

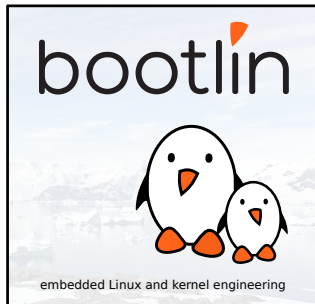


Embedded Linux system development

© Copyright 2004-2021, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: June 29, 2021.

Document updates and sources:
<https://bootlin.com/doc/training/embedded-linux>

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





Rights to copy

© Copyright 2004-2021, Bootlin

License: Creative Commons Attribution - Share Alike 3.0

<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

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.

Document sources: <https://github.com/bootlin/training-materials/>



Hyperlinks in the document

There are many hyperlinks in the document

- ▶ Regular hyperlinks:

<https://kernel.org/>

- ▶ Kernel documentation links:

[dev-tools/kasan](#)

- ▶ Links to kernel source files and directories:

[drivers/input/](#)

[include/linux/fb.h](#)

- ▶ Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):

[platform_get_irq\(\)](#)

[GFP_KERNEL](#)

[struct file_operations](#)



Company at a glance

- ▶ Engineering company created in 2004, named "Free Electrons" until Feb. 2018.
- ▶ Locations: Orange, Toulouse, Lyon (France)
- ▶ Serving customers all around the world
- ▶ Head count: 13 - Only Free Software enthusiasts!
- ▶ Focus: Embedded Linux, Linux kernel, build systems and low level Free and Open Source Software for embedded and real-time systems.
- ▶ Feb. 2021: Bootlin is the 20th all-time Linux kernel contributor
- ▶ Activities: development, training, consulting, technical support.
- ▶ Added value: get the best of the user and development community and the resources it offers.



No.1	Unknown	140019(15.26%)
No.2	Intel	94806(10.33%)
No.3	Red Hat	78140(8.52%)
No.4	Hobbyists	73603(8.02%)
No.5	Novell	39218(4.27%)
No.6	IBM	35085(3.82%)
No.7	Linaro	28288(3.08%)
No.8	AMD	22426(2.44%)
No.9	Google	20489(2.23%)
No.10	Renesas Electronics	18443(2.01%)
No.11	Oracle	17729(1.93%)
No.12	Samsung	17514(1.91%)
No.13	Texas Instruments	16372(1.78%)
No.14	HuaWei	13377(1.46%)
No.15	Mellanox Technologies	11477(1.25%)
No.16	ARM	8919(0.97%)
No.17	Academics	8560(0.93%)
No.18	Consultants	8073(0.88%)
No.19	Broadcom	8011(0.87%)
No.20	Bootlin	7611(0.83%)
No.21	NXP	7549(0.82%)
No.22	Linutronix	7430(0.81%)
No.23	NVIDIA	6951(0.76%)
No.24	Canonical	6855(0.75%)
No.25	Linux Foundation	6369(0.69%)
No.26	Code Aurora Forum	6260(0.68%)
No.27	Pengutronix	6201(0.68%)
No.28	VISION Engraving and Routing Systems	6045(0.66%)
No.29	Analog Devices	5944(0.65%)
No.30	Fujitsu	5120(0.56%)
No.31	QUALCOMM	4903(0.53%)
No.32	Freescale	4694(0.51%)
No.33	Wolfson Microelectronics	4180(0.46%)
No.34	Marvell	4178(0.46%)
No.35	Nokia	4097(0.45%)
No.36	Cisco	4071(0.44%)
No.37	Parallels	3841(0.42%)
No.38	Imagination Technologies	3774(0.41%)
No.39	Facebook	3484(0.38%)
No.40	QLogic	3394(0.37%)
No.41	ST Microelectronics	3188(0.35%)
No.42	Astaro	2981(0.32%)
No.43	NetApp	2860(0.31%)

Top Linux contributors since git (2005)



Bootlin on-line resources

- ▶ All our training materials and technical presentations:
<https://bootlin.com/docs/>
- ▶ Technical blog:
<https://bootlin.com/>
- ▶ Quick news (Mastodon):
<https://fosstodon.org/@bootlin>
- ▶ Quick news (Twitter):
<https://twitter.com/bootlincom>
- ▶ Quick news (LinkedIn):
<https://www.linkedin.com/company/bootlin>
- ▶ Elixir - browse Linux kernel sources on-line:
<https://elixir.bootlin.com>



Mastodon is a free and decentralized social network created in the best interests of its users.

Image credits: Jin Nguyen - <https://frama.link/bQwcWHTP>

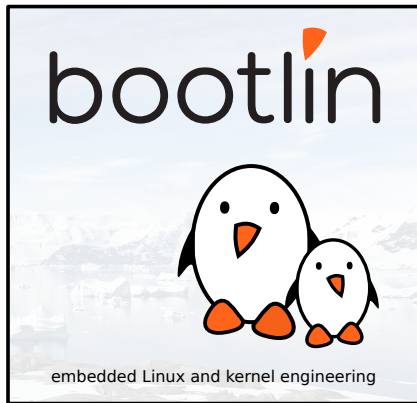


Generic course information

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!

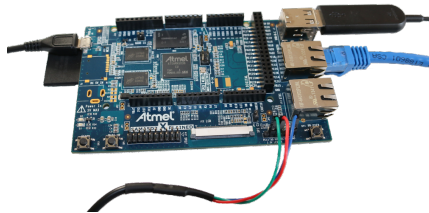




Hardware used in this training session

Using Microchip (formerly Atmel) SAMA5D3 Xplained boards in all practical labs

- ▶ SAMA5D36 (ARM 32 bit Cortex A5) CPU from Microchip
- ▶ 256 MB DDR2 RAM, 256 MB NAND flash
- ▶ 2 Ethernet ports (Gigabit + 100 Mbit)
- ▶ 2 USB 2.0 host, 1 USB device
- ▶ 1 MMC/SD slot
- ▶ 3.3 V serial port (like Beaglebone Black)
- ▶ Misc: Arduino R3-compatible header, JTAG, buttons, LEDs
- ▶ Currently sold at 93 EUR + VAT at Mouser



Board and CPU documentation, design files, software: <https://bit.ly/2Ghv10p>



Shopping list: hardware for this course

- ▶ Microchip SAMA5D3 Xplained board - Available from Microchip and multiple distributors (Mouser, Digikey...). See <https://bit.ly/2Ghv10p> (Microchip's website)
- ▶ USB Serial Cable - 3.3 V - Female ends (for serial console): Olimex: <https://frama.link/zWJDToXP>
- ▶ Logitech USB H340 audio headsets (for cross-compiling labs) https://support.logitech.com/en_us/product/usb-headset-h340
- ▶ An SD card with at least 128 MB of capacity





Labs proposed on another platform

After this course, you can also run most labs on the STM32MP157A-DK1 Discovery board (<https://www.st.com/en/evaluation-tools/stm32mp157a-dk1.html>)



Lab instructions available on <https://bootlin.com/doc/training/embedded-linux-4d/>



Labs proposed on another platform

After this course, you can also run most labs on the QEMU emulated ARM Versatile Express Cortex A9 board



Lab instructions available on

<https://bootlin.com/doc/training/embedded-linux-qemu/>



Participate!

During the lectures...

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



Practical lab guidelines

During practical labs...

- ▶ We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- ▶ So, if you are more than 8 participants, please form up to 8 working groups.
- ▶ Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- ▶ Don't hesitate to copy and paste commands from the PDF slides and labs.



Advise: write down your commands!

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

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

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

Lab commands

Cross-compiling kernel:

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-
make sama5_defconfig
```

Booting kernel through tftp:

```
setenv bootargs console=ttyS0 root=/dev/nfs
setenv bootcmd tftp 0x21000000 zimage; tftp
0x22000000 dtb; bootz 0x21000000 - 0x2200...
```

Making ubifs images:

```
mkfs.ubifs -d rootfs -o root.ubifs -e 124KiB
-m 2048 -c 1024
```

Encountered issues:

Restart NFS server after editing /etc/exports!



Cooperate!

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

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

embedded-linux-nov2020 Channel for

Srinath

Michael What should be CROSS_COMPILE variable set to in case of the xplained board? I ran into some issues with my USB hub so doing the u-boot again

Michael you should look at the name of the cross-compiler in the toolchain's bin/ directory. CROSS_COMPILE should be set to what's before "gcc" in the name, including the trailing "-". Like if the compiler is arm-buildroot-linux-gcc, CROSS_COMPILE should be arm-buildroot-linux-

2 messages deleted.


Srinath

Will ask them here since I am going to do labs after the session is over! Thanks!

Srinath changed their display name to **srinath.**

@accedre-matrix.org

I tried to finalize Kernel - Cross-compiling task, but my system is not able to restart the new kernel. Does anyone know what can be the root cause?



arnaud.a

I had the same because I accidentally renamed the module... from the journal

Send an encrypted message...



Command memento sheet

- ▶ This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- ▶ It saves us 1 day of UNIX / Linux command line training.
- ▶ Our best tip: in the command line shell, always hit the **Tab** key to complete command names and file paths. This avoids 95% of typing mistakes.
- ▶ Get an electronic copy on https://bootlin.com/doc/legacy/command-line/command_memento.pdf





vi basic commands

- ▶ The `vi` editor is very useful to make quick changes to files in an embedded target.
- ▶ Though not very user friendly at first, `vi` is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- ▶ Get an electronic copy on https://bootlin.com/doc/legacy/command-line/vi_memento.pdf
- ▶ You can also take the quick tutorial by running `vimtutor`. This is a worthy investment!



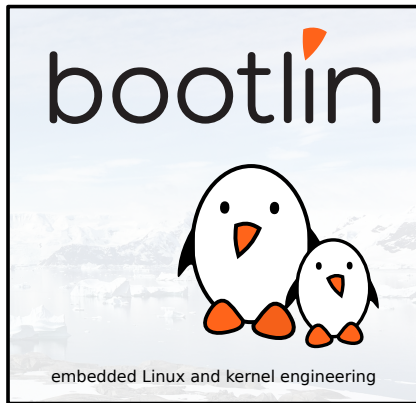


Introduction to Embedded Linux

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

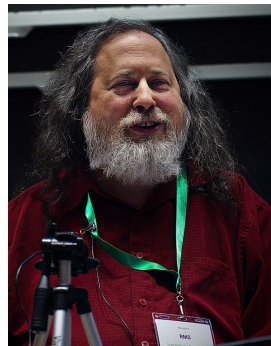
Corrections, suggestions, contributions and translations are welcome!





Birth of Free Software

- ▶ 1983, Richard Stallman, **GNU project** and the **free software** concept. Beginning of the development of *gcc*, *gdb*, *glibc* and other important tools
- ▶ 1991, Linus Torvalds, **Linux kernel project**, a UNIX-like operating system kernel. Together with GNU software and many other open-source components: a completely free operating system, GNU/Linux
- ▶ 1995, Linux is more and more popular on server systems
- ▶ 2000, Linux is more and more popular on **embedded systems**
- ▶ 2008, Linux is more and more popular on mobile devices and phones
- ▶ 2012, Linux is available on cheap, extensible hardware: Raspberry Pi, BeagleBone Black



Richard Stallman in 2019
Image credits (Wikipedia):
<https://frama.link/qC73jkk4>



Free software?

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



What is embedded Linux?

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



Advantages of Linux and open-source for embedded systems



Re-using components

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



Low cost

- ▶ Free software can be duplicated on as many devices as you want, free of charge.
- ▶ If your embedded system uses only free software, you can reduce the cost of software licenses to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
- ▶ Of course, using Linux is not free of cost. You still need substantial learning and engineering efforts to achieve your goals.
- ▶ **Allows to have a higher budget for the hardware or to increase the company's skills and knowledge**



Full control

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

- ▶ 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 high-quality support: you can directly contact the main developers of the component you are using. The likelihood of getting an answer doesn't depend what company you work for.
- ▶ Often better than traditional support, but one needs to understand how the community works to properly use the community support possibilities
- ▶ **Allows to speed up the resolution of problems when developing your system**



Taking part into the community

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



A few examples of embedded systems running Linux



Wireless routers



Image credits: Evan Amos (<https://bit.ly/2JzDIkv>)



Video systems



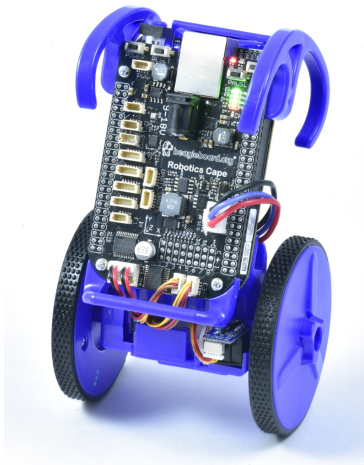
Image credits: <https://bit.ly/2HbwyVq>



Bike computers



Product from BLOKS (<http://bloks.de>). Permission to use this picture only in this document, in updates and in translations.



eduMIP robot (<https://www.ucsdrobotics.org/edumip>)



In space

SpaceX Starlink satellites



SpaceX Falcon 9 and Falcon Heavy rockets



Mars Ingenuity Helicopter

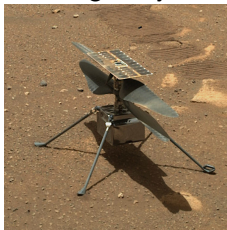


Image credits: Wikipedia



Embedded hardware for Linux systems



Processor and architecture (1)

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

- ▶ x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
- ▶ ARM, with hundreds of different *System on Chips* (SoC: CPU + on-chip devices, for all sorts of products)
- ▶ RISC-V, the rising architecture with a free instruction set (from high-end cloud computing to the smallest embedded systems)
- ▶ PowerPC (mainly real-time, industrial applications)
- ▶ MIPS (mainly networking applications)
- ▶ Microblaze (Xilinx), Nios II (Altera): soft cores on FPGAs
- ▶ Others: ARC, m68k, Xtensa, SuperH...



Processor and architecture (2)

- ▶ Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- ▶ Linux does not support small microcontrollers (8 or 16 bit)
- ▶ 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.
 - ▶ **Block storage:** SD/MMC/eMMC, USB mass storage, SATA, etc,
 - ▶ **Raw flash storage** is supported too, both NAND and NOR flash, with specific filesystems
- ▶ 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.

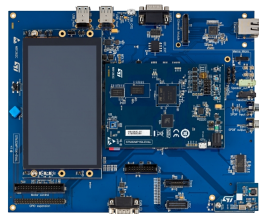


- ▶ The Linux kernel has support for many common communication buses
 - ▶ I2C
 - ▶ SPI
 - ▶ 1-wire
 - ▶ SDIO
 - ▶ PCI
 - ▶ USB
 - ▶ CAN (mainly used in automotive)
- ▶ And also extensive networking support
 - ▶ Ethernet, Wifi, Bluetooth, CAN, etc.
 - ▶ IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
 - ▶ Firewalling,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



Types of hardware platforms (1)

- ▶ **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products, but best for product development.
- ▶ **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.



STM32MP157C-EV1

evaluation board

Image credits (st.com):

<https://frama.link/NySnaxuV>



PocketBeagle

Image credits (Beagleboard.org):

<https://beagleboard.org/pocket>



Types of hardware platforms (2)

- ▶ **Community development platforms**, to make a particular SoC popular and easily available. These are ready-to-use and low cost, but usually have fewer 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.



Beaglebone Black board



Olimex Open hardware
ARM laptop main board

Image credits (Olimex):

<https://www.olimex.com/Products/DIY-Laptop/>



Criteria for choosing the hardware

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

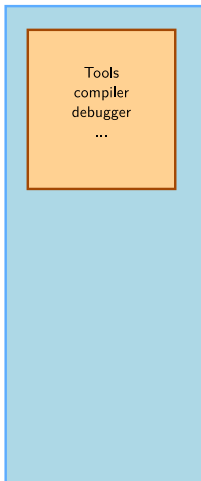


Embedded Linux system architecture

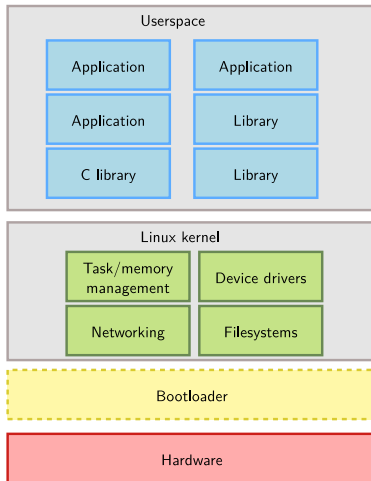


Host and target

Development PC (*host*)



Embedded system (*target*)



The bootloader disappears
after starting the kernel



Software components

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



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 course](#)*

- ▶ **System integration**

- ▶ Integrate all the components, bootloader, kernel, third-party libraries and applications and in-house applications into a working system
- ▶ Purpose of *this* course

- ▶ **Development of applications**

- ▶ Normal Linux applications, but using specifically chosen libraries

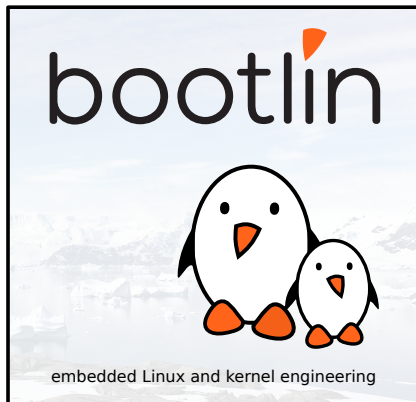


Embedded Linux development environment

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





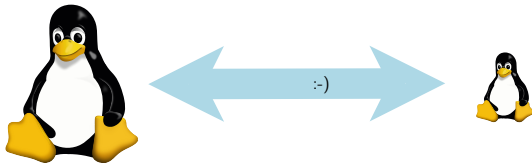
- ▶ Two ways to switch to embedded Linux
 - ▶ Use **solutions provided and supported by vendors** like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
 - ▶ Use **community solutions**. They are completely open, supported by the community.
- ▶ In Bootlin 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 GNU/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.
- ▶ 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.
- ▶ If you are stuck with a Windows desktop, at least you should use GNU/Linux in a virtual machine (such as VirtualBox which is open source), though there could be a small performance penalty. With Windows 10, you can also run your favorite native Linux distro through Windows Subsystem for Linux (WSL2)





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!



Image credits:

<https://tinyurl.com/f4zxj5kw>



Linux root and non-root users

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



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.
- ▶ Note: *Snap* and *Flatpak* offer new ways of packaging applications in a self-contained way. See <https://www.atechtown.com/flatpak-vs-snap/>.



Managing software packages (1)

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

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



Managing software packages (2)

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

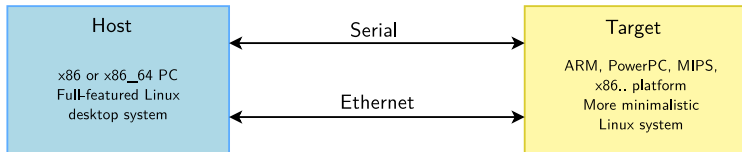
Further details on package management:

<https://www.debian.org/doc/manuals/debian-reference/ch02.en.html>



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





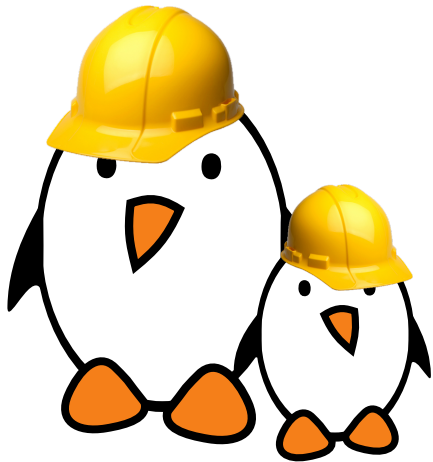
Serial line communication program

- ▶ 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*, *screen* and the new *tio* (<https://github.com/tio/tio>).
- ▶ In this training session, we recommend using the simplest of them: *Picocom*
 - ▶ Installation with `sudo apt install picocom`
 - ▶ Run with `picocom -b BAUD_RATE /dev/SERIAL_DEVICE`.
 - ▶ Exit with `[Ctrl][a] [Ctrl][x]`
- ▶ `SERIAL_DEVICE` is typically
 - ▶ `ttUSBx` for USB to serial converters
 - ▶ `ttSx` for real serial ports
- ▶ Most frequent command: `picocom -b 115200 /dev/ttyUSB0`



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 directly copy/paste paths from the file manager to Gnome Terminal by drag-and-drop.



Prepare your lab environment

- ▶ Download and extract the lab archive

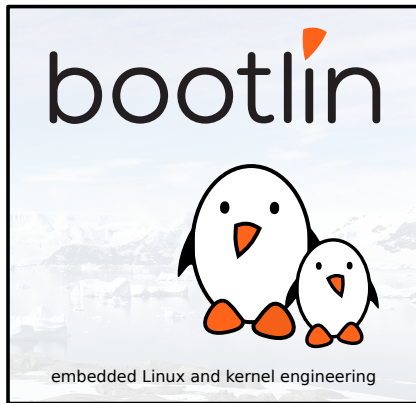


Cross-compiling toolchains

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Definition and Components

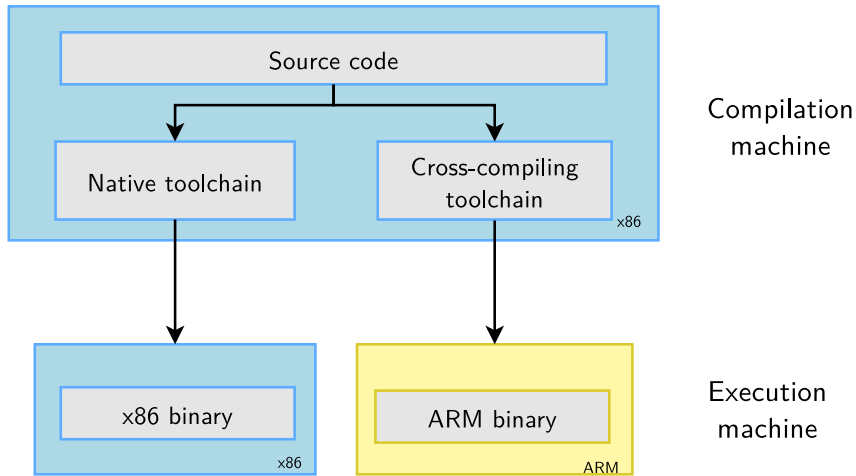


Toolchain definition (1)

- ▶ 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.



Toolchain definition (2)



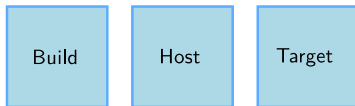


Machines in build procedures

- ▶ 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

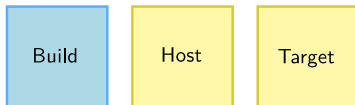


Different toolchain build procedures



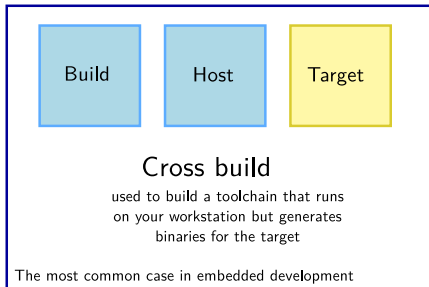
Native build

used to build the normal gcc
of a workstation



Cross-native build

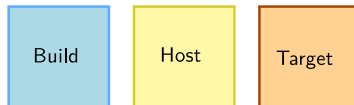
used to build a toolchain that runs on your
target and generates binaries for the target



Cross build

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

The most common case in embedded development

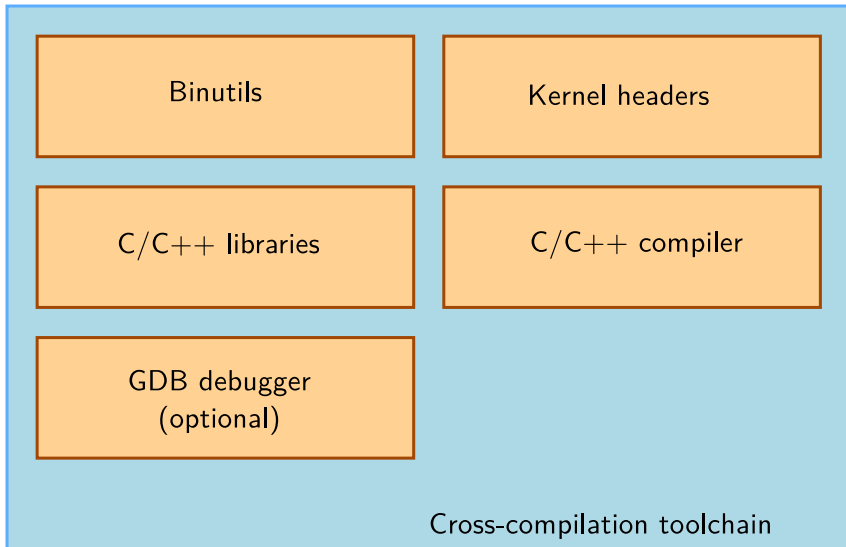


Canadian cross build

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



Components of gcc toolchains



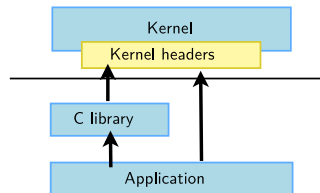


- ▶ **Binutils** is a set of tools to generate and manipulate binaries for a given CPU architecture
 - ▶ `as`, the assembler, that generates binary code from assembler source code
 - ▶ `ld`, the linker
 - ▶ `ar`, `ranlib`, to generate `.a` archives (static libraries)
 - ▶ `objdump`, `readelf`, `size`, `nm`, `strings`, to inspect binaries. Very useful analysis tools!
 - ▶ `objcopy`, to modify binaries
 - ▶ `strip`, to strip parts of binaries that are just needed for debugging (reducing their size).
- ▶ GNU Binutils: <https://www.gnu.org/software/binutils/>, GPL license
- ▶ The LLVM project now provides alternatives to GNU Binutils: `llvm-strip`, `llvm-readelf`, `lld`... (<https://www.llvm.org/docs/CommandGuide/>)



Kernel headers (1)

- ▶ The C library and compiled programs need 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/uapi/` and in `arch/<arch>/include/uapi` in the kernel sources
- ▶ The kernel headers are extracted from the kernel sources using the `headers_install` kernel Makefile target.





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>` (used by the `stat` command)

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




Kernel headers (3)

The kernel to user space ABI is **backward compatible**

- ▶ ABI = *Application Binary Interface* - It's about binary compatibility
- ▶ Kernel developers are doing their best to **never** break existing programs when the kernel is upgraded. Otherwise, users would stick to older kernels, which would be bad for everyone.
- ▶ Hence, 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 only if they don't use the recent features, otherwise they will break.

What to remember: updating your kernel shouldn't break your programs; it's usually fine to keep an old toolchain as long as it works fine for your project.



C/C++ compiler

- ▶ GCC: GNU Compiler Collection, the famous free software compiler
- ▶ <https://gcc.gnu.org/>
- ▶ Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, Go, etc. Can generate code for a large number of CPU architectures, including x86, ARM, RISC-V, and many others.
- ▶ Available under the GPL license, libraries under the GPL with linking exception.
- ▶ Alternative: Clang / LLVM compiler (<https://clang.llvm.org/>) getting increasingly popular and able to compile most programs (license: MIT/BSD type). It can offer better optimizations and make errors easier to interpret.

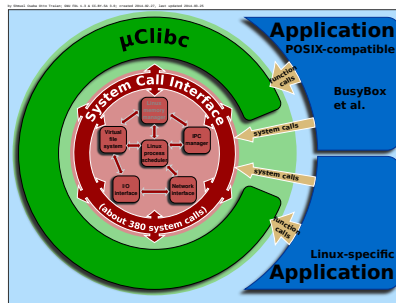




C library

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

Comparing libcs by feature: https://www.etalabs.net/compare_libcs.html



Source: Wikipedia (<https://bit.ly/2zrGve2>)



C Libraries

- ▶ License: LGPL
- ▶ C library from the GNU project
- ▶ Designed for performance, standards compliance and portability
- ▶ Found on all GNU / Linux host systems
- ▶ Of course, actively maintained
- ▶ By default, quite big for small embedded systems. On armv7hf, version 2.31: `libc`: 1.5 MB, `libm`: 432 KB, source: <https://toolchains.bootlin.com>
- ▶ But some features not needed in embedded systems can be configured out (merged from the old *eglibc* project).
- ▶ <https://www.gnu.org/software/libc/>



Image: <https://bit.ly/2EzHl6m>



- ▶ <https://uclibc-ng.org/>
- ▶ A continuation of the old uClibc project, license: LGPL
- ▶ Lightweight C library for small embedded systems
 - ▶ High configurability: many features can be enabled or disabled through a menuconfig interface.
 - ▶ Supports most embedded architectures, including MMU-less ones (ARM Cortex-M, Blackfin, etc.). The only library supporting ARM noMMU.
 - ▶ No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
 - ▶ Some features may be implemented later than on glibc (real-time, floating-point operations...)
 - ▶ Focus on size rather than performance
 - ▶ Size on armv7hf, version 1.0.34: `libc`: 712 KB, source: <https://toolchains.bootlin.com>
- ▶ Actively supported, but Yocto Project stopped supporting it.



<https://www.musl-libc.org/>

- ▶ A lightweight, fast and simple library for embedded systems
- ▶ Created while uClibc's development was stalled
- ▶ In particular, great at making small static executables
- ▶ More permissive license (MIT), making it easier to release static executables. We will talk about the requirements of the LGPL license (glibc, uClibc) later.
- ▶ Supported by build systems such as Buildroot and Yocto Project.
- ▶ Used by the Alpine Linux lightweight distribution (<https://www.alpinelinux.org/>)
- ▶ Size on armv7hf, version 1.2.0: `libc`: 748 KB, source: <https://toolchains.bootlin.com>





glibc vs uclibc-ng vs musl - small static executables

Let's compile and strip a `hello.c` program **statically** and compare the size

- ▶ With musl 1.2.0:
9,084 bytes
- ▶ With uclibc-ng 1.0.34 :
21,916 bytes.
- ▶ With glibc 2.31:
431,140 bytes

Tests run with gcc 10.0.2 toolchains for armv7-eabihf
(from <https://toolchains.bootlin.com>)



glibc vs uclibc vs musl - more realistic example

Let's compile and strip BusyBox 1.32.1 **statically**
(with the `defconfig` configuration) and compare the size

- ▶ With musl 1.2.0:
1,176,744 bytes
- ▶ With uclibc-ng 1.0.34 :
1,251,080 bytes.
- ▶ With glibc 2.31:
1,852,912 bytes

Notes:

- ▶ Tests run with `gcc 10.0.2` toolchains for `armv7-eabi`
- ▶ BusyBox is automatically compiled with `-Os` and stripped.
- ▶ Compiling with shared libraries will mostly eliminate size differences



Other smaller C libraries

- ▶ Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- ▶ They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- ▶ Choices:
 - ▶ Newlib, <https://sourceware.org/newlib/>, maintained by Red Hat, used mostly in Cygwin, in bare metal and in small POSIX RTOS.
 - ▶ Klibc, <https://en.wikipedia.org/wiki/Klibc>, from the kernel community, designed to implement small executables for use in an *initramfs* at boot time.



Advise for choosing the C library

- ▶ Advice to start developing and debugging your applications with *glibc*, which is the most standard solution.
- ▶ Then, when everything works, if you have size constraints, try to compile your app and then the entire filesystem with *uClibc* or *musl*.
- ▶ If you run into trouble, it could be because of missing features in the C library.
- ▶ In case you wish to make static executables, *musl* will be an easier choice in terms of licensing constraints. The binaries will be smaller too. Note that static executables built with a given C library can be used in a system with a different C library.



Toolchain Options

- ▶ 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 are typically 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*
- ▶ https://en.wikipedia.org/wiki/Application_Binary_Interface



Floating point support

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



CPU optimization flags

- ▶ A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC)
- ▶ However, gcc offers further options:
 - ▶ `-march` allows to select a specific target instruction set
 - ▶ `-mtune` allows to optimize code for a specific CPU
 - ▶ For example: `-march=armv7 -mtune=cortex-a8`
 - ▶ `-mcpu=cortex-a8` can be used instead to allow gcc to infer the target instruction set (`-march=armv7`) and cpu optimizations (`-mtune=cortex-a8`)
 - ▶ <https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html>
- ▶ 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`, `-mtune`, `-mcpu` options are passed
 - ▶ To compile the C library
- ▶ Even if the C library has been compiled for armv5t, it doesn't prevent from compiling bare-metal programs or the kernel for armv7.



Obtaining a Toolchain



Building a toolchain manually

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

- ▶ Lots of details to learn: many components to build, complicated configuration. Need to be familiar with building and configuring tools.
- ▶ Many decisions to make about the components (such as C library, gcc and binutils versions, ABI, floating point mechanisms...). Not trivial to find working combinations of such components!
- ▶ Need to be familiar with current gcc issues and patches on your platform
- ▶ See the *Crosstool-NG* docs/ directory for details on how toolchains are built.



Get a pre-compiled toolchain

- ▶ Solution that many people choose
 - ▶ Advantage: it is the simplest and most convenient solution
 - ▶ Drawback: you can't fine tune the toolchain to your needs
- ▶ Make sure the toolchain you find meets your requirements: CPU, endianness, C library, component versions, ABI, soft float or hard float, etc.
- ▶ Possible choices
 - ▶ Toolchains packaged by your distribution
For example, Ubuntu toolchains (glibc only):
`sudo apt install gcc-arm-linux-gnueabi`
 - ▶ Bootlin's GNU toolchains (for most architectures):
<https://toolchains.bootlin.com>
 - ▶ ARM GNU toolchains released by ARM (previously shipped by Linaro):
<https://tinyurl.com/2fkvvrju>



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 and musl, hard and soft float, many architectures
- ▶ Actively maintained
- ▶ <https://crosstool-ng.github.io/>

`.config - crosstool-NG Configuration`

Target options

