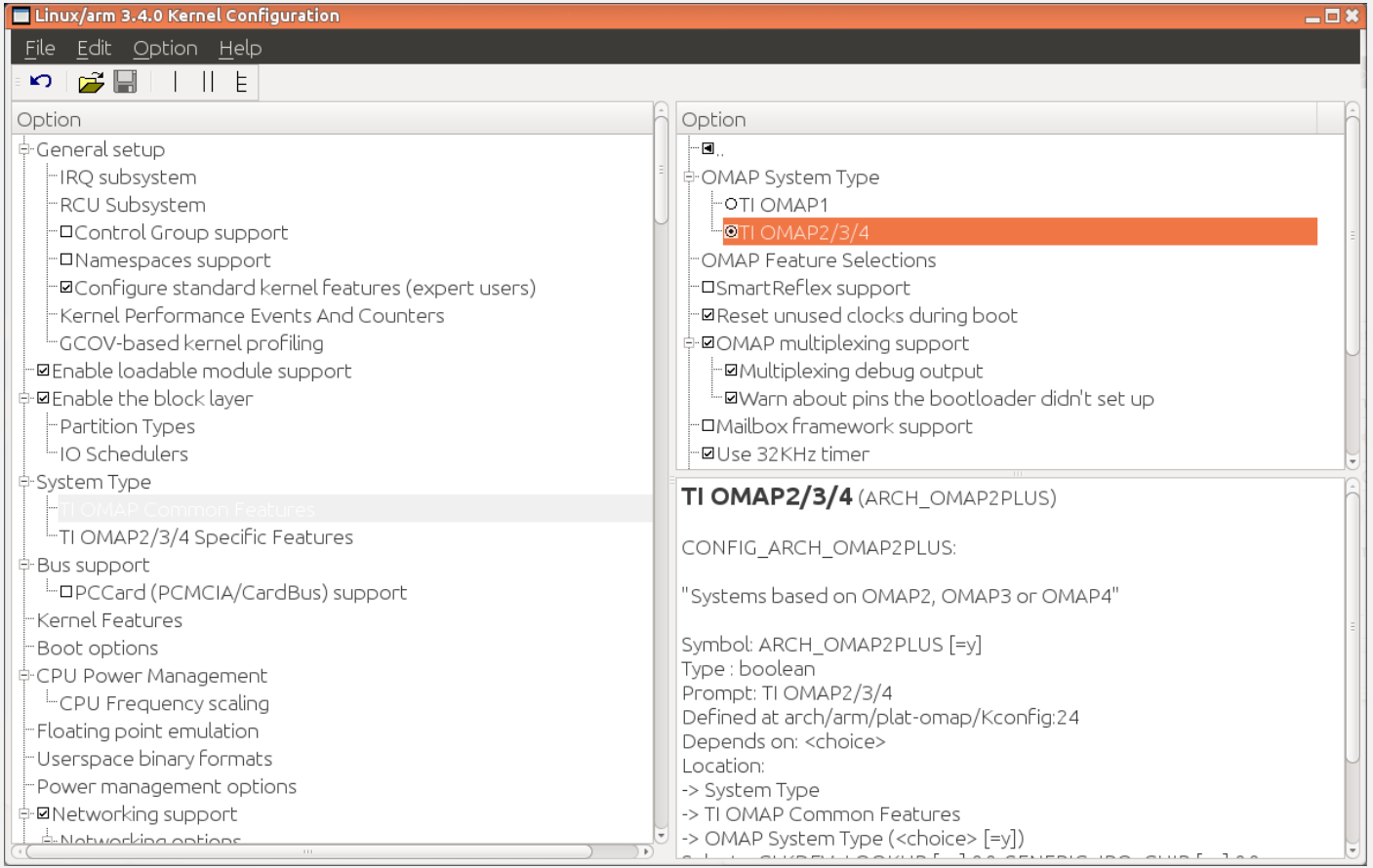
- There are dependencies between kernel options
- For example, enabling a network driver requires the network stack to be enabled
- Two types of dependencies
  - `depends on` dependencies. In this case, option A that depends on option B is not visible until option B is enabled
  - `select` dependencies. 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 those that cannot be selected because of missing dependencies. In this case, they are displayed in gray

make xconfig

`make xconfig`

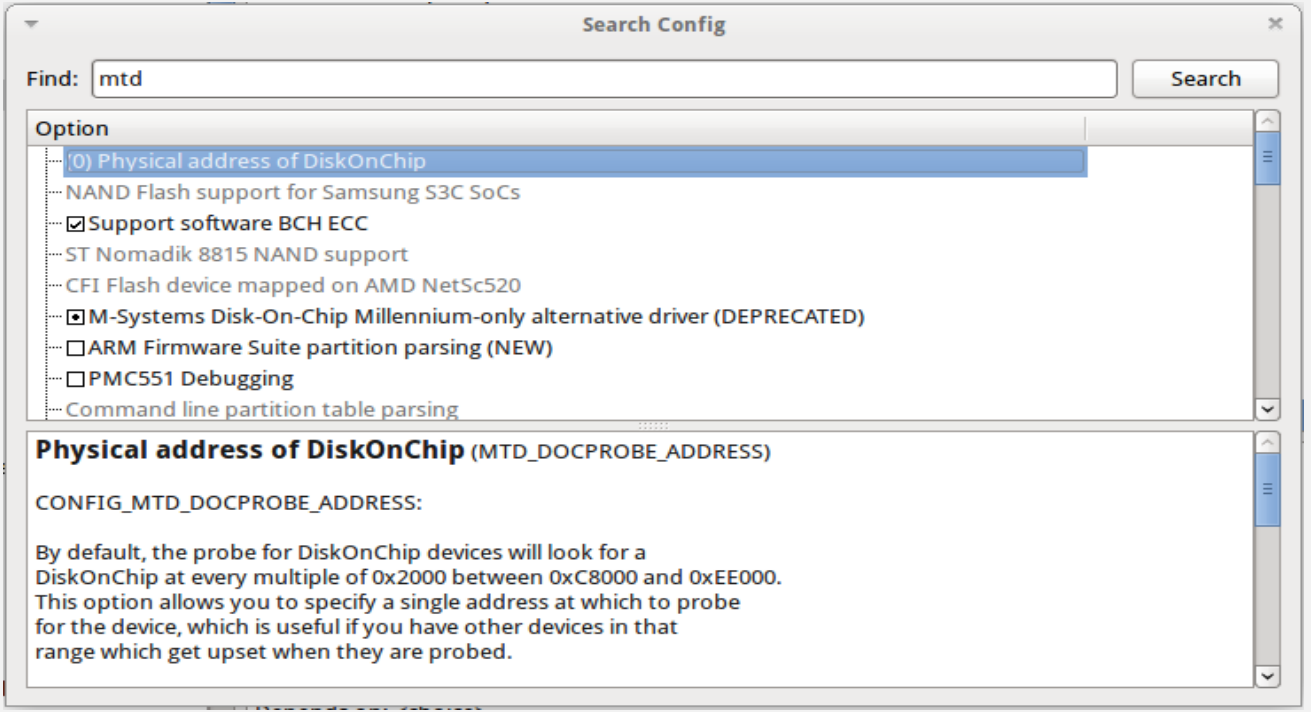
- The most common graphical interface to configure the kernel.
- Make sure you read
  `help -> introduction: useful options!`
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt4-dev g++` (`libqt3-mt-dev` for older kernel releases)

make xconfig screenshot



make xconfig search interface

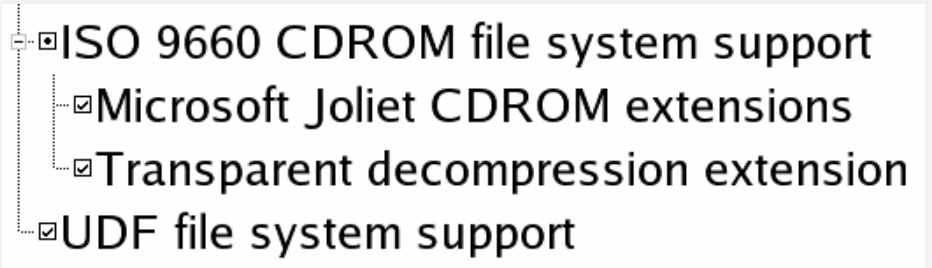
Looks for a keyword in the parameter name. Allows to select or unselect found parameters.



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

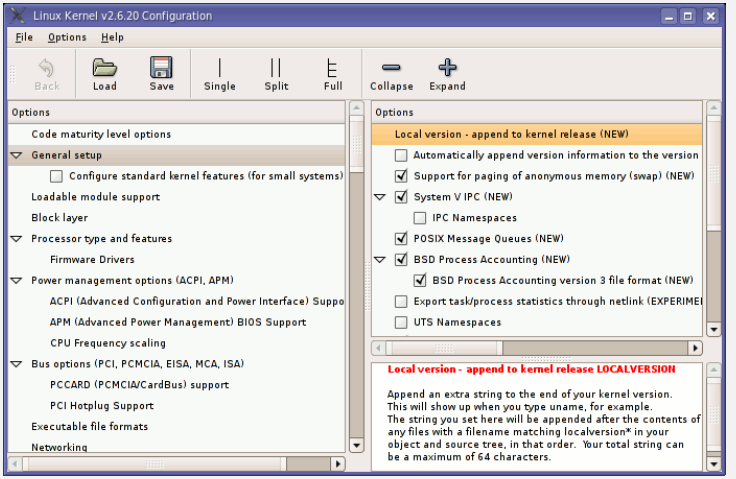
#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
```

```
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

make gconfig

`make gconfig`

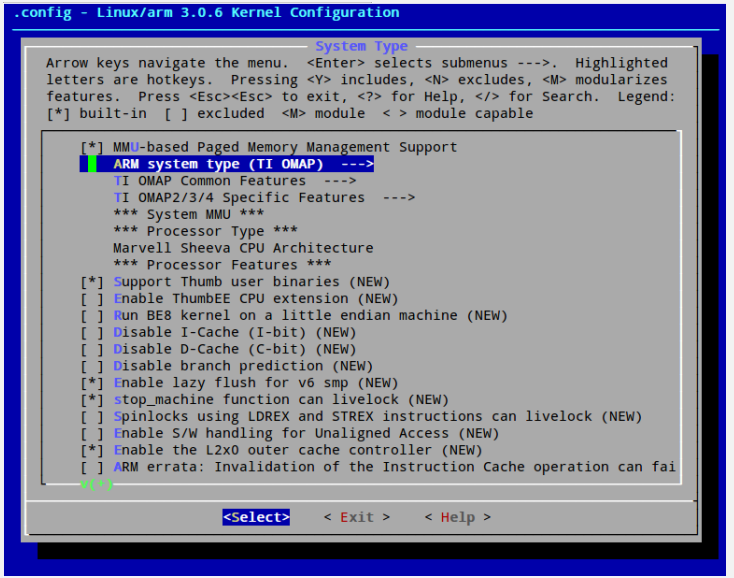
- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`



make menuconfig

`make menuconfig`

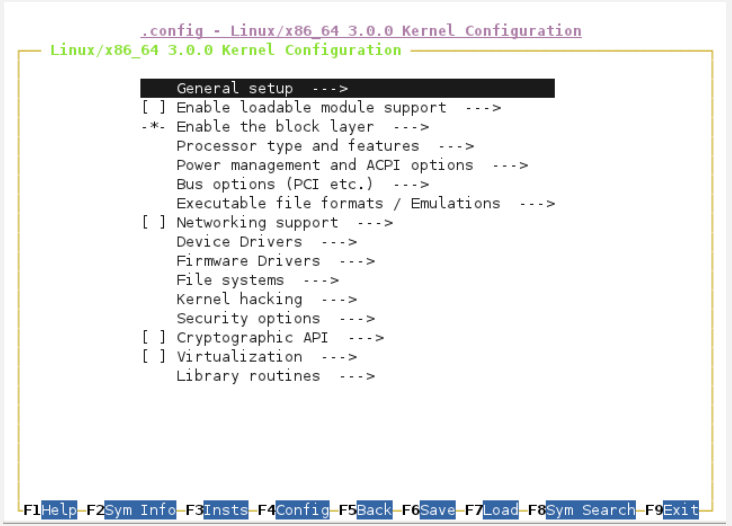
- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot...
- Required Debian packages: `libncurses-dev`



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

`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

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

make allnoconfig

`make allnoconfig`

- Only sets strongly recommended settings to `y`.
- Sets all other settings to `n`.
- Very useful in embedded systems to select only the minimum required set of features and drivers.
- Much more convenient than unselecting hundreds of features one by one!

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`, `allnoconfig`...) 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

Overview of kernel options (1)

- General setup
  - *Local version - append to kernel release* allows to concatenate an arbitrary string to the kernel version that a user can get using `uname -r`. Very useful for support!
  - *Support for swap*, can usually be disabled on most embedded devices
  - *Configure standard kernel features (expert users)* allows to remove features from the kernel to reduce its size. Powerful, but use with care!

Overview of kernel options (2)

- Loadable module support
  - Allows to enable or completely disable module support. If your system doesn't need kernel modules, best to disable since it saves a significant amount of space and memory
- Enable the block layer
  - If `CONFIG_EXPERT` is enabled, the block layer can be completely removed. Embedded systems using only flash storage can safely disable the block layer
- Processor type and features (x86) or System type (ARM) or CPU selection (MIPS)
  - Allows to select the CPU or machine for which the kernel must be compiled
  - On x86, only optimization-related, on other architectures very important since there's no compatibility

Overview of kernel options (3)

- Kernel features
  - Tickless system, which allows to disable the regular timer tick and use on-demand ticks instead. Improves power savings
  - High resolution timer support. By default, the resolution of timer is the tick resolution. With high resolution timers, the resolution is as precise as the hardware can give
  - Preemptible kernel enables the preemption inside the kernel code (the user-space code is always preemptible). See our real-time presentation for details
- Power management
  - Global power management option needed for all power management related features
  - Suspend to RAM, CPU frequency scaling, CPU idle control, suspend to disk

Overview of kernel options (4)

- Networking support
  - The network stack
  - Networking options
    - ⋆ Unix sockets, needed for a form of inter-process communication
    - ⋆ TCP/IP protocol with options for multicast, routing, tunneling, Ipsec, Ipv6, congestion algorithms, etc.
    - ⋆ Other protocols such as DCCP, SCTP, TIPC, ATM
    - ⋆ Ethernet bridging, QoS, etc.
  - Support for other types of network
    - ⋆ CAN bus, Infrared, Bluetooth, Wireless stack, WiMax stack, etc.

Overview of kernel options (5)

- Device drivers
  - MTD is the subsystem for flash (NOR, NAND, OneNand, battery-backed memory, etc.)
  - Parallel port support
  - Block devices, a few misc block drivers such as loopback, NBD, etc.
  - ATA/ATAPI, support for IDE disk, CD-ROM and tapes. A new stack exists
  - SCSI
    - The SCSI core, needed not only for SCSI devices but also for USB mass storage devices, SATA and PATA hard drives, etc.
    - SCSI controller drivers

Overview of kernel options (6)

- Device drivers (cont)
  - SATA and PATA, the new stack for hard disks, relies on SCSI
  - RAID and LVM, to aggregate hard drivers and do replication
  - Network device support, with the network controller drivers. Ethernet, Wireless but also PPP
  - Input device support, for all types of input devices: keyboards, mice, joysticks, touchscreens, tablets, etc.
  - Character devices, contains various device drivers, amongst them
    - serial port controller drivers
    - PTY driver, needed for things like SSH or telnet

- I2C, SPI, 1-wire, support for the popular embedded buses
- Hardware monitoring support, infrastructure and drivers for thermal sensors

Overview of kernel options (7)

- Device drivers (cont)
  - Watchdog support
  - Multifunction drivers are drivers that do not fit in any other category because the device offers multiple functionality at the same time
  - Multimedia support, contains the V4L and DVB subsystems, for video capture, webcams, AM/FM cards, DVB adapters
  - Graphics support, infrastructure and drivers for framebuffers
  - Sound card support, the OSS and ALSA sound infrastructures and the corresponding drivers
  - HID devices, support for the devices that conform to the HID specification (Human Input Devices)

Overview of kernel options (8)

- Device drivers (cont)
  - USB support
    - Infrastructure
    - Host controller drivers
    - Device drivers, for devices connected to the embedded system
    - Gadget controller drivers
    - Gadget drivers, to let the embedded system act as a mass-storage device, a serial port or an Ethernet adapter
  - MMC/SD/SDIO support
  - LED support

- Real Time Clock drivers
- Voltage and current regulators
- Staging drivers, crappy drivers being cleaned up

Overview of kernel options (9)

- For some categories of devices the driver is not implemented inside the kernel
  - Printers
  - Scanners
  - Graphics drivers used by X.org
  - Some USB devices
- For these devices, the kernel only provides a mechanism to access the hardware, the driver is implemented in userspace

Overview of kernel options (10)

- File systems
  - The common Linux filesystems for block devices: ext2, ext3, ext4
  - Less common filesystems: XFS, JFS, ReiserFS, GFS2, OCFS2, Btrfs
  - CD-ROM filesystems: ISO9660, UDF
  - DOS/Windows filesystems: FAT and NTFS
  - Pseudo filesystems: proc and sysfs
  - Miscellaneous filesystems, with amongst other flash filesystems such as JFFS2, UBIFS, SquashFS, cramfs
  - Network filesystems, with mainly NFS and SMB/CIFS
- Kernel hacking
  - Debugging features useful for kernel developers

Kernel compilation

- `make`
  - in the main kernel source directory
  - Remember to run `make -j 4` if you have multiple CPU cores to speed up the compilation process
  - No need to run as root!
- Generates
  - `vmlinux`, the raw uncompressed kernel image, at 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.
  - 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, and 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`
    Module dependencies

- `modules.symbols`
  Tells which module a given symbol belongs to.

Kernel cleanup targets

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

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

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

Specifying cross-compilation (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
- You can also start the configuration from scratch without loading a default configuration file
- 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 protocol, architecture-independent drivers, etc.)
- Make sure you have the support for the right CPU, the right board and the right device drivers.

Building and installing the kernel

- Run `make`
- Copy the final kernel image to the target storage
  - can be `uImage`, `zImage`, `vmlinux`, `bzImage` in `arch/<arch>/boot`
- `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.

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
- 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.

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 in `/lib/modules/<kernel-version>/modules.dep`

  This file is 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 userspace too:
  `echo "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)

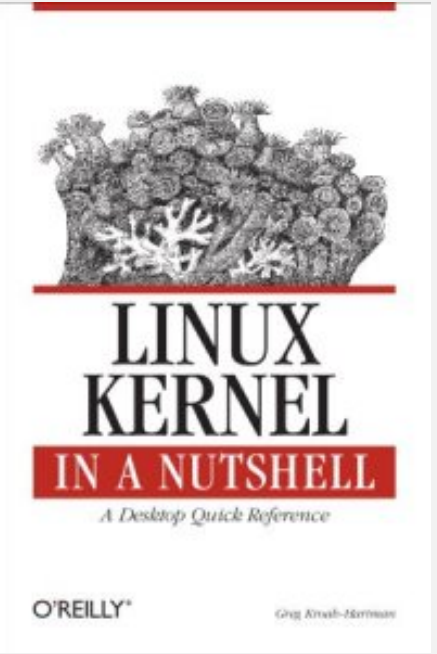
Passing parameters to modules

- 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*

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/
- Our rating: 2 stars



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 those 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

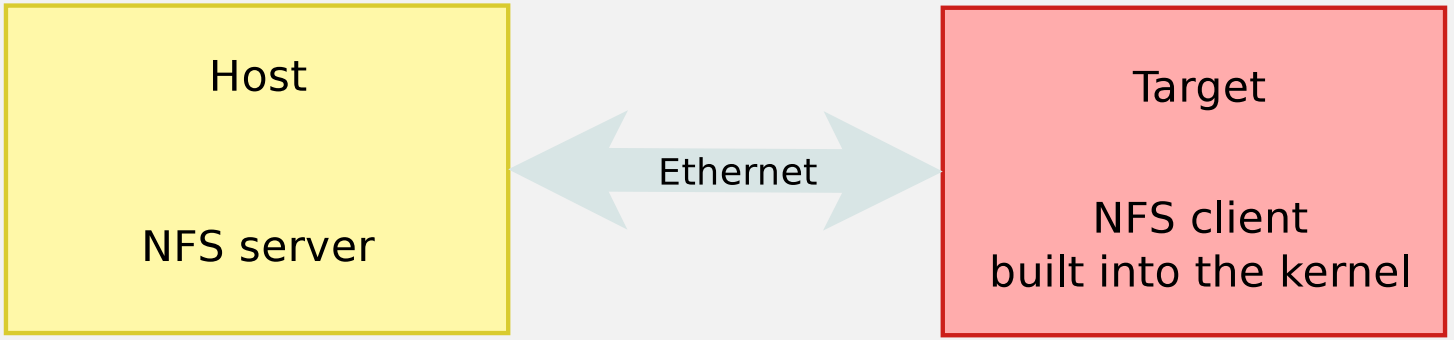
- Partitions of a hard disk or USB key
  - `root=/dev/sdXY`, where `X` is a letter indicating the device, and `Y` a number indicating the partition
  - `/dev/sdb2` is the second partition of the second disk drive (either USB key or ATA hard drive)
- Partitions of an SD card
  - `root=/dev/mmcblkXpY`, 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).


Host
NFS server
Ethernet
Target
NFS client
built into the kernel

Mounting rootfs over the network (2)

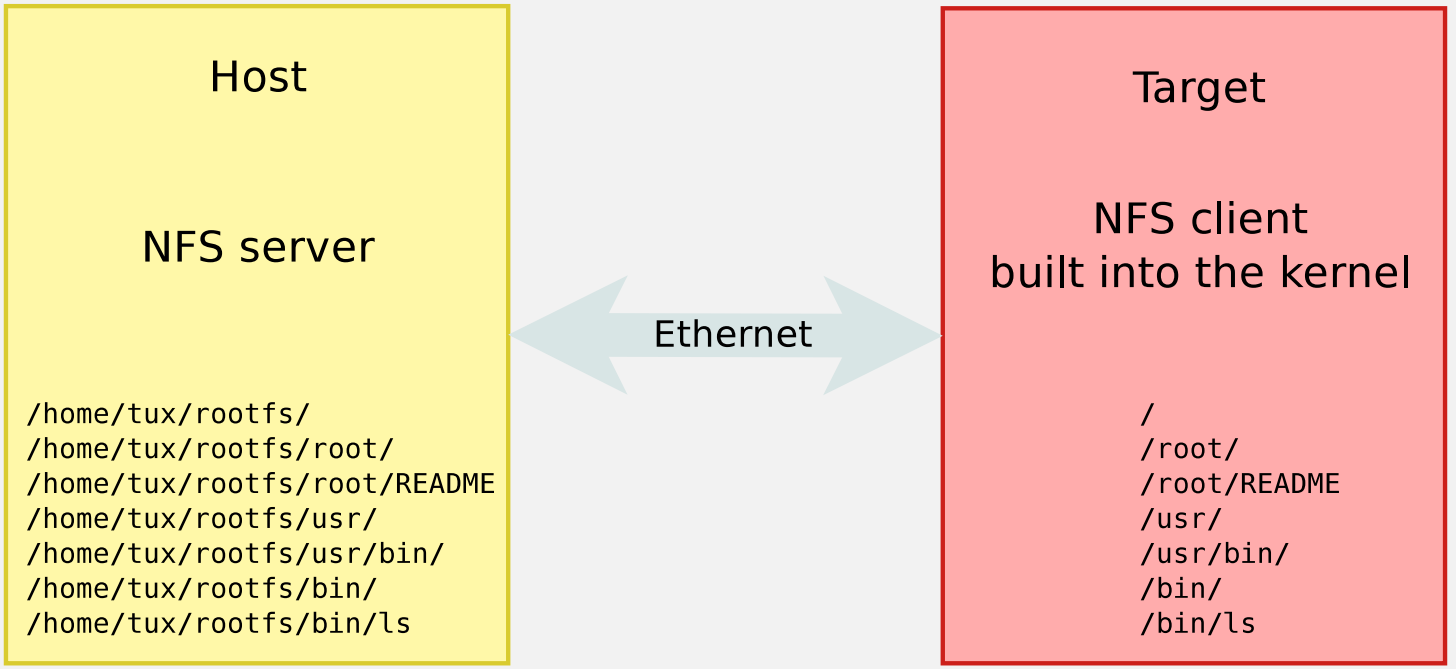
On the development workstation side, a NFS server is needed

- 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)

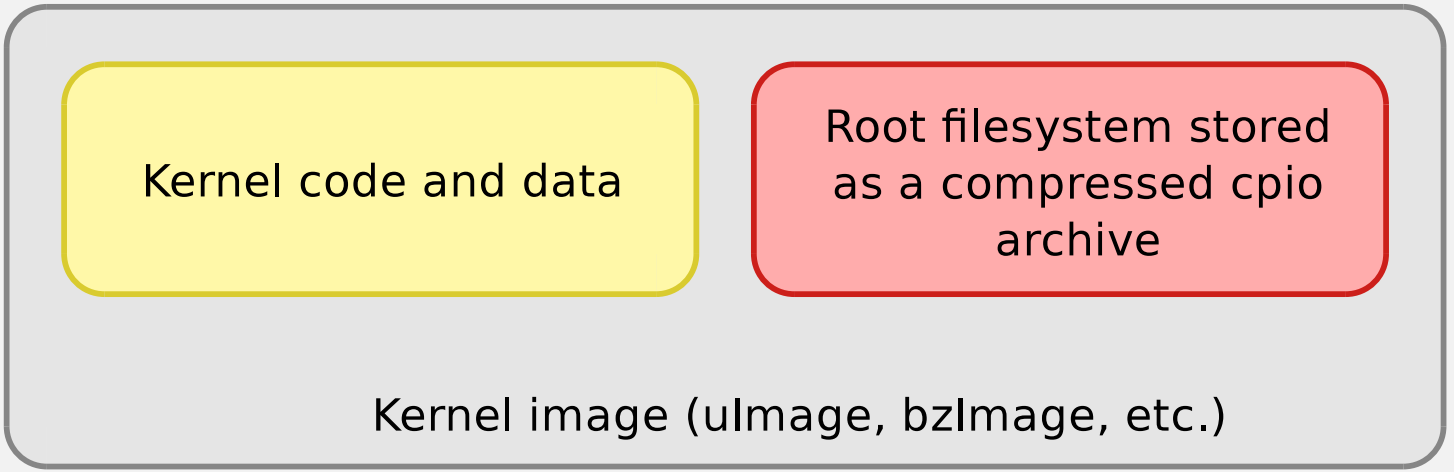
Mounting rootfs over the network (4)



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
- 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
- filesystems/ramfs-rootfs-initramfs.txt
  early-userspace/README

Root filesystem organization

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

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`

Devices

- One of the kernel important role is to **allow applications to access hardware devices**
- In the Linux kernel, most devices are presented to userspace 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

Devices: everything is a file

- A very important Unix design decision was to represent most of the ``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 userspace 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