```
Target Architecture (arm) --->
*** Options for arm ***
Default instruction set mode (arm) --->
[ ] Use Thumb-interworking (READ HELP)
-*- Use EABI
[*] append 'hf' to the tuple (EXPERIMENTAL)
    () Suffix to the arch-part
[ ] Omit vendor part of the target tuple
*** Generic target options ***
[ ] Build a multilib toolchain (READ HELP!!!)
[*] Attempt to combine libraries into a single directory
[*] Use the MMU
Endianness: (Little endian) --->
Bitness: (32-bit) --->
*** Target optimisations ***
(cortex-a5) Emit assembly for CPU
(vfpv4-d16) Use specific FPU
Floating point: (hardware (FPU)) --->
    () Target CFLAGS
    () Target LDFLAGS
```



Toolchain building utilities (3)

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

- ▶ **Buildroot**

- ▶ Makefile-based. Can build glibc, uClibc and musl based toolchains, for a wide range of architectures. Use `make sdk` to only generate a toolchain.
- ▶ <https://buildroot.org>

- ▶ **PTXdist**

- ▶ Makefile-based, maintained mainly by *Pengutronix*. It only supports uClibc and glibc (version 2021.03 status)
- ▶ <https://www.ptxdist.org/>

- ▶ **OpenEmbedded / Yocto Project**

- ▶ A featureful, but more complicated build system
- ▶ <http://www.openembedded.org/>
- ▶ <https://www.yoctoproject.org/>



Crosstool-NG: installation and usage

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

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

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

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

- ▶ To load a sample configuration

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

- ▶ To adjust the configuration

```
./ct-ng menuconfig or ./ct-ng nconfig (according to your preference)
```

- ▶ To build the toolchain

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



Toolchain contents

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



Time to build your toolchain

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

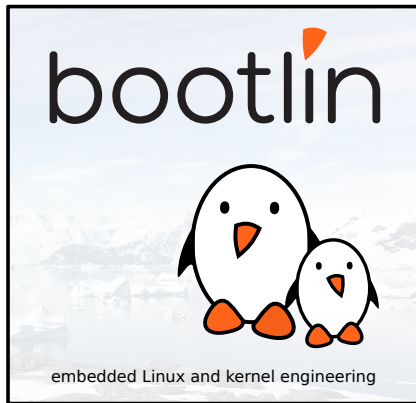


Bootloaders

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Boot Sequence

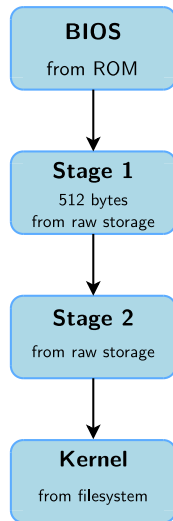


- ▶ The bootloader is a piece of code responsible for
 - ▶ Basic hardware initialization
 - ▶ Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
 - ▶ Possibly decompression of the application binary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



Bootloaders on BIOS-based x86 (1)

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





Bootloaders on x86 (2)

- ▶ GRUB, Grand Unified Bootloader, the most powerful one.
<https://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:
<https://bootlin.com/doc/legacy/grub/>
- ▶ Syslinux, for network and removable media booting (USB key, CD-ROM)
<https://kernel.org/pub/linux/utils/boot/syslinux/>



Booting on embedded CPUs: case 1

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



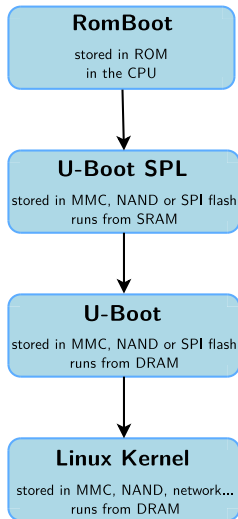


Booting on embedded CPUs: case 2

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



Booting on Microchip ARM SAMA5D3

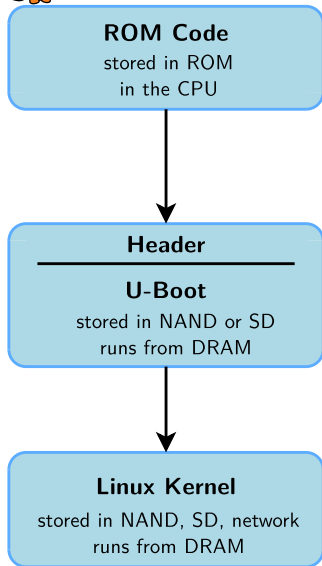


- ▶ **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 64 KB. No user interaction possible in standard boot mode.
- ▶ **U-Boot SPL**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads 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 (the bootloader no longer exists).

Note: same process on other Microchip AT91 SoCs, but the SRAM size is smaller on the older ones.



Booting on Marvell SoCs



- ▶ **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

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, RISC-V, etc.
 - ▶ The de-facto standard nowadays. We will study it in detail.
 - ▶ <https://www.denx.de/wiki/U-Boot>
- ▶ **Barebox**, an architecture-neutral bootloader created by Pengutronix.
 - ▶ It doesn't have as much hardware support as U-Boot yet.
 - ▶ U-Boot has improved quite a lot thanks to this competitor.
 - ▶ <https://www.barebox.org>

Bootloader: modern expectation



See the nice introduction to Barebox from Ahmad Fatoum at ELCE 2020:
Video: <https://youtu.be/Oj7lKbFtyM0>
Slides: <https://elinux.org/images/9/9d/Barebox-bells-n-whistles.pdf>

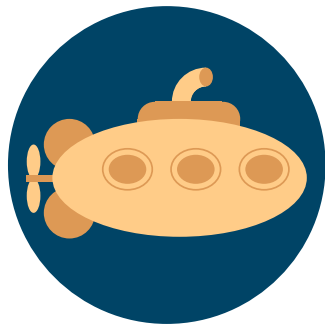


The U-boot bootloader



U-Boot is a typical free software project

- ▶ License: GPLv2 (same as Linux)
- ▶ Freely available at <https://www.denx.de/wiki/U-Boot>
- ▶ Documentation available at <https://www.denx.de/wiki/U-Boot/Documentation>
- ▶ The latest development source code is available in a Git repository: <https://gitlab.denx.de/u-boot/u-boot>
- ▶ Development and discussions happen around an open mailing-list <https://lists.denx.de/pipermail/u-boot/>
- ▶ Follows a regular release schedule. Every 2 or 3 months, a new version is released. Versions are named YYYY.MM.



U-Boot

Image credits:

<https://frama.link/rwCUFc-T>



U-Boot configuration

- ▶ Get the source code from the website or from git, and uncompress it
- ▶ The `configs/` directory contains one or several configuration file(s) 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.
 - ▶ Examples:
`configs/stm32mp15_basic_defconfig`
`configs/stm32mp15_trusted_defconfig`
- ▶ Note: U-Boot is migrating from board configuration defined in C header files (`include/configs/`) to *defconfig* like in the Linux kernel (`configs/`)
 - ▶ Not all boards have been converted to the new configuration system.
 - ▶ Many boards still have *both* hardcoded configuration settings in `.h` files, and configuration settings in `defconfig` files that can be overridden with configuration interfaces.



U-Boot configuration file

CHIP_defconfig

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



Configuring and compiling U-Boot

- ▶ U-Boot must be configured before being compiled
 - ▶ Configuration stored in a `.config` file
 - ▶ `make BOARDNAME_defconfig`
 - ▶ Where `BOARDNAME` is the name of a configuration, as visible in the `configs/` directory.
 - ▶ You can then run `make menuconfig` to further customize U-Boot's configuration!
- ▶ Make sure that the cross-compiler is available in `PATH`
- ▶ Compile U-Boot, by specifying the cross-compiler prefix.
Example, if your cross-compiler executable is `arm-linux-gcc`:
`make CROSS_COMPILE=arm-linux-`
- ▶ The main result is a `u-boot.bin` file, which is the U-Boot image. Depending on your specific platform, or what storage device you're booting from (NAND or MMC), there may be other specialized images: `u-boot.img`, `u-boot.kwb`...
- ▶ This also generates the U-Boot SPL image to be flashed together with U-Boot. The exact file name can vary too, depending on what the romcode expects.



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 the 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 2020.04 (May 26 2020 - 16:05:43 +0200)
```

```
CPU: SAMA5D36
Crystal frequency:    12 MHz
CPU clock      :    528 MHz
Master clock   :    132 MHz
DRAM:  256 MiB
NAND:   256 MiB
MMC:   Atmel mci: 0, Atmel mci: 1
Loading Environment from NAND... OK
In:    serial@ffffee00
Out:   serial@ffffee00
Err:   serial@ffffee00
Net:   eth0: ethernet@f0028000
Error: ethernet@f802c000 address not set.
```

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

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



Information commands (1)

Flash information (NOR and SPI flash)

```
=> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (R0) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C00041FFF (R0) U-Boot
```

NAND flash information

```
=> nand info

Device 0: nand0, sector size 128 KiB
  Page size      2048 b
  OOB size       64 b
  Erase size     131072 b
  subpagesize    2048 b
  options        0x40004200
  bbt options    0x00008000
```



Information commands (2)

Version details

```
=> version
```

```
U-Boot 2020.04 (May 26 2020 - 16:05:43 +0200)
```

```
arm-linux-gcc (crosstool-NG 1.24.0.105_5659366) 9.2.0
```

```
GNU ld (crosstool-NG 1.24.0.105_5659366) 2.34
```



Important commands (1)

- ▶ The exact set of commands depends on the U-Boot configuration
- ▶ `help` and `help command`
- ▶ `fatload`, loads a file from a FAT filesystem to RAM
 - ▶ Example: `fatload usb 0:1 0x21000000 zImage`
 - ▶ And also `fatinfo`, `fatls`, `fatsize`, `fatwrite`...
- ▶ `ext2load`, loads a file from an ext2 filesystem to RAM
 - ▶ And also `ext2info`, `ext2ls`, `ext2size`, `ext2write`...
- ▶ Similar commands for other filesystems: `ext4load`, `ext4ls`, `sqfsload`, `sqfsls`...
([SquashFS support contributed by Bootlin](#))...
- ▶ `tftp`, loads a file from the network to RAM (example given later)
- ▶ `ping`, to test the network
- ▶ `bootd` (can be abbreviated as `boot`), runs the default boot command, stored in `bootcmd`
- ▶ `bootz <address>`, starts a kernel image loaded at the given address in RAM



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.



U-Boot bdfinfo command

```
=> bdfinfo
arch_number = 0x00000000
boot_params = 0x20000100
DRAM bank   = 0x00000000
-> start     = 0x20000000
-> size      = 0x10000000
baudrate     = 115200 bps
TLB addr     = 0x2FFF0000
relocaddr    = 0x2FF27000
reloc off    = 0x09027000
irq_sp       = 0x2FB1DC40
sp start     = 0x2FB1DC30
Early malloc usage: 135c / 2000
fdt_blob     = 2fb1dc50
```

Allow to find valid RAM
addresses without needing
the SoC datasheet or
board manual

Source: U-Boot 2018.01
on Microchip SAMA5D3 Xplained



Environment variables: principle

- ▶ U-Boot can be configured through environment variables
 - ▶ Some specific environment variables impact the behavior of the different commands
 - ▶ Custom environment variables can be added, and used in scripts
- ▶ Environment variables are loaded from persistent storage to RAM at U-Boot startup. They can be defined or modified and saved back to storage for persistence.



Environment variables: implementation

Depending on the configuration, the U-Boot environment is typically stored in:

- ▶ At a fixed offset in NAND flash
- ▶ At a fixed offset on MMC or USB storage, before the beginning of the first partition.
- ▶ In a file (`uboot.env`) on a FAT or ext4 partition
- ▶ In a UBI volume

`.config - U-Boot 2020.07 Configuration`

Environment

```
[ ] Environment is not stored
[ ] Environment in EEPROM
[*] Environment is in a FAT filesystem
[ ] Environment is in a EXT4 filesystem
[ ] Environment in flash memory
[ ] Environment in an MMC device
[ ] Environment in a NAND device
[ ] Environment in a non-volatile RAM
[ ] Environment is in OneNAND
[ ] Environment is in remote memory space
[ ] Environment in a UBI volume
(mmc) Name of the block device for the environment
(0) Device and partition for where to store the environment in FAT
(uboot.env) Name of the FAT file to use for the environment
(0x4000) Environment Size
[*] Relocate gd->env_addr
[ ] Create default environment from file
[ ] Add run-time information to the environment
[ ] Block forced environment operations
```

U-Boot environment configuration menu



Environment variables commands

Commands to manipulate environment variables:

- ▶ `printenv`
Shows all variables
- ▶ `printenv <variable-name>`
Shows the value of a variable
- ▶ `setenv <variable-name> <variable-value>`
Changes the value of a variable or defines a new one, only in RAM
- ▶ `editenv <variable-name>`
Edits the value of a variable in-place, only in RAM
- ▶ `saveenv`
Saves the current state of the environment to storage for persistence.



Environment variables commands - Example

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



Important U-Boot env variables

- ▶ `bootcmd`, specifies the commands that U-Boot will automatically execute at boot time after a configurable delay (`bootdelay`), if the process is not interrupted. See next page for an example.
- ▶ `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
- ▶ `filesize`, the size of the latest copy to memory (from `tftp`, `fatload`, `nand read...`)



Scripts in environment variables

- ▶ Environment variables can contain small scripts, to execute several commands and test the results of commands.
 - ▶ Useful to automate booting or upgrade processes
 - ▶ Several commands can be chained using the ; operator
 - ▶ Tests can be done using if command ; then ... ; else ... ; fi
 - ▶ Scripts are executed using `run <variable-name>`
 - ▶ You can reference other variables using `${variable-name}`
- ▶ Examples
 - ▶ `setenv bootcmd 'tftp 0x21000000 zImage; tftp 0x22000000 dtb; bootz 0x21000000 - 0x22000000'`
 - ▶ `setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 zImage; then run mmc-do-boot; fi; fi'`

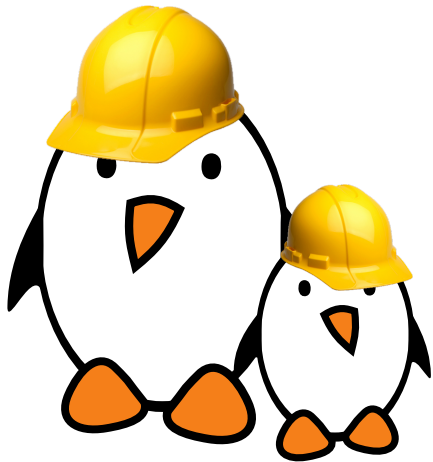


Transferring files to the target

- ▶ U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.
- ▶ Files must be exchanged between the target and the development workstation. This is possible:
 - ▶ Through the network (Ethernet if a network port is available, Ethernet over USB device...), if U-Boot has drivers for such networking. This is the fastest and most efficient solution.
 - ▶ Through a USB key, if U-Boot supports the USB controller of your platform
 - ▶ Through a SD or microSD card, if U-Boot supports the MMC controller of your platform
 - ▶ Through the serial port (`loadb`, `loadx` or `loady` command)



- ▶ Network transfer from the development workstation to U-Boot on the target takes place through TFTP
 - ▶ *Trivial File Transfer Protocol*
 - ▶ Somewhat similar to FTP, but without authentication and over UDP
- ▶ A TFTP server is needed on the development workstation
 - ▶ `sudo apt install tftpd-hpa`
 - ▶ All files in `/var/lib/tftpboot` or in `/srv/tftp` (if `/srv` exists) 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`, `serverip`, and `ethaddr` environment variables
 - ▶ Use `tftp <address> <filename>` to load file contents to the specified RAM address
 - ▶ Example: `tftp 0x21000000 zImage`



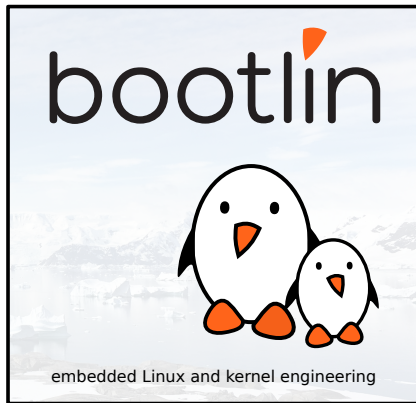
Time to start the practical lab!

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



Linux kernel introduction

© Copyright 2004-2021, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Linux features



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



Linus Torvalds in 2014

Image credits (Wikipedia):

<https://bit.ly/2UIa1TD>

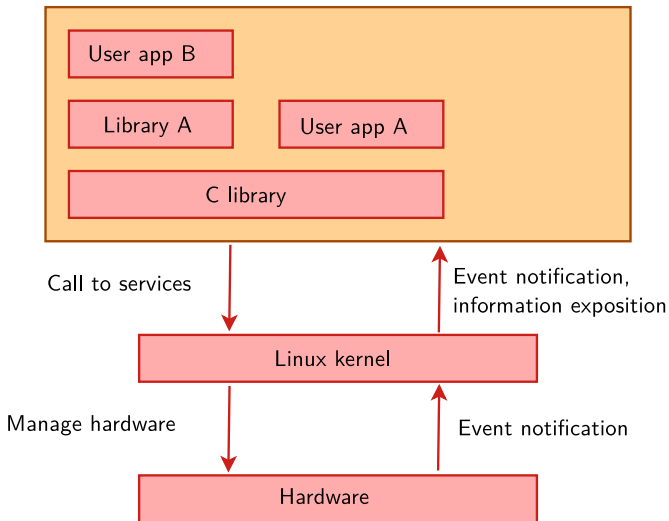


Linux kernel key features

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



Linux kernel in the system





Linux kernel main roles

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



System calls

- ▶ The main interface between the kernel and user space is the set of system calls
- ▶ About 400 system calls that provide the main kernel services
 - ▶ File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- ▶ This interface is stable over time: only new system calls can be added by the kernel developers
- ▶ This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

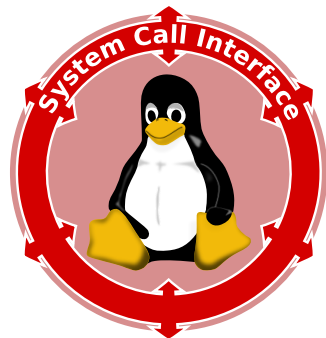


Image credits (Wikipedia):
<https://bit.ly/2U2rdGB>

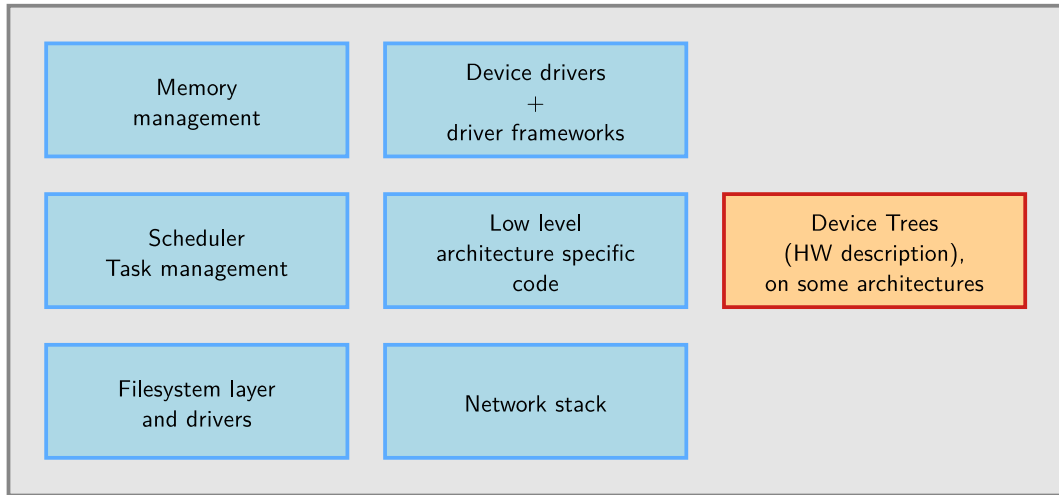


Pseudo filesystems

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



Linux Kernel





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



Supported hardware architectures

See the [arch/](#) directory in the kernel sources

- ▶ Minimum: 32 bit processors, with or without MMU, supported by `gcc`
- ▶ 32 bit architectures ([arch/](#) subdirectories)
Examples: [arm](#), [arc](#), [m68k](#), [microblaze](#) (soft core on FPGA)...
- ▶ 64 bit architectures:
Examples: [alpha](#), [arm64](#), [ia64](#)...
- ▶ 32/64 bit architectures
Examples: [mips](#), [powerpc](#), [riscv](#), [sh](#), [sparc](#), [x86](#)...
- ▶ Note that unmaintained architectures can also be removed when they have compiling issues and nobody fixes them.
- ▶ Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`



Linux versioning scheme and development process



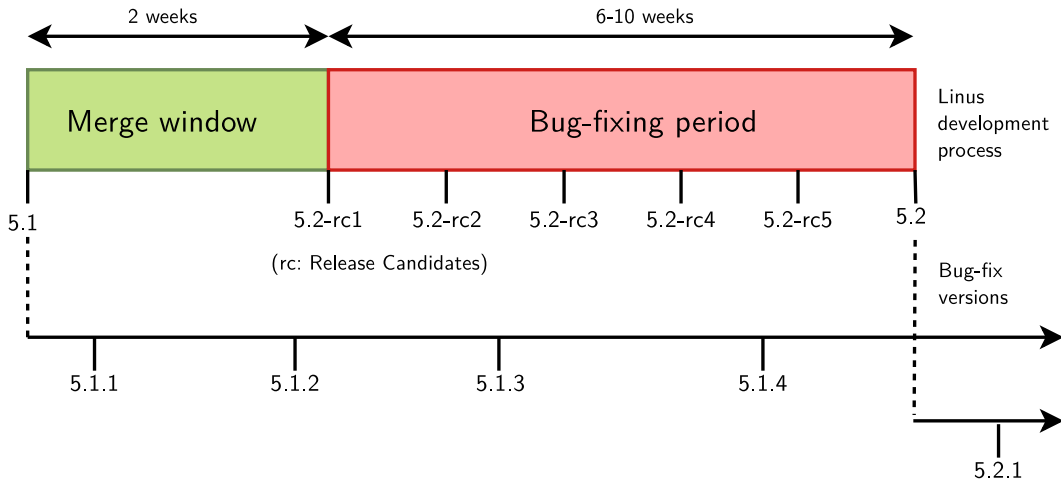
Linux versioning scheme

- ▶ Until 2003, there was a new stable release branch of Linux every 2 or 3 years (2.0, 2.2, 2.4). New development branches took 2-3 years to become stable (too slow!).
- ▶ Since 2003, there is a new stable release of Linux about every 10 weeks:
 - ▶ Versions 2.6 (Dec. 2003) to 2.6.39 (May 2011)
 - ▶ Versions 3.0 (Jul. 2011) to 3.19 (Feb. 2015)
 - ▶ Versions 4.0 (Apr. 2015) to 4.20 (Dec. 2018)
 - ▶ Version 5.0 was released in Mar. 2019.
- ▶ Features are added to the kernel in a progressive way. Since 2003, kernel developers have managed to do so without having to introduce a massively incompatible development branch.
- ▶ For each release, there are bugfix and security updates: 5.0.1, 5.0.2, etc.



Linux development model

Using merge and bug fixing windows





Need for long term support (1)

- ▶ Issue: bug and security fixes only released for most recent stable kernel versions.
- ▶ Only the last release of each year is made an LTS (*Long Term Support*) release, and is supposed to be supported for up to 6 years.

Version	Maintainer	Released	Projected EOL
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2025
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2024
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2024
4.9	Greg Kroah-Hartman & Sasha Levin	2016-12-11	Jan, 2023
4.4	Greg Kroah-Hartman & Sasha Levin	2016-01-10	Feb, 2022

Captured on <https://kernel.org> in Feb. 2021, following the [Releases](#) link.

- ▶ Example at Google: starting from *Android O* (2017), all new Android devices will have to run such an LTS kernel.



Need for long term support (2)

- ▶ You could also get long term support from a commercial embedded Linux provider.
 - ▶ Wind River Linux can be supported for up to 15 years.
 - ▶ Ubuntu Core can be supported for up to 10 years.
- ▶ *"If you are not using a supported distribution kernel, or a stable / longterm kernel, you have an insecure kernel"* - Greg KH, 2019
Some vulnerabilities are fixed in stable without ever getting a CVE.
- ▶ The *Civil Infrastructure Platform* project is an industry / Linux Foundation effort to support selected LTS versions (starting with 4.4) much longer (> 10 years).
See <https://bit.ly/2hy1QYC>.



What's new in each Linux release? (1)

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

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

```
at91: at91-ohci: support overcurrent notification
```

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

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

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

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

Very difficult to find out the key changes and to get the global picture out of individual changes.