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

Creating device files

- On a basic Linux system, the device files have to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needs root privileges
  - Coherency between device files and devices handled by the kernel is left to the system developer
- On more elaborate Linux systems, mechanisms can be added to create/remove them automatically when devices appear and disappear
  - `devtmpfs` virtual filesystem, since kernel 2.6.32
  - `udev` daemon, solution used by desktop and server Linux systems
  - `mdev` program, a lighter solution than `udev`

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 userspace 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`
- `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 `sysctl/` in kernel sources.
  - Example
    `echo 3 > /proc/sys/vm/drop_caches`

sysfs filesystem

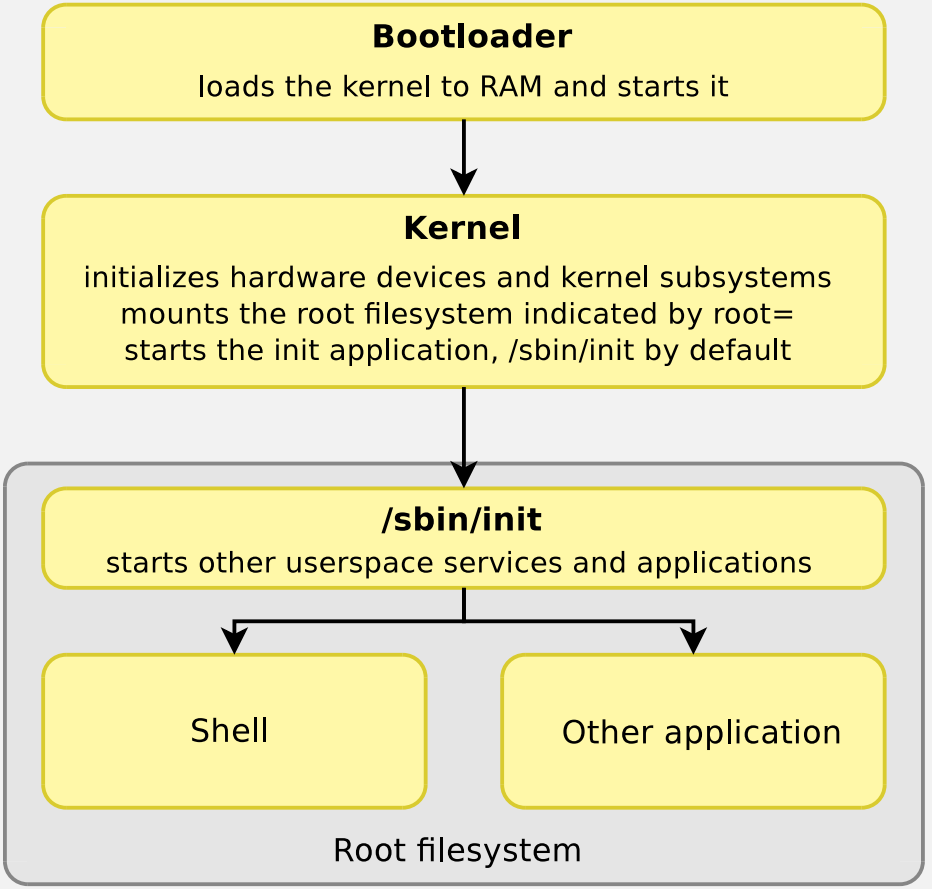
- The `sysfs` filesystem is a feature integrated in the 2.6 Linux kernel
- It allows to represent in userspace the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various userspace applications that need to list and query the available hardware, for example `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 modulepower
```

Basic applications

- In order to work, a Linux system needs at least a few applications
- An `init` application, which is the first userspace application started by the kernel after mounting the root filesystem
  - The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
  - If none of them are found, the kernel panics and the boot process is stopped.
  - The init application is responsible for starting all other userspace applications and services
- Usually a shell, to 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.)
- Those basic components have to be integrated into the root filesystem to make it usable

Overall booting process



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, those 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.

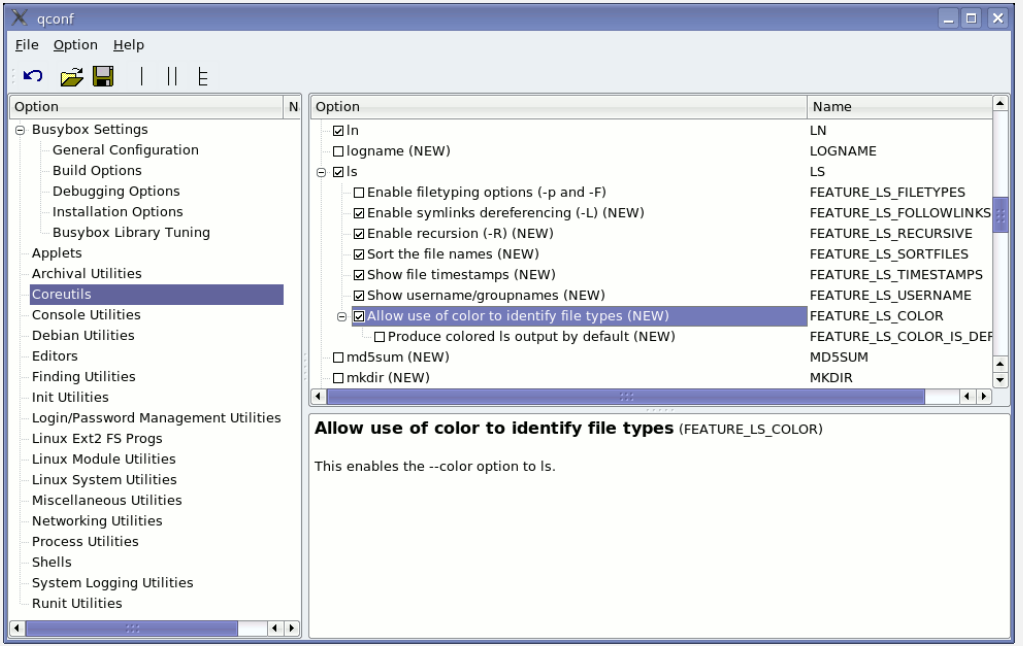
Configuring BusyBox

- 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).

BusyBox make xconfig

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`

Block vs. flash

- Storage devices are classified in two main types: **block devices** and **flash devices**
  - They are handled by different subsystems and different filesystems
- **Block devices** can be read and written to on a per-block basis, without erasing, and do not wear out when being used for a long time
  - Hard disks, floppy disks, RAM disks
  - USB keys, Compact Flash, SD card, these are based on flash storage, but have an integrated controller that emulates a block device
- **Flash devices** can be read, but writing requires erasing, and often occurs on a larger size than the "block" size
  - NOR flash, NAND flash

Block device list

- The list of all block devices available in the system can be found in `/proc/partitions`

```
$ cat /proc/partitions
major minor #blocks name

 179        0    3866624 mmcblk0
 179        1      73712 mmcblk0p1
 179        2    3792896 mmcblk0p2
   8        0  976762584 sda
   8        1    1060258 sda1
   8        2  975699742 sda2
```

- And also in `/sys/block/`

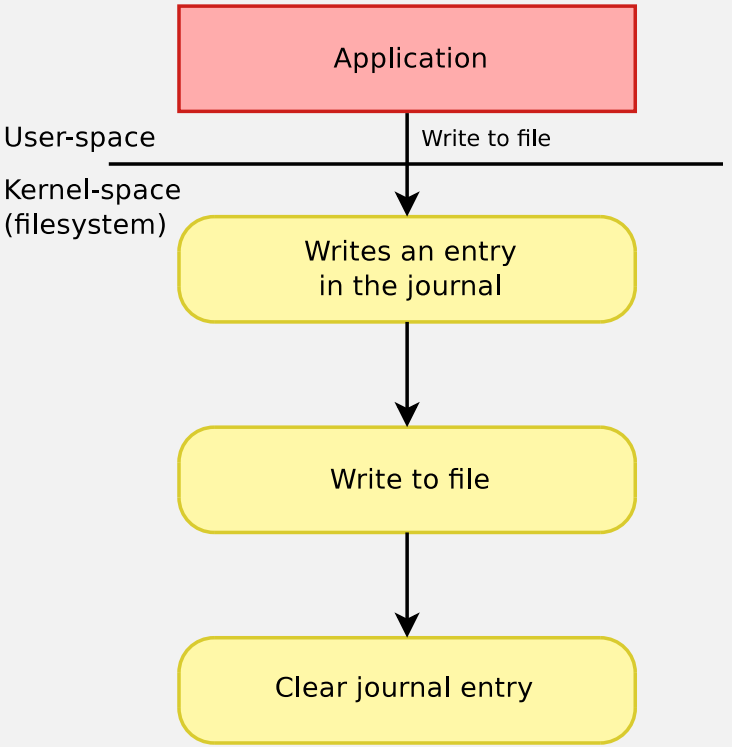
Traditional block filesystems

Traditional filesystems

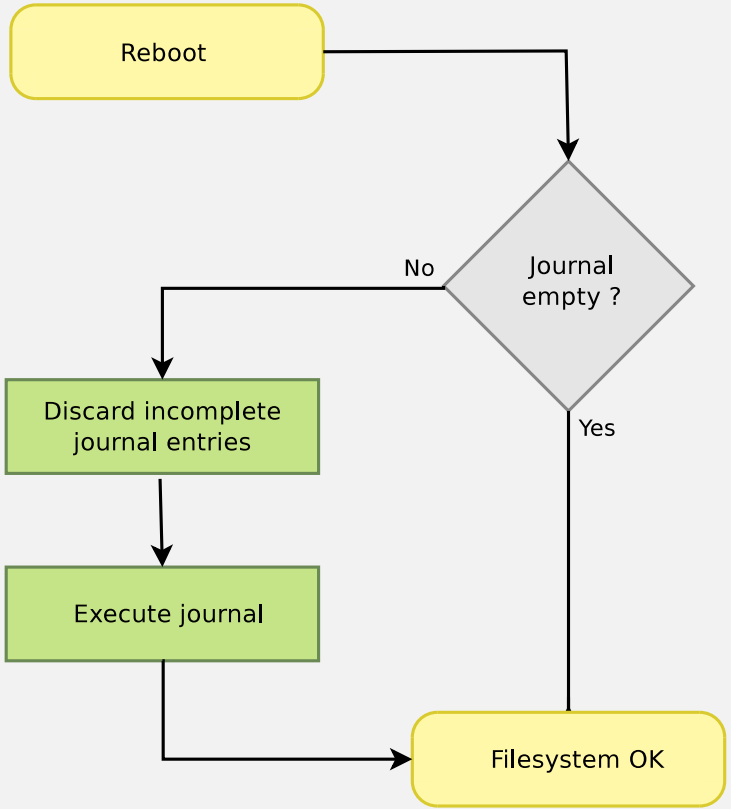
- Can be left in a non-coherent state after a system crash or sudden poweroff, which requires a full filesystem check after reboot.
- `ext2`: traditional Linux filesystem
  (repair it with `fsck.ext2`)
- `vfat`: traditional Windows filesystem
  (repair it with `fsck.vfat` on GNU/Linux or Scandisk on Windows)

Journaled filesystems

- Designed to stay in a correct state even after system crashes or a sudden poweroff
- All writes are first described in the journal before being committed to files



Filesystem recovery after crashes



- Thanks to the journal, the filesystem is never left in a corrupted state
- Recently saved data could still be lost

Journaled block filesystems

Journaled filesystems

- `ext3`: `ext2` with journal extension
  `ext4`: the new generation with many improvements.
  Ready for production. They are the default filesystems for all Linux systems in the world.
- The Linux kernel supports many other filesystems: `reiserFS`, `JFS`, `XFS`, etc. Each of them have their own characteristics, but are more oriented towards server or scientific workloads
- `Btrfs` (``Butter FS'')
  The next generation. In mainline but still experimental.

We recommend `ext2` for very small partitions ($<$ 5 MB), because other filesystems need too much space for metadata (`ext3` and `ext4` need about 1 MB for a 4 MB partition).

Creating ext2/ext3/ext4 volumes

- To create an empty ext2/ext3/ext4 filesystem on a block device or inside an already-existing image file
  - `mkfs.ext2 /dev/hda3`
  - `mkfs.ext3 /dev/sda2`
  - `mkfs.ext4 /dev/sda3`
  - `mkfs.ext2 disk.img`
- To create a filesystem image from a directory containing all your files and directories
  - Use the `genext2fs` tool, from the package of the same name
  - `genext2fs -d rootfs/ rootfs.img`
  - Your image is then ready to be transferred to your block device

Mounting filesystem images

- Once a filesystem image has been created, one can access and modifies its contents from the development workstation, using the **loop** mechanism
- Example:
```
genext2fs -d rootfs/ rootfs.img
mkdir /tmp/tst
mount -t ext2 -o loop rootfs.img /tmp/tst
```
- In the `/tmp/tst` directory, one can access and modify the contents of the `rootfs.img` file.
- This is possible thanks to `loop`, which is a kernel driver that emulates a block device with the contents of a file.
- Do not forget to run `umount` before using the filesystem image!

Squashfs

Squashfs: http://squashfs.sourceforge.net

- Read-only, compressed filesystem for block devices. Fine for parts of a filesystem which can be read-only (kernel, binaries...)
- Great compression rate and read access performance
- Used in most live CDs and live USB distributions
- Supports LZO compression for better performance on embedded systems with slow CPUs (at the expense of a slightly degraded compression rate)
- Now supports XZ encryption, for a much better compression rate, at the expense of higher CPU usage and time.

Benchmarks: (roughly 3 times smaller than ext3, and 2-4 times faster)
http://elinux.org/Squash_Fs_Comparisons

Squashfs - How to use

- Need to install the `squashfs-tools` package
- Creation of the image
  - On your workstation, create your filesystem image:
    `mksquashfs rootfs/ rootfs.sqfs`
  - Caution: if the image already exists remove it first,
    or use the `-noappend` option.
- Installation of the image
  - Let's assume your partition on the target is in `/dev/sdc1`
  - Copy the filesystem image on the device
    `dd if=rootfs.sqfs of=/dev/sdc1`
    Be careful when using `dd` to not overwrite the incorrect partition!
- Mount your filesystem:
  `mount -t squashfs /dev/sdc1 /mnt/root`

tmpfs

Not a block filesystem of course!
Perfect to store temporary data in RAM: system log files, connection data, temporary files...

- `tmpfs` configuration: `File systems -> Pseudo filesystems`
  Lives in the Linux file cache. Doesn't waste RAM: unlike ramdisks, no need to copy files to the file cache, grows and shrinks to accommodate stored files. Saves RAM: can swap out pages to disk when needed.

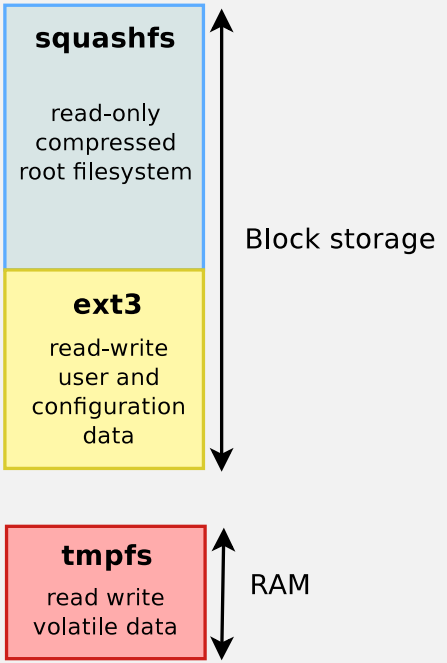
- How to use: choose a name to distinguish the various tmpfs instances you could have. Examples:
  `mount -t tmpfs varrun /var/run`
  `mount -t tmpfs udev /dev`

See `filesystems/tmpfs.txt` in kernel sources.

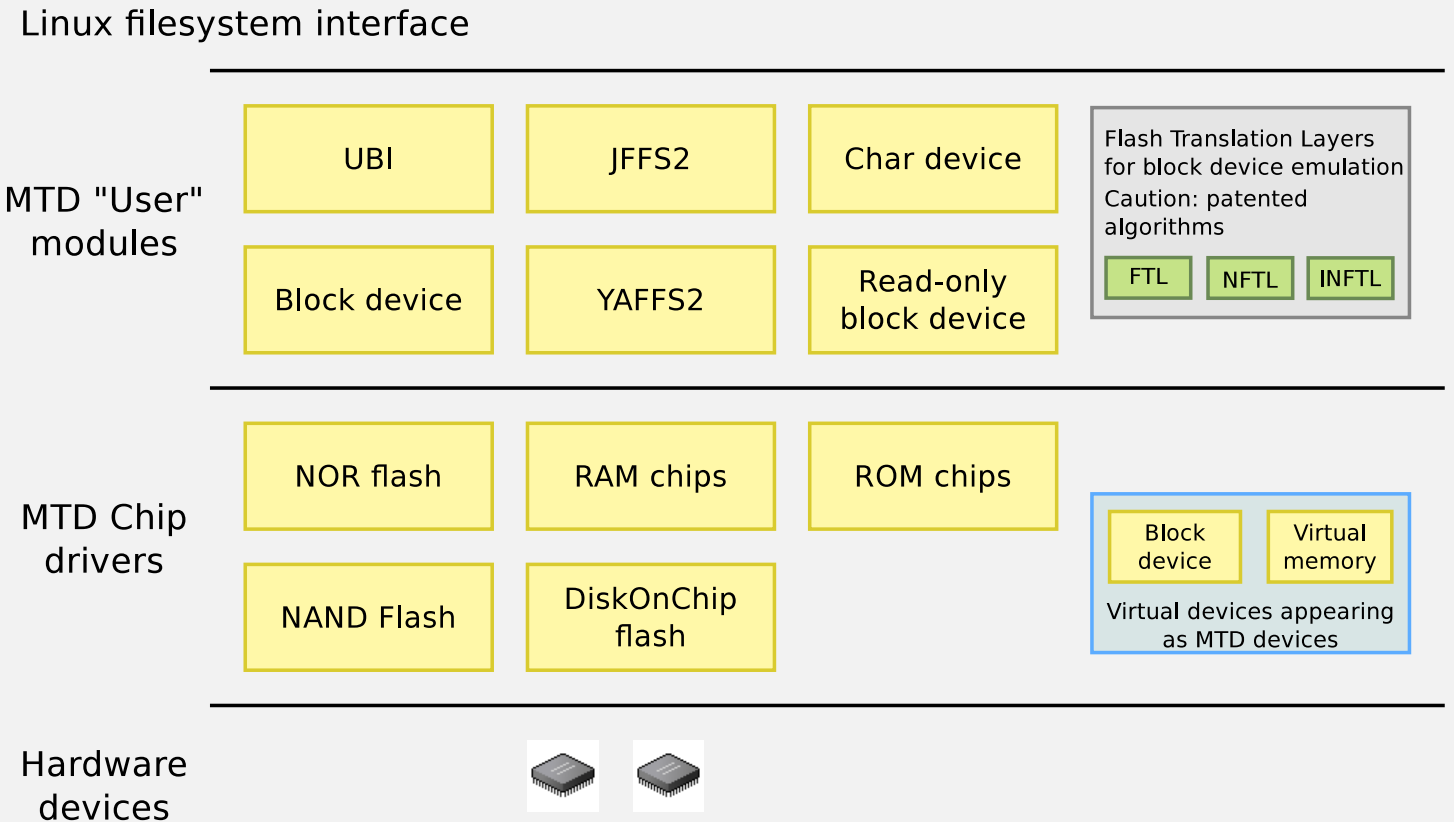
Mixing read-only and read-write filesystems

Good idea to split your block storage into:

- A compressed read-only partition (`Squashfs`)
  Typically used for the root filesystem (binaries, kernel...).
  Compression saves space. Read-only access protects your system from mistakes and data corruption.
- A read-write partition with a journaled filesystem (like `ext3`)
  Used to store user or configuration data.
  Guarantees filesystem integrity after power off or crashes.
- Ram storage for temporary files (`tmpfs`)



The MTD subsystem



MTD devices

- MTD devices are visible in `/proc/mtd`
- The **mtdchar** driver creates a character device for each MTD device of the system
  - Usually named `/dev/mtdX`, major 90. Even minors for read-write access, odd minors for read-only access
  - Provide `ioctl()` to erase and manage the flash
  - Used by the *mtd-utils* utilities
- The **mtdblock** driver creates a block device for each MTD device of the system
  - Usually named `/dev/mtdblockX`, major 31. Minor is the number of the MTD device
  - Allows read/write block-level access. But bad blocks are not handled, and no wear leveling is done for writes.

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
  - Hard-coded into the kernel code
  - 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

Definition of MTD partitions

MTD partitions are defined in the kernel, in the board definitions. Example from `arch/arm/mach-omap2/board-igep0020.c`:

```
static struct mtd_partition igep2_flash_partitions[] = {
    {
        .name   = "X-Loader",
        .offset = 0,
        .size   = 2 * (64*(2*2048))
    },
    {
        .name   = "U-Boot",
        .offset = MTDPART_OFS_APPEND,
        .size   = 6 * (64*(2*2048)),
    },
    [...]
    {
        .name   = "File System",
        .offset = MTDPART_OFS_APPEND,
        .size   = MTDPART_SIZ_FULL,
    },
};
```

Modifying MTD partitions (1)

- MTD partitions can fortunately be defined through the kernel command line.
- First need to find the name of the MTD device. Look at the kernel log at boot time. In our case, the MTD device name is `omap2-nand.0`:

```
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"
```

Modifying MTD partitions (2)

- You can now use the `mtdparts` kernel boot parameter
- Example:
  `mtdparts=omap2-nand.0:512k(X-Loader)ro,1536k(U-Boot)ro,512k(Environment),4m(Kernel`
- We've just defined 6 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` (16 MiB)
  - `Data filesystem` (Remaining space)

Modifying MTD partitions (3)

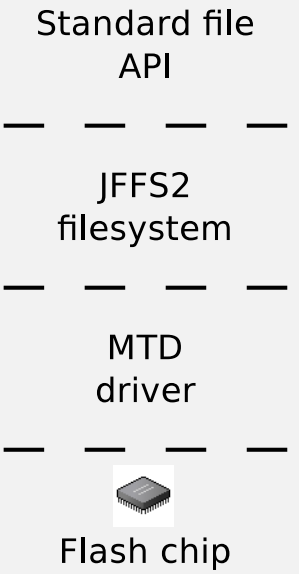
- Partition sizes must be multiple of the erase block size.
  You can use sizes in hexadecimal too. Remember the below sizes:
  `0x20000` = 128k, `0x100000` = 1m, `0x1000000` = 16m
- `ro` lists the partition as read only
- `-` is used to use all the remaining space.

mtd-utils

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

jffs2

- Today's standard filesystem for MTD flash
- Nice features: on the fly compression (saves storage space and reduces I/O), power down reliable, wear-leveling and ECC.
- Drawbacks: doesn't scale well
  - Mount time depending on filesystem size: the kernel has to scan the whole filesystem at mount time, to read which block belongs to each file.
  - Need to use the `CONFIG_JFFS2_SUMMARY` kernel option to store such information in flash. This dramatically reduces mount time (from 16 s to 0.8s for a 128 MB partition).
- http://www.linux-mtd.infradead.org/doc/jffs2.html



jffs2 - How to use

On the Linux **target**

- Need either the `mtd-utils` package from the MTD project, or their embedded variants from Busybox
- Erase and format a partition with jffs2:
  `flash_eraseall -j /dev/mtd2`
- Mount the partition:
  `mount -t jffs2 /dev/mtdblock2 /mnt/flash`
- Fill the contents by writing
  (copying from NFS or from external storage)
- Other possibility: use a *jffs2* image (see next page to produce it):
  `flash_eraseall /dev/mtd2`
  `nandwrite -p /dev/mtd2 rootfs.jffs2`

How to create a jffs2 image

- `mkfs.jffs2` command available in the mtd-utils package.
  Caution: unlike some `mkfs` commands, it doesn't create a filesystem, but a filesystem image.
- First, find the erase block size (on the target running Linux):
  `cat /proc/mtd`
  For example: `00040000` (256 KiB)
- Then create the image on your workstation:
  `mkfs.jffs2 --pad --no-cleanmarkers --eraseblock=256 -d rootfs/ -o rootfs.jffs2`
- The `--pad` option pads the jffs2 image contents
  until the end of the final erase block.
- It is fine if the jffs2 image is smaller than the MTD partition.
  The jffs2 file system will use the entire partition anyway.
- The `--no-cleanmarkers` option is for NAND flash only.

Mounting a jffs2 image on your host

Useful to edit `jffs2` images on your development system
Mounting an MTD device as a loop device is a bit complex task.
Here's an example for `jffs2`, for your reference:

- First find the erase block size used to create the jffs2 image.
  Let's assume it is 256KiB (262144 bytes).
- Create a block device from the image
  `losetup /dev/loop0 root.jffs2`
- Emulate an MTD device from a block device,
  using the `block2mtd` kernel module
  `modprobe block2mtd block2mtd=/dev/loop0,262144`
- Load the mtdblock driver if needed
  `modprobe mtdblock`
- Finally, mount the filesystem (create `/mnt/jffs2` if needed)
  `mount -t jffs2 /dev/mtdblock0 /mnt/jffs2`

Initializing jffs2 partitions from U-boot

You may not want to have `mtd-utils` on your target!

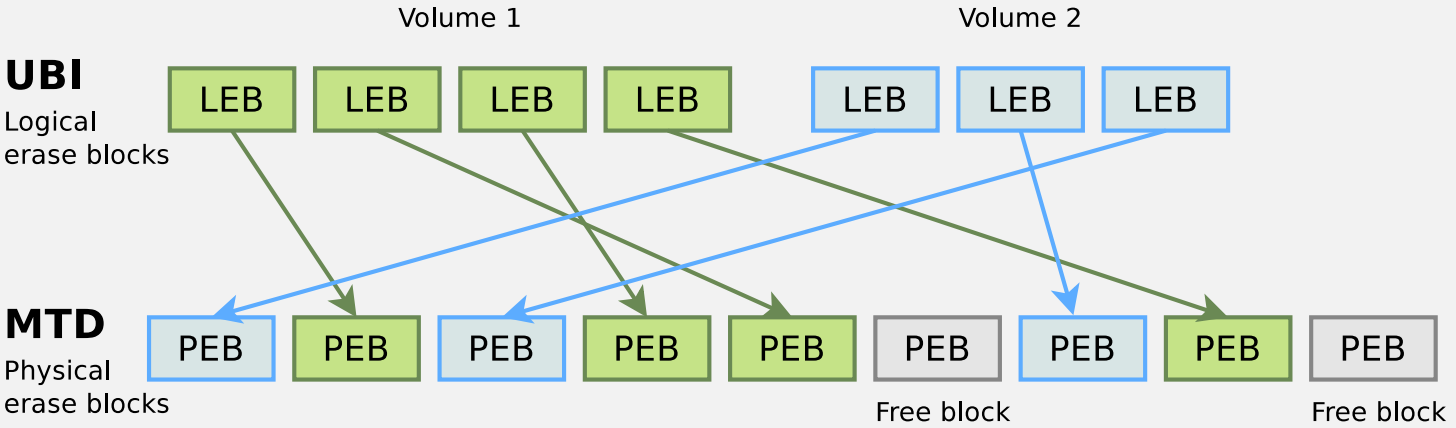
- Create a JFFS2 image on your workstation
- In the U-Boot prompt:
  - Download the jffs2 image to RAM with `tftp`
    Or copy this image to RAM from external storage
    (U-boot understands FAT filesystems and supports USB storage)
  - Flash it inside an MTD partition
    (exact instructions depending on flash type, NOR or NAND,
    reuse the instructions used to flash your kernel). Make sure to write only the size of the image, not more!
  - If you boot on a jffs2 root filesystem, add `root=/dev/mtdblock<x>` and `rootfstype=jffs2` to the Linux command line arguments.
- Limitation: need to split the jffs2 image in several chunks
  if bigger than the RAM size.

UBI (1)

Unsorted Block Images

- http://www.linux-mtd.infradead.org/doc/ubi.html
- Volume management system on top of MTD 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).

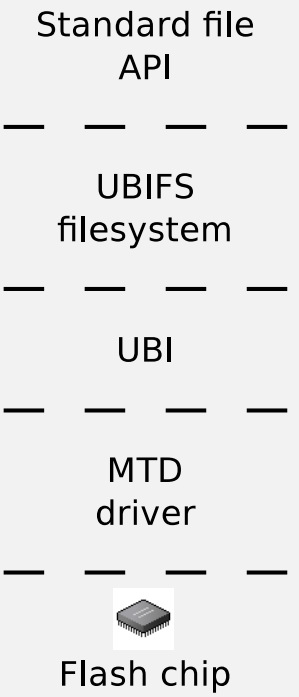
UBI (2)


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
Free block

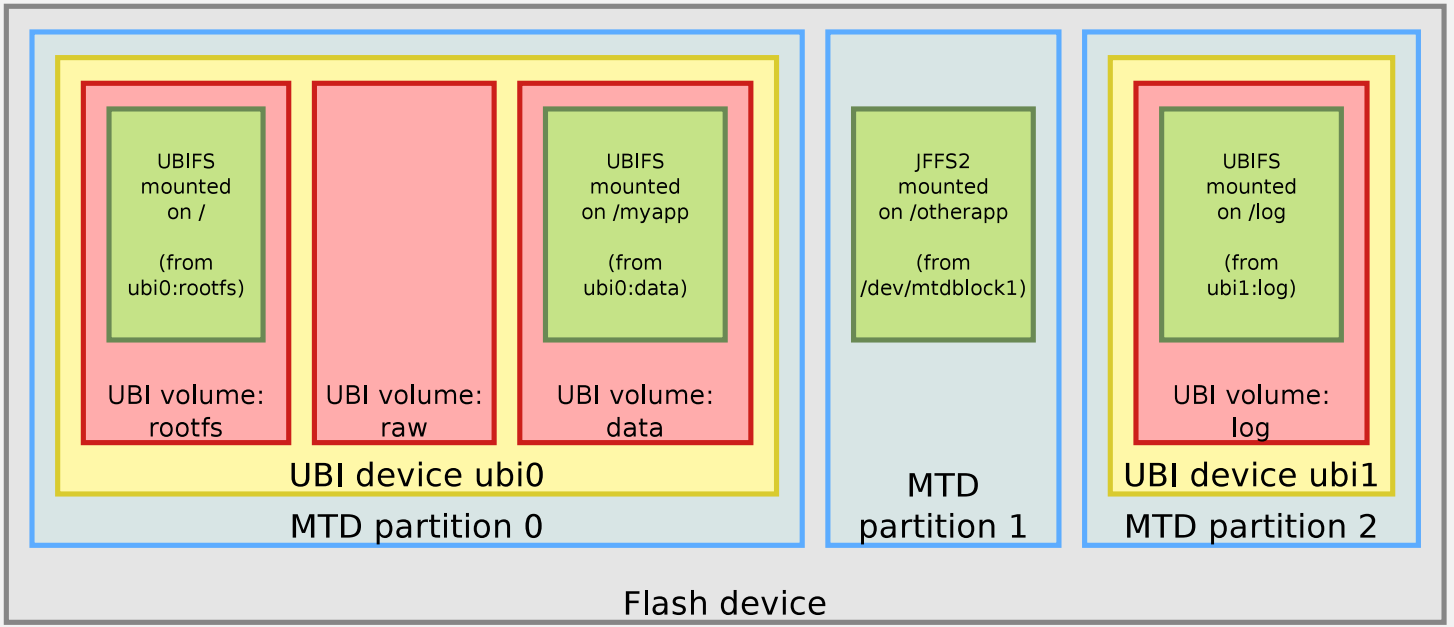
UBIFS

http://www.linux-mtd.infradead.org/doc/ubifs.html

- The next generation of the jffs2 filesystem, from the same linux-mtd developers.
- Available since Linux 2.6.27
- Works on top of UBI volumes
- Has a noticeable metadata overhead on very small partitions (4M, 8M)



UBI layout example



UBI - Preparation

- Have `/dev/` mounted as a `devtmpfs` filesystem
- Erase your flash partition while preserving your erase counters
  `ubiformat /dev/mtd1`
  See http://www.linux-mtd.infradead.org/faq/ubi.html if you face problems
- Attach UBI to one (of several) of the MTD partitions:
  `ubiattach /dev/ubi_ctrl -m 1`
- This command creates the `ubi0` device, which represent the full UBI space stored on MTD device 1 (new `/dev/ubi0` character device)

UBI - Volume management

- Volume creation with `ubimkvol`
  - `ubimkvol /dev/ubi0 -N test -s 116MiB`
  - `ubimkvol /dev/ubi0 -N test -m` (max available size)
  - The volume is then identified as `ubi0:test` for the `mount`/`umount` commands
- Volume removal with `ubirmvol`
  - `ubirmvol /dev/ubi0 -N test`

UBIFS - How to use

- When a UBI volume is created, creating an empty UBIFS filesystem is just a matter of mounting it
  - `mount -t ubifs ubi0:test /mnt/flash`
- Images of UBIFS filesystems can be created using the `mkfs.ubifs` utility
  - `mkfs.ubifs -m 4096 -e 256KiB -c 1000 -r rootfs/ ubifs.img`
    - `-m 4096`, minimal I/O size
    - `-e 256 KiB`, erase block size
    - `-c 1000`, maximum number of logical erase blocks. See http://www.linux-mtd.infradead.org/faq/ubifs.html#L_max_leb_cnt for details.
  - Can be written to a UBI volume using `ubiupdatevol` and the `/dev/ubiX_Y devices`

Ubinize

- Images of a full UBI space, containing several volumes can be created using the `ubinize` utility
  - Can be written to a raw MTD partition using `nand write` in U-boot.
  - Caution: `nand erase` will also erase the Erase Counters
- `ubinize -o ubi.img -p 256KiB -m 4096 ubi.ini`
  - Creates `ubi.img`, with 256KiB physical erase blocks, 4096 minimum I/O size (`-m`).

UBIFS - How to prepare a root fs

- Create the UBIFS image from the target directory
- Write the configuration file for the UBI device (`ubi.ini`):

```
[RFS-volume]
mode=ubi
image=rootfs.ubifs
vol_id=1
vol_size=30MiB
vol_type=dynamic
vol_name=rootfs
vol_flags=autoresize
vol_alignment=1
```

- Create the UBI device image
- Flash it using a bad block aware command from the bootloader
- Pass UBI layout information to the kernel:
  - `rootfstype=ubifs ubi.mtd=1 root=ubi0:rootfs`

Flash filesystem benchmarks

http://elinux.org/Flash_Filesystem_Benchmarks

- **jffs2**
  - Worst performance
  - Requires `CONFIG_SUMMARY` to have acceptable boot time
- **yaffs2**
  - Good performance, but not in mainline Linux
- **ubifs**
  - Best solution and performance for medium and big partitions
  - Too much metadata overhead small partitions (only case when `yaffs2` and `jffs2` are still useful)

Issues with flash-based block storage

- Flash storage made available only through a block interface.
- Hence, no way to access a low level flash interface and use the Linux filesystems doing wear leveling.
- No details about the layer (Flash Translation Layer) they use. Details are kept as trade secrets, and may hide poor implementations.
- Hence, it is highly recommended to limit the number of writes to these devices.

Reducing the number of writes

- Of course, 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.
- You may decide to do without journaled filesystems. They cause more writes, but are also much more power down resistant (trade-off).

Useful reading

- Arnd Bergmann: Optimizing Linux with cheap flash drives
  In depth coverage of flash storage with a block interface.
  http://lwn.net/Articles/428584/
- Introduction to JFFS2 and LogFS:
  http://lwn.net/Articles/234441/
- Nice UBI presentation from Toshiba:
  http://free-electrons.com/redirect/celf-ubi.html
- Documentation on the linux-mtd website:
  http://www.linux-mtd.infradead.org/

Contents

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

- Freecode, a website referencing most open-source projects
  http://www.freecode.com
- 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, so dynamic linking must be used
- 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 you used and to keep your modifications and adaptations well-separated from the original version
  - Conform to the license requirements before shipping the product to the customers
- Free Software licenses have been enforced successfully in courts
  - GPL-violations.org, http://www.gpl-violations.org
  - Software Freedom Law Center, http://www.softwarefreedom.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

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
- *lighttpd*: http://lighttpd.net
  Low footprint server good at managing high loads.
  May be useful in embedded systems too
- Other possibilities: *Boa*, *thttpd*, *nginx*, etc

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 userspace tools associated to the Linux firewall, Netfilter
- **iw and wireless tools**, the userspace 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

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 userspace 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

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

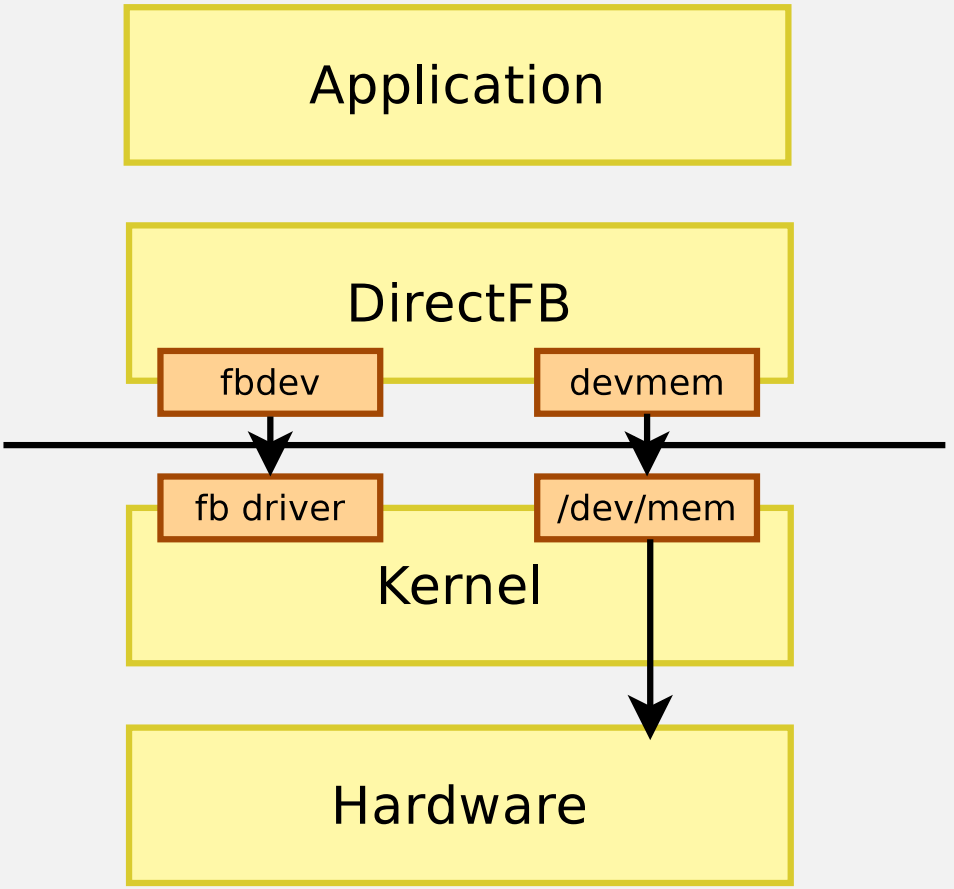
Audio, video and multimedia

- **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 provider by SoC vendors.
- **alsa-lib**, the userspace 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.

DirectFB (1)

- Low-level graphical library
  - Lines, rectangles, triangles drawing and filling
  - Blitting, flipping
  - Text drawing
  - Windows and transparency
  - Image loading and video display
- But also handles input event handling: mouse, keyboard, joystick, touchscreen, etc.
- Provides accelerated graphic operations on a few hardware platforms
- Single-application by default, but multiple applications can share the framebuffer thanks to *fusion*
- License: LGPL 2.1
- http://www.directfb.org

DirectFB: architecture



DirectFB: usage

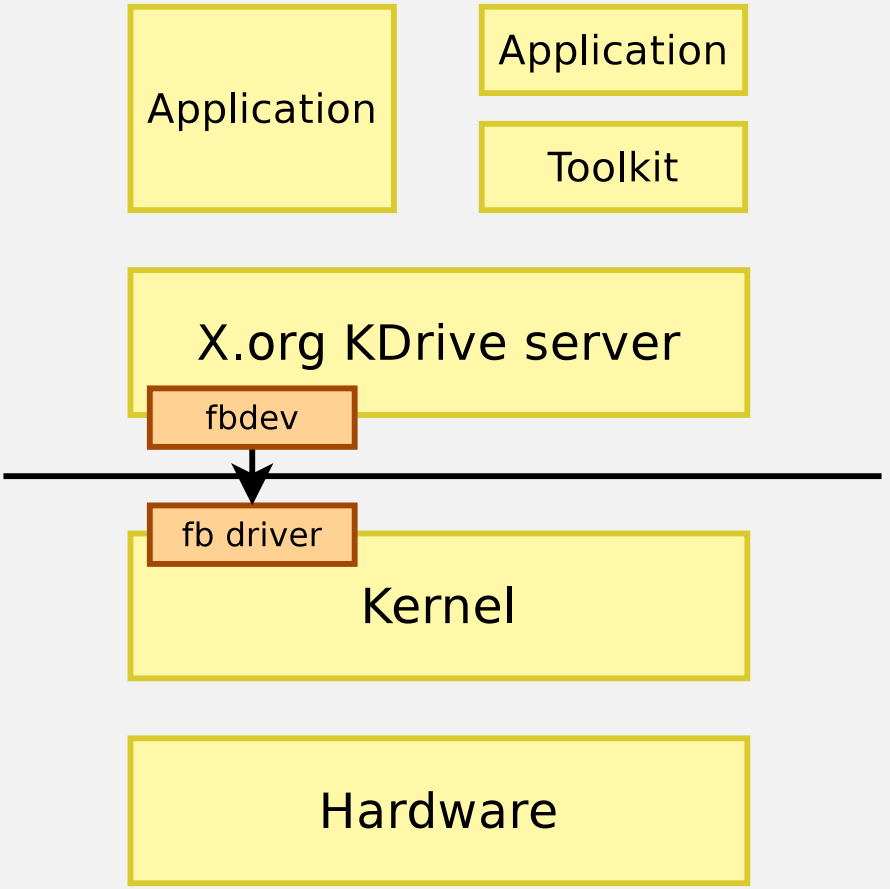
- Multimedia applications
  - For example the Disko framework, for set-top box related applications
- ``Simple'' graphical applications
  - Industrial control
  - Device control with limited number of widgets
- Visualization applications
- As a lower layer for higher-level graphical libraries

DirectFB: architecture

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
  - Probably doesn't make sense since DirectFB is a more lightweight solution for an API of roughly the same level (no widgets)
- Or, usually used with a toolkit on top of it
  - Gtk
  - Qt
  - Fltk
  - WxEmbedded

Gtk

- 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*

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
  - DirectFB back-end integrated in version 4.4, which allows to take advantage of the acceleration provided by DirectFB drivers

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

Other less frequent solutions

- Enlightenment foundation libraries
  - Very powerful, but complicated to use due to the lack of documentation
  - http://www.enlightenment.org/p.php?p=about/efl

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

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 everywhere (MacOS X, iPhone, Google Android and Chrome...). Many applications (browsers, e-mail clients...) are already using WebKit: http://trac.webkit.org/wiki/Applications using WebKit
- Multiple graphical back-ends: Qt4, GTK...
- Lightweight web-browsers: Midori (GTK), Arora (Qt)
- You could use it to create your custom browser.

Industrial applications

- In many industrial applications, the system is only responsible for monitoring and control a device
- Such a system is usually relatively simple in terms of components
  - Kernel
  - BusyBox
  - C library
  - Applications relying directly on the C library, sometimes using the real-time capabilities of the Linux kernel
  - Sometimes a Web server for remote control, or another server implementing a custom protocol

Digital Photo Frame: requirements

- Example taken from a conference of Matt Porter, Embedded Alley at ELC 2008
- Hardware: ARM SoC with DSP, audio, 800x600 LCD, MMC/SD, NAND, buttons, speakers
- The photo frame must be able to
  - Display to the LCD
  - Detect SD card insertion, notify applications of the insertion so that applications can build a catalog of the pictures on the SD card
  - Modern 3D GUI with nice transitions
  - Navigation through buttons
  - Support audio playback (MP3, playlists, ID3 tag)
  - JPEG resizing and rotation

Digital Photo Frame: components (1)

- Base system
  - Components present in virtually all embedded Linux systems
  - The U-Boot bootloader
  - Linux Kernel
    - Drivers for SD/MMC, framebuffer, sound, input devices
  - Busybox
  - Build system, in this case was OpenEmbedded
  - Components: **u-boot**, **Linux**, **busybox**

Digital Photo Frame: components (2)

- Event handling to detect SD card insertion
  - `udev`, that receives events from the kernel, creates device nodes, and sends events to `hal`
  - `hal`, which maintains a database of available devices and provides a `dbus` API
  - `dbus` to connect `hal` with the application. The application subscribes to `hal` event through `dbus` and gets notified when they are triggered
  - Components: **udev**, **hal**, **dbus**

Digital Photo Frame: components (3)

- JPEG display
  - **libjpeg** to decode the pictures
  - **jpegtran** to resize and rotate them
  - **FIM** (Fbi Improved) for dithering
- MP3 support
  - **libmad** for playing
  - **libid3** for ID3 tags reading
  - **libm3u** to support playlists
  - Used vendor-provided components to leverage the DSP to play MP3

Digital Photo Frame: components (4)

- 3D interface
  - Vincent, an open-source implementation of OpenGL ES
  - Clutter, higher-level API to develop 3D applications
- Application itself
  - Manages media events
  - Uses the JPEG libraries to decode and render pictures
  - Receives Linux input events from buttons and draws OpenGL-based UI developed with Clutter
  - Manage a user-defined configuration
  - Play the music with the MP3-related libraries
  - Display photo slideshow

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

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

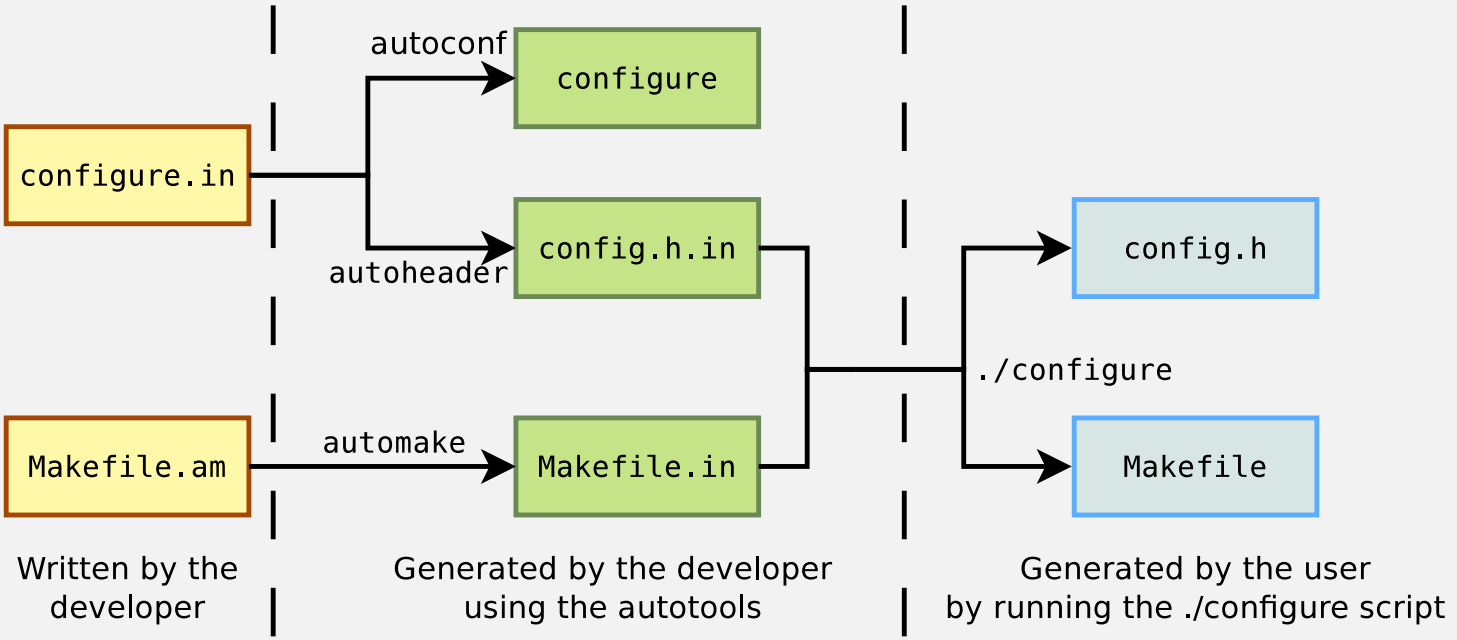
Build systems