What's new in each Linux release? (2)

Fortunately, there are some useful resources available

- ▶ <https://kernelnewbies.org/LinuxChanges>
In depth coverage of the new features in each kernel release
- ▶ <https://lwn.net/Kernel>
Coverage of the features accepted in each merge window

January 18, 2021	Resource limits in user namespaces
January 15, 2021	Fast commits for ext4
January 14, 2021	MAINTAINERS truth and fiction
January 11, 2021	Old compilers and old bugs
January 7, 2021	Restricted DMA
January 5, 2021	Portable and reproducible kernel builds with TuxMake
→ December 28, 2020	5.11 Merge window, part 2
→ December 18, 2020	5.11 Merge window, part 1



Linux kernel sources



Location of kernel sources

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



Getting Linux sources

- ▶ The kernel sources are available from <https://kernel.org/pub/linux/kernel> as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- ▶ However, more and more people use the `git` version control system. Absolutely needed for kernel development!
 - ▶ Fetch the entire kernel sources and history

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```
 - ▶ Create a branch that starts at a specific stable version

```
git checkout -b <name-of-branch> v5.6
```
 - ▶ Web interface available at <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/>
 - ▶ Read more about Git at <https://git-scm.com/>



Linux kernel size (1)

- ▶ Linux 5.10.11 sources:
 - ▶ 70,639 files (`git ls-files | wc -l`)
 - ▶ 29,746,102 lines (`git ls-files | xargs cat | wc -l`)
 - ▶ 962,810,769 bytes (`git ls-files | xargs cat | wc -c`)
- ▶ A minimum uncompressed Linux kernel just sizes 1-2 MB
- ▶ 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 5.7 (in percentage of total number of lines).

- ▶ `drivers/`: 60.1%
- ▶ `arch/`: 12.9%
- ▶ `fs/`: 4.7%
- ▶ `sound/`: 4.2%
- ▶ `net/`: 4.0%
- ▶ `include/`: 3.6%
- ▶ `tools/`: 3.2%
- ▶ `Documentation/`: 3.2%
- ▶ `kernel/`: 1.3%
- ▶ `lib/`: 0.6%
- ▶ `mm/`: 0.5%
- ▶ `scripts/`: 0.4%
- ▶ `crypto/`: 0.4%
- ▶ `security/`: 0.3%
- ▶ `block/`: 0.2%
- ▶ `samples/`: 0.1%
- ▶ `virt/`: 0.1%
- ▶ ...



Getting Linux sources

► Full tarballs

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

- Example:

<https://kernel.org/pub/linux/kernel/v4.x/linux-4.20.13.tar.xz>

- Extract command:

```
tar xf linux-4.20.13.tar.xz
```

► Incremental patches between versions

- It assumes you already have a base version and you apply the correct patches in the right order to upgrade to the next one. Quick to download and apply

- Examples:

<https://kernel.org/pub/linux/kernel/v4.x/patch-4.20.xz>

(from 4.19 to 4.20)

<https://kernel.org/pub/linux/kernel/v4.x/patch-4.20.13.xz>

(from 4.20 to 4.20.13)

- All previous kernel versions are available in

<https://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.
See <https://en.wikipedia.org/wiki/Diff>
- ▶ 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 (*hunk*) per modified part of the file, starting with a header with the starting line number and the number of lines the change hunk applies to

```
@@ -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

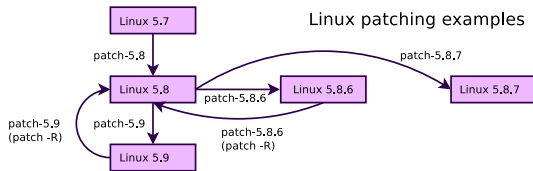
`patch` usage examples:

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

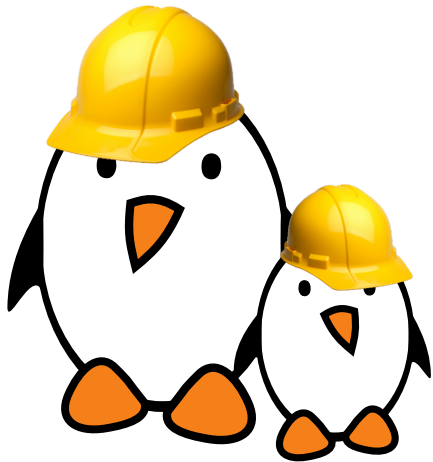


Applying a Linux patch

- ▶ Two types of Linux patches:
 - ▶ Either to be applied to the previous stable version
(from $x.<y-1>$ to $x.y$)
 - ▶ Or implementing fixes to the current stable version
(from $x.y$ to $x.y.z$)
- ▶ Can be downloaded in `gzip` or `xz` (much smaller) compressed files.
- ▶ Always produced for `patch -p1`
- ▶ Need to run the `patch` command inside the **toplevel** kernel source directory



```
cd linux-5.7
# From 5.7 to 5.8.6
xzcat ../patch-5.8.xz | patch -p1
xzcat ../patch-5.8.6.xz | patch -p1
# Back to 5.8 from 5.8.6
xzcat ../patch-5.8.6.xz | patch -R -p1
# From 5.8 to 5.8.7
xzcat ../patch-5.8.7.xz | patch -p1
# Renaming directory
cd ..; mv linux-5.7 linux-5.8.7
```



Time to start the practical lab!

- ▶ Get the Linux kernel sources
- ▶ Apply patches



Building the kernel



Kernel configuration



Kernel configuration

- ▶ 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 the target architecture and on your hardware (for device drivers, etc.)
 - ▶ On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.). Such generic options are available in all architectures.



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-4.14.x/`
 - ▶ `make <target>`



Specifying the target architecture

First, specify the architecture for the kernel to build

- ▶ Set ARCH to the name of a directory under `arch/`:
`export ARCH=arm`
- ▶ By default, the kernel build system assumes that the kernel is configured and built for the host architecture (`x86` in our case, native kernel compiling)
- ▶ The kernel build system will use this setting to:
 - ▶ Use the configuration options for the target architecture.
 - ▶ Compile the kernel with source code and headers for the target architecture.



Choose a compiler

The compiler invoked by the kernel Makefile is `$(CROSS_COMPILE)gcc`

- ▶ Specifying the compiler is already needed at configuration time, as some kernel configuration options depend on the capabilities of the compiler.
- ▶ When compiling natively
 - ▶ Leave `CROSS_COMPILE` undefined and the kernel will be natively compiled for the host architecture using `gcc`.
- ▶ When using a cross-compiler
 - ▶ 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-`
 - ▶ So, you can specify your cross-compiler as follows:
`export CROSS_COMPILE=arm-linux-gnueabi-`

`CROSS_COMPILE` is actually the prefix of the cross compiling tools (`gcc`, `as`, `ld`, `objcopy`, `strip`...).



Specifying ARCH and CROSS_COMPILE

There are actually two ways of defining 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.



Kernel configuration details

- ▶ The configuration is stored in the `.config` file at the root of kernel sources
 - ▶ Simple text file, `CONFIG_PARAM=value` (included by the kernel Makefile)
- ▶ 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



Initial configuration

Difficult to find which kernel configuration will work with your hardware and root filesystem. Start with one that works!

- ▶ Desktop or server case:
 - ▶ Advisable to start with the configuration of your running kernel, usually available in `/boot`:

```
cp /boot/config-`uname -r` .config
```
- ▶ Embedded platform case (at least on ARM 32 bit):
 - ▶ Default configuration files are available, usually for each CPU family.
 - ▶ They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files (only settings different from default ones).
 - ▶ Run `make help` to find if one is available for your platform
 - ▶ To load a default configuration file, just run

```
make cpu_defconfig
```
 - ▶ This will overwrite your existing `.config` file!

Now, you can make configuration changes (`make menuconfig...`).



Create your own default configuration

To create your own default configuration file:

- ▶ `make savedefconfig`

This creates a minimal configuration (non-default settings)

- ▶ `mv defconfig arch/<arch>/configs/myown_defconfig`

This way, you can share a reference configuration inside the kernel sources.



Kernel or module?

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



Kernel option types

There are different types of options, defined in `Kconfig` files:

- ▶ `bool` options, they are either
 - ▶ *true* (to include the feature in the kernel) or
 - ▶ *false* (to exclude the feature from the kernel)
- ▶ `tristate` options, they are either
 - ▶ *true* (to include the feature in the kernel image) or
 - ▶ *module* (to include the feature as a kernel module) or
 - ▶ *false* (to exclude the feature)
- ▶ `int` options, to specify integer values
- ▶ `hex` options, to specify hexadecimal values
Example: `CONFIG_PAGE_OFFSET=0xC0000000`
- ▶ `string` options, to specify string values
Example: `CONFIG_LOCALVERSION=-no-network`
Useful to distinguish between two kernels built from different options



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 B that depends on option A is not visible until option A is enabled
 - ▶ `select` dependencies. In this case, with option B depending on option A, when option A is enabled, option B is automatically enabled. In particular, such dependencies are used to declare what features a hardware architecture supports.

```
menuconfig ATA
tristate "Serial ATA and Parallel ATA drivers (libata)"
depends on HAS_IOMEM
depends on BLOCK
select SCSI
select GLOB
--help--
If you want to use an ATA hard disk, ATA tape drive, ATA CD-ROM or
any other ATA device under Linux, say Y and make sure that you know
the name of your ATA host adapter (the card inside your computer
that "speaks" the ATA protocol, also called ATA controller),
because you will be asked for it.
```

Kconfig file excerpt



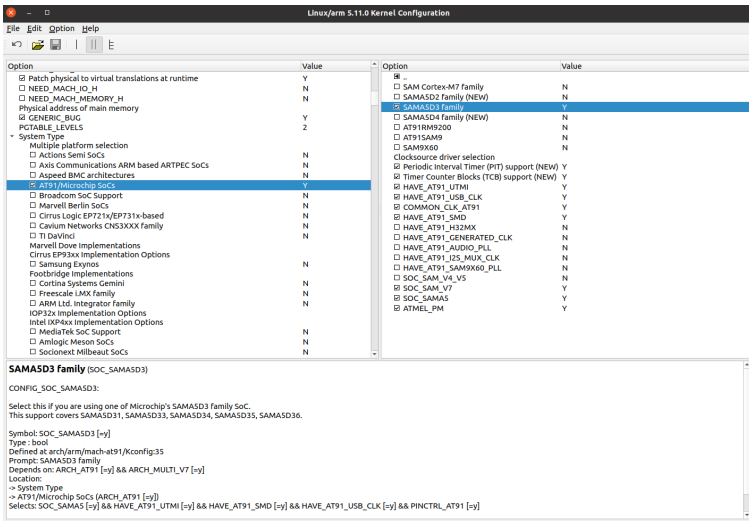
make xconfig

make xconfig

- ▶ The most common graphical interface to configure the kernel.
- ▶ File browser: easier to load configuration files
- ▶ Search interface to look for parameters
- ▶ Required Debian / Ubuntu packages: `qt5-default`



make xconfig screenshot

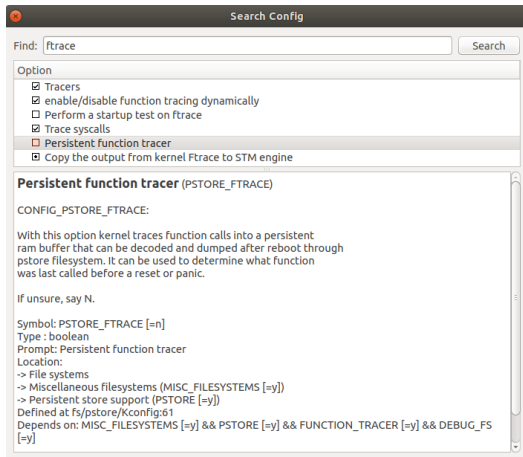




make xconfig search interface

Looks for a keyword in the parameter name (shortcut: [Ctrl] + [f]).

Allows to set values to 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`

- ☐ ISO 9660 CDROM file system support
 - ☒ Microsoft Joliet CDROM extensions
 - ☒ Transparent decompression extension
 - ☒ UDF file system support

Values in resulting .config file

Parameter values as displayed in make xconfig



Corresponding .config file excerpt

Options are grouped by sections and are prefixed with CONFIG_.

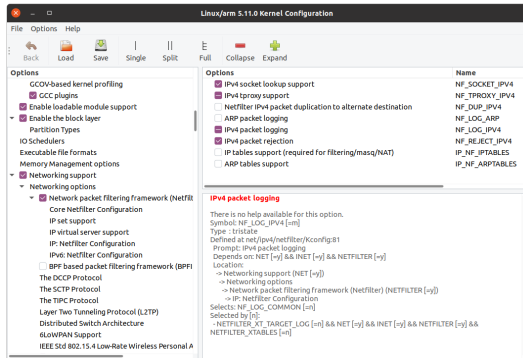
```
#  
# CD-ROM/DVD Filesystems  
#  
CONFIG_ISO9660_FS=m  
CONFIG_JOLIET=y  
CONFIG_ZISOFS=y  
CONFIG_UDF_FS=y  
CONFIG_UDF_NLS=y  
  
#  
# DOS/FAT/NT Filesystems  
#  
# CONFIG_MSDOS_FS is not set  
# CONFIG_VFAT_FS is not set  
CONFIG_NTFS_FS=m  
# CONFIG_NTFS_DEBUG is not set  
CONFIG_NTFS_RW=y
```



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.
Very efficient interface.
- ▶ Same interface found in other tools:
BusyBox, Buildroot...
- ▶ Convenient number shortcuts to jump
directly to search results.
- ▶ Required Debian packages:
`libncurses-dev`

```
Linux/arm 5.11.0 Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus --- (or empty submenu ---).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
built-in [ ] excluded <M> module <> module capable

General setup --->
(8) Maximum PAGE_SIZE order of alignment for DMA IOMMU buffers
System Type --->
Bus support --->
Kernel Features --->
Boot options --->
CPU Power Management --->
Floating point emulation --->
Power management options --->
Firmware Drivers --->
[*] ARM Accelerated Cryptographic Algorithms --->
General architecture-dependent options --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
IO Schedulers --->
Executable file formats --->
Memory Management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Security options --->
*- Cryptographic API --->
Library routines --->
Kernel hacking --->

<Select> < Exit > < Help > < Save > < Load >
```




make nconfig

make nconfig

- ▶ A newer, similar text interface
- ▶ More user friendly (for example, easier to access help information).
- ▶ However, lacking the shortcuts that menuconfig offers in search results. Therefore, much less convenient than menuconfig.
- ▶ Required Debian packages:
libncurses-dev

```
.config - Linux/arm 5.11.0 Kernel Configuration
Linux/arm 5.11.0 Kernel Configuration

General setup --->
(8) Maximum PAGE_SIZE order of alignment for DMA IOMMU buffers
System Type --->
Bus support --->
Kernel Features --->
Boot options --->
CPU Power Management --->
Floating point emulation --->
Power management options --->
Firmware Drivers --->
[*] ARM Accelerated Cryptographic Algorithms --->
General architecture-dependent options --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
IO Schedulers --->
Executable file formats --->
Memory Management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Security options --->
-* Cryptographic API --->
Library routines --->
Kernel hacking --->

F1 Help F2 SymInfo F3 Help 2 F4 ShowAll F5 Back F6 Save F7 Load F8 SymSearch F9 Exit
```



make oldconfig

make oldconfig

- ▶ Needed very often!
- ▶ Useful to upgrade a `.config` file from an earlier kernel release
- ▶ Asks for values for new parameters.
- ▶ ... unlike `make menuconfig` and `make xconfig` which silently set default values for new parameters.

If you edit a `.config` file by hand, it's useful to run `make oldconfig` afterwards, to set values to new parameters that could have appeared because of dependency changes.



Undoing configuration changes

A frequent problem:

- ▶ After changing several kernel configuration settings, your kernel no longer works.
- ▶ If you don't remember all the changes you made, you can get back to your previous configuration:

```
$ cp .config.old .config
```
- ▶ All the configuration interfaces of the kernel (`xconfig`, `menuconfig`, `oldconfig`...) keep this `.config.old` backup copy.



Compiling and installing the kernel



Kernel compilation

make

- ▶ Run it in the main kernel source directory!
- ▶ Remember to run multiple jobs in parallel if you have multiple CPU cores / threads. Our advice: `ncpus * 2` or `ncpus + 2`, to fully load the CPU and I/Os at all times.

Example: `make -j 8`

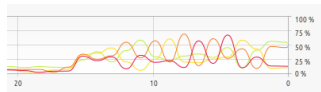
- ▶ No need to run as root!
- ▶ To **recompile** faster (7x according to some benchmarks), use the `ccache` compiler cache:
`export CROSS_COMPILE="ccache riscv64-linux-"`

Benefits on parallel make

Tests on Linux 5.11 on arm

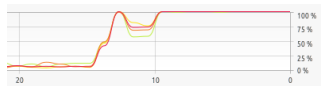
`gnome-system-monitor` showing the load of the 4 CPUs

`make allnoconfig` configuration



`make`

total time: 129 s



`make -j8`

total time: 67 s



Kernel compilation results

- ▶ `vmlinux`, the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted
- ▶ `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
 - ▶ `bzImage` for x86, `zImage` for ARM, `Image.gz` for RISC-V, `vmlinux.bin.gz` for ARC, etc.
- ▶ `arch/<arch>/boot/dts/*.dtb`, compiled Device Tree files (on some architectures)
- ▶ All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.



Kernel installation: native case

- ▶ `make install`
 - ▶ Does the installation for the host system by default, so needs to be run as root.
- ▶ Installs
 - ▶ `/boot/vmlinuz-<version>`
Compressed kernel image. Same as the one in `arch/<arch>/boot`
 - ▶ `/boot/System.map-<version>`
Stores kernel symbol addresses for debugging purposes (obsolete: such information is usually stored in the kernel itself)
 - ▶ `/boot/config-<version>`
Kernel configuration for this version
- ▶ In GNU/Linux distributions, typically re-runs the bootloader configuration utility to make the new kernel available at the next boot.



Kernel installation: embedded case

- ▶ `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle.
- ▶ Another reason is that there is no standard way to deploy and use the kernel image.
- ▶ Therefore making the kernel image available to the target is usually manual or done through scripts in build systems.
- ▶ It is however possible to customize the `make install` behavior in `arch/<arch>/boot/install.sh`



Module installation: native case

- ▶ `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`, `modules.alias.bin`
Aliases for module loading utilities. Used to find drivers for devices. Example line:
`alias usb:v066Bp20F9d*dc*dsc*dp*ic*isc*ip*in* asix`
 - ▶ `modules.dep`, `modules.dep.bin`
Module dependencies
 - ▶ `modules.symbols`, `modules.symbols.bin`
Tells which module a given symbol belongs to.



Module installation: embedded case

- ▶ In embedded development, you can't directly use `make modules_install` as it would install target modules in `/lib/modules` on the host!
- ▶ The `INSTALL_MOD_PATH` variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem:

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



Kernel cleanup targets

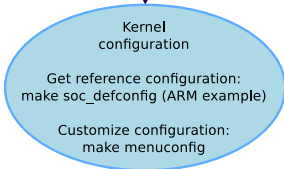
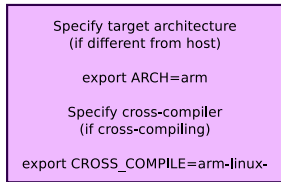
- ▶ Clean-up generated files (to force re-compilation):
`make clean`
- ▶ Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your `.config` file!
`make mrproper`
- ▶ Also remove editor backup and patch reject files (mainly to generate patches):
`make distclean`
- ▶ If you are in a git tree, remove all files not tracked (and ignored) by git:
`git clean -fdx`



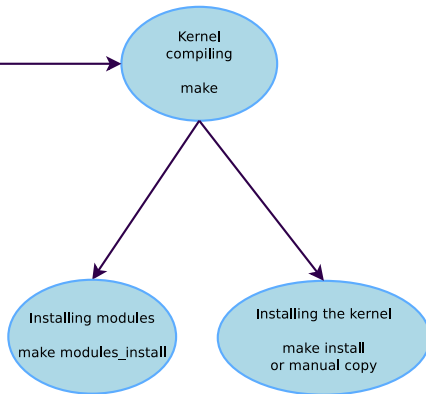


Kernel building overview

Environment setup and configuration



Kernel building and deployment





Booting the kernel



Device Tree (1)

- ▶ Many embedded architectures have a lot of non-discoverable hardware (serial, Ethernet, I2C, Nand flash, USB controllers...)
- ▶ Depending on the architecture, such hardware is either described in BIOS ACPI tables (x86), using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ▶ The Device Tree (DT) was created for PowerPC, and later was adopted by other architectures (ARM, ARC...). Now Linux has DT support in most architectures, at least for specific systems (for example for the OLPC on x86).



Device Tree (2)

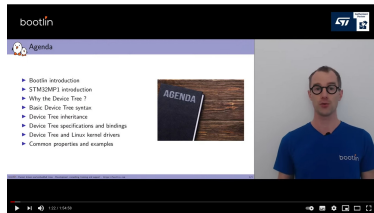
- ▶ A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, and needs to be passed to the kernel at boot time.
 - ▶ There is one different Device Tree for each board/platform supported by the kernel, available in `arch/arm/boot/dts/<board>.dtb`.
 - ▶ See [arch/arm/boot/dts/at91-sama5d3_xplained.dts](#) for example.
- ▶ The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.



Customize your board device tree!

Often needed for embedded board users:

- ▶ To describe external devices attached to non-discoverable busses (such as I2C) and configure them.
- ▶ To configure pin muxing: choosing what SoC signals are made available on the board external connectors. See <http://linux.tanzilli.com/> for a web service doing this interactively.
- ▶ To configure some system parameters: flash partitions, kernel command line (other ways exist)
- ▶ Device Tree 101 webinar, Thomas Petazzoni (2021):
Slides: <https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/>
Video: <https://youtu.be/a9CZ1Uk30YQ>





Booting with U-Boot

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



Kernel command line

- ▶ In addition to the compile time configuration, the kernel behavior 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
 - ▶ Example: `console=ttyS0 root=/dev/mmcblk0p2 rootwait`
 - ▶ Many more exist. The most important ones are documented in [admin-guide/kernel-parameters](#) in kernel documentation.
- ▶ This kernel command line can be, in order of priority (highest to lowest):
 - ▶ Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel.
 - ▶ Specified in the Device Tree (for architectures which use it)
 - ▶ Built into the kernel, using the `CONFIG_CMDLINE` option.
 - ▶ A combination of the above depending on the kernel configuration.



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



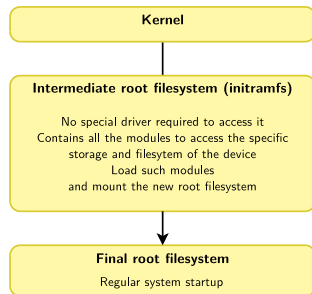
Using kernel modules



Advantages of modules

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

Using kernel modules to support many different devices and setups



The modules in the initramfs are updated every time a kernel upgrade is available.



Module dependencies

- ▶ Some kernel modules can depend on other modules, which need to be loaded first.
- ▶ Example: the `ubifs` module depends on the `ubi` and `mtd` modules.
- ▶ Dependencies are described both in
`/lib/modules/<kernel-version>/modules.dep` and in
`/lib/modules/<kernel-version>/modules.dep.bin` (binary hashed format)
These files are generated when you run `make modules_install`.



Kernel log

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

- ▶ The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- ▶ Kernel log messages are available through the `dmesg` command (**d**iagnostics **m**essage)
- ▶ Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel command line parameter, or completely disabled with the `quiet` parameter). Example:

```
console=ttyS0 root=/dev/mmcblk0p2 loglevel=5
```

- ▶ Note that you can write to the kernel log from user space too:

```
echo "<n>Debug info" > /dev/kmsg
```



Module utilities (1)

`<module_name>`: name of the module file without the trailing `.ko`

- ▶ `modinfo <module_name>` (for modules in `/lib/modules`)

`modinfo <module_path>.ko`

Gets information about a module without loading it: parameters, license, description and dependencies.

- ▶ `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 usb-storage
```

- ▶ Through `insmod`:

```
sudo insmod ./usb-storage.ko delay_use=0
```

- ▶ Through `modprobe`:

Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:

```
options usb-storage delay_use=0
```

- ▶ Through the kernel command line, when the driver is built statically into the kernel:

```
usb-storage.delay_use=0
```

- ▶ `usb-storage` is the *driver name*
- ▶ `delay_use` is the *driver parameter name*. It specifies a delay before accessing a USB storage device (useful for rotating devices).
- ▶ `0` is the *driver parameter value*



Check module parameter values

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

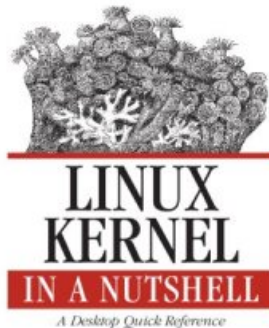
- ▶ Check `/sys/module/<name>/parameters`.
- ▶ There is one file per parameter, containing the parameter value.
- ▶ Also possible to change parameter values if these files have write permissions (depends on the module code).
- ▶ Example:

```
echo 0 > /sys/module/usb_storage/parameters/delay_use
```



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
<https://bootlin.com/community/kernel/lkn/>
- ▶ Getting old but still containing useful content.



O'REILLY*

Greg Kroah-Hartman

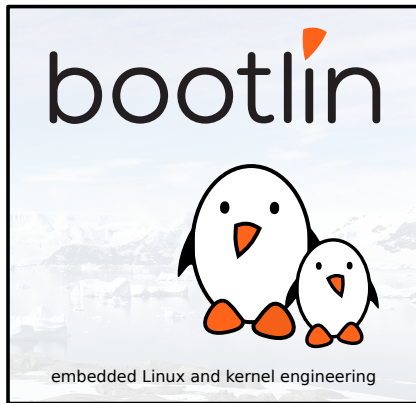


Linux Root Filesystem

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Principle and solutions



- ▶ 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

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

- ▶ It is empty

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

```
$
```

- ▶ Mount a storage device in this mount point

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



Root filesystem

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

Please append a correct `"root="` boot option

Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)



Location of the root filesystem

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



Mounting rootfs from storage devices

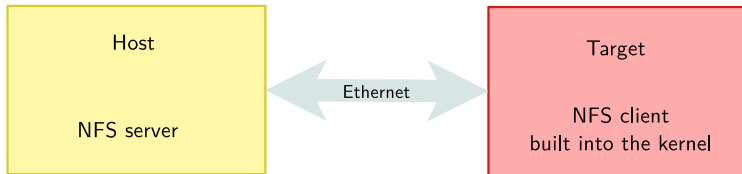
- ▶ Partitions of a hard disk or USB key
 - ▶ `root=/dev/sdXY`, where `X` is a letter indicating the device, and `Y` a number indicating the partition
 - ▶ `/dev/sdb2` is the second partition of the second disk drive (either USB key or ATA hard drive)
- ▶ Partitions of an SD card
 - ▶ `root=/dev/mmcblkXpY`, 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 flash partition in the system (there could be multiple flash chips)



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.
- ▶ Can have a big root filesystem even if you don't have support for internal or external storage yet.
- ▶ The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).





Mounting rootfs over the network (2)

On the development workstation side, a NFS server is needed

- ▶ Install an NFS server (example: Debian, Ubuntu)

```
sudo apt 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
```

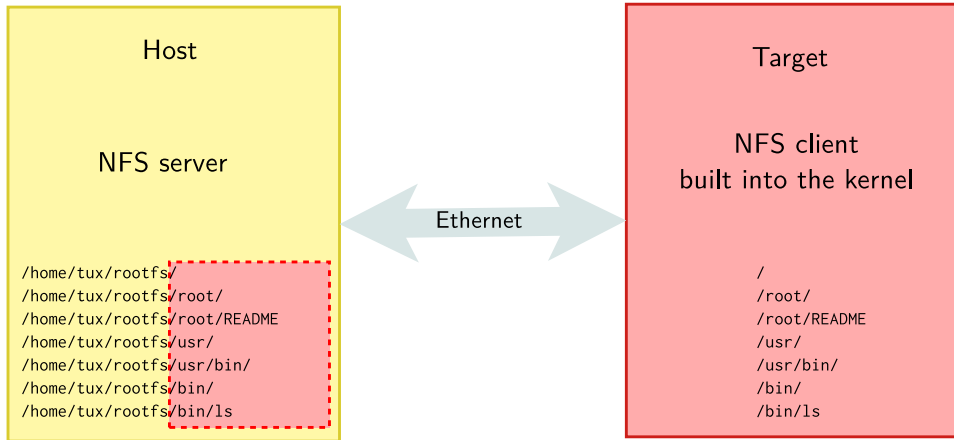



Mounting rootfs over the network (3)

- ▶ On the target system
- ▶ The kernel must be compiled with
 - ▶ `CONFIG_NFS_FS=y` (NFS **client** 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)
 - ▶ You may need to add `",nfsvers=3,tcp"` to the `nfsroot` setting, as an NFS version 2 client and UDP may be rejected by the NFS server in recent GNU/Linux distributions.



Mounting rootfs over the network (4)





Root filesystem in memory: *initramfs*

It is also possible to boot the system with a filesystem in memory: *initramfs*

- ▶ Either from a compressed CPIO archive integrated into the kernel image
- ▶ Or from such an archive loaded by the bootloader into memory
- ▶ At boot time, this archive is extracted into the Linux file cache
- ▶ 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.
- ▶ Details (in kernel sources):
`filesystems/ramfs-rootfs-initramfs`



External initramfs

- ▶ To create one, first create a compressed CPIO archive:

```
cd rootfs/  
find . | cpio -H newc -o > ../initramfs.cpio  
cd ..  
gzip initramfs.cpio
```

- ▶ If you're using U-Boot, you'll need to include your archive in a U-Boot container:

```
mkimage -n 'Ramdisk Image' -A arm -O linux -T ramdisk -C gzip \  
-d initramfs.cpio.gz uInitramfs
```

- ▶ Then, in the bootloader, load the kernel binary, DTB and uInitramfs in RAM and boot the kernel as follows:

```
bootz kernel-addr initramfs-addr dtb-addr
```



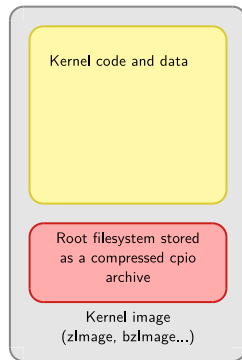
Built-in initramfs

To have the kernel Makefile include an initramfs archive in the kernel image: use the `CONFIG_INITRAMFS_SOURCE` option.