- 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 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.
- The `--host` argument must be passed to the `configure` script.
  - `--build` argument is 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.

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

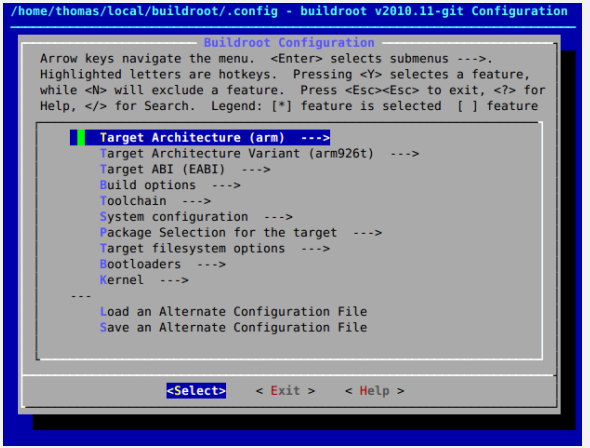
- **Buildroot**, developed by the community
  http://www.buildroot.net
- **PTXdist**, developed by Pengutronix
  http://www.pengutronix.de/software/ptxdist/index_en.html
- **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project
  http://www.openwrt.org
- **LTIB**. 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** and vendor-specific derivatives such as **Arago**
- 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 toolchains only, but can use external uClibc or glibc toolchains
- Over 700+ applications or libraries integrated, from basic utilities to more elaborate software stacks: X.org, Gstreamer, Qt, Gtk, WebKit, 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.

Buildroot: adding a new package (2)

- 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

- 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

- Available on ARM, MIPS and PowerPC architectures
- Provides a ready-to-use filesystem with all the software you need.
- Huge flexibility thanks to the package management system, but only works only systems with enough storage size (300 MB) and RAM (64 MB).
- Software is compiled natively by default.
- You can build your own root filesystem images on x86 by using the `debootstrap` command.
- Emdebian is a project to make Debian better for embedded systems: leverage Debian package descriptions, but reduces dependencies, smaller configuration, removes documentation, supports uClibc... See http://emdebian.org.

Distributions - Ubuntu

Ubuntu GNU/Linux

- https://wiki.ubuntu.com/ARM
- Based on Debian, same benefits
- New release every 6 months. Same release schedule as for x86.
- Supported on ARM, but only on Cortex A8 and beyond. Supplies Thumb2 binaries. Neon not supported.
- Good solution for mobile multimedia devices. Getting ready for ARM servers too.
- Training participants - try Ubuntu on your IGEPv2 board:
  Run the `linaro-media-create` utility with images found on http://releases.linaro.org/12.03/ubuntu/oneiric-images/ubuntu-desktop/

Distributions - Others

Fedora

- http://fedoraproject.org/wiki/Architectures/ARM
- Also has some support for ARM, but ARM releases get much less attention and are currently made more than one year later.
- Expect Fedora to be back in the next years, to meet demand for ARM servers.

Embedded distributions

Distributions designed for specific types of devices

- **Meego**: http://meego.com/
  Distribution targeting phones, media players, netbooks, TVs and In Vehicle Infotainment.
  Dropped by Nokia and Intel, and will be replaced by Tizen, a new project (https://www.tizen.org/).
- **Android**: http://www.android.com/
  Google's distribution for phones and tablet PCs.
  Except the Linux kernel, very different userspace than other Linux distributions. Very successful, lots of applications available (many proprietary).
- **Ångström**: http://www.angstrom-distribution.org/
  Targets PDAs and webpads (Siemens Simpad...)
  Binaries available for arm little endian.

System emulator: QEMU

http://qemu.org
Fast processor emulator

- Useful to develop root filesystems and applications when the hardware is not available yet (binary compatible development boards are also possible).
- Emulates the processor and various peripherals (“System emulation): x86, ppc, arm, sparc, mips, m68k...
- Many different ARM boards supported:
  `qemu-system-arm -M ?`
- QEMU also allows to just run binaries for other architectures (``User emulation''). Used inside some building tools.

Commercial embedded Linux solutions

Caution: *commercial* doesn't mean *proprietary!*

- Vendors play fair with the GPL and do make their source code available to their users, and most of the time, eventually to the community.
  - As long as they distribute the sources to their users, the GPL doesn't require vendors to share their sources with any third party.
- Graphical toolkits developed by the vendors are usually proprietary, trying to make it easier to create and embedded Linux systems.
- Major players: Wind River, Montavista, TimeSys

Commercial solution strengths

- Technical advantages
  - Well tested and supported kernel and tool versions
  - Often supporting patches not supported by the mainline kernel yet (example: real-time patches)
- Complete development tool sets: kernels, toolchains, utilities, binaries for impressive lists of target platforms
- Integrated utilities for automatic kernel image, initramfs and filesystem generation.
- Graphical developments tools
- Development tools available on multiple platforms: GNU / Linux, Solaris, Windows...
- Support services
  - Useful if you don't have your own support resources
  - Long term support commitment, even for versions considered as obsolete by the community, but not by your users!

Commercial or community solutions?

Commercial distributions and tool sets

- Best if you don't have your own support resources and have a sufficient budget
- Help focusing on your primary job: making an embedded device.
- You can even subcontract driver development to the vendor

Community distributions and tools

- Best if you are on a tight budget
- Best if you are willing to build your own embedded Linux expertise, investigate issues by yourselves, and train your own support resources.

In any case, your products are based on Free Software!

Contents

Application development

- An embedded Linux system is just a normal Linux system, with usually a smaller selection of components
- In terms of application development, developing on embedded Linux is exactly the same as developing on a desktop Linux system
- All existing skills can be re-used, without any particular adaptation
- All existing libraries, either third-party or in-house, can be integrated into the embedded Linux system
  - Taking into account, of course, the limitation of the embedded systems in terms of performance, storage and memory

Programming language

- The default programming language for system-level application in Linux is usually C
  - The C library is already present on your system, nothing to add
- C++ can be used for larger applications
  - The C++ library must be added to the system
  - Some libraries, including Qt, are developed in C++ so they need the C++ library on the system anyway
- Scripting languages can also be useful for quick application development, web applications or scripts
  - But they require an interpreter on the embedded system and have usually higher memory consumption and slightly lower performance
- Languages: Python, Perl, Lua, Ada, Fortran, etc.

C library or higher-level libraries?

- For many applications, the C library already provides a relatively large set of features
  - file and device I/O, networking, threads and synchronization, inter-process communication
  - Thoroughly described in the glibc manual, or in any *Linux system programming* book
  - However, the API carries a lot of history and is not necessarily easy to grasp for new comers
- Therefore, using a higher level framework, such as Qt or the Gtk stack, might be a good idea
  - These frameworks are not only graphical libraries, their core is separate from the graphical part
  - But of course, these libraries have some memory and storage footprint, in the order of a few megabytes

Building your applications

- For simple applications that do not need to be really portable or provide compile-time configuration options, a simple Makefile will be sufficient
- For more complicated applications, or if you want to be able to run your application on a desktop Linux PC and on the target device, using a build system is recommended
  - Look at the *autotools* (ancient, complicated but very widely used) or *CMake* (modern, simpler, smaller but growing user base)
- The QT library is a special case, since it comes with its own build system for applications, called *qmake*.

Simple Makefile (1)

Case of an application that only uses the C library, contains two source files and generates a single binary

```
CROSS_COMPILE?=arm-linux-
CC=$(CROSS_COMPILE)gcc
OBJS=foo.o bar.o

all: foobar

foobar: $(OBJS)
	$(CC) -o $@ $^

clean:
	$(RM) -f foobar $(OBJS)
```

Simple Makefile (2)

- Case of an application that uses the Glib and the GPS libraries

```
CROSS_COMPILE?=arm-linux-
LIBS=libgps glib-2.0
OBJS=foo.o bar.o

CC=$(CROSS_COMPILE)gcc
CFLAGS=$(shell pkg-config --cflags $(LIBS))
LDFLAGS=$(shell pkg-config --libs $(LIBS))

all: foobar

foobar: $(OBJS)
        $(CC) -o $@ $^ $(LDFLAGS)

clean:
        $(RM) -f foobar $(OBJS)
```

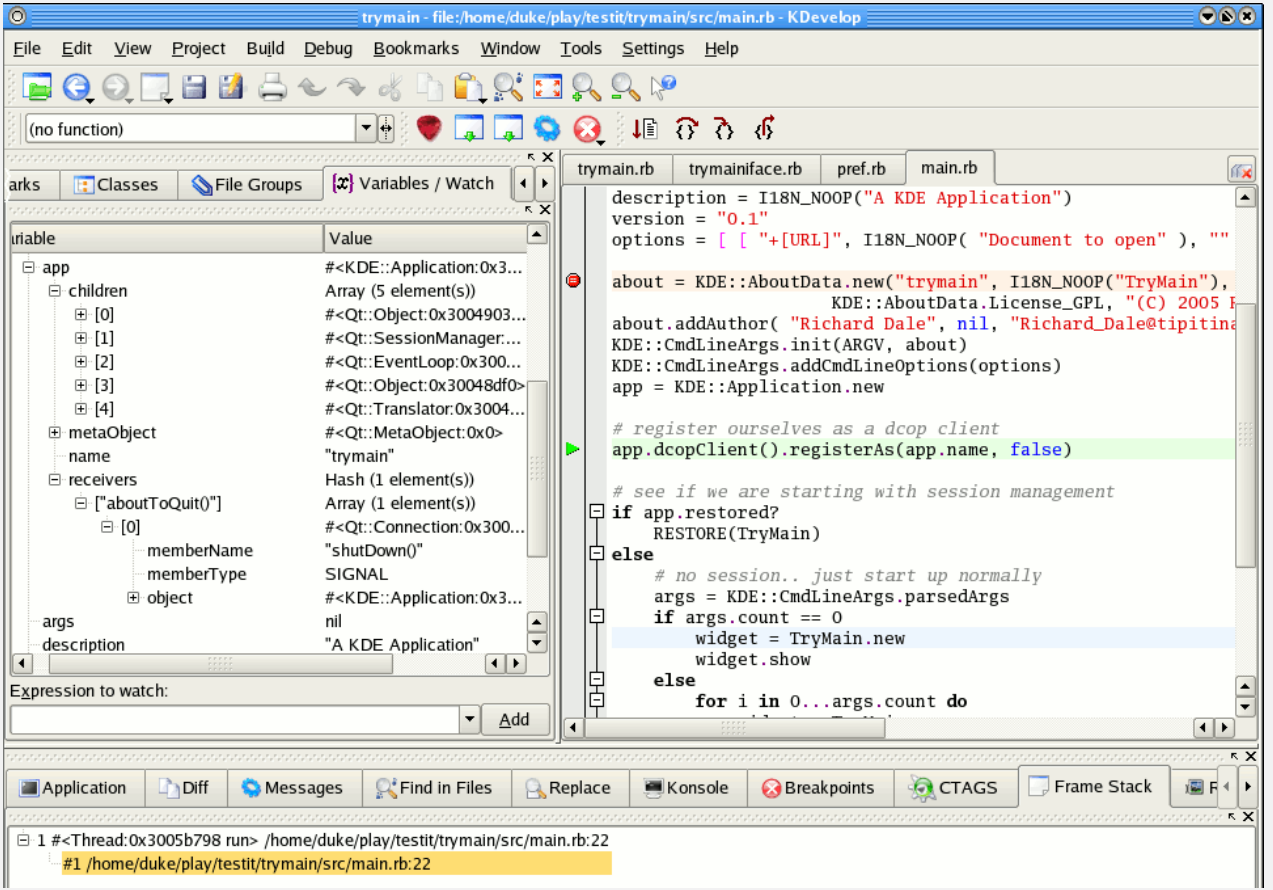
KDevelop

http://kdevelop.org

- A full featured IDE!
- License: GPL
- Supports many languages: Ada, C, C++, Database, Java, Perl, PHP, Python, Ruby, Shell
- Supports many kinds of projects: KDE, but also GTK, Gnome, kernel drivers, embedded (Opie)...
- Many features: editor, syntax highlighting, code completion, compiler interface, debugger interface, file manager, class browser...

Nice overview: http://en.wikipedia.org/wiki/Kdevelop

KDevelop screenshot



Eclipse (1)

http://www.eclipse.org/

- An extensible, plug-in based software development kit, typically used for creating IDEs.
- Supported by the Eclipse foundation, a non-profit consortium of major software industry vendors (IBM, Intel, Borland, Nokia, Wind River, Zend, Computer Associates...).
- Free Software license (Eclipse Public License). Incompatible with the GPL.
- Supported platforms: GNU/Linux, Unix, Windows

Extremely popular: created a lot of attraction.

Eclipse (2)

- Eclipse is actually a platform composed of many projects: http://www.eclipse.org/projects/
  - Some projects are dedicated at integrating into Eclipse features useful for embedded developers (cross-compilation, remote development, remote debugging, etc.)
- The platform is used by major embedded Linux software vendors for their (proprietary) system development kits: MontaVista DevRocket, TimeSys TimeStorm, Wind River Workbench, Sysgo ELinOS.

Eclipse is a huge project. It would require an entire training session!

Other popular solutions

- Many embedded Linux developers simply use **Vim** or **Emacs**. They can integrate with debuggers, source code browsers such as *cscope*, offer syntax highlighting and more.
- **Geany** is an easy-to-use graphical code editor.
- **CodeBlocks** is also quite popular, since it's also available on the Windows platform.

All those editors are available in most Linux distributions, simply install them and try them out!

Version control systems

Real projects can't do without them

- Allow multiple developers to contribute on the same project. Each developer can see the latest changes from the others, or choose to stick with older versions of some components.
- Allow to keep track of changes, and revert them if needed.
- Allow developers to have their own development branch (branching)
- Supposed to help developers resolving conflicts with different branches (merging)

Traditional version control systems

Rely on a central repository. The most popular open-source ones:

- **CVS - Concurrent Versions System**
  - Still quite popular in enterprise contexts. Almost no longer exists in the open-source community.
  - Should no longer be used for new projects
  - http://en.wikipedia.org/wiki/Concurrent_Versions_System
- **Subversion**
  - Created as a replacement of CVS, removing many of its limitations.
  - Commits on several files, proper renaming support, better performance, etc.
  - The user interface is very similar to CVS
  - http://en.wikipedia.org/wiki/Subversion_(software)

Distributed source control systems (1)

No longer have a central repository

- More adapted to the way the Free Software community develops software and organizes
- Allows each developer to have a full local history of the project, to create local branches. Makes each developer's work easier.
- People get working copies from other people's working copies, and exchange changes between themselves. Branching and merging is made easier.
- Make it easier for new developers to join, making their own experiments without having to apply for repository access.

Distributed source control systems (2)

- **Git**
  - Initially designed and developed by Linus Torvalds for Linux kernel development
  - Extremely popular in the community, and used by more and more projects (kernel, U-Boot, Barebox, uClibc, GNOME, X.org, etc.)
  - Outstanding performance, in particular in big projects
  - http://en.wikipedia.org/wiki/Git_(software)
- **Mercurial**
  - Another system, created with the same goals as Git.
  - Used by some big projects too
  - http://en.wikipedia.org/wiki/Mercurial

http://en.wikipedia.org/wiki/Version_control_systems#Distributed_revision_control

GDB

The **GNU Project Debugger**
http://www.gnu.org/software/gdb/

- The debugger on GNU/Linux, available for most embedded architectures.
- Supported languages: C, C++, Pascal, Objective-C, Fortran, Ada...
- Console interface (useful for remote debugging).
- Graphical front-ends available.
- Can be used to control the execution of a program, set breakpoints or change internal variables. You can also use it to see what a program was doing when it crashed (by loading its memory image, dumped into a core file).

See also http://en.wikipedia.org/wiki/Gdb

GDB crash course

A few useful GDB commands

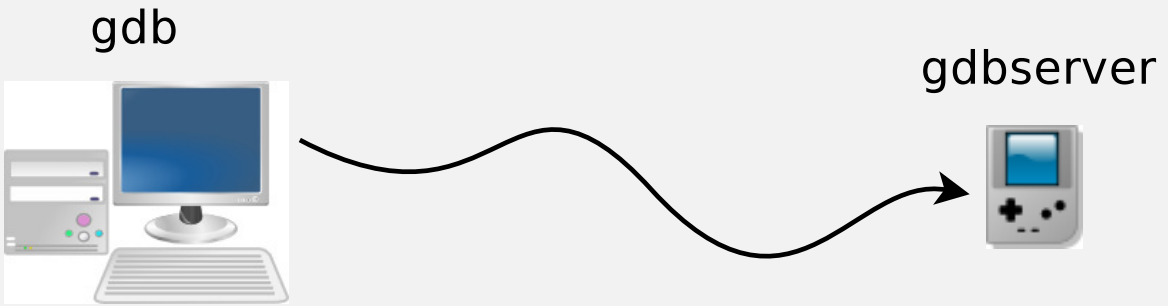
- `break foobar`
  puts a breakpoint at the entry of function `foobar()`
- `break foobar.c:42`
  puts a breakpoint in `foobar.c`, line 42
- `print var` or `print task->files[0].fd`
  prints the variable `var`, or a more complicated reference. GDB can also nicely display structures with all their members
- `continue`
  continue the execution
- `next`
  continue to the next line, stepping over function calls
- `step`
  continue to the next line, entering into subfunctions

GDB graphical front-ends

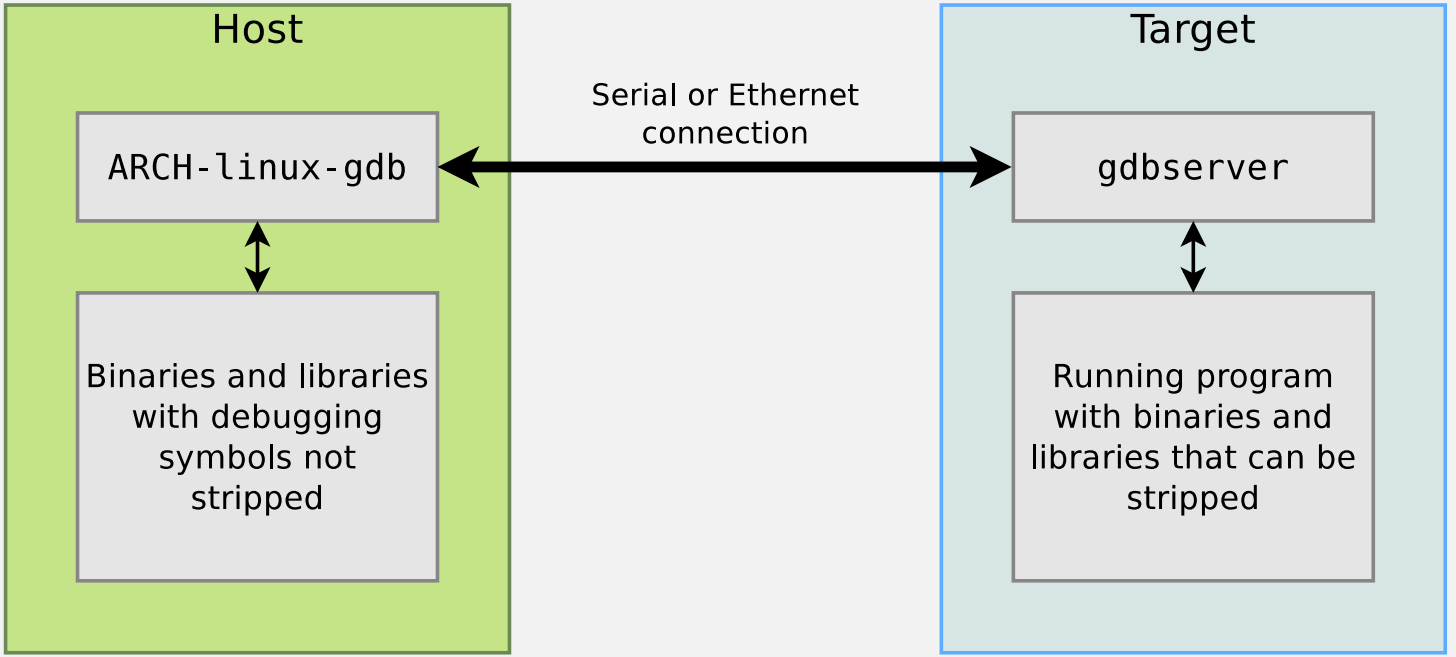
- **DDD** - Data Display Debugger
  http://www.gnu.org/software/ddd/
  A popular graphical front-end, with advanced data plotting capabilities.
- **GDB/Insight**
  http://sourceware.org/insight/
  From the GDB maintainers.
- **KDbg**
  http://www.kdbg.org/
  Another front-end, for the K Display Environment.
- Integration with other IDEs: Eclipse, Emacs, KDevelop, etc.

Remote debugging

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



Remote debugging: architecture



Remote debugging: usage

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

Post mortem analysis

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

memcheck

http://hald.dnsalias.net/projects/memcheck/

- GNU GPL tool for dynamic memory checking
- Works by replacing glibc's memory management functions by its own.
- Supports most useful CPU architectures.

DUMA

Detect Unintended Memory Access
http://duma.sourceforge.net/

- Fork and replacement for Electric Fence
- Stops your program on the exact instruction that overruns or underruns a `malloc()` memory buffer.
- GDB will then display the source-code line that causes the bug.
- Works by using the virtual-memory hardware to create a red-zone at the border of each buffer - touch that, and your program stops.
- Works on any platform supported by Linux, whatever the CPU (provided virtual memory support is available).

Valgrind (1)

http://valgrind.org/

- GNU GPL Software suite for debugging and profiling programs.
- Supported platforms: Linux on x86, x86_64, ppc32, ppc64 and arm (armv7 only: Cortex A8, A9 and A5)
- Can detect many memory management and threading bugs.
- Profiler: provides information helpful to speed up your program and reduce its memory usage.
- The most popular tool for this usage. Even used by projects with hundreds of programmers.

Valgrind (2)

- Can be used to run any program, without the need to recompile it.
- Example usage
  `Valgrind --leak-check=yes ls -la`
- Works by adding its own instrumentation to your code and then running in on its own virtual cpu core.
  Significantly slows down execution, but still fine for testing!
- More details on http://valgrind.org/info/ and http://valgrind.org/docs/manual/coregrind_core.html#howworks

strace

System call tracer
http://sourceforge.net/projects/strace/

- Available on all GNU/Linux systems
  Can be built by your cross-compiling toolchain generator.
- Allows to see what any of your processes is doing:
  accessing files, allocating memory...
  Often sufficient to find simple bugs.
- Usage:
  `strace <command>` (starting a new process)
  `strace -p <pid>` (tracing an existing process)

See `man strace` for details.

strace example output