- ▶ It can be the path to a directory containing the root filesystem contents
- ▶ It can be the path to a ready made cpio archive
- ▶ It can be a text file describing the contents of the initramfs

See the kernel documentation for details:

[driver-api/early-userspace/early_userspace_support](#)





Contents



Root filesystem organization

- ▶ The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- ▶ <https://wiki.linuxfoundation.org/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 images, configurations and initramfs (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`, `ip`, `cp`, `bash`, etc.
- ▶ `/lib` contains the C library and sometimes a few other basic libraries
- ▶ All other programs and libraries are in `/usr`



Device Files



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



Types of devices

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



Devices: everything is a file

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



Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda /dev/sda1 /dev/sda2 /dev/sdc1 /dev/zero
brw-rw---- 1 root disk      8,  0 2011-05-27 08:56 /dev/sda
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
brw-rw---- 1 root disk      8, 32 2011-05-27 08:56 /dev/sdc
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

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



Pseudo Filesystems



proc virtual filesystem

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



- ▶ 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 adjust kernel parameters
 - ▶ They are called *sysctl*. See [admin-guide/sysctl/](#) in kernel sources.
 - ▶ Example (free the page cache and slab objects):
`echo 3 > /proc/sys/vm/drop_caches`



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



Minimal filesystem

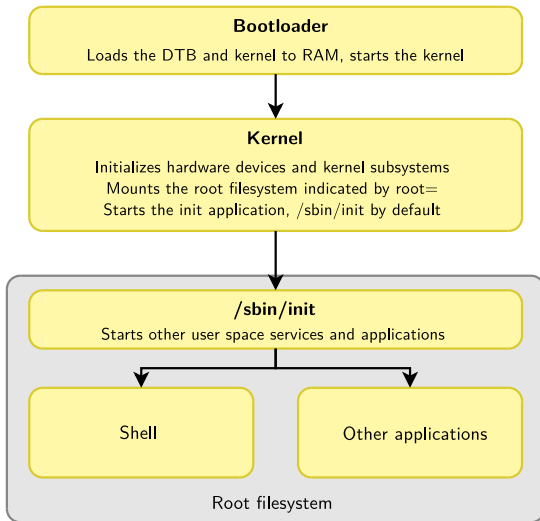


Basic applications

- ▶ In order to work, a Linux system needs at least a few applications
- ▶ An `init` application, which is the first user space application started by the kernel after mounting the root filesystem (see <https://en.wikipedia.org/wiki/Init>):
 - ▶ The kernel tries to run the command specified by the `init=` command line parameter if available.
 - ▶ Otherwise, it tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
 - ▶ In the case of an `initramfs`, it will only look for `/init`. Another path can be supplied by the `rdinit=` kernel argument.
 - ▶ If none of this works, the kernel panics and the boot process is stopped.
 - ▶ The `init` application is responsible for starting all other user space applications and services
- ▶ A shell, to implement scripts, automate tasks, and allow a user to interact with the system
- ▶ Basic UNIX executables, for use in system scripts or in interactive shells: `mv`, `cp`, `mkdir`, `cat`, `modprobe`, `mount`, `ip`, etc.
- ▶ These basic components have to be integrated into the root filesystem to make it usable

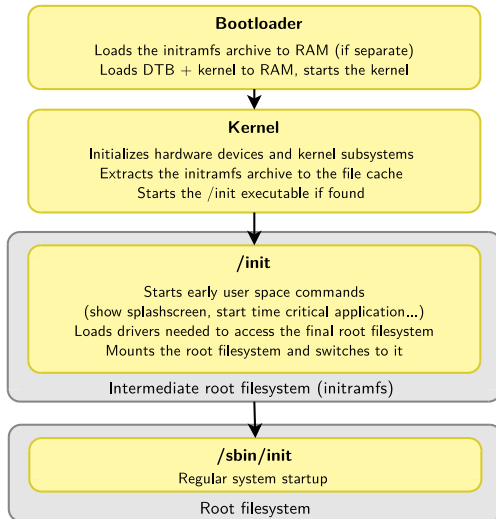


Overall booting process





Overall booting process with initramfs



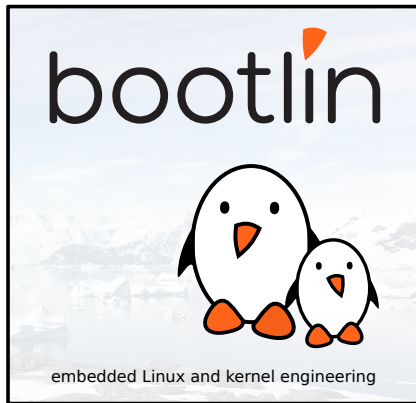


Busybox

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Why Busybox?

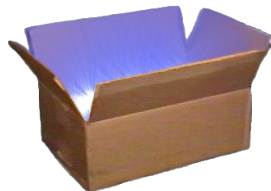
- ▶ A Linux system needs a basic set of programs to work
 - ▶ An init program
 - ▶ A shell
 - ▶ Various basic utilities for file manipulation and system configuration
- ▶ In normal GNU/Linux systems, these programs are provided by different projects
 - ▶ `coreutils`, `bash`, `grep`, `sed`, `tar`, `wget`,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



General purpose toolbox: BusyBox

<https://www.busybox.net/>

- ▶ Rewrite of many useful UNIX command line utilities
 - ▶ Created in 1995 to implement a rescue and installer system for Debian, fitting in a single floppy disk (1.44 MB)
 - ▶ Integrated into a single project, which makes it easy to work with
 - ▶ Designed with embedded systems in mind: highly configurable, no unnecessary features
 - ▶ Called the *Swiss Army Knife of Embedded Linux*
- ▶ License: GNU GPLv2
- ▶ Alternative: Toybox, BSD licensed
(<https://en.wikipedia.org/wiki/Toybox>)





BusyBox in the root filesystem

- ▶ 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).

```
rootfs
├── bin
│   ├── ash -> busybox
│   ├── busybox
│   ├── cat -> busybox
│   ├── ls -> busybox
│   ├── mount -> busybox
│   └── sh -> busybox
├── sbin
│   ├── halt -> ../bin/busybox
│   ├── ifconfig -> ../bin/busybox
│   └── init -> ../bin/busybox
└── usr
    └── sbin
        └── httpd -> ../../bin/busybox
```



BusyBox commands!

[, [[, acpid, add-shell, addgroup, adduser, adjtimex, ar, arch, arp, arping, awk, base64, basename, bbconfig, bc, beep, blkdiscard, blkid, blockdev, bootchartd, brctl, bunzip2, busybox, bzipcat, bzip2, cal, cat, chat, chatr, chcon, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, conspy, cp, cpio, crond, crontab, cryptpw, ctttyhack, cut, date, dc, dd, dealloctv, delgroup, deluser, depmod, devmem, df, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dupleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand, expr, factor, fakeidentd, fallocation, false, fatattr, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole, fgrep, find, findfs, flash_eraseall, flash_lock, flash_unlock, flashcp, flock, fold, free, freeramdisk, fsck, fsck.minix, fsfreeze, fstrim, fsync, ftpd, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, groups, gunzip, gzip, halt, hd, hdparm, head, hexdump, hexedit, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init, insmod, install, ionice, iostat, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, ipneigh, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, last, less, link, linux32, linux64, linuxrc, ln, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lsof, lspci, lsscsi, lsusb, lzcat, lzma, lzop, lzopcat, makedevs, makemime, man, matchpathcon, md5sum, mdev, msg, microcom, minips, mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.reiser, mkfs.vfat, mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount, mountpoint, mpstat, mt, mv, nameif, nanddump, nandwrite, nbd-client, nc, netcat, netstat, nice, nl, nmeter, nohup, nologin, nproc, nsenter, nslookup, ntpd, nuke, od, openvt, partprobe, passwd, paste, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, pmap, popmaildir, poweroff, printenv, printf, ps, pscan, pstree, pwd, pwdx, raidautorun, rdact, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, remove-shell, renice, reset, resize, restorecon, resume, rev, rfdkill, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-init, run-parts, runcon, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfattr, setfiles, setfont, setkeycodes, setlogcons, setpriv, setsebool, setserial, setsid, setuidgid, sh, sha1sum, sha256sum, sha3sum, sha512sum, showkey, shred, shuf, slattach, sleep, smemcap, softlimit, sort, split, ssl_client, start-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svc, svlogd, svok, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tc, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, timeout, top, touch, tr, traceroute, traceroute6, true, truncate, ts, tty, ttysize, tuncctl, tune2fs, ubiattach, ubidetach, ubimkvol, ubirename, ubirmvol, ubirsvol, ubiupdatevol, udhcpc, udhcpd, udpsvd, uevent, umount, uname, uncompress, unexpand, uniq, unit, unix2dos, unlink, unlzma, unlzop, unxz, unzip, uptime, users, usleep, uudecode, uuencode, vconfig, vi, vlock, volname, w, wall, watch, watchdog, wc, wget, which, who, whoami, whois, xargs, xxd, xz, xzcat, yes, zcat, zcip

Source: `run /bin/busybox`



Configuring BusyBox

- ▶ Get the latest stable sources from <https://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 menuconfig` (text)
Same configuration interfaces as the ones used by the Linux kernel (though older versions are used, causing `make xconfig` to be broken in recent distros).



BusyBox make menuconfig

You can choose:

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

Coreutils

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
(-)  
[ ] link (3.2 kb)  
[*] ln (4.9 kb)  
[ ] logname (1.1 kb)  
[*] ls (14 kb)  
[*] Enable filetyping options (-p and -F)  
[ ] Enable symlinks dereferencing (-L)  
[*] Enable recursion (-R)  
[*] Enable -w WIDTH and window size autodetection  
[*] Sort the file names  
[*] Show file timestamps  
[*] Show username/groupnames  
[ ] Allow use of color to identify file types  
[*] md5sum (6.5 kb)  
!(+)
```

<Select> < Exit > < Help >



Compiling BusyBox

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



Applet highlight: Busybox init

- ▶ Busybox provides an implementation of an `init` program
- ▶ Simpler than the `init` implementation found on desktop/server systems (*SysV init* or *systemd*)
- ▶ 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



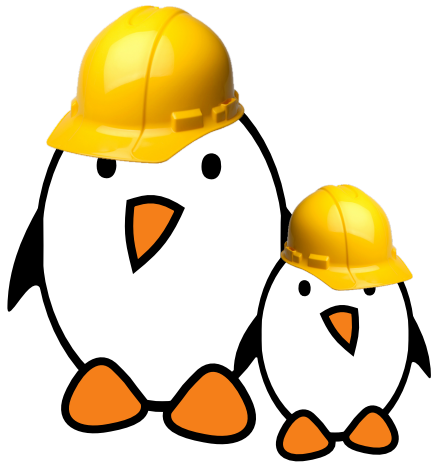
Applet highlight - BusyBox vi

- ▶ If you are using BusyBox, adding `vi` support only adds about 20K
- ▶ 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.

```
[ ] cmp (4.9 kb)
[ ] diff (13 kb)
[ ] ed (21 kb)
[ ] patch (9.4 kb)
[ ] sed (12 kb)
[*] vi (23 kb)
(4896) Maximum screen width
[*] Allow to display 8-bit chars (otherwise shows dots)
[*] Enable ":" colon commands (no "ex" mode)
[*] Enable yank/put commands and mark cmds
[*] Enable search and replace cmds
[ ] Enable regex in search and replace
[*] Catch signals
[*] Remember previous cmd and "." cmd
[*] Enable -R option and "view" mode
[*] Enable settable options, ai ic showmatch
[*] Support :set
[ ] Handle window resize
[ ] Use 'tell me cursor position' ESC sequence to measure window
[*] Support undo command "u"
[*] Enable undo operation queuing
(256) Maximum undo character queue size
[ ] Allow vi and awk to execute shell commands
```



Practical lab - A tiny embedded system



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

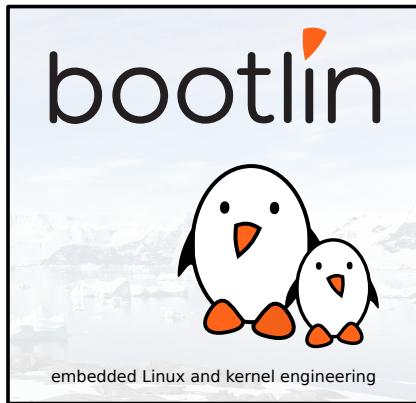


Block filesystems

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Block devices



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, in random order, without erasing.
 - ▶ Hard disks, RAM disks
 - ▶ USB keys, SSD, SD cards, eMMC: these are based on flash storage, but have an integrated controller that emulates a block device, managing the flash in a transparent way.
- ▶ **Raw flash devices** are driven by a controller on the SoC. They can be read, but writing requires prior 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
```

- ▶ `/sys/block/` also stores information about each block device, for example whether it is removable storage or not.



Partitioning

- ▶ Block devices can be partitioned to store different parts of a system
- ▶ The partition table is stored inside the device itself, and is read and analyzed automatically by the Linux kernel
 - ▶ `mmcblk0` is the entire device
 - ▶ `mmcblk0p2` is the second partition of `mmcblk0`
- ▶ Two partition table formats:
 - ▶ *MBR*, the legacy format
 - ▶ *GPT*, the new format, not used everywhere yet, supporting disks bigger than 2 TB.
- ▶ Numerous tools to create and modify the partitions on a block device: `fdisk`, `cfdisk`, `sfdisk`, `parted`, etc.



Transferring data to a block device

- ▶ It is often necessary to transfer data to or from a block device in a *raw* way
 - ▶ Especially to write a *filesystem image* to a block device
- ▶ This directly writes to the block device itself, bypassing any filesystem layer.
- ▶ The block devices in `/dev/` allow such *raw* access
- ▶ `dd` (**d**isk **d**uplicate) is the tool of choice for such transfers:
 - ▶ `dd if=/dev/mmcblk0p1 of=testfile bs=1M count=16`
Transfers 16 blocks of 1 MB from `/dev/mmcblk0p1` to `testfile`
 - ▶ `dd if=testfile of=/dev/sda2 bs=1M seek=4`
Transfers the complete contents of `testfile` to `/dev/sda2`, by blocks of 1 MB, but starting at offset 4 MB in `/dev/sda2`
 - ▶ **Typical mistake:** copying a file to a filesystem without mounting it first:
`dd if=zImage of=/dev/sde1`
Instead, you should use:
`sudo mount /dev/sde1 /boot`
`cp zImage /boot/`



Available block filesystems



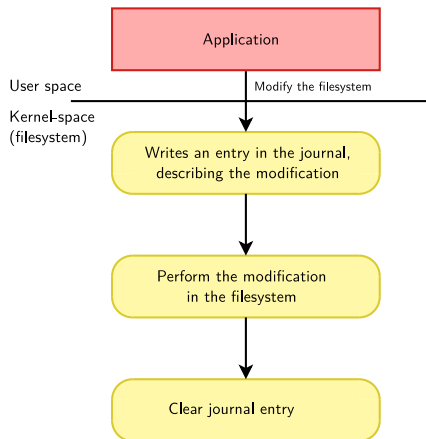
Standard Linux filesystem format: ext2, ext3, ext4

- ▶ The standard filesystem used on Linux systems is the series of `ext{2,3,4}` filesystems
 - ▶ ext2
 - ▶ ext3, brought *journaling* compared to ext2, now obsoleted by ext4.
 - ▶ ext4, mainly brought performance improvements and support for very big partitions.
- ▶ It supports all features Linux needs in a root filesystem: permissions, ownership, device files, symbolic links, etc.



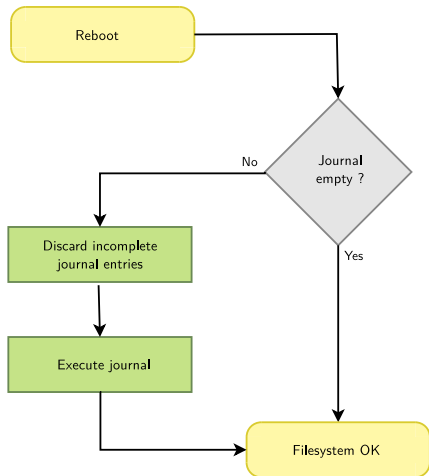
Journalized filesystems

- ▶ Unlike simpler filesystems (`ext2`, `vfat...`), designed to stay in a coherent state even after system crashes or a sudden poweroff.
- ▶ Writes are first described in the journal before being committed to files (can be all writes, or only metadata writes depending on the configuration)
- ▶ Allows to skip a full disk check at boot time after an unclean shutdown.





Filesystem recovery after crashes



- ▶ Thanks to the journal, the recovery at boot time is quick, since the operations in progress at the moment of the unclean shutdown are clearly identified. There's no need for a full filesystem check.
- ▶ Does not mean that the latest writes made it to the storage: this depends on syncing the changes to the filesystem.

See https://en.wikipedia.org/wiki/Journaling_file_system for further details.



Other journaled Linux/UNIX filesystems

- ▶ `btrfs`, intended to become the next standard filesystem for Linux. Integrates numerous features: data checksumming, integrated volume management, snapshots, etc.
- ▶ `XFS`, high-performance filesystem inherited from SGI IRIX, still actively developed.
- ▶ `JFS`, inherited from IBM AIX. No longer actively developed, provided mainly for compatibility.
- ▶ `reiserFS`, used to be a popular filesystem, but its latest version `Reiser4` was never merged upstream.
- ▶ `ZFS`, provides standard and advanced filesystem and volume management (CoW, snapshot, etc.). Due to license it cannot be mainlined into Linux but present into few distributions (see `OpenZFS`).

All those filesystems provide the necessary functionalities for Linux systems: symbolic links, permissions, ownership, device files, etc.



F2FS: filesystem for flash-based storage

<https://en.wikipedia.org/wiki/F2FS>

- ▶ Filesystem that takes into account the characteristics of flash-based storage: eMMC, SD cards, SSD, etc.
- ▶ Developed and contributed by Samsung
- ▶ Now supporting transparent compression (LZO, LZ4, zstd) and encryption.
- ▶ For optimal results, need a number of details about the storage internal behavior which may not easy to get
- ▶ Benchmarks: best performer on flash devices most of the time:
See <https://lwn.net/Articles/520003/>
- ▶ Technical details: <https://lwn.net/Articles/518988/>
- ▶ Not as widely used as `ext4` and `btrfs`, even on flash-based storage.



SquashFS: read-only filesystem

- ▶ Read-only, compressed filesystem for block devices. Fine for parts of a filesystem which can be read-only (kernel, binaries...)
- ▶ Great compression rate, which generally brings improved read performance
- ▶ Used in most live CDs and live USB distributions
- ▶ Supports several compression algorithms (LZO, XZ, etc.)
- ▶ Benchmarks: roughly 3 times smaller than ext3, and 2-4 times faster (https://elinux.org/Squash_Fs_Comparisons)
- ▶ New alternative to SquashFS: EROFS
<https://en.wikipedia.org/wiki/EROFS>



Our advice for choosing the best filesystem

- ▶ Some filesystems will work better than others depending on how you use them.
- ▶ For example, `reiserFS` had the reputation to be best at handling many small files.
- ▶ Fortunately, filesystems are easy to benchmark, being transparent to applications:
 - ▶ Format your storage with each filesystem
 - ▶ Copy your data to it
 - ▶ Run your system on it and benchmark its performance.
 - ▶ Keep the one working best in your case.
- ▶ For read/write partitions, the best choices are probably `btrfs` and `f2fs`.



Compatibility filesystems

Linux also supports several other filesystem formats, mainly to be interoperable with other operating systems:

- ▶ `vfat` for compatibility with the FAT filesystem used in the Windows world and on numerous removable devices
 - ▶ Also convenient to store bootloader binaries (FAT easy to understand for ROM code)
 - ▶ This filesystem does *not* support features like permissions, ownership, symbolic links, etc. Cannot be used for a Linux root filesystem.
 - ▶ Linux now supports the exFAT filesystem too (`exfat`).
- ▶ `ntfs` for compatibility with the NTFS filesystem used on Windows
- ▶ `hfs` for compatibility with the HFS filesystem used on Mac OS
- ▶ `iso9660`, the filesystem format used on CD-ROMs, obviously a read-only filesystem



tmpfs: filesystem in RAM

- ▶ Not a block filesystem of course!
- ▶ Perfect to store temporary data in RAM: system log files, connection data, temporary files...
- ▶ More space-efficient than ramdisks: files are directly in the file cache, grows and shrinks to accommodate stored files
- ▶ How to use: choose a name to distinguish the various tmpfs instances you have (unlike in most other filesystems, each tmpfs instance is different). Examples:

```
mount -t tmpfs run /var/run
```

```
mount -t tmpfs shm /dev/shm
```
- ▶ See [filesystems/tmpfs](#) in kernel sources.



Using block filesystems



Creating ext2/ext4 filesystems

- ▶ To create an empty ext2/ext4 filesystem on a block device or inside an already-existing image file
 - ▶ `mkfs.ext2 /dev/hda3`
 - ▶ `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
 - ▶ This tool only supports ext2. Alternative for other filesystems: create a disk image, format it, mount it (see next slides), copy contents and umount.
 - ▶ `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.
- ▶ Note: `-o loop` no longer necessary with recent versions of `mount` from *GNU Coreutils*. Not true with BusyBox `mount`.
- ▶ Do not forget to run `umount` before using the filesystem image!



Creating squashfs filesystems

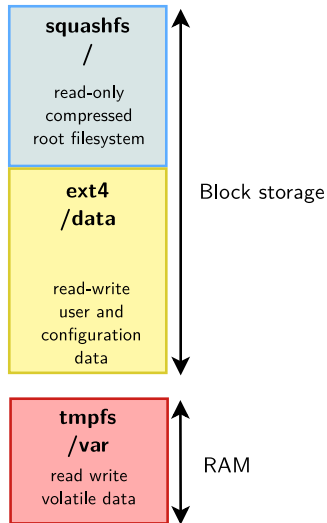
- ▶ Need to install the `squashfs-tools` package
- ▶ Can only create an image: creating an empty *squashfs* filesystem would be useless, since it's read-only.
- ▶ To create a *squashfs* image:
 - ▶ `mksquashfs rootfs/ rootfs.sqfs -noappend`
 - ▶ `-noappend`: re-create the image from scratch rather than appending to it
- ▶ Examples mounting a squashfs filesystem:
 - ▶ Same way as for other block filesystems
 - ▶ `mount -o loop rootfs.sqfs /mnt` (filesystem image on the host)
 - ▶ `mount /dev/<device> /mnt` (on the target)



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 **ext4**)
Used to store user or configuration data.
Guarantees filesystem integrity after power off or crashes.
- ▶ Ram storage for temporary files (**tmpfs**)

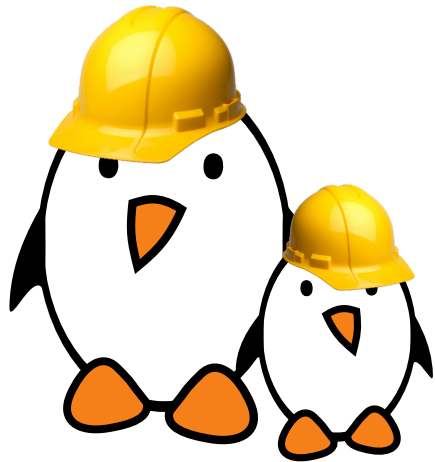




Issues with flash-based block storage

- ▶ 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.
- ▶ Not knowing about the wear leveling algorithm, it is highly recommended to limit the number of writes to these devices.
- ▶ Using industrial grade storage devices (MMC/SD, USB) is also recommended.

See the *Optimizing Linux with cheap flash drives* article from Arnd Bergmann and try his *flashbench* tool (<http://git.linaro.org/people/arnd/flashbench.git/about/>) for finding out the erase block and page size for your storage, and optimizing your partitions and filesystems for best performance.

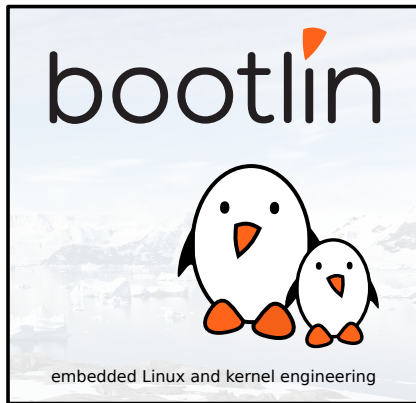


- ▶ Creating partitions on your block storage
- ▶ Booting your system with a mix of filesystems: SquashFS for the root filesystem (including applications), ext4 for configuration and user data, and tmpfs for temporary system files.



Flash storage and filesystems

© Copyright 2004-2021, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Block devices vs flash devices: reminder

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

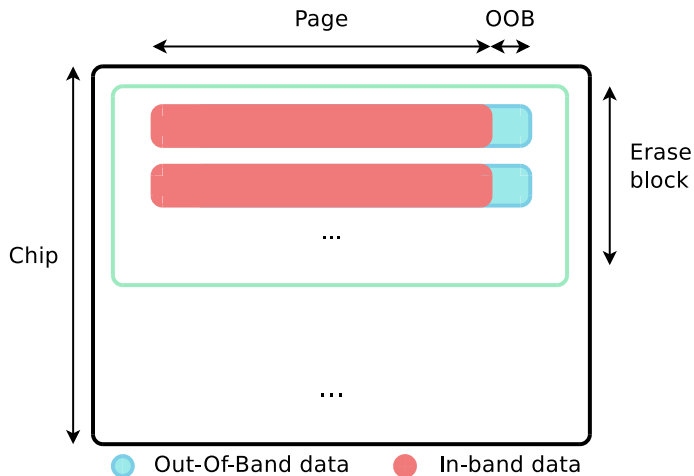


NAND flash chips: how they work?

- ▶ Encode bits with voltage levels
 - ▶ SLC: *Single Level Cell* - 1 bit per memory cell
 - ▶ MLC: *Multi Level Cell* - Multiple bits per cell
- ▶ Start with all bits set to 1
- ▶ Writing implies changing some bits from 1 to 0 (assuming 1 bit per cell)
- ▶ Restoring bits to 1 is done via the ERASE operation
- ▶ Writing and erasing are not done on a per bit or per byte basis
- ▶ Organization
 - ▶ Page: minimum unit for PROGRAM operation. Example size: 4 K.
 - ▶ Block: minimum unit for ERASE operation. Example size: 128 K.



NAND flash storage: organization



Example sizes on
Microchip SAMA5D3 Xplained:

- Page size: 2048 b
- OOB size: 64 b
- Erase block size: 131072 b



NAND flash storage: constraints

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



NAND flash: ECC

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



The MTD subsystem (1)

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



The MTD subsystem (2)

Linux filesystem interface

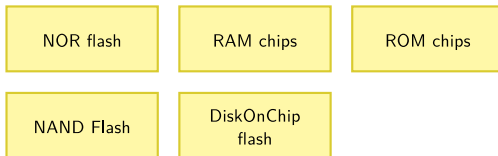
MTD "User"
modules



Flash Translation Layers
for block device emulation
Caution: patented
algorithms

FTL NFTL INFTL

MTD Chip
drivers



Block
device Virtual
memory

Virtual devices appearing
as MTD devices

Hardware
devices





MTD partitioning

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



Linux: definition of MTD partitions

The Device Tree is the standard place to define *default* MTD partitions and MTD device properties (such as ECC) for platforms with Device Tree support.

```
nand@0,0 {  
    linux,mtd-name= "micron,mt29c4g96maz";  
    [...]  
    ti,nand-ecc-opt = "bch8"  
    [...]  
    partition@0 {  
        label = "SPL";  
        reg = <0 0x80000>; /* 512KiB */  
    };  
    partition@80000 {  
        label = "U-Boot";  
        reg = <0x80000 0x1C0000>; /* 1792KiB */  
    };  
    partition@1c0000 {  
        label = "Environment";  
        reg = <0x240000 0x40000>; /* 256KiB */  
    };  
    [...]
```

Example from [arch/arm/boot/dts/omap3-overo-base.dtsi](#)



U-Boot: defining MTD partitions (1)

- ▶ U-Boot allows to define MTD partitions on flash devices, using the same syntax as Linux for declaring them.
- ▶ Sharing definitions allows to eliminate the risk of mismatches between Linux and U-Boot.
- ▶ Named partitions are also easier to use, and much less error prone than using offsets.
- ▶ Use flash specific commands (detailed soon), and pass partition names instead of numerical offsets
- ▶ Example: `nand erase.part <partname>`



U-Boot: defining MTD partitions (2)

► Example:

```
# Association between U-Boot name and Linux name
setenv mtdids nand0=omap2-nand.0
# Partition definitions
setenv mtdparts mtdparts=omap2-nand.0:512k(XLoader)ro,1536k(UBoot)ro,512k(Env),4m(Kernel),-(Root)
```

► This defines 5 partitions in the `omap2-nand.0` device:

- 1st stage bootloader (512 KiB, read-only)
- U-Boot (1536 KiB, read-only)
- U-Boot environment (512 KiB)
- Kernel (4 MiB)
- Root filesystem (Remaining space)



U-Boot: defining MTD partitions (3)

- ▶ Partition sizes must be multiple of the erase block size. You can use sizes in hexadecimal too.
- ▶ `ro` lists the partition as read only
- ▶ `-` is used to use all the remaining space.
- ▶ Remember the below sizes:
 - ▶ `0x100` = 256
 - ▶ `0x1000` = 4 KiB
 - ▶ `0x10000` = 64 KiB
 - ▶ `0x100000` = 1 MiB
 - ▶ `0x1000000` = 16 MiB

Tip: use the `python` shell for hexadecimal/decimal/binary conversion:

```
>>> hex(4096)
'0x1000'
>>> 0x10000
65536
>>> bin(0xbeef)
'0b1011111011101111'
```



U-Boot: defining MTD partitions (4)

`mtdids` associates a U-Boot flash device name to a Linux flash device name:
`setenv mtdids <devid>=<mtdid>[,<devid>=<mtdid>]`

That's required because the Linux name is used in partition definitions.