```
> strace cat Makefile
execve("/bin/cat", ["cat", "Makefile"], [/* 38 vars */]) = 0
brk(0) = 0x98b4000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f85000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111585, ...}) = 0
mmap2(NULL, 111585, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f69000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0320h\1\0004\0\0\0\344"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442180, ...}) = 0
mmap2(NULL, 1451632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e06000
mprotect(0xb7f62000, 4096, PROT_NONE) = 0
mmap2(0xb7f63000, 12288, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15c) = 0xb7f63000
mmap2(0xb7f66000, 9840, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3) = 0
```

ltrace

A tool to trace library calls used by a program and all the signals it receives

- Very useful complement to strace, which shows only system calls.
- Of course, works even if you don't have the sources
- Allows to filter library calls with regular expressions, or just by a list of function names.
- Manual page: http://linux.die.net/man/1/ltrace

See http://en.wikipedia.org/wiki/Ltrace for details

ltrace example output

```
ltrace nedit index.html
sscanf(0x8274af1, 0x8132618, 0x8248640, 0xbfaadfe8, 0) = 1
sprintf("const 0", "const %d", 0) = 7
strcmp("startScan", "const 0") = 1
strcmp("ScanDistance", "const 0") = -1
strcmp("const 200", "const 0") = 1
strcmp("$list_dialog_button", "const 0") = -1
strcmp("$shell_cmd_status", "const 0") = -1
strcmp("$read_status", "const 0") = -1
strcmp("$search_end", "const 0") = -1
strcmp("$string_dialog_button", "const 0") = -1
strcmp("$rangeset_list", "const 0") = -1
strcmp("$calltip_ID", "const 0") = -1
```

ltrace summary

Example summary at the end of the ltrace output (-c option)

```
Process 17019 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000050          50         1           set_thread_area
  0.00    0.000000           0        48           read
  0.00    0.000000           0        44           write
  0.00    0.000000           0        80        63 open
  0.00    0.000000           0        19           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         1           mprotect
  0.00    0.000000           0        19           mmap2
  0.00    0.000000           0        50        46 stat64
  0.00    0.000000           0        18           fstat64
------ ----------- ----------- --------- --------- ----------------
100.00    0.000050 288 111 total
```

OProfile

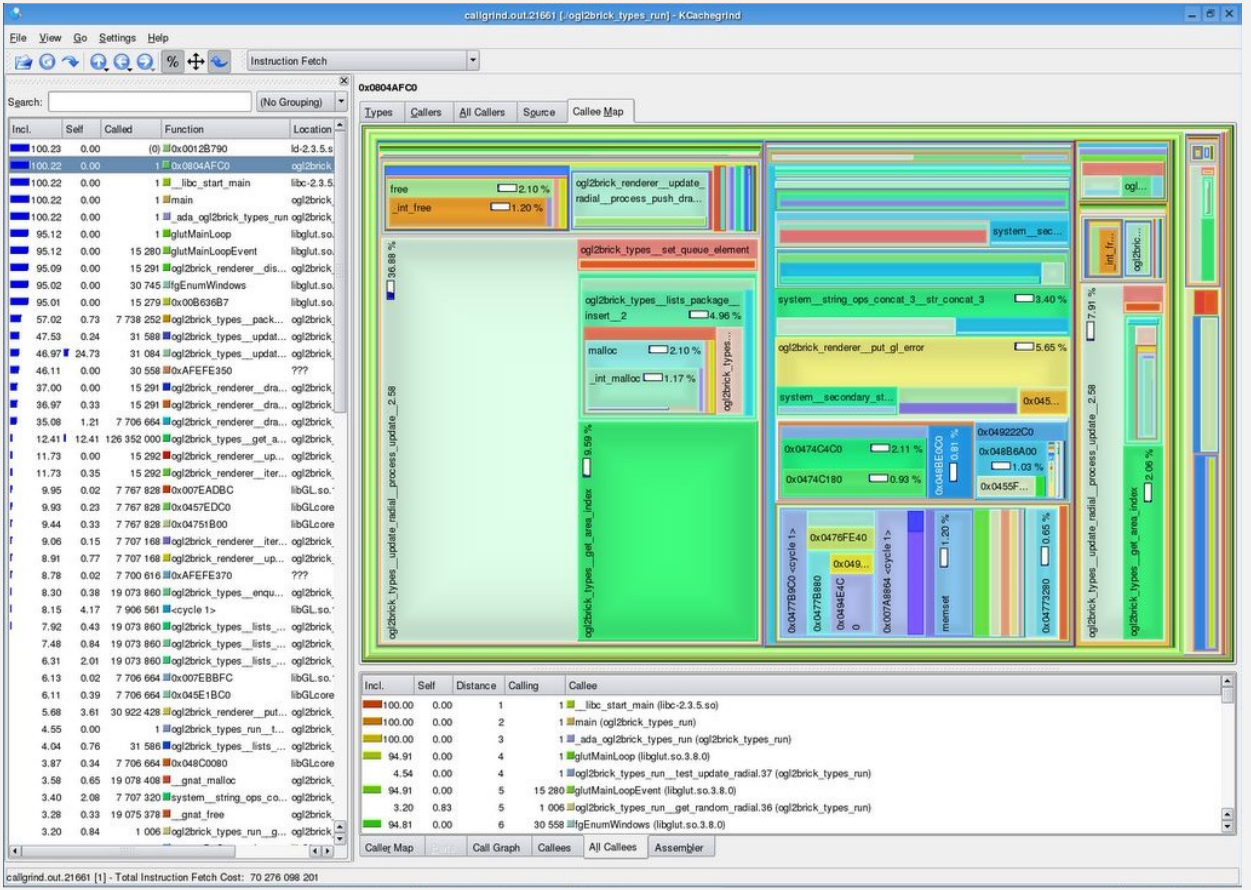
http://oprofile.sourceforge.net

- A system-wide profiling tool
- Can collect statistics like the top users of the CPU.
- Works without having the sources.
- Requires a kernel patch to access all features, but is already available in a standard kernel.
- Requires more investigation to see how it works.
- Ubuntu/Debian packages: `oprofile`, `oprofile-gui`

Callgrind / KCachegrind

- **Cachegrind** / **Callgrind**: part of the Valgrind tool suite
  Collects function call statistics and call graphs. Useful to know in which functions most time is spent.
- KCachegrind: http://kcachegrind.sourceforge.net/
  An amazing visualizer for Cachegrind / Callgrind data.
- KCachegrind can also import data from other profilers (such as OProfile), and from profiling output from Python, Perl and PHP.

KCachegrind screenshot



Developing on Windows!?

Using a GNU/Linux workstation is the easiest way to create software for GNU/Linux or embedded Linux

- You use the same tools and environment as all community developers do. Much fewer issues you are the only one to face.
- You get familiar with the system. Essential for understanding issues.

However, some developers have no choice: Windows is the only desktop OS allowed in their company.

Cygwin

http://cygwin.com/
Linux (POSIX)-like environment for Windows

- 2 components:
  Linux API emulation layer: `cygwin1.dll`
  A collection of tools originally found in GNU/Linux
- Allows to compile and run many GNU/Linux programs on Windows: shells, compiler, http servers, X Window, GTK...
- Very easy to install. Can choose which tools to download and install.
- For embedded Linux system developers: makes it possible to use GNU toolchains (compiled for Windows) required to build Linux binaries (kernel, libraries or applications).

Cygwin limitations

Cygwin is not a complete substitute for a real GNU/Linux system.

- Almost all developers work on GNU/Linux or on another Unix platform (typically BSD). Don't expect them to test that their tools build on Windows with Cygwin.
- The number of Cygwin users is quite small.
  You may be the first to face or report building issues on this platform for a given compiler or tool version.
- Cygwin is very slow.

So, the best solution is to run Linux inside Windows!

VMware

http://en.wikipedia.org/wiki/VMware

- License: proprietary
- Can run a GNU/Linux PC from Windows, almost at the host speed.
- VMware Player is now available free of charge. Many Free Software system images available for download.

The most popular solution in the corporate world.

VirtualBox

http://virtualbox.org from Sun Microsystems

- PC emulation solution available on both Windows and GNU/Linux
- 2 licenses:
  - Proprietary: free of cost for personal use and evaluation.
    Binaries available for Windows. Full features.
  - Open Source Edition (OSE): GPL license.
    Most features (except in particular USB support).
    No binaries released for Windows so far (but possible).
- Based on QEMU's core engine. Performance similar to that of VMware.

See http://en.wikipedia.org/wiki/VirtualBox

Books

- **Embedded Linux Primer, Second Edition, Prentice Hall**
  By Christopher Hallinan, October 2010
  Covers a very wide range of interesting topics.

- **Building Embedded Linux Systems, O'Reilly**
  By Karim Yaghmour, Jon Masters, Gilad Ben-Yossef and Philippe Gerum, and others (including Michael Opdenacker), August 2008
  http://oreilly.com/catalog/9780596529680/

- **Embedded Linux System Design and Development**
  P. Raghavan, A. Lad, S. Neelakandan, Auerbach, Dec. 2005.
  http://free-electrons.com/redirect/elsdd-book.html
  Useful book covering most aspects of embedded Linux system development (kernel and tools).

Web sites

- **ELinux.org**, http://elinux.org, a Wiki entirely dedicated to embedded Linux. Lots of topics covered: real-time, filesystem, multimedia, tools, hardware platforms, etc. Interesting to explore to discover new things.
- **LWN**, http://lwn.net, very interesting news site about Linux in general, and specifically about the kernel. Weekly newsletter, available for free after one week for non-paying visitors.
- **Linux Gizmos**, http://linuxgizmos.com, a news site about embedded Linux, mainly oriented on hardware platforms related news.

International conferences

Useful conferences featuring embedded Linux and kernel topics

- Embedded Linux Conference: http://embeddedlinuxconference.com/
  Organized by the CE Linux Forum: California (San Francisco, Spring), in Europe (Fall). Very interesting kernel and userspace topics for embedded systems developers. Presentation slides freely available
- Linux Plumbers, http://linuxplumbersconf.org
  Conference on the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc.
- FOSDEM: http://fosdem.org (Brussels, February)
  For developers. Presentations about system development.
- Android Builder Summit: http://events.linuxfoundation.org/events/android-builders-summit
- Free-electrons free conference videos on http://free-electrons.com/community/videos/conferences/!

Last slide

Thank you!
And may the Source be with you