- ▶ `devid`: **U-Boot** device identifier (from `nand info` or `flinfo`)
- ▶ `mtdid`: **Linux** mtd identifier. Displayed when booting the Linux kernel:

```
NAND device: Manufacturer ID: 0x2c, Chip ID: 0xbc (Micron NAND 512MiB 1,8V 16-bit)
Creating 5 MTD partitions on "omap2-nand.0":
0x0000000000000-0x0000000080000 : "X-Loader"
0x0000000080000-0x0000000200000 : "U-Boot"
0x0000000200000-0x0000000280000 : "Environment"
0x0000000280000-0x0000000580000 : "Kernel"
0x0000000580000-0x0000002000000 : "File System"
```




U-Boot: defining MTD partitions (5)

`mtdparts` defines partitions for the different devices

- ▶ `setenv mtdparts mtdparts=<mtdid>:<partition>[,partition]`
- ▶ **partition format:** `<size>[@offset](<name>)[ro]`

You can use the `mtdparts` command to check that your partitions definitions are understood correctly by U-Boot.



U-Boot: sharing partition definitions with Linux

Here is a recommended way to pass partition definitions from U-Boot to Linux:

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



U-Boot: manipulating NAND devices (1)

```
=> help nand
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
    read/write 'size' bytes starting at offset 'off'
    to/from memory address 'addr', skipping bad blocks.
nand read.raw - addr off|partition [count]
nand write.raw[.noverify] - addr off|partition [count]
    Use read.raw/write.raw to avoid ECC and access the flash as-is.
nand write.trimffs - addr off|partition size
    write 'size' bytes starting at offset 'off' from memory address
    'addr', skipping bad blocks and dropping any pages at the end
    of eraseblocks that contain only 0xFF
nand erase[.spread] [clean] off size - erase 'size' bytes from offset 'off'
    With '.spread', erase enough for given file size, otherwise,
    'size' includes skipped bad blocks.
nand erase.part [clean] partition - erase entire mtd partition'
nand erase.chip [clean] - erase entire chip'
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub [-y] off size | scrub.part partition | scrub.chip
    really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
```



U-Boot: manipulating NAND devices (2)

► `nand info`

Show available NAND devices and characteristics

```
Device 0: nand0, sector size 128 KiB
Page size      2048 b
OOB size       64 b
Erase size     131072 b
subpagesize    2048 b
options        0x00004200
bbt options    0x00008000
```

► `nand device [dev]`

Select or display the active NAND device

```
=> nand device 0
```



U-Boot: manipulating NAND devices (3)

- ▶ `nand read[.option] <addr> <offset|partname> <size>`

Read data from NAND

```
=> nand read 0x21000000 0x100000 100
```

```
NAND read: device 0 offset 0x100000, size 0x100  
256 bytes read: OK
```

- ▶ `nand erase <offset> <size>`

Erase a NAND region

```
=> nand erase 0x4000000 100000
```

```
NAND erase: device 0 offset 0x4000000, size 0x100000  
Erasing at 0x40e0000 -- 100% complete.
```



U-Boot: manipulating NAND devices (4)

- ▶ `nand erase.part <partname>`

Erase a NAND partition

```
=> nand erase.part UBI
```

```
NAND erase.part: device 0 offset 0x200000, size 0xfe00000  
Erasing at 0xffe0000 -- 100% complete.  
OK
```

- ▶ `nand write[.option] <addr> <offset|partname> <size>`

Write data on NAND. Use `nand write.trimffs` to avoid writing empty pages (those filled with `0xff`)

```
=> nand write 0x21000000 0x4000000 100000
```

```
NAND write: device 0 offset 0x4000000, size 0x100000  
1048576 bytes written: OK
```



U-Boot: manipulating NAND devices (5)

- ▶ `nand bad`
Shows bad blocks

```
=> nand bad
```

```
Device 0 bad blocks:
```

- ▶ More commands for debugging purposes



Linux: MTD devices interface with user space

- ▶ MTD devices are visible in `/proc/mtd`
- ▶ User space only sees MTD partitions, not the flash device under those partitions (unless the kernel is compiled with `CONFIG_MTD_PARTITIONED_MASTER`)
- ▶ The **mtdchar** driver creates a character device for each MTD device/partition of the system
 - ▶ Named `/dev/mtdX` and `/dev/mtdXro`
 - ▶ Provide `ioctl()` to erase and manage the flash
 - ▶ Used by the *mtd-utils* utilities



Linux: user space flash management tools

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



Flash wear leveling (1)

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



Flash wear leveling (2)

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

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



Flash file-systems

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



Legacy flash filesystems: JFFS2

Journaling Flash File System version 2

- ▶ Supports on the fly compression
- ▶ Wear leveling, power failure resistant
- ▶ Available in the official Linux kernel
- ▶ Boot time depends on the filesystem size: doesn't scale well for large partitions because needs to scan the whole storage at boot time. Need to enable `CONFIG_JFFS2_SUMMARY` to address this issue.
- ▶ <http://www.linux-mtd.infradead.org/doc/jffs2.html>

Standard file
API



JFFS2
filesystem



MTD
driver





Legacy flash filesystems: YAFFS2

Yet Another Flash File System version 2

- ▶ Mainly supports NAND flash
- ▶ No compression
- ▶ Wear leveling, power failure resistant
- ▶ Fast boot time
- ▶ Not part of the official Linux kernel: code only available separately (Dual GPLv2 / Commercial license for non GPL operating systems)
- ▶ <https://yaffs.net/>

Standard file
API



YAFFS2
filesystem



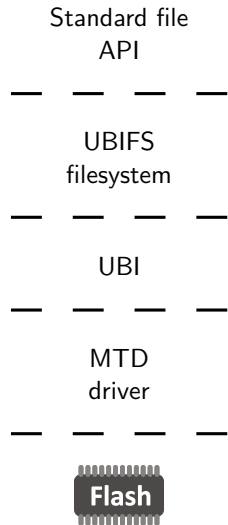
MTD
driver





Unsorted Block Images

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



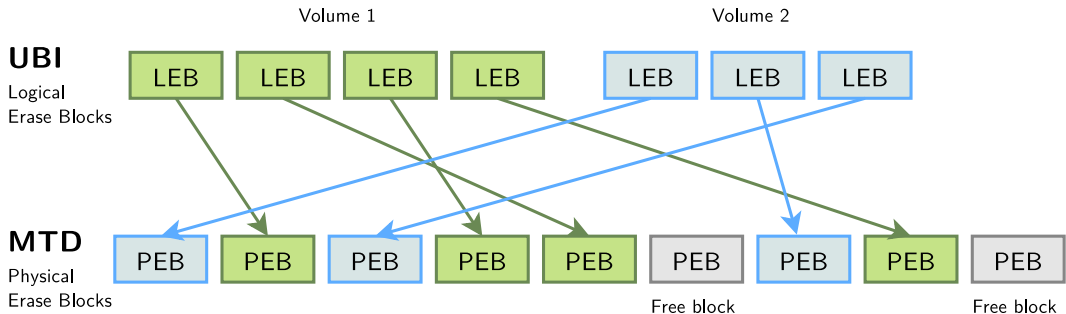


Unsorted Block Images

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



UBI (2)

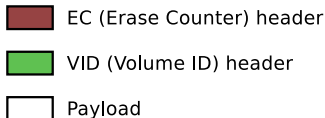
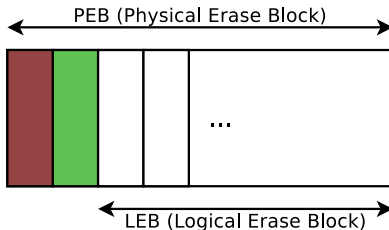


When there is too much activity on an LEB, UBI can decide to move it to another PEB with a lower erase count. Even read-only volumes participate to wear leveling!



UBI: internals

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



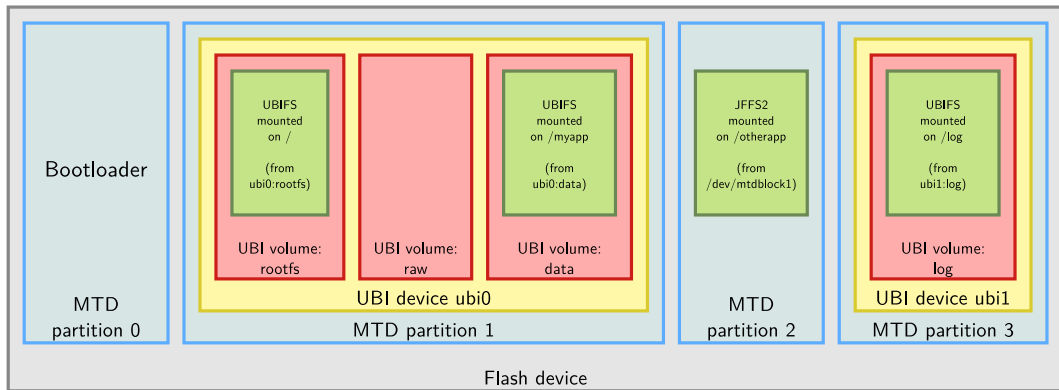


UBI: good practice

- ▶ UBI is responsible for distributing erases all over the flash device: the more space you assign to a partition attached to the UBI layer the more efficient the wear leveling will be
- ▶ If you need partitioning, use UBI volumes not MTD partitions
- ▶ Some partitions will still have to be MTD partitions: e.g. the bootloaders.
- ▶ U-Boot now even supports storing its environment in a UBI volume!
- ▶ If you do need extra MTD partitions, try to group them at the end or the beginning of the flash device (often more reliable areas).

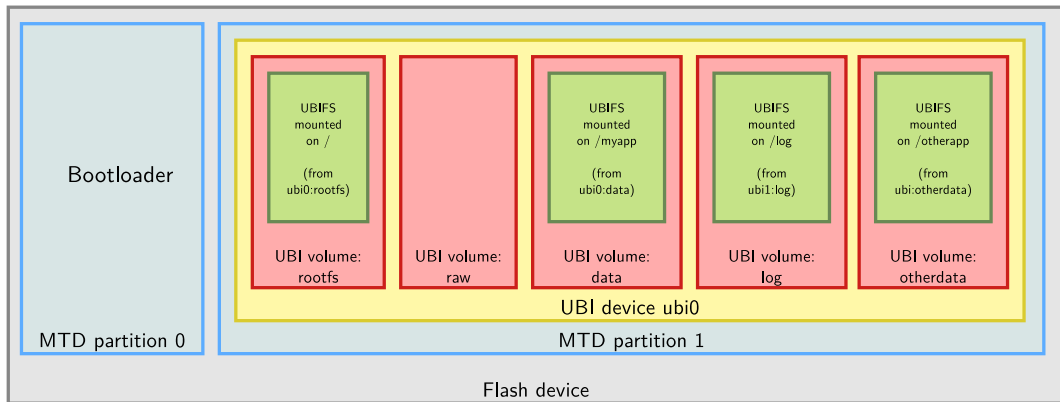


UBI layout: bad example





UBI layout: good example





Unsorted Block Images File System

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



Linux: UBI host tools

- ▶ `ubinize` is the only host tool for the UBI layer
- ▶ It creates a UBI image to be flashed on an MTD partition, from a specification of the contents of its volumes:

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

```
[rootfs-volume]
mode=ubi
image=rootfs.ubifs
vol_id=2
vol_size=2MiB
vol_type=dynamic
vol_name=rootfs
```

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



ubinize configuration file

- ▶ Each section describes a UBI volume
- ▶ `static` volumes are meant to store **read-only** blobs of data, and get the minimum corresponding size. CRC checks are done on them.
- ▶ A read-only UBIFS filesystem can go in a `static` volume, but in this case `dynamic` volumes are best for performance (CRC checking also done at UBIFS level).
- ▶ `autoresize`: allows to fill all remaining UBI space

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

```
[rootfs-volume]
mode=ubi
image=rootfs.ubifs
vol_id=2
vol_size=2MiB
vol_type=dynamic
vol_name=rootfs
```

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




ubinize command line

- ▶ `ubinize` the following arguments:
 - ▶ `-o <output-file-path>`
Path to the output image file
 - ▶ `-p <peb-size>`
The PEB size (MTD erase block size)
 - ▶ `-m <min-io-size>`
The minimum write unit size (MTD write size)
 - ▶ `-s <subpage-size>`
Subpage size, only needed if both your flash and your flash controller are supporting subpage writes
 - ▶ The last argument is the path to the `ubinize` configuration file
- ▶ Example: `ubinize -o ubi.img -p 256KiB -m 4096 -s 2048 ubinize.cfg`



U-Boot: UBI tools (1)

Grouped under the `ubi` command

- ▶ `ubi part <part-name>`
Attach an MTD partition to the UBI layer. Example:
`ubi part UBI`
- ▶ `ubi info [layout]`
Display UBI device information
(or volume information if the `layout` string is passed)
- ▶ `ubi check <vol-name>`
Check if a volume exists
- ▶ `ubi readvol <dest-addr> <vol-name> [<size>]`
Read volume contents. Example:
`ubi readvol 0x21000000 kernel`



U-Boot: UBI tools (2)

U-Boot also has commands to modify the UBI volumes:

- ▶ `ubi createvol <vol-name> [<size>] [<type>]`

Create a new volume. Example:

```
ubi createvol initramfs 0x100000 static
```

- ▶ `ubi removevol <vol-name>`

Remove an existing volume

- ▶ `ubi writevol <src-addr> <vol-name> <size>`

Write to volume. Example:

```
tftp 0x21000000 data.ubifs
```

```
ubi writevol 0x21000000 data ${filesize}
```



U-Boot: UBI tools (3)

U-Boot also has commands to access UBIFS partitions:

- ▶ `ubismount <volume-name>`
Mount the specified volume. Example:
`ubismount root`
- ▶ `ubifsload <addr> <filename> [bytes]`
Load file contents at the specified address. Example:
`ubifsload 0x21000000 boot/zImage`
- ▶ Other commands: `ubifsls` and `ubifsumount`

This shows that to optimize space, the kernel and DTB can also be stored in a UBIFS partition. Otherwise, the DTB volume consumes an entire LEB.



Linux: UBI target tools (1)

- ▶ Tools used on the target to dynamically create and modify UBI elements
- ▶ UBI device management:
 - ▶ `ubiformat /dev/mtdx`
Format an MTD partition and preserve Erase Counter information if any.
Example: `ubiformat /dev/mtd1`
 - ▶ `ubiattach -m <MTD-device-id> /dev/ubi_ctrl`
Attach an MTD partition/device to the UBI layer, and create a UBI device
Example: `ubiattach -m 1 /dev/ubi_ctrl`
 - ▶ `ubidetach -m <MTD-device-id> /dev/ubi_ctrl`
Detach an MTD partition/device from the UBI layer, and remove the associated UBI device
Example: `ubidetach -m 1 /dev/ubi_ctrl`
 - ▶ `ubinfore -a`
Print information about all UBI devices and volumes



Linux: UBI target tools (2)

UBI volume management:

- ▶ `ubimkvol /dev/ubi<UBI-device-id> -N <name> -s <size>`

Create a new volume. Use `-m` in place of `-s <size>` if you want to assign all the remaining space to this volume.

Example: `ubimkvol /dev/ubi0 -N varlog -s 16MiB`

- ▶ `ubirmvol /dev/ubi<UBI-device-id> -N <name>`

Delete a UBI volume

Example: `ubirmvol /dev/ubi0 -N varlog`

- ▶ `ubiupdatevol /dev/ubi<UBI-device-id>_<UBI-vol-id> [-s <size>] <img-file>`

Update volume contents

Example: `ubiupdatevol /dev/ubi0_1 rootfs.img`



UBI volume management (continued):

- ▶ `ubirsvol /dev/ubi<UBI-device-id> -N <name> -s <size>`

Resize a UBI volume

Example: `ubirsvol /dev/ubi0 -N varlog -s 32MiB`

- ▶ `ubirename /dev/ubi<UBI-device-id> <old-name> <new-name>`

Rename a UBI volume

Example: `ubirename /dev/ubi0 varlog var`



Beware that some implementations of UBI commands in BusyBox are still incomplete. In particular:

- ▶ `ubiformat` and `ubininfo` are not implemented yet
- ▶ `ubirsvol` doesn't support `-N <name>`. You have to specify the volume to resize by its id (`-n num`):
`ubirsvol /dev/ubi0 -n 4 -s 64 MiB`
- ▶ `ubidetach` doesn't support `-m <MTD-device-id>`. You have to specify the UBI device id by `-d <UBI-device-id>`



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

- ▶ `mkfs.ubifs -m 4096 -e 258048 -c 1000 -r rootfs/ ubifs.img`
 - ▶ `-m 4096`, minimal I/O size (see `/sys/class/mtd/mtdx/writesize`).
 - ▶ `-e 258048`, **logical** erase block size (smaller than PEB size, can be found in the kernel log after running `ubiattach`)
 - ▶ `-c 1000`, maximum size of the UBI volume the image will be flashed into, in number of logical erase blocks. Do not make this number unnecessary big, otherwise the UBIFS data structures will be bigger than needed and performance will be degraded. Details: http://linux-mtd.infradead.org/faq/ubifs.html#L_max_leb_cnt
- ▶ Once created
 - ▶ Can be written to a UBI volume from the target either using `ubiupdatevol` in Linux, or `ubi writevol` in U-Boot.
 - ▶ Or, can be included in a UBI image (using `ubinize` on the host)

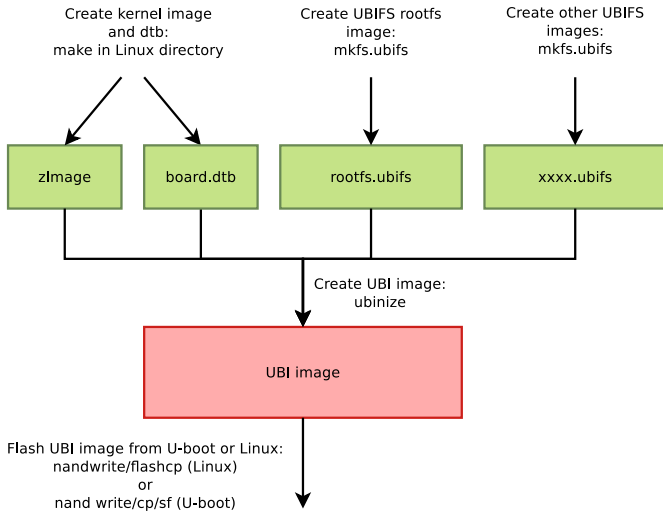


Linux: UBIFS target tools

- ▶ No specific tools are required to create a UBIFS filesystem. An empty filesystem is created the first time it is mounted. The same applies to JFFS2.
- ▶ Mounting a UBIFS filesystem is done with `mount`:
`mount -t ubifs <ubi-device-id>:<volume-name> <mount-point>`
- ▶ Example:
`mount -t ubifs ubi0:data /data`



Linux: UBI image creation workflow





Linux: Using a UBIFS filesystem as root filesystem

- ▶ You just have to pass the following information on the kernel command line:
 - ▶ `ubi.mtd=UBI`
Attach the MTD partition named `UBI` to the UBI layer and create `ubi0`.
Note: you can also use the MTD partition number (more error prone): `ubi.mtd=1`
 - ▶ `rootfstype=ubifs root=ubi0:rootfs`
Mount the `rootfs` volume on `ubi0` as a UBIFS filesystem
 - ▶ `rootfstype=` lets the kernel know what filesystem to mount as root filesystem. It's mandatory for UBIFS, but it can also be used for block filesystems. This way the kernel doesn't have to try all the filesystems it supports. This reduces boot time.
- ▶ Example: `rootfstype=ubifs ubi.mtd=UBI root=ubi0:rootfs`



Summary: how to boot on a UBIFS filesystem

In U-Boot:

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

```
setenv bootargs_base console=ttyS0 rootfstype=ubifs root=ubi0:rootfs ubi.mtd=UBI
```

- ▶ Define the boot command sequence, initializing UBI, loading kernel and DTB images from UBI partitions, and adding `mtdparts` to the kernel command line. Example:

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



Linux: Block emulation layers

- ▶ Sometimes we need block devices to re-use existing block filesystems, especially read-only ones like SquashFs
- ▶ Linux provides two block emulation layers:
 - ▶ `mtdblock`: block devices emulated on top of MTD devices
 - ▶ `ubiblock`: block devices emulated on top of UBI volumes



Linux: mtdblock

- ▶ The `mtdblock` layer creates a block device for each MTD device of the system
- ▶ Usually named `/dev/mtdblockX`.
- ▶ Allows read/write block-level access. However bad blocks are not handled, and no wear leveling is done for writes.
- ▶ For historical reasons, JFFS2 and YAFFS2 filesystems require a block device for the `mount` command.
- ▶ **Do not write on `mtdblock` devices**

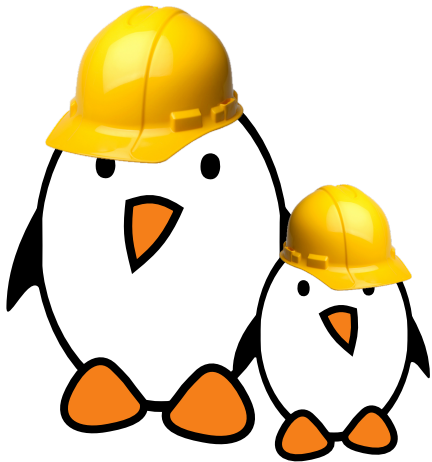


- ▶ `CONFIG_MTD_UBI_BLOCK`
- ▶ Implemented by Ezequiel Garcia from Bootlin.
- ▶ Preferred over `mtdblock` if UBI is available (UBI accounts for data retention and wear leveling issues, while MTD does not)
- ▶ The `ubiblock` layer creates **read-only** block devices on demand
- ▶ The user specifies which static volumes (s)he would like to attach to `ubiblock`
 - ▶ Through the kernel command line: by passing
`ubi.block=<ubi-dev-id>,<volume-name>`
Example: `ubi.block=0,rootfs`
 - ▶ In Linux, using the `ubiblock` utility provided by `mtd-utils`:
`ubiblock --create <ubi-volume-dev-file>`
- ▶ Usually named `/dev/ubiblockX_Y`, where X is the UBI device id and Y is the UBI volume id (example: `/dev/ubiblock0_3`)



Useful reading on Linux on raw flash storage

- ▶ Documentation on the linux-mtd website:
<http://www.linux-mtd.infradead.org/>
 - ▶ You will find answers to your questions there



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

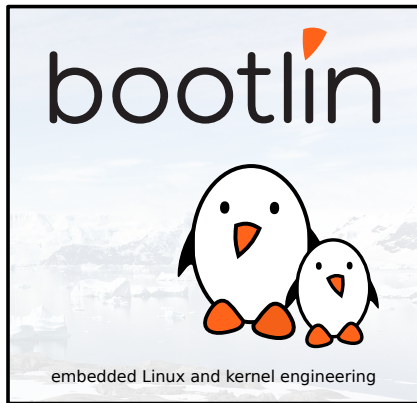


Embedded Linux system development

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Contents

- ▶ Using open-source components
- ▶ Tools for the target device
 - ▶ Networking
 - ▶ System utilities
 - ▶ Language interpreters
 - ▶ Audio, video and multimedia
 - ▶ Graphical toolkits
 - ▶ Databases
 - ▶ Web browsers
- ▶ System building



Leveraging open-source components in an Embedded Linux system



Third party libraries and applications

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



Find existing components

- ▶ Look at the list of software packaged by embedded Linux build systems
 - ▶ These are typically chosen for their suitability to embedded systems
- ▶ Look at other embedded Linux products, and see what their components are.
<https://linuxgizmos.com/> is a good source of information.
- ▶ Ask the community or your favorite search engine
- ▶ 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!



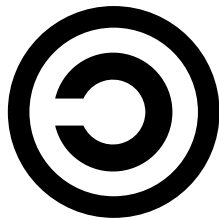
Licenses (1)

- ▶ 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 <https://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 <https://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 made proprietary, but they still require attribution





- ▶ **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, U-Boot...)
- ▶ A number of programs are covered by version 3, released in 2007: gcc, bash, grub, samba, Qt...
 - ▶ 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



- ▶ 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 sections 4. and 5.



- ▶ **GNU Lesser General Public License**
- ▶ Covers around 10% of the free software projects
- ▶ A copyleft license
 - ▶ Modified versions must be released under the same license
 - ▶ But, programs linked against a library under the LGPL do not need to be released under the LGPL and can be kept proprietary.
 - ▶ However, the user must keep the ability to update the library independently from the program. Dynamic linking is the easiest solution. Statically linked executables are only possible if the developer provides a way to relink with an update (with source code or linkable object files).
 - ▶ If this constraint is too strong for you, use a library with a more permissive license if you can (such as the *musl* C library, with MIT license).
- ▶ Used instead of the GPL for most of the libraries, including the C libraries
- ▶ Also available in two versions, v2 and v3



Licensing: examples

- ▶ You distribute a system including GPL or LGPL software
 - ▶ You must be ready to distribute the corresponding source code to your customers.
- ▶ 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.
- ▶ 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 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 about 10 well-known licenses, so it is fairly easy for the majority of projects to get a good understanding of the license
- ▶ Otherwise, read the license text
- ▶ Check Free Software Foundation's opinion
<https://www.fsf.org/licensing/licenses/>
- ▶ Check Open Source Initiative's opinion
<https://www.opensource.org/licenses>



Respect free software licenses

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



Keeping changes separate (1)

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



Keeping changes separate (2)

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



Tools for the target device: Networking



ssh server and client: Dropbear

<https://matt.ucc.asn.au/dropbear/dropbear.html>

- ▶ Very small memory footprint ssh server for embedded systems
- ▶ Satisfies most needs. Both client and server!
- ▶ Size: 204 KB, dynamically compiled with musl on ARM (Buildroot 2020.11 with Bootlin musl toolchain)
- ▶ Useful to:
 - ▶ Get a remote console on the target device
 - ▶ Copy files to and from the target device (`scp` or `rsync`).
- ▶ 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*: <https://busybox.net>
 - ▶ Tiny: only adds 20 K to BusyBox (dynamically linked on arm, with all features enabled.)
 - ▶ Sufficient features for many devices with a web interface, including CGI, http authentication, script support (like PHP, with a separate interpreter), reverse proxy...
 - ▶ License: GPL
- ▶ Other possibilities: lightweight servers like *Boa*, *thttpd*, *lighttpd*, *nginx*, etc
- ▶ Some products are using *Node.js*, which is lightweight enough to be used. *low.js* (<https://github.com/neonious/lowjs>) is even lighter, and is available on Linux and microcontrollers.





Network utilities (1)

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



Network utilities (2)

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



Tools for the target device: System utilities



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



Tools for the target device: Language interpreters



Language interpreters

- ▶ Interpreters for the most common scripting languages are available. Useful for
 - ▶ Application development
 - ▶ Web services development
 - ▶ Scripting
- ▶ Languages supported
 - ▶ Lua, easy to embedded in C applications
 - ▶ Python
 - ▶ Perl
 - ▶ Ruby
 - ▶ TCL
 - ▶ PHP



Tools for the target device: Audio, video and multimedia



Audio, video and multimedia

- ▶ **GStreamer**, a multimedia framework
 - ▶ Allows to decode/encode a wide variety of codecs.
 - ▶ Supports hardware encoders and decoders through plugins, proprietary/specific plugins are often provided by SoC vendors.
- ▶ **alsa-lib**, the user space tools associated to the ALSA kernel sound subsystem
- ▶ Directly using encoding and decoding libraries, if you decide not to use GStreamer:
 - ▶ **libavcodec**: from the *ffmpeg* project, used in players such *vlc* and *mpplayer*, and supporting most audio and video codecs (mpeg4, h264, vp8, vp9...)
 - ▶ **libvpx**: vp8 and vp9 video encoding
 - ▶ **libflac**: *FLAC: Free Lossless Audio Codec*
 - ▶ **libvorbis**: lossy audio codec
 - ▶ **libopus**: lossy audio codec (replacement for Vorbis)
 - ▶ **libmad**: mp3 audio codec
 - ▶ **libspeex**: audio codec optimized for human speech



Tools for the target device: Graphical toolkits



Graphical toolkits: “Low-level” solutions and layers

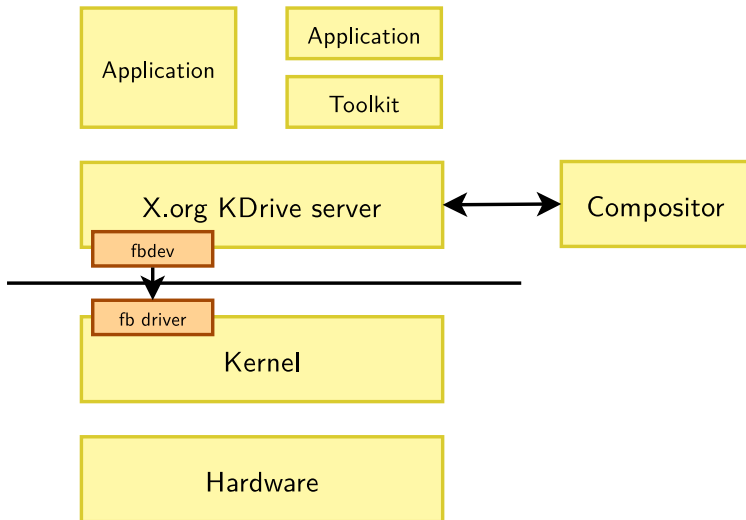


- ▶ Stand-alone simplified version of the X server, for embedded systems
 - ▶ Formerly know as Tiny-X
 - ▶ Kdrive is integrated in the official X.org server
- ▶ Works on top of the Linux frame buffer, thanks to the Xfbdev variant of the server
- ▶ Real X server
 - ▶ Fully supports the X11 protocol: drawing,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





Kdrive: architecture





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

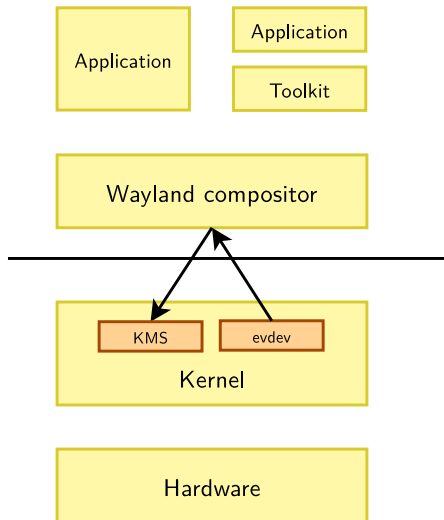


- ▶ A simpler replacement for X
- ▶ *Wayland is a protocol for a compositor to talk to its clients as well as a C library implementation of that protocol.*
- ▶ Weston: a minimal and fast reference implementation of a Wayland compositor, and is suitable for many embedded and mobile use cases.
- ▶ Most graphical toolkits (Gtk, Qt, EFL...) support Wayland now.
- ▶ Most desktop distributions support it: Fedora, Debian, Ubuntu (from 21.04 on)
- ▶ [https://en.wikipedia.org/wiki/Wayland_\(display_server_protocol\)](https://en.wikipedia.org/wiki/Wayland_(display_server_protocol))





Wayland: architecture

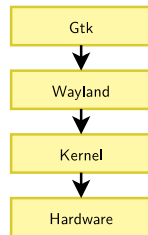
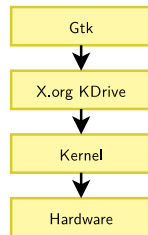




Graphical toolkits: “High-level” solutions



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



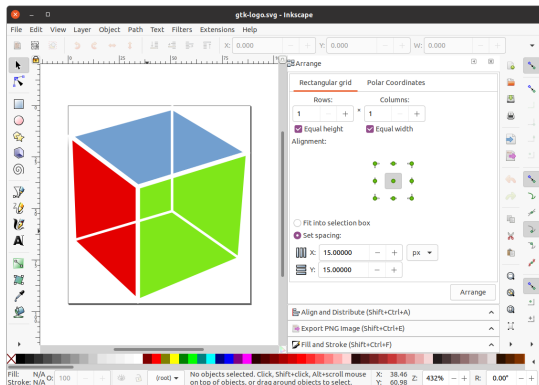


Gtk stack components

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



Gtk example (Inkscape)



Unfortunately GTK is losing traction in embedded.

Mer, the descendent of Maemo, a GTK based framework for tablets and phones, has now been implemented in EFL (see next slides).



Qt (1)

- ▶ The other famous toolkit, providing widget-based high-level APIs to develop graphical applications
- ▶ Implemented in C++
 - ▶ the C++ library is required on the target system
 - ▶ standard API in C++, but with bindings for other languages
- ▶ Works either on top of
 - ▶ Framebuffer
 - ▶ X11
 - ▶ Wayland
- ▶ Multiplatform: Linux, MacOS, Windows.
- ▶ <https://www.qt.io/>





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): <https://j.mp/W4PK85>
- ▶ Qt Embedded has an integrated windowing system, allowing several applications to share the same screen
- ▶ Very well documented
- ▶ License: mix of LGPLv3 and GPLv3 (and LGPLv2 and GPLv2 for some parts), making it difficult to implement non GPL applications. According to customers, the commercial license is very expensive (about 5 USD per unit for volumes in thousands of devices).



Qt's usage



Source: <https://www.qt.io/qt-for-device-creation/>



Other graphical toolkits

- ▶ Enlightenment Foundation Libraries (EFL) / Elementary
 - ▶ Very powerful. Supported by Samsung, Intel and Free.fr.
 - ▶ Work on top of X or Wayland.
 - ▶ License: BSD
 - ▶ <https://www.enlightenment.org/about-efl>
- ▶ Fast Light Toolkit (FLTK)
 - ▶ Very lightweight, multi-platform, widget library written in C++
 - ▶ The "hello" program fits in 100 KiB, statically linked
 - ▶ Work on top of X or Wayland (port in progress).
 - ▶ License: LGPL
 - ▶ <https://www.fltk.org/>

See https://en.wikipedia.org/wiki/List_of_widget_toolkits



Further details on Linux graphics

Check out the freely available materials from our training course on Linux graphics:

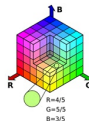
- ▶ Image processing theory, hardware, kernel and userspace aspects...
- ▶ More than 200 pages

<https://bootlin.com/training/graphics>



Light representation, color quantization

- ▶ Light itself must be quantized in digital representations *distinct from and unrelated to spatial quantization*
- ▶ Translating light information (colors) to numbers:
 - ▶ Using a translation referential called **colorspace**
 - ▶ The translated color has **coordinates** in the colorspace e.g. 3 for a human-eye-alike referential: red, green, blue
- ▶ Color coordinates are quantized with:
 - ▶ A given **resolution**: the smallest possible color difference
 - ▶ A given **range**: the span of representable colors
- ▶ Different approaches exist for color quantization:
 - ▶ **Uniform** quantization in the color range (most common) values are attributed to colors with a regular step (*resolution*)
 - ▶ **Irregular** quantization with indexed colors (palettes) values are attributed to colors as needed

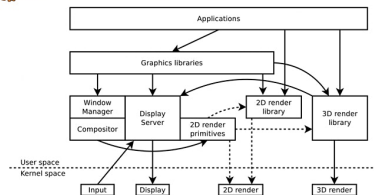


bootlin - Kernel, drivers and embedded Linux - Development, consulting, training and support - <https://bootlin.com>

21/208



System-agnostic overview (illustrated)



bootlin - Kernel, drivers and embedded Linux - Development, consulting, training and support - <https://bootlin.com>

21/208



Tools for the target device: Databases



Lightweight database - SQLite

<https://www.sqlite.org>

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



Tools for the target device: Web browsers



<https://webkit.org/>

- ▶ Web browser engine. Application framework that can be used to develop web browsers or add HTML rendering capability to your applications. You could also replace your application by a full-screen browser (easier to implement).
- ▶ License: portions in LGPL and others in BSD. Proprietary applications allowed.
- ▶ Used by many web browsers: Safari, iPhone and Android default browsers ... Google Chrome now uses a fork of its WebCore component). Used by e-mail clients too to render HTML.
- ▶ Multiple graphical back-ends: Qt, GTK, EFL...



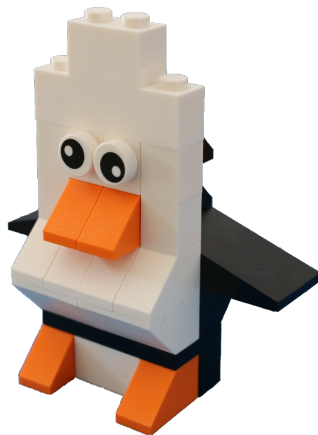


System building



System building: goal and solutions

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



Penguin picture: <https://bit.ly/1PwDklz>



System building: manually

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



System building: manually (2)

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

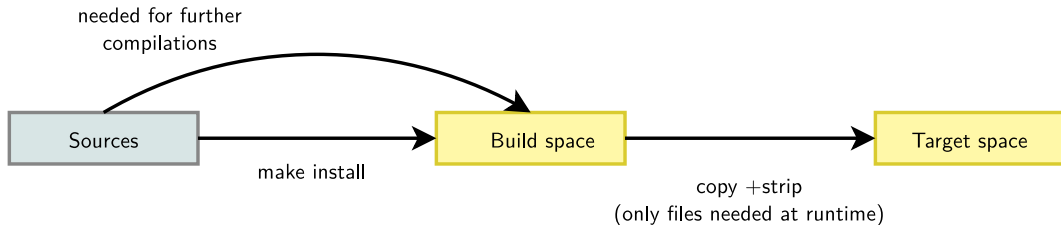


- ▶ A basic root file system needs at least
 - ▶ A traditional directory hierarchy, with `/bin`, `/etc`, `/lib`, `/root`, `/usr/bin`, `/usr/lib`, `/usr/share`, `/usr/sbin`, `/var`, `/sbin`
 - ▶ A set of basic utilities, providing at least the `init` program, a shell and other traditional UNIX command 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 systems, 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 at least the headers, binaries and configuration files.





Build systems

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

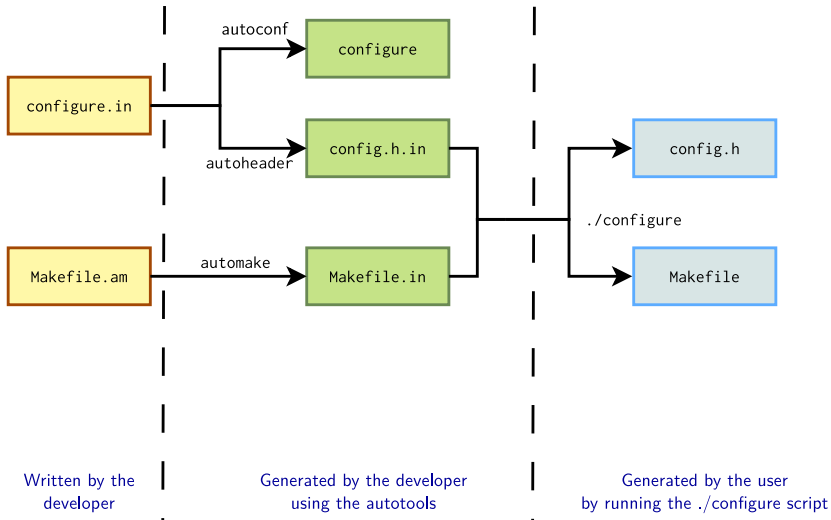
- ▶ A basic `Makefile`
 - ▶ Need to read the `Makefile` to understand how it works and how to tweak it for cross-compilation
- ▶ A build system based on the *Autotools*
 - ▶ As this is the most common build system, we will study it in details
- ▶ CMake, <https://cmake.org/>
 - ▶ Newer and simpler than the *autotools*. Used by (sometimes large) projects such as KDE, KiCad, LLVM / Clang, Scribus, OpenCV, Qt (since version 6).
- ▶ Scons, <https://scons.org/>
- ▶ Waf, <https://waf.io/>
- ▶ Other manual build systems



- ▶ 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





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



Configuring and compiling: native case

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



Configuring and compiling: cross case (1)

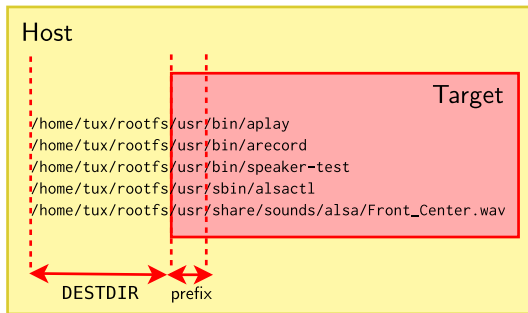
- ▶ For cross-compilation, things are a little bit more complicated.
- ▶ At least some of the environment variables `AR`, `AS`, `LD`, `NM`, `CC`, `GCC`, `CXX`, `STRIP`, `OBJCOPY` must be defined to point to the proper cross-compilation tools. The host tuple is also by default used as prefix.
- ▶ `configure`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



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.

```
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/rootfs install
```





Installation (1)

- ▶ The autotools based software packages provide both `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` (older alternative to `pkgconfig`)
 - ▶ 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 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_LIBDIR` 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_LIBDIR` 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.



Further details about autotools

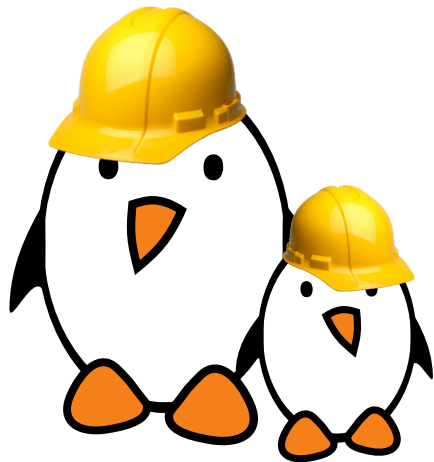
See our *Demystification tutorial* presentation about the GNU Autotools by Thomas Petazzoni, 2016: <https://bit.ly/2hjpojv> (101 pages!)



Note: the Youtube video has been deleted by mistake



Practical lab - Third party libraries and applications



- ▶ Manually cross-compiling applications and libraries
- ▶ Learning about common techniques and issues.
- ▶ Compile and run an audio player application!



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 support 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
<https://buildroot.org>
See our dedicated course and training materials:
<https://bootlin.com/training/buildroot/>
- ▶ **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project
<https://openwrt.org>
- ▶ **PTXdist**, developed by Pengutronix
<https://www.ptxdist.org>
Similar configuration interface (menuconfig), but a bit difficult to grasp at first.
- ▶ **OpenEmbedded**, more flexible but also far more complicated
<http://www.openembedded.org>, its industrialized version **Yocto Project** and vendor-specific derivatives such as **Arago**.
See our dedicated course and training materials:
<https://bootlin.com/training/yocto/>.



Buildroot (1)



- ▶ Allows to build a toolchain, a root filesystem image with many applications and libraries, a bootloader and a kernel image
 - ▶ Or any combination of the previous items
- ▶ Supports building uClibc, glibc and musl toolchains, either built by Buildroot, or external
- ▶ Over 2000+ applications or libraries integrated, from basic utilities to more elaborate software stacks: Wayland, GStreamer, Qt, Gtk, WebKit, Python, PHP, etc.
- ▶ Good for small to medium embedded systems, with a fixed set of features
 - ▶ No support for generating packages (.deb or .ipk)
 - ▶ Needs complete rebuild for most configuration changes.
- ▶ Active community, releases published every 3 months. One LTS release made every year (YYYY.02 so far).



Buildroot (2)

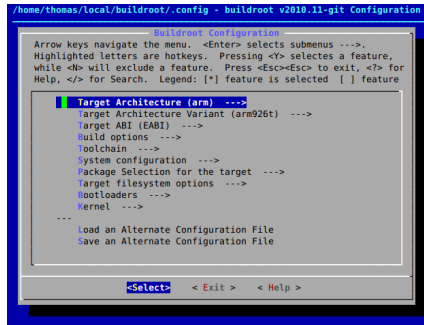
- Configuration takes place through a *config interface similar to the kernel

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.
 - ▶ A `.hash` file (optional, but recommended), containing hashes for the files to download.
 - ▶ Patches (optional). Each file of the form `*.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"
GQVIEW_LICENSE = GPL-2.0
GQVIEW_LICENSE_FILES = COPYING

$(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.



- ▶ 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
- ▶ Allows to generate and maintain custom distributions
- ▶ Good for larger embedded Linux systems, or people looking for more configurability and extensibility
- ▶ Drawbacks: very steep learning curve, very long first build.



Debian GNU/Linux, <https://www.debian.org>

- ▶ Provides the easiest environment for quickly building prototypes and developing applications. Countless runtime and development packages available.
- ▶ But probably too costly to maintain and unnecessarily big for production systems.
- ▶ Available on multiple architectures: ARM (`armel`, `armhf`, `arm64`), MIPS, PowerPC, RISC-V (in progress)...
- ▶ Software is compiled natively by default.
- ▶ Use the `debootstrap` command to build a root filesystem for your architecture, with a custom selection of packages.
- ▶ ELBE (<https://elbe-rfs.org>) is a more advanced environment for generating custom root filesystems based on Debian.





Fedora

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



Image credits: <https://bit.ly/2EzmJLF>

Ubuntu

- ▶ <https://ubuntu.com/download/iot>
- ▶ Ubuntu Desktop supported on Raspberry Pi
- ▶ Ubuntu Core targeting more real embedded projects, packaging and securing applications through *Snaps*, and offering up to 10 years of security updates.



Image credits:
<https://tinyurl.com/f4zxj5kw>



Even if you don't use them for final products, they can be useful to make demos quickly

▶ **Alpine Linux:** <https://www.alpinelinux.org/>

- ▶ Security oriented distribution based on *Musl* and *BusyBox*
- ▶ Supports `x86` and `arm`, both 32 and 64 bit, plus `ppc64` and `s390`
- ▶ Multiple types of downloads supported
 - ▶ Standard version: about 130 MB
 - ▶ Mini root filesystem: about 4 MB (without kernel)
 - ▶ Other images: Raspberry Pi, Virtual, Xen, Generic ARM...





Application frameworks

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

► **Tizen:** <https://www.tizen.org/>

Targeting smartphones, wearables (watches), smart TVs and In Vehicle Infotainment devices.

Supported by big phone manufacturers (mostly Samsung) and operators

HTML5 base application framework.

Wikipedia: 21% of the smart TVs market share in 2018

See <https://en.wikipedia.org/wiki/Tizen>

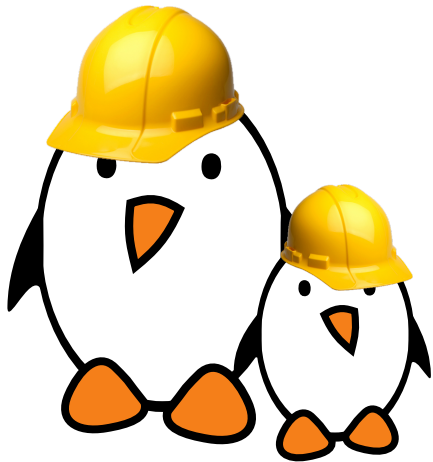
► **Android:** <https://www.android.com/>

Google's distribution for phones, tablets, TVs, cars...

Except the Linux kernel, very different user space than other Linux distributions. Mostly successful in its target markets though.



Image credits:
https://frama.link/yhPuj_oS



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

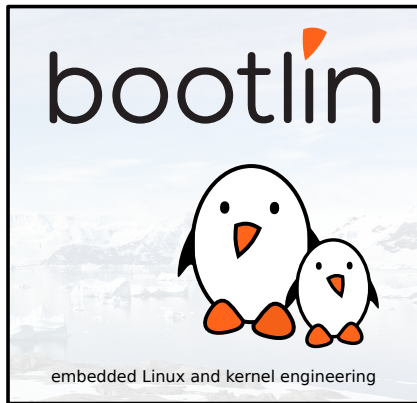


Embedded Linux application development

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Contents

- ▶ Application development
 - ▶ Developing applications on embedded Linux
 - ▶ Building your applications
- ▶ Source management
 - ▶ Integrated development environments (IDEs)
 - ▶ Version control systems
- ▶ Debugging and analysis tools
 - ▶ Debuggers
 - ▶ Memory checkers
 - ▶ System analysis



Developing applications on embedded Linux



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
- ▶ Application development could start on x86, even before the hardware is available.



Programming language (1)

- ▶ The programming language for system-level applications 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
- ▶ The Rust language is increasingly popular in embedded and system applications, as an alternative to C and C++.
See <https://www.rust-lang.org/what/embedded> for attractive features.
- ▶ Suggestion to start with Rust if you neither know C and C++.



Programming language (2)

- ▶ 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
 - ▶ Most popular: Python, shell
- ▶ All programming languages can be used: Lua, Ada, Java, Go...



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/Glib 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
 - ▶ *autotools* is ancient, complicated but very widely used.
 - ▶ We recommend to invest in *CMake* instead: modern, simpler, smaller but growing user base. See an example on <https://cmake.org/examples/>



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)
```

```
↩Tab➡ $(CC) -o $@ $^
```

```
clean:
```

```
↩Tab➡ $(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)
    <Tab> $(CC) -o $@ $^ $(LDFLAGS)

clean:
    <Tab> $(RM) -f foobar $(OBJS)
```



Integrated Development Environments (IDE)



Visual Studio Code

- ▶ <https://code.visualstudio.com/>
- ▶ Created by Microsoft, MIT licensed
- ▶ Extensible, language agnostic text editor
- ▶ Built-in git commands
- ▶ The most popular IDE (open-source and proprietary) according Stack Overflow's 2019 survey
- ▶ Try it on Ubuntu:
`sudo snap install --classic code`
- ▶ See Michael Opdenacker's *Using Visual Studio Code for Embedded Development* presentation ([slides](#), [video](#)) at ELCE 2020.



Image credits (Wikipedia):
<https://frama.link/RjFqWGBS>





Eclipse (1)

<https://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.



Image credits:

<https://frama.link/C6FJ7Py5>



Eclipse (2)

- ▶ Eclipse is actually a platform composed of many projects:
<https://www.eclipse.org/projects/>
 - ▶ Some projects are dedicated to 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.
- ▶ Used by free software build systems and development environments too, such as Yocto Project and Buildroot.
- ▶ Eclipse now supports the Theia project (<https://theia-ide.org/>) that supports VS Code extensions, but with its own architecture. It's now used in many IDEs: new Arduino PRO IDE, ARM mbed, Eclipse Che...



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.
- ▶ People also use **QtCreator**, even for non Qt projects
- ▶ **Atom** (from GitHub) is a very popular text editor too
- ▶ See Stack Overflow's survey of most popular IDEs (2019): <https://frama.link/bfPgbb88>

All these tools are available in most Linux distributions, simply install them and try them out!

Vim

```
return show_irq_affinity(t, n, w);

static ssize_t write_irq_affinity(int type, struct file *file,
const char __user *buffer, size_t count, loff_t *pos)
{
    unsigned int irq = (int)(long)POSIX_DATA(file_inode(file));
    cpumask_var_t new_value;
    int err;

    if (!irq_can_set_affinity(irq) || !no_irq_affinity)
        return -EIO;
    if (!alloc_cpumask_var(&new_value, GFP_KERNEL))
        return -ENOMEM;

    if (type)
        err = cpumask_parselist_user(buffer, count, new_value);
    else
        err = cpumask_parse_user(buffer, count, new_value);
    if (err)
        goto free_cpumask;

    if (!is_affinity_mask_valid(new_value)) {
        err = -EINVAL;
        goto free_cpumask;
    }

    /*
     * Do not allow disabling IRQs completely - it's a too easy
     * way to make the system unusable accidentally -> At least
     * one online CPU still has to be targeted.
     */
    if (!cpumask_intersects(new_value, cpu_online_mask)) {
        /* Special case for empty set - allow the architecture
         * code to set default SMP affinity. */
        err = irq_set_affinity_par(irq, new_value) ? -EINVAL : count;
    } else {
        irq_set_affinity(irq, new_value);
    }
}
```

Emacs

```
See the file Options Buffers Tools C Help.
#sigid int systcl_sched_latency = 600000000;
unsigned int normalized_systcl_sched_latency = 600000000;

/*
 * The initial- and re-scaling of tunables is configurable
 * (default SCHED_TUNABLESCALING_LOG = "(i+log10cpu)");
 */
/*
 * Options are:
 * SCHED_TUNABLESCALING_NONE = untested, always "1"
 * SCHED_TUNABLESCALING_LOG = scaled logarithmic, "(i+log10cpu)"
 * SCHED_TUNABLESCALING_LINEAR = scaled linear, "cpu"
 */
#define sched_tunable_scaling systcl_sched_tunable_scaling
      = SCHED_TUNABLESCALING_LOG;

/*
 * Minimal preemption granularity for CPU-bound tasks:
 * (default: 0.75 msec * (1 + log10cpu)), units: nanoseconds
 */
unsigned int systcl_sched_min_granularity = 75000000;
unsigned int normalized_systcl_sched_min_granularity = 75000000;

/*
 * Is kept at systcl_sched_latency / systcl_sched_min_granularity
 */
static unsigned int sched_nr_latency = 8;

/*
 * After fork, child runs first. If set to 0 (default) then
 * parent will try to run first.
 */
unsigned int systcl_sched_child_runs_first __read_mostly;

/*
 * SCHED_OTHER wake-up granularity.
 * (default: 1 msec * (1 + log10cpu)), units: nanoseconds
 */
```



Version control systems



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. **Subversion** is the **last** popular open-source one:

- ▶ <https://subversion.apache.org/>
- ▶ Created as a replacement of the old CVS, removing many of its limitations.
- ▶ Commits on several files, proper renaming support, better performance, etc.
- ▶ The user interface is very similar to CVS
- ▶ [https://en.wikipedia.org/wiki/Subversion_\(software\)](https://en.wikipedia.org/wiki/Subversion_(software))
- ▶ Fun: see Linus Torvalds expressing his love for CVS and Subversion:
<https://youtu.be/4XpnKHJAok8>

No longer recommended for new projects. Use distributed source control systems instead!



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 (Linux, U-Boot, Barebox, uClibc, GNOME, X.org, etc.)
- ▶ Outstanding performance, in particular in big projects
- ▶ [https://en.wikipedia.org/wiki/Git_\(software\)](https://en.wikipedia.org/wiki/Git_(software))

▶ **Mercurial**

- ▶ Another system, created with the same goals as Git.
- ▶ Used by some big projects too
- ▶ <https://en.wikipedia.org/wiki/Mercurial>

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



Debuggers



The **GNU Project Debugger**

<https://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).
- ▶ Can also be used through graphical IDEs
- ▶ 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).
- ▶ New alternative: *lldb* (<https://lldb.llvm.org/>) from the LLVM project.

See also <https://en.wikipedia.org/wiki/Gdb>





GDB crash course (1)

A few useful GDB commands

- ▶ `break foobar (b)`
Put a breakpoint at the entry of function `foobar()`
- ▶ `break foobar.c:42`
Put a breakpoint in `foobar.c`, line 42
- ▶ `print var` or `print task->files[0].fd (p)`
Print the variable `var`, or a more complicated reference. GDB can also nicely display structures with all their members



GDB crash course (2)

- ▶ `continue (c)`
Continue the execution after a breakpoint
- ▶ `next (n)`
Continue to the next line, stepping over function calls
- ▶ `step (s)`
Continue to the next line, entering into subfunctions
- ▶ `backtrace (bt)`
Display the program stack



Remote debugging



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
 - ▶ `ARCH-linux-gdb` is used on the development workstation, offering all its features.
 - ▶ `gdbserver` is used on the target system (only 100 KB on arm).

`ARCH-linux-gdb`

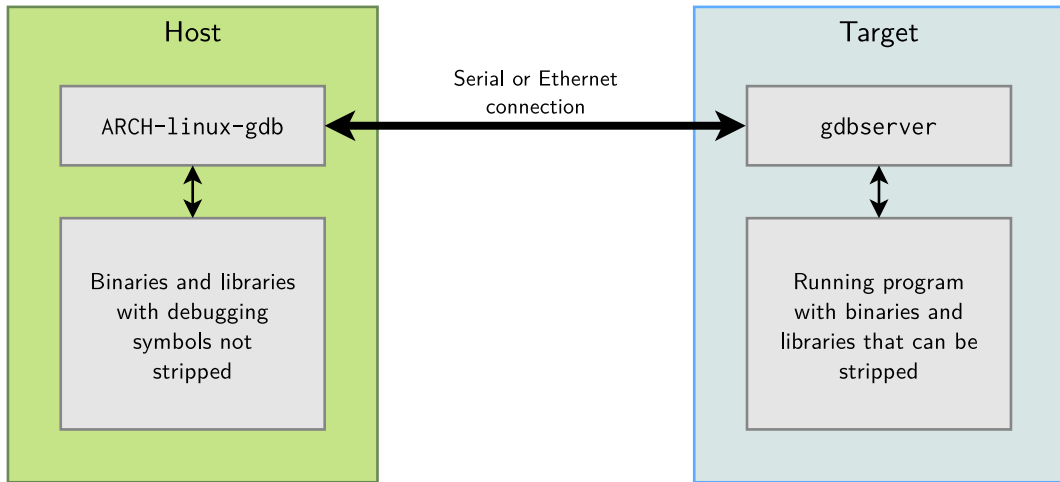


`gdbserver`





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, start `ARCH-linux-gdb <executable>`,
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` in the shell starting the application, 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`



Memory checkers



Valgrind (1)

<https://valgrind.org/>

- ▶ GNU GPL Software suite for debugging and profiling programs.
- ▶ Supported platforms: Linux on x86, x86_64, arm (armv7 only), arm64, mips32, s390, ppc32 and ppc64. Also supported on other operating systems (Android, Darwin, Illumos, Solaris...)
- ▶ 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 <https://valgrind.org/info/> and https://valgrind.org/docs/manual/coregrind_core.html
- ▶ The Valgrind tool suite is easy to add to your root filesystem with Buildroot.





System analysis



System call tracer - <https://strace.io>

- ▶ Available on all GNU/Linux systems
Can be built by your cross-compiling toolchain generator or by your build system.
- ▶ 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)
`strace -c <command>` (statistics of system calls taking most time)

See [the strace manual](#) for details.



Image credits: <https://strace.io/>



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\0\320h\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
```

Hint: follow the open file descriptors returned by open().



strace -c example output

```
> strace -c cheese
```

% time	seconds	usecs/call	calls	errors	syscall
36.24	0.523807	19	27017		poll
28.63	0.413833	5	75287	115	ioctl
25.83	0.373267	6	63092	57321	recvmsg
3.03	0.043807	8	5527		writev
2.69	0.038865	10	3712		read
2.14	0.030927	3	10807		getpid
0.28	0.003977	1	3341	34	futex
0.21	0.002991	3	1030	269	openat
0.20	0.002889	2	1619	975	stat
0.18	0.002534	4	568		mmap
0.13	0.001851	5	356		mprotect
0.10	0.001512	2	784		close
0.08	0.001171	3	461	315	access
0.07	0.001036	2	538		fstat
...					



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. Library calls include system calls too!
- ▶ 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: <https://linux.die.net/man/1/ltrace>
- ▶ Works better with *glibc*. `ltrace` was broken with *uClibc* and may still be.

See <https://en.wikipedia.org/wiki/Ltrace> for details



ltrace example output

```
ltrace nedit index.html
scanf(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
```



Application development

- ▶ Compile your own application with the ncurses library

Remote debugging

- ▶ Set up remote debugging tools on the target: `strace`, `ltrace` and `gdbserver`.
- ▶ Debug a simple application running on the target using remote debugging

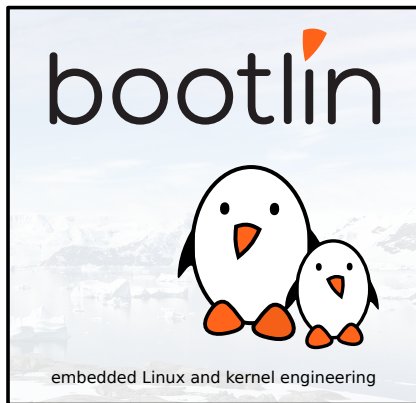


Real-time in embedded Linux systems

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



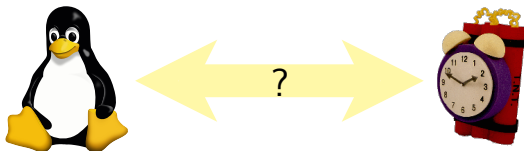


Introduction



Embedded Linux and real time

- ▶ Due to its advantages, Linux and open-source software are more and more commonly used in embedded applications
- ▶ However, some applications also have real-time constraints
- ▶ They, at the same time, want to
 - ▶ Get all the nice advantages of Linux: hardware support, components re-use, low cost, etc.
 - ▶ Get their real-time constraints met





Embedded Linux and real time

- ▶ Linux is an operating system part of the large UNIX family
- ▶ It was originally designed as a time-sharing system
 - ▶ The main goal was to get the best throughput from the available hardware, by making the best possible usage of resources (CPU, memory, I/O)
 - ▶ Time determinism was not taken into account
- ▶ On the opposite, real-time constraints imply time determinism, even at the expense of lower global throughput
- ▶ Best throughput and time determinism are contradictory requirements



Linux and real-time approaches (1)

- ▶ Over time, two major approaches have been taken to bring real-time requirements into Linux
- ▶ **Approach 1**
 - ▶ Improve the Linux kernel itself so that it matches real-time requirements, by providing bounded latencies, real-time APIs, etc.
 - ▶ Approach taken by the mainline Linux kernel and the **PREEMPT_RT** project.
- ▶ **Approach 2**
 - ▶ Add a layer below the Linux kernel that will handle all the real-time requirements, so that the behavior of Linux doesn't affect real-time tasks.
 - ▶ Approach taken by RTLinux, RTAI and **Xenomai**



Linux and real-time approaches (2)

An alternative approach is to use specific hardware to run real-time work on:

- ▶ Dedicating a CPU core to running a separate real-time OS or bare-metal real-time application, without Linux control.
- ▶ Running real-time work on an FPGA
- ▶ Running real-time work on a dedicated microcontroller
 - ▶ Can be a separate micro-controller
 - ▶ Can be a micro-controller embedded in the main system-on-chip: PRU (Programmable Real-Time Unit) on TI AM335x, Cortex-M4 co-processors on NXP i.MX or STM32MP1 processors, etc.

This section will not cover such approaches: we will limit ourselves to approaches where Linux is in full control of the system, including all the real-time processing.



*available for STM32MP157C and STM32MP157F only

STM32MP157 SoC block diagram

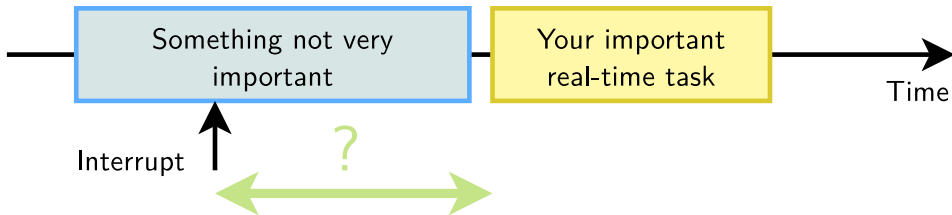


Improving the mainline Linux kernel with PREEMPT_RT



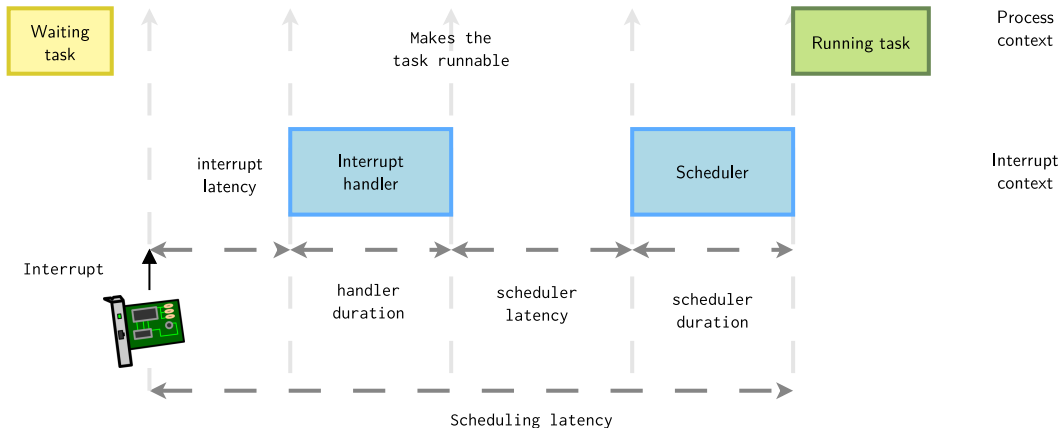
Understanding latency

- ▶ When developing real-time applications with a system such as Linux, the typical scenario is the following
 - ▶ An event from the physical world happens and gets notified to the CPU by means of an interrupt
 - ▶ The interrupt handler recognizes and handles the event, and then wake-up the user space task that will react to this event
 - ▶ Some time later, the user space task will run and be able to react to the physical world event
- ▶ Real-time is about providing guaranteed worst case latencies for this reaction time, called latency





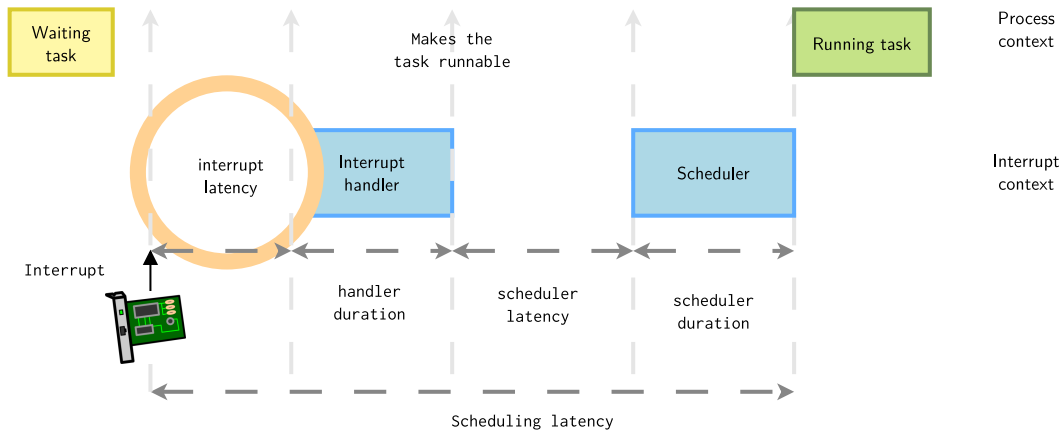
Linux kernel latency components



$$\text{kernel latency} = \text{interrupt latency} + \text{handler duration} + \text{scheduler latency} + \text{scheduler duration}$$



Interrupt latency





Sources of interrupt latency

There can be multiple causes delaying the execution of an interrupt handler.

- ▶ Interrupts disabled by evil drivers. Such drivers may still exist but can be fixed.
- ▶ Already running interrupt handlers. In Linux, there are no nested interrupts. Only one interrupt handler can run at the same time on a given CPU core. Therefore, your critical interrupt handler may wait for the completion of another, non-critical handler.

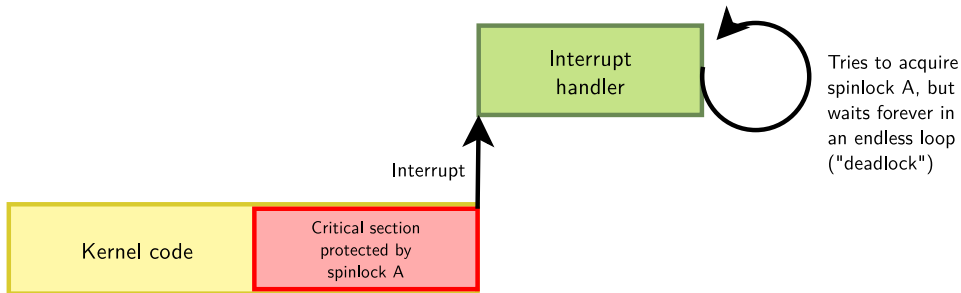
Solution: make interrupt handlers run as quickly as possible.

- ▶ Sections of kernel code when disabling interrupts is necessary, typically in some *spinlocks*.



Disabling interrupts in spinlocks (1)

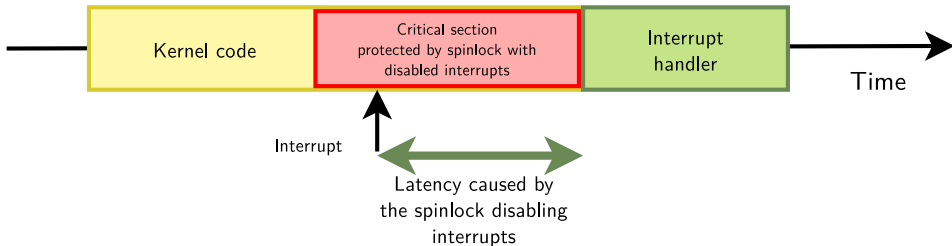
- ▶ One of the concurrency prevention mechanism used in the kernel is the **spinlock**
- ▶ It has several variants, but one commonly used is to prevent concurrent accesses between a process context and an interrupt context works by disabling interrupts
- ▶ We want to avoid situations in which the interrupt handler waits forever for a spinlock that was held by the code it interrupted:





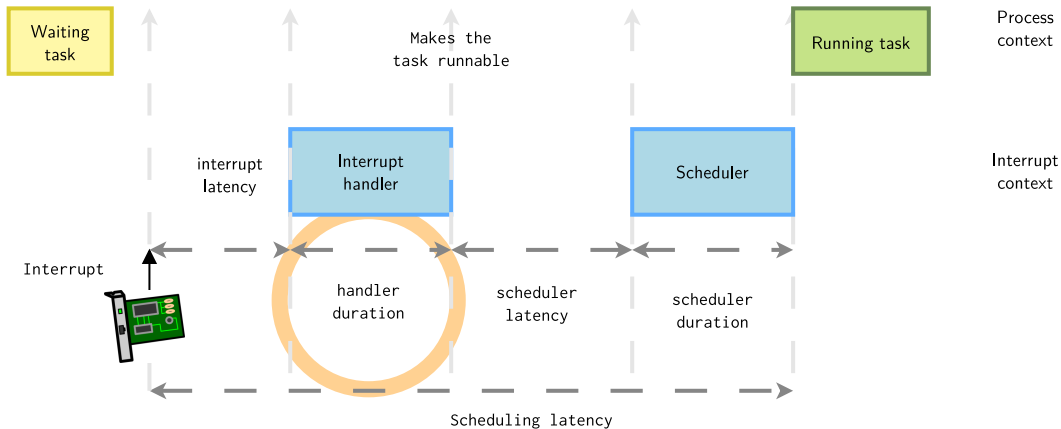
Disabling interrupts in spinlocks (2)

- ▶ To avoid such issues, when there can be concurrency between regular kernel code and interrupt code, spinlocks are held with interrupts disabled.
- ▶ This can delay the execution of a critical interrupt handler
- ▶ The duration of these critical sections in which spinlocks are held with interrupts disabled is unbounded:





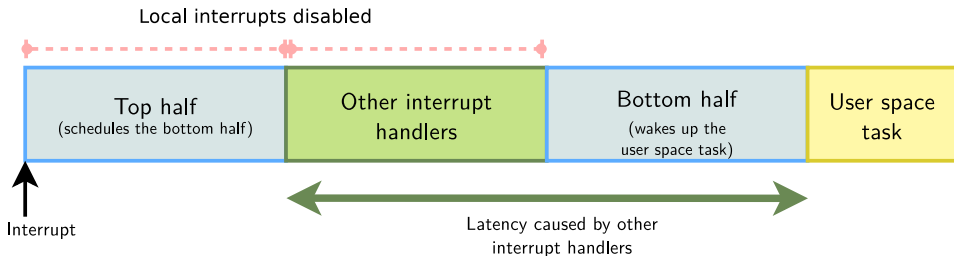
Interrupt handler duration





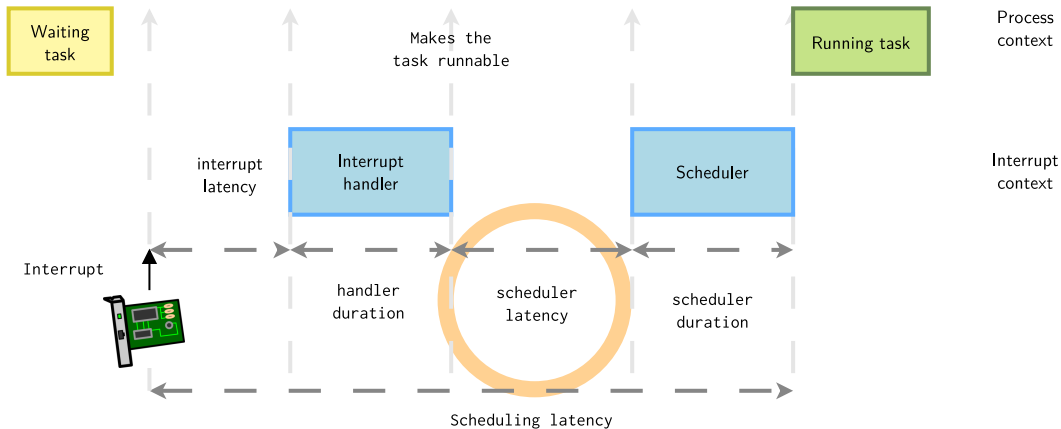
Interrupt handler implementation

- ▶ In Linux, many interrupt handlers are split in two parts
 - ▶ A top-half, started by the CPU as soon as interrupts are enabled. It runs with the interrupt line disabled and is supposed to complete as quickly as possible.
 - ▶ A bottom-half, scheduled by the top-half, which starts after all pending top-halves have completed their execution.
- ▶ Therefore, for real-time critical interrupts, bottom-halves shouldn't be used: their execution is delayed by all other interrupts in the system.





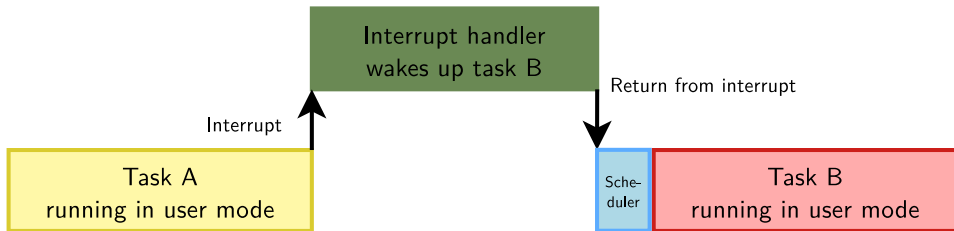
Scheduler latency





Understanding preemption (1)

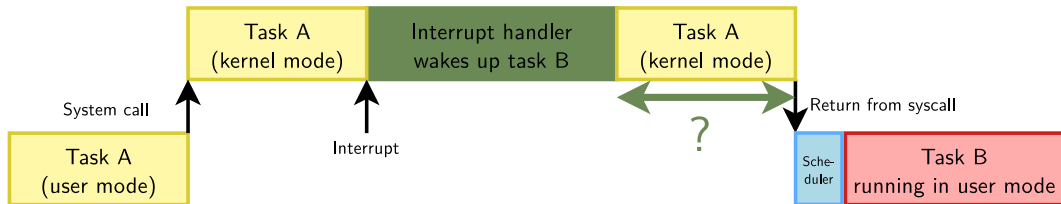
- ▶ The Linux kernel is a preemptive operating system
- ▶ When a task runs in user space mode and gets interrupted by an interruption, if the interrupt handler wakes up another task, this task can be scheduled as soon as we return from the interrupt handler.





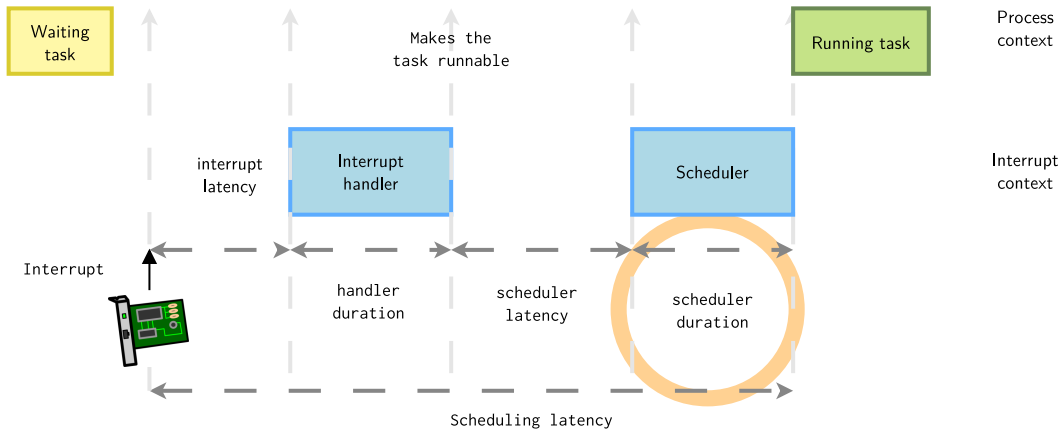
Understanding preemption (2)

- ▶ However, when the interrupt comes while the task is executing a system call, this system call has to finish before another task can be scheduled.
- ▶ By default, the Linux kernel does not do kernel preemption.
- ▶ This means that the time before which the scheduler will be called to schedule another task is unbounded.





Scheduler duration





Sources of scheduler duration

- ▶ Before Linux 2.6, the execution time of the scheduler depended on the number of processes on the system.
- ▶ Since Linux 2.6, the Linux kernel chooses the next process to run in constant time ("O(1) scheduler" feature).



Other non-deterministic mechanisms

Outside of the critical path detailed previously, other non-deterministic mechanisms of Linux can affect the execution time of real-time tasks

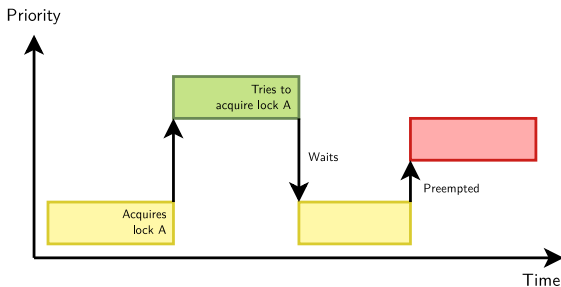
- ▶ Linux is highly based on virtual memory, as provided by an MMU, so that memory is allocated on demand. Whenever an application accesses code or data for the first time, it is loaded on demand, which can create huge delays.
- ▶ Many C library services or kernel services are not designed with real-time constraints in mind.
- ▶ To avoid such sources of non-determinism, your system should allocate all the resources it needs ahead of time, before it is ready to react to events in a real-time way. For the virtual memory needs, it will be done through the `mlock()` and `mlockall()` system calls.

Issues can be addressed with a correct system design!



Priority inversion

A process with a low priority might hold a lock needed by a higher priority process, effectively reducing the priority of this process. Things can be even worse if a middle priority process uses the CPU.

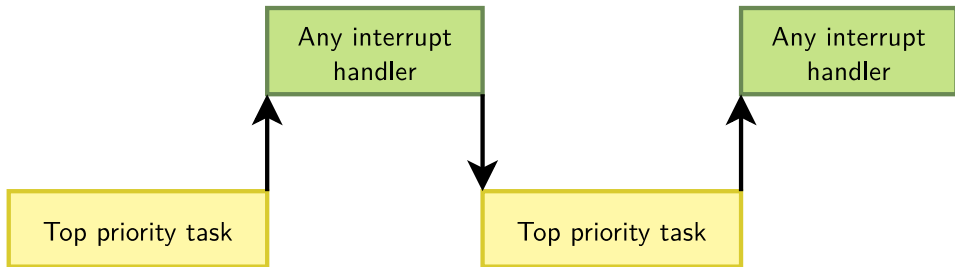


Best known case: Mars Pathfinder mission in 1997 (running VxWorks).
See <https://www.embedded.com/introduction-to-priority-inversion/>



Interrupt handler priority

In Linux, interrupt handlers are executed directly by the CPU interrupt mechanisms, and not under control of the Linux scheduler. Therefore, all interrupt handlers have a higher priority than all tasks running on the system.





The PREEMPT_RT project

- ▶ Long-term project (started in 2004!) lead by Linux kernel developers such as Thomas Gleixner, Steven Rostedt and Ingo Molnar
 - ▶ <https://wiki.linuxfoundation.org/realtime/>
- ▶ The goal is to gradually improve the Linux kernel regarding real-time requirements and to get these improvements merged into the mainline kernel
 - ▶ PREEMPT_RT development works very closely with the mainline development
- ▶ Many of the improvements designed, developed and debugged inside PREEMPT_RT over the years are now part of the mainline Linux kernel
 - ▶ The project is a long-term branch of the Linux kernel that ultimately should disappear as everything will have been merged



Improvements to the mainline kernel

Brought by PREEMPT_RT project since its beginning

- ▶ O(1) scheduler
- ▶ Kernel preemption
- ▶ Better POSIX real-time API support
- ▶ *ftrace* kernel function tracer
- ▶ Priority inheritance support for mutexes
(*mutex*: *MUTual EXclusion*, locking mechanisms)
- ▶ High-resolution timers
- ▶ Threaded interrupts
- ▶ Spinlock annotations



Preemption models offered by the mainline kernel (Linux 5.7 status):

Preemption Model

No Forced Preemption (Server)

<X> Voluntary Kernel Preemption (Desktop)

Preemptible Kernel (Low-Latency Desktop)



1st option: no forced preemption

CONFIG_PREEMPT_NONE

Kernel code (interrupts, exceptions, system calls) never preempted. Default behavior in standard kernels.

- ▶ Best for systems making intense computations, on which overall throughput is key.
- ▶ Best to reduce task switching to maximize CPU and cache usage (by reducing context switching).
- ▶ Still benefits from some Linux real-time improvements: $O(1)$ scheduler, increased multiprocessor safety (work on RT preemption was useful to identify hard to find SMP bugs).
- ▶ Can also benefit from a lower timer frequency (several possible values between 100 Hz and 1000 Hz).



2nd option: voluntary kernel preemption

CONFIG_PREEMPT_VOLUNTARY

Kernel code can preempt itself

- ▶ Typically for desktop systems, for quicker application reaction to user input.
- ▶ Adds explicit rescheduling points (`might_sleep()`) throughout kernel code.
- ▶ Minor impact on throughput.
- ▶ Still used in: Ubuntu Desktop 20.04

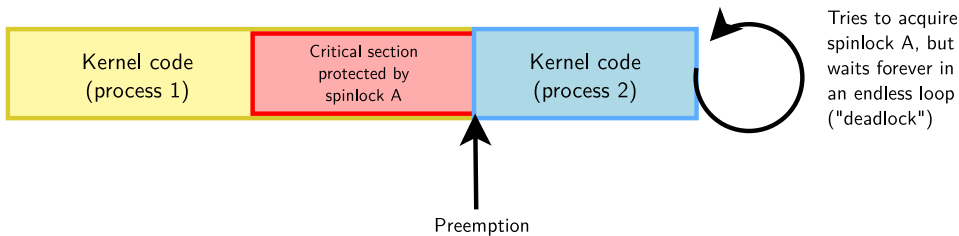


3rd option: preemptible kernel

CONFIG_PREEMPT

Most kernel code can be involuntarily preempted at any time. When a process becomes runnable, no more need to wait for kernel code (typically a system call) to return before running the scheduler.

- ▶ Exception: kernel critical sections (holding spinlocks):

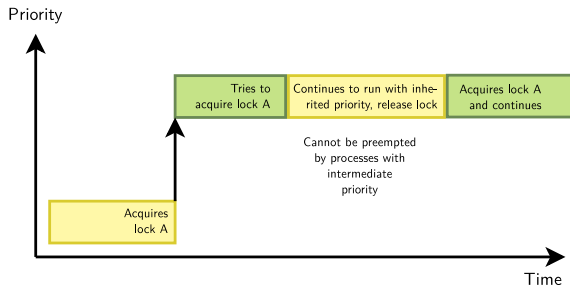


- ▶ Typically for desktop or embedded systems with latency requirements in the milliseconds range. Still a relatively minor impact on throughput.



Priority inheritance

- ▶ One classical solution to the priority inversion problem is called priority inheritance



- ▶ In the Linux kernel, mutexes support priority inheritance
- ▶ In user space, priority inheritance must be explicitly enabled on a per-mutex basis.



High resolution timers

- ▶ The resolution of the timers used to be bound to the resolution of the regular system tick
 - ▶ Usually 100 Hz or 250 Hz, depending on the architecture and the configuration
 - ▶ A resolution of only 10 ms or 4 ms.
 - ▶ Increasing the regular system tick frequency is not an option as it would consume too many resources
- ▶ The high-resolution timers infrastructure allows to use the available hardware timers to program interrupts at the right moment.
 - ▶ Hardware timers are multiplexed, so that a single hardware timer is sufficient to handle a large number of software-programmed timers.
 - ▶ Usable directly from user space using the usual timer APIs



Threaded interrupts

- ▶ To solve the interrupt inversion problem, PREEMPT_RT has introduced the concept of threaded interrupts
- ▶ The interrupt handlers run in normal kernel threads, so that the priorities of the different interrupt handlers can be configured
- ▶ The real interrupt handler, as executed by the CPU, is only in charge of masking the interrupt and waking-up the corresponding thread
- ▶ The idea of threaded interrupts also allows to use sleeping spinlocks (see later)
- ▶ The conversion of interrupt handlers to threaded interrupts is not automatic: drivers must be modified
- ▶ In PREEMPT_RT, all interrupt handlers are switched to threaded interrupts



PREEMPT_RT patchset specifics



New real-time preemption option (1)

The PREEMPT_RT patchset adds one new level of preemption, `CONFIG_PREEMPT_RT`:

Preemption Model

No Forced Preemption (Server)

Voluntary Kernel Preemption (Desktop)

Preemptible Kernel (Low-Latency Desktop)

<X> Fully Preemptible Kernel (Real-Time)



New real-time preemption option (2)

Fully Preemptible Kernel (Real-Time)

`CONFIG_PREEMPT_RT`:

This option turns the kernel into a real-time kernel by replacing various locking primitives (spinlocks, rwlocks, etc.) with preemptible priority-inheritance aware variants, enforcing interrupt threading and introducing mechanisms to break up long non-preemptible sections. This makes the kernel, except for very low level and critical code paths (entry code, scheduler, low level interrupt handling) fully preemptible and brings most execution contexts under scheduler control.

Select this if you are building a kernel for systems which require real-time guarantees.



This level of preemption replaces all kernel spinlocks by mutexes (or so-called sleeping spinlocks)

- ▶ Instead of providing mutual exclusion by disabling interrupts and preemption, they are just normal locks: when contention happens, the process is blocked and another one is selected by the scheduler.
- ▶ Works well with threaded interrupts, since threads can block, while usual interrupt handlers could not.
- ▶ Some core, carefully controlled, kernel spinlocks remain as normal spinlocks (*spinlock annotations*: differentiation now in mainline between spinning spinlocks and those that can be converted to sleeping spinlocks).
- ▶ With `CONFIG_PREEMPT_RT`, virtually all kernel code becomes preemptible
 - ▶ An interrupt can occur at any time, when returning from the interrupt handler, the woken up process can start immediately.



Setting up PREEMPT_RT



PREEMPT_RT setup (1)

- ▶ PREEMPT_RT is delivered as a patch against the mainline kernel
 - ▶ Only the latest and long term stable releases of the kernel are actively supported and maintained, currently: 5.12, 5.10, 5.4, 4.19, 4.14, 4.9, 4.4. This is yet another advantage of using a mainline LTS kernel release! It will be difficult to port the patches to other kernel versions.
 - ▶ Access to supported versions:
https://wiki.linuxfoundation.org/realtime/preempt_rt_versions
- ▶ Quick set up:
 - ▶ Download the latest PREEMPT_RT patch
from <https://kernel.org/pub/linux/kernel/projects/rt/>
 - ▶ Download and extract the corresponding mainline kernel version
 - ▶ Apply the patch to the mainline kernel tree



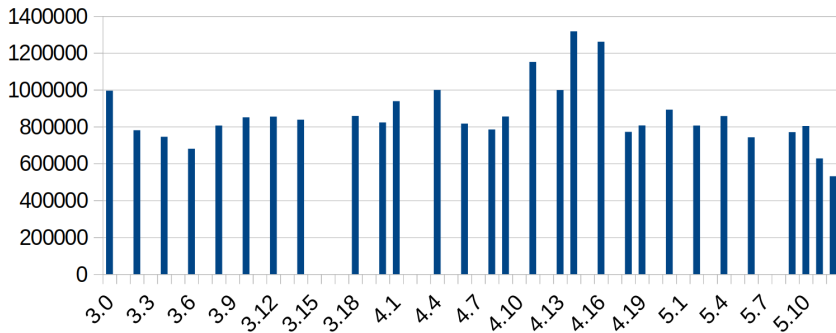
PREEMPT_RT setup (2)

- ▶ In the kernel configuration, be sure to enable
 - ▶ `CONFIG_PREEMPT_RT`
 - ▶ High-resolution timers
- ▶ Compile your kernel, and boot
- ▶ You are now running the real-time Linux kernel
- ▶ Of course, some system configuration remains to be done, in particular setting appropriate priorities to the interrupt threads, which depend on your application.



PREEMPT_RT mainlining status (1)

- ▶ The project is making good progress since it got funding from the Linux Foundation in 2015 (Linux version 4.1 at that time).
- ▶ We can now observe a slow but steady reduction of the size of the PREEMPT_RT patchset:





PREEMPT_RT mainlining status (2)

- ▶ However, the mainline Linux kernel is a moving target too, introducing new issues for real-time (such as disabling preemption in BPF... see <https://lwn.net/Articles/802884/>).
- ▶ A major achievement though in Linux 5.3: the `CONFIG_PREEMPT_RT` configuration option is now in mainline, though it cannot be selected yet (missing dependencies). This simplifies further mainlining work.
- ▶ In 5.12, new `CONFIG_PREEMPT_DYNAMIC` switch to change the preemption model at boot time: `preempt=none`, `preempt=voluntary` or `preempt=full`
- ▶ See the latest news on <https://wiki.linuxfoundation.org/realtime/>



Useful resources

About real-time support in the standard Linux kernel

- ▶ All LWN.net articles about Linux and real-time:
<https://lwn.net/Kernel/Index/#Realtime>
- ▶ A realtime's developer checklist, John Ogness and Marta Rybczyńska:
<https://lwn.net/Articles/837019/>
- ▶ Real-Time is coming to Linux - What does that mean for you, Steven Rostedt, ELCE 2018:
<https://www.elinux.org/images/7/7e/Rostedt-elc-eu-2018-rt-what-does-it-mean.pdf>
Video: <https://youtu.be/BxJm-Ujipcg>
- ▶ The home page of the PREEMPT_RT project:
<https://wiki.linuxfoundation.org/realtime/>



Real-time application development with PREEMPT_RT



Development and compilation

- ▶ No special library is needed, the POSIX real-time API is part of the standard C library
- ▶ The glibc C library is recommended, as support for some real-time features is not mature in other C libraries
 - ▶ Priority inheritance mutexes or NPTL on some architectures, for example
- ▶ Compile a program
 - ▶ `ARCH-linux-gcc -o myprog myprog.c -lrt`
- ▶ To get the documentation of the POSIX API
 - ▶ Install the `manpages-posix-dev` package
 - ▶ Run `man function-name`



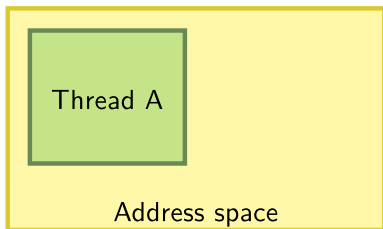
Process, thread?

- ▶ Confusion about the terms *process*, *thread* and *task*
- ▶ In UNIX, a process is created using `fork()` and is composed of
 - ▶ An address space, which contains the program code, data, stack, shared libraries, etc.
 - ▶ One thread, that starts executing the `main()` function.
 - ▶ Upon creation, a process contains one thread
- ▶ Additional threads can be created inside an existing process, using `pthread_create()`
 - ▶ They run in the same address space as the initial thread of the process
 - ▶ They start executing a function passed as argument to `pthread_create()`

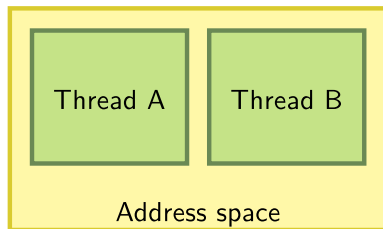


Process, thread: kernel point of view

- ▶ The kernel represents each thread running in the system by a `struct task_struct` structure.
- ▶ From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`



Process after `fork()`



Process after `pthread_create()`



Creating threads

- ▶ Linux supports the POSIX thread API
- ▶ To create a new thread

```
pthread_create(pthread_t *thread, pthread_attr_t *attr,  
               void *(*routine)(void*), void *arg);
```

- ▶ The new thread will run in the same address space, but will be scheduled independently
- ▶ Exiting from a thread

```
pthread_exit(void *value_ptr);
```

- ▶ Waiting for the termination of a thread

```
pthread_join(pthread_t *thread, void **value_ptr);
```



Scheduling classes (1)

The Linux kernel scheduler supports different scheduling classes

- ▶ The default class (`SCHED_OTHER`), in which threads are started by default, is a *time-sharing* class
 - ▶ All threads, regardless of their priority, get some CPU time
 - ▶ The proportion of CPU time they get is dynamic and affected by the *nice* value, which ranges from -20 (highest) to 19 (lowest). Can be set using the `nice` or `renice` commands



Scheduling classes (2)

- ▶ The real-time classes `SCHED_FIFO`, `SCHED_RR` and `SCHED_DEADLINE`
 - ▶ RT Priorities ranging from 0 (lowest) to 99 (highest)
 - ▶ The highest RT priority thread gets all the CPU time, until it blocks.
 - ▶ With `SCHED_FIFO`, *First In, First Out*, each additional threads with the same RT priority has to wait for the first ones to block.
 - ▶ In `SCHED_RR`, *Round-Robin* scheduling between the threads with the same RT priority. All must block before lower priority threads get CPU time.
 - ▶ In `SCHED_DEADLINE`: guarantees that an RT task will be given an exact amount of cpu time every period.

See <https://man7.org/linux/man-pages/man7/sched.7.html> for details.



Using scheduling classes (1)

- ▶ An existing program can be started in a specific scheduling class with a specific priority using the `chrt` command line tool
 - ▶ Example: `chrt -f 99 ./myprog`
 - f: `SCHED_FIFO`
 - r: `SCHED_RR`
 - d: `SCHED_DEADLINE`
- ▶ The `sched_setscheduler()` API can be used to change the scheduling class and priority of a threads

```
int sched_setscheduler(pid_t pid, int policy,  
                      const struct sched_param *param);
```

- ▶ `policy` can be `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`, etc. (others exist).
- ▶ `param` is a structure containing the priority



Using scheduling classes (2)

- ▶ The priority can be set on a per-thread basis when a thread is created

```
struct sched_param parm;  
pthread_attr_t attr;  
  
pthread_attr_init(&attr);  
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);  
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);  
parm.sched_priority = 42;  
pthread_attr_setschedparam(&attr, &parm);
```

- ▶ Then the thread can be created using `pthread_create()`, passing the `attr` structure.
- ▶ Several other attributes can be defined this way: stack size, etc.



Memory locking

- ▶ In order to solve the non-determinism introduced by virtual memory, memory can be locked
 - ▶ Guarantee that the system will keep it allocated
 - ▶ Guarantee that the system has pre-loaded everything into memory
- ▶ `mlockall(MCL_CURRENT | MCL_FUTURE);`
 - ▶ Locks all the memory of the current address space, for currently mapped pages and pages mapped in the future
- ▶ Other, less useful parts of the API: `munlockall`, `mlock`, `munlock`.
- ▶ Watch out for non-currently mapped pages
 - ▶ Stack pages
 - ▶ Dynamically-allocated memory



Mutexes

- ▶ Allows mutual exclusion between two threads in the same address space
- ▶ Initialization/destruction

```
pthread_mutex_init(pthread_mutex_t *mutex,  
                   const pthread_mutexattr_t *mutexattr);  
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- ▶ Lock/unlock

```
pthread_mutex_lock(pthread_mutex_t *mutex);  
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- ▶ Priority inheritance must be activated explicitly

```
pthread_mutexattr_t attr;  
pthread_mutexattr_init (&attr);  
pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
```



▶ Timer creation

```
timer_create(clockid_t clockid, struct sigevent *evp,  
             timer_t *timerid);
```

- ▶ `clockid` is usually `CLOCK_MONOTONIC`. `sigevent` defines what happens upon timer expiration: send a signal or start a function in a new thread. `timerid` is the returned timer identifier.

▶ Configure the timer for expiration at a given time

```
timer_settime(timer_t timerid, int flags,  
              struct itimerspec *newvalue,  
              struct itimerspec *oldvalue);
```



Timers (2)

- ▶ Delete a timer

```
timer_delete(timer_t timerid)
```

- ▶ Get the resolution of a clock, `clock_getres`
- ▶ Other functions: `timer_getoverrun()`, `timer_gettime()`
- ▶ You can also directly use `nanosleep()` or `clock_nanosleep()` to transparently setup timers for specified amounts of time.



- ▶ Signals are asynchronous notification mechanisms
- ▶ Notification occurs either
 - ▶ By the call of a signal handler. Be careful with the limitations of signal handlers!
 - ▶ By being unblocked from the `sigwait()`, `sigtimedwait()` or `sigwaitinfo()` functions. Usually better.
- ▶ Signal behavior can be configured using `sigaction()`
- ▶ The mask of blocked signals can be changed with `pthread_sigmask()`
- ▶ Delivery of a signal using `pthread_kill()` or `tgkill()`
- ▶ All signals between `SIGRTMIN` and `SIGRTMAX`, 32 signals under Linux.



▶ Semaphores

- ▶ Usable between different processes using named semaphores
- ▶ `sem_open()`, `sem_close()`, `sem_unlink()`, `sem_init()`, `sem_destroy()`, `sem_wait()`, `sem_post()`, etc.

▶ Message queues

- ▶ Allows processes to exchange data in the form of messages.
- ▶ `mq_open()`, `mq_close()`, `mq_unlink()`, `mq_send()`, `mq_receive()`, etc.

▶ Shared memory

- ▶ Allows processes to communicate by sharing a segment of memory
- ▶ `shm_open()`, `ftruncate()`, `mmap()`, `munmap()`, `close()`, `shm_unlink()`



Debugging latencies in PREEMPT_RT



ftrace - Kernel function tracer

Infrastructure that can be used for debugging or analyzing latencies and performance issues in the kernel.

- ▶ Very well documented in [trace/ftrace](#)
- ▶ Negligible overhead when tracing is not enabled at run-time.
- ▶ Can be used to trace any kernel function!



Using ftrace

- ▶ Tracing information available through the `tracefs` virtual fs
- ▶ Mount this filesystem as follows:
`mount -t tracefs nodev /sys/kernel/tracing`
- ▶ Check available tracers in `/sys/kernel/tracing/available_tracers`



Scheduling latency tracer

CONFIG_SCHED_TRACER (*Kernel Hacking* section)

- ▶ Maximum recorded time between waking up a top priority task and its scheduling on a CPU, expressed in us.
- ▶ Check that `wakeup` is listed in `/sys/kernel/tracing/available_tracers`
- ▶ To select, reset and enable this tracer:

```
echo wakeup > /sys/kernel/tracing/current_tracer
echo 0 > /sys/kernel/tracing/tracing_max_latency
echo 1 > /sys/kernel/tracing/tracing_enabled
```

- ▶ Let your system run, in particular real-time tasks.
Dummy example: `chrt -f 5 sleep 1`
- ▶ Disable tracing:

```
echo 0 > /sys/kernel/tracing/tracing_enabled
```

- ▶ Read the maximum recorded latency and the corresponding trace:

```
cat /sys/kernel/tracing/tracing_max_latency
```



Real-time extensions to the Linux kernel



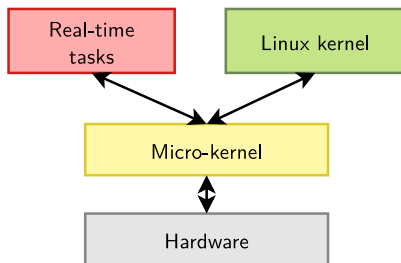
Linux real-time extensions

Three generations

- ▶ RTLinux
- ▶ RTAI
- ▶ Xenomai

A common principle

- ▶ Add an extra layer between the hardware and the Linux kernel, to manage real-time tasks separately.





First real-time extension for Linux, created by Victor Yodaiken.

- ▶ However, the author filed a software patent covering the addition of real-time support to general operating systems as implemented in RTLinux!
- ▶ Its Open Patent License drew many developers away and frightened users. Community projects like RTAI and Xenomai now attract most developers and users.
- ▶ February, 2007: RTLinux rights sold to Wind River. Today, no longer advertised by Wind River.
- ▶ Project completely dead.



<https://www.rtai.org/> - **Real-Time Application Interface for Linux**

- ▶ Created in 1999, by Prof. Paolo Mantegazza (long time contributor to RTLinux), Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM).
- ▶ Community project. Significant user base. Attracted contributors frustrated by the RTLinux legal issues.
- ▶ Only really actively maintained on x86
- ▶ May offer slightly better latencies than Xenomai, at the expense of a less maintainable and less portable code base
- ▶ Since RTAI is not really maintained on ARM and other embedded architectures, our presentation is focused on Xenomai.



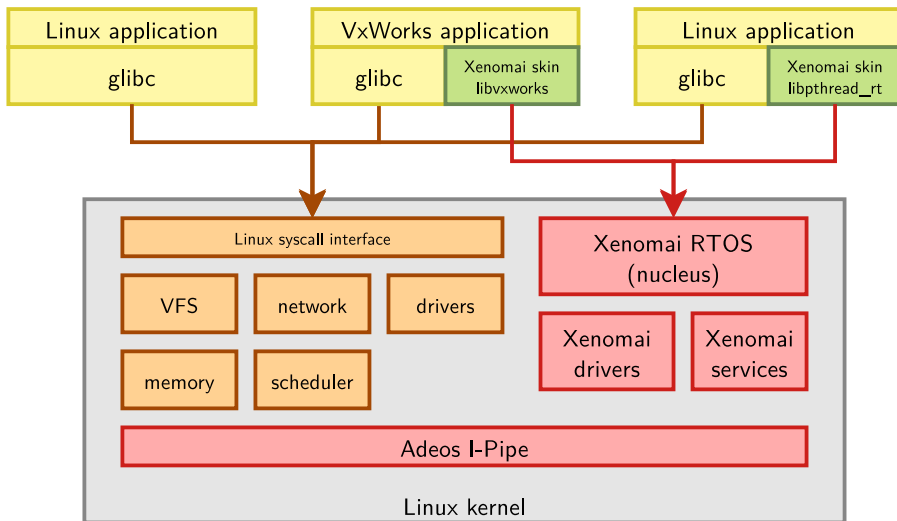
<https://www.xenomai.org/>



- ▶ Started in 2001, as an extension to RTAI, to emulate traditional RTOS APIs.
- ▶ Initial goals: facilitate the porting of programs to GNU / Linux.
- ▶ Skins mimicking the APIs of the VxWorks and pSOS traditional real-time operating systems, as well as the POSIX API, and offering a *native* API too.
- ▶ Offers two solutions:
 - ▶ *Cobalt*: (like **Co**-kernel), using a micro-kernel dedicated to running real-time tasks. This will never be merged in the mainline kernel.
 - ▶ *Mercury*: (like **Mer**ge), based on the mainline kernel with PREEMPT_RT.



Xenomai Cobalt architecture





Xenomai user space real-time support

- ▶ Supported architectures are the ones supported by the *ipipe* patch: arm, arm64, blackfin, powerpc, x86
- ▶ Two modes are defined for a thread
 - ▶ the primary mode, where the thread is handled by the Xenomai scheduler
 - ▶ the secondary mode, when it is handled by the Linux scheduler.
- ▶ Thanks to the services of the Adeos I-pipe service, Xenomai system calls are defined.
 - ▶ A thread migrates from secondary mode to primary mode when such a system call is issued
 - ▶ It migrates from primary mode to secondary mode when a Linux system call is issued, or to handle gracefully exceptional events such as exceptions or Linux signals.



Life of a Xenomai application

- ▶ Xenomai applications are started like normal Linux processes, they are initially handled by the Linux scheduler and have access to all Linux services
- ▶ After their initialization, they declare themselves as real-time applications, which migrates them to primary mode. In this mode:
 - ▶ They are scheduled directly by the Xenomai scheduler, so they have the real-time properties offered by Xenomai
 - ▶ They don't have access to any Linux service, otherwise they get migrated back to secondary mode and lose all real-time properties
 - ▶ They can only use device drivers that are implemented in Xenomai, not the ones of the Linux kernel
- ▶ Need to implement device drivers in Xenomai, and to split real-time and non real-time parts of your applications.



Real Time Driver Model (RTDM)

- ▶ An approach to unify the interfaces for developing device drivers and associated applications under real-time Linux
 - ▶ An API very similar to the native Linux kernel driver API
- ▶ Allows to develop in kernel space:
 - ▶ Character-style device drivers
 - ▶ Network-style device drivers
- ▶ Current notable RTDM based drivers:
 - ▶ Serial port controllers;
 - ▶ RTnet UDP/IP stack;
 - ▶ RT socket CAN, drivers for CAN controllers;
 - ▶ Analogy, fork of the Comedi project, drivers for acquisition cards.

https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__rtdm.html



Setting up Xenomai



How to build the Xenomai kernel (Cobalt)

- ▶ Download Xenomai sources at
<https://xenomai.org/downloads/xenomai/stable/latest/>
- ▶ Download the latest **i-pipe** patch for your architecture:
<https://xenomai.org/downloads/ipipe/>
- ▶ Download the corresponding kernel.org version
- ▶ Xenomai offers a script to patch your kernel. Example (arm):

```
./scripts/prepare-kernel.sh \
    --linux=../linux-4.19.124 \
    --ipipe=../ipipe-core-4.19.124-cip27-arm-09.patch
    --arch=arm \
```
- ▶ Run the kernel configuration menu.



Linux options for Xenomai configuration

Linux/arm 3.8.13 Kernel Configuration

File Edit Option Help

Option

- General setup
 - IRQ subsystem
 - Timers subsystem
 - CPU/Task time and stats accounting
 - RCU Subsystem
 - Control Group support
 - Group CPU scheduler
 - Namespaces support
 - Configure standard kernel features (e
 - Kernel Performance Events And Counte
 - GCOV-based kernel profiling
 - Enable loadable module support
 - Enable the block layer
 - Partition Types
 - IO Schedulers
- Real-time sub-system
- System Type
 - Multiple platform selection
- Bus support
 - PCCard (PCMCIA/CardBus) support
- Kernel Features
- Boot options
- CPU Power Management
- Floating point emulation
- Userspace binary formats
- Power management options
- Networking support
 - Networking options
 - Network packet filtering framework
 - Core Netfilter Configuration
 - IP set support
 - IP virtual server support
 - IP: Netfilter Configuration
 - IPv6: Netfilter Configuration

Option

Option	Name
Xenomai (NEW)	XENOMAI
Nucleus (NEW)	XENO_OPT_NUCLEUS
Pervasive real-time support in user-space (NEW)	XENO_OPT_PERVASIVE
Priority coupling support (DEPRECATED) (NEW)	XENO_OPT_PRIOCP
Optimize as pipeline head (DEPRECATED) (NEW)	XENO_OPT_PIPELINE_HEAD
Extra scheduling classes (NEW)	XENO_OPT_SCHED_CLASSES
(32) Number of pipe devices (NEW)	XENO_OPT_PIPE_NRDEV
(512) Number of registry slots (NEW)	XENO_OPT_REGISTRY_NRSLOTS
(256) Size of the system heap (Kb) (NEW)	XENO_OPT_SYS_HEAPSZ
(128) Size of the private stack pool (Kb) (NEW)	XENO_OPT_SYS_STACKPOOLSZ
(12) Size of private semaphores heap (Kb) (NEW)	XENO_OPT_SEM_HEAPSZ
(12) Size of global semaphores heap (Kb) (NEW)	XENO_OPT_GLOBAL_SEM_HEAPSZ
Statistics collection (NEW)	XENO_OPT_STATS
Debug support (NEW)	XENO_OPT_DEBUG
Nucleus Debugging support (NEW)	XENO_OPT_DEBUG_NUCLEUS
Spinlocks Debugging support (NEW)	XENO_OPT_DEBUG_XNLOCK
Queue Debugging support (NEW)	XENO_OPT_DEBUG_QUEUEUES
Registry Debugging support (NEW)	XENO_OPT_DEBUG_REGISTRY
Timer Debugging support (NEW)	XENO_OPT_DEBUG_TIMERS
Detect mutexes held in relaxed sections (NEW)	XENO_OPT_DEBUG_SYNCH_RELAX
Watchdog support (NEW)	XENO_OPT_WATCHDOG
(4) Watchdog timeout (NEW)	XENO_OPT_WATCHDOG_TIMEOUT
Shared interrupts (NEW)	XENO_OPT_SHIRQ
Timing	
Scalability	
Machine	
Interfaces	
Drivers	

Xenomai (XENOMAI)

CONFIG_XENOMAI:

Xenomai is a real-time extension to the Linux kernel. Note that Xenomai relies on Adeos interrupt pipeline (CONFIG_IPIPE option) to be enabled, so enabling this option selects the



How to build Xenomai user space

- ▶ User space libraries are compiled using the traditional autotools
 - ▶ `./configure --host=arm-linux && make && make DESTDIR=/your/rootfs/ install`
- ▶ Xenomai installs a *xeno-config* script, which helps you compile your own programs against the Xenomai libraries.
- ▶ See Xenomai's `examples` directory.
- ▶ Installation details may be found at <http://xenomai.org/installing-xenomai-3-x/>
- ▶ Build systems such as Buildroot can compile the Xenomai user space for you.



Developing applications on Xenomai



The POSIX skin

- ▶ The POSIX skin allows to recompile without changes a traditional POSIX application so that instead of using Linux real-time services, it uses Xenomai services
 - ▶ https://www.xenomai.org/index.php/Porting_POSIX_applications_to_Xenomai
 - ▶ Clocks and timers, condition variables, message queues, mutexes, semaphores, shared memory, signals, thread management
 - ▶ Good for existing code or programmers familiar with the POSIX API
- ▶ Of course, if the application uses any Linux service that isn't available in Xenomai, it will switch back to secondary mode
- ▶ To link an application against the POSIX skin

```
CFL='xeno-config --skin=posix --cflags`  
LDF='xeno-config --skin=posix --ldflags`  
ARCH-gcc $CFL -o rttest rttest.c $LDF
```



Communication with a normal task

- ▶ If a Xenomai real-time application using the POSIX skin wishes to communicate with a separate non-real-time application, it must use the rtpc mechanism
- ▶ In the Xenomai application, create an `IPCPROTO_XDDP` socket

```
socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP);
setsockopt(s, SOL_RTIPC, XDDP_SETLOCALPOOL,
           &poolsz, sizeof(poolsz));
memset(&saddr, 0, sizeof(saddr));
saddr.sipc_family = AF_RTIPC;
saddr.sipc_port = PORTX;
ret = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
```

- ▶ And then the normal socket API `sendto()` / `recvfrom()`
- ▶ In the Linux application
 - ▶ Open `/dev/rtpPORTX`, where `PORTX` is the XDDP port
 - ▶ Use `read()` and `write()`



Alchemy: the native API

Xenomai proposes its own API for developing real-time tasks

- ▶ https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy.html
- ▶ Xenomai developers find it a more coherent and more flexible API than the POSIX API, and easier to learn and understand.
- ▶ However, beware that programming with the POSIX API is much more portable (ability to use regular Linux, Linux with PREEMPT_RT, Xenomai and other POSIX operating systems).



Benchmarks: mainline vs PREEMPT_RT vs Xenomai

cyclicttest results on BeagleBone Black (ARM), with different kernel setups.
Wake-up time measured in μ s.

Table: Idle CPU

	min	avg	max
Mainline Linux	39	43	1046
PREEMPT_RT	16	21	52
Xenomai 2	8	9	41
Xenomai 3	7	7	35

Table: Stressed CPU

	min	avg	max
Mainline Linux	39	52	1097
PREEMPT_RT	18	25	62
Xenomai 2	8	19	45
Xenomai 3	6	16	37

http://wiki.csie.ncku.edu.tw/embedded/xenomai/rtlws_paper.pdf



Organizations

- ▶ <https://wiki.linuxfoundation.org/realtime/start>

The Linux Foundation, now funding work on PREEMPT_RT

- ▶ <https://www.osadl.org>

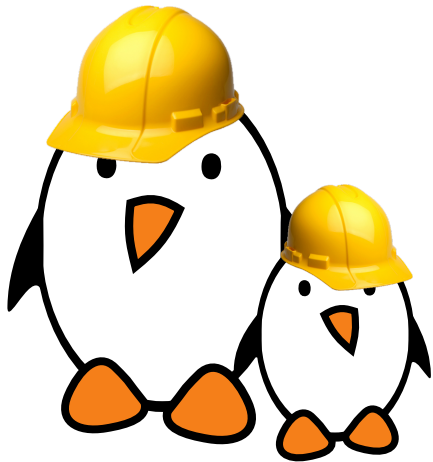
Open Source Automation Development Lab (OSADL)

Targets machine and plant control systems. Most member companies are German (Thomas Gleixner is on board). Supports the use of PREEMPT_RT and Xenomai and contributes to these projects. Shares useful documentation and resources.

They also organize a yearly Real Time Linux Workshop.



Practical lab - Real-time - Scheduling latency

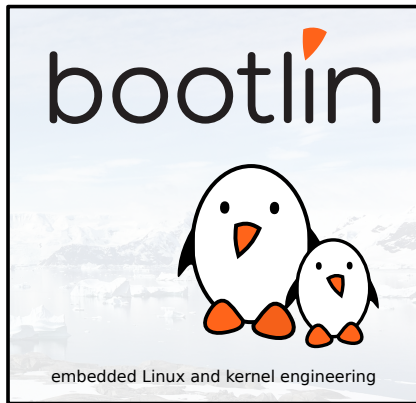


- ▶ Check clock accuracy.
- ▶ Start processes with real-time priority.
- ▶ Build a real-time application against the standard POSIX real-time API, and against Xenomai's POSIX skin
- ▶ Compare scheduling latency on your system, between a standard kernel, a kernel with PREEMPT_RT and a kernel with Xenomai.



References

© Copyright 2004-2021, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





- ▶ **Mastering Embedded Linux, Second Edition, Packt Publishing**

By Chris Simmonds, June 2017

An up-to-date resource covering most aspects of embedded Linux development.

<https://bit.ly/2A9Pb5Y>

- ▶ **The Linux Programming Interface, Oct 2010**

- ▶ Michael Kerrisk, No Starch Press

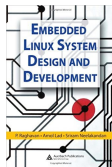
- ▶ <https://man7.org/tlpi/>

- ▶ A gold mine about Linux system programming

- ▶ **Embedded Linux System Design and Development**

P. Raghavan, A. Lad, S. Neelakandan, Auerbach, Dec. 2005. Very good coverage of the POSIX programming API (still up to date).

<https://j.mp/19X8iu2>



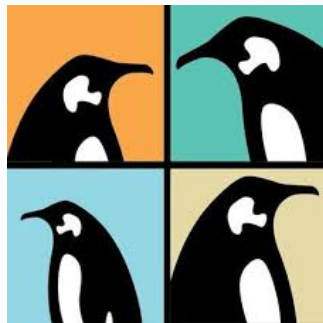


- ▶ **ELinux.org**, <https://elinux.org>, a Wiki entirely dedicated to embedded Linux. Lots of topics covered: real-time, filesystem, multimedia, tools, hardware platforms, etc. Interesting to explore to discover new things.
- ▶ **LWN**, <https://lwn.net>, very interesting news site about Linux in general,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



International conferences (1)

- ▶ Embedded Linux Conference:
 - ▶ <https://embeddedlinuxconference.com/>
 - ▶ Organized by the Linux Foundation every year in North America and in Europe
 - ▶ Very interesting kernel and user space topics for embedded systems developers. Many kernel and embedded project maintainers are present.
 - ▶ Presentation slides and videos freely available on https://elinux.org/ELC_Presentations
- ▶ Linux Plumbers: <https://linuxplumbersconf.org>
 - ▶ About the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc. Not really a conventional conference with formal presentations, but rather a place where contributors on each topic meet, share their progress and make plans for work ahead.





International conferences (2)

- ▶ FOSDEM: <https://fosdem.org> (Brussels, February)
For developers. Presentations about system development.
- ▶ Live Embedded Event: <https://live-embedded-event.com/>
A new free live event about embedded topics. Co-organized by Bootlin!
- ▶ Currently, most conferences are available on-line. They are much more affordable and often free.

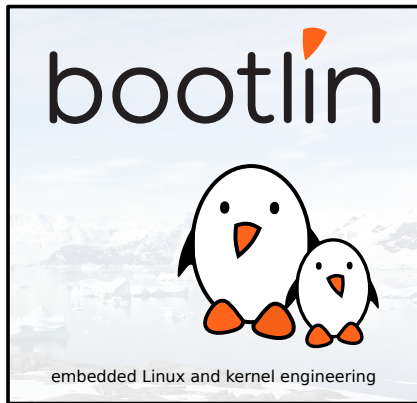


Last slides

© Copyright 2004-2021, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Thank you!
And may the Source be with you