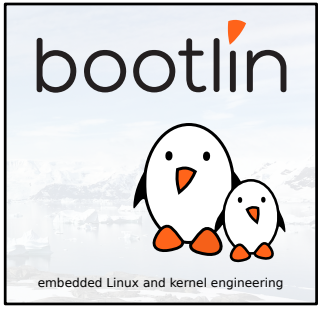
Embedded Linux system development training

Latest update: September 06, 2022.

Document updates and training details:
https://bootlin.com/training/embedded-linux

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

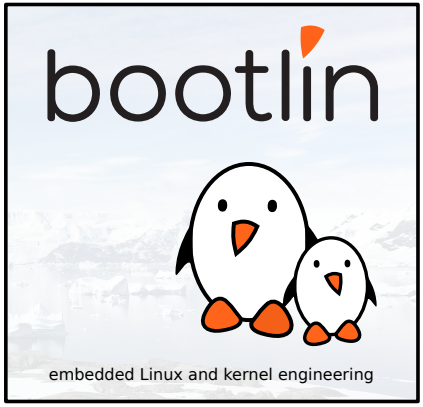


Embedded Linux system development training

- These slides are the training materials for Bootlin's *Embedded Linux system development* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/embedded-linux
- Contact: training@bootlin.com

Icon by Eucalyp, Flaticon

About Bootlin



Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

bootlin

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development

On-site: 4 or 5 days
Online: 7 * 4 hours

Linux kernel driver development

On-site: 5 days
Online: 7 * 4 hours

Yocto Project system development

On-site: 3 days
Online: 4 * 4 hours

Buildroot system development

On-site: 3 days
Online: 5 * 4 hours

Understanding the Linux graphics stack

On-site: 2 days
Online: 4 * 4 hours

Embedded Linux boot time optimization

On-site: 3 days
Online: 4 * 4 hours

Real-Time Linux with PREEMPT_RT

On-site: 2 days
Online: 3 * 4 hours

Bootlin, an open-source contributor

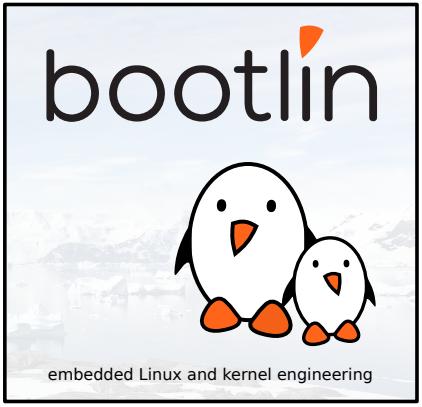
- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 8000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 5000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- Twitter:
  https://twitter.com/bootlincom
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Generic course information



STMicroelectronics STM32MP157D-DK1 Discovery board

- STM32MP157D (Dual Cortex-A7 + Cortex-M4) CPU from STMicroelectronics
- 512 MB DDR3L RAM
- Gigabit Ethernet port
- 4 USB 2.0 host ports
- 1 USB-C OTG port
- 1 Micro SD slot
- On-board ST-LINK/V2-1 debugger
- Misc: buttons, LEDs, Audio codec
- Currently sold at 65 EUR + VAT at Mouser

Board and CPU documentation, design files, software:

https://www.st.com/en/evaluation-tools/stm32mp157d-dk1.html

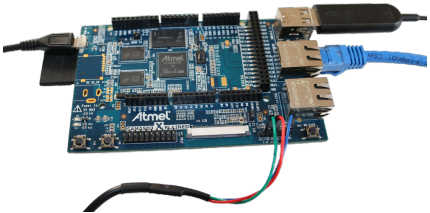
Shopping list: hardware for this course

- STMicroelectronics STM32MP157D-DK1 Discovery kit - Available from Mouser (65 EUR + VAT)
- USB-C cable for the power supply
- USB-A to micro B cable for the serial console
- RJ45 cable for networking
- A standard USB audio headset. We're using Logitech USB H340 [1]
- A micro SD card with at least 128 MB of capacity

[1] https://support.logitech.com/en_us/product/usb-headset-h340

Labs proposed on another platform

After this course, you can also run all labs on the Microchip SAMA5D3 Xplained ARM board.
In addition, you will also have real-time and NAND flash labs!



Lab instructions available on https://bootlin.com/doc/training/embedded-linux/

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/

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

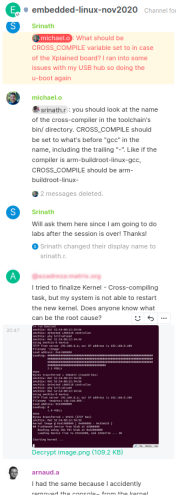
During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please keep your camera on too if you have one.
  - Also make sure your name is properly filled.
  - If Jitsi Meet is used, you can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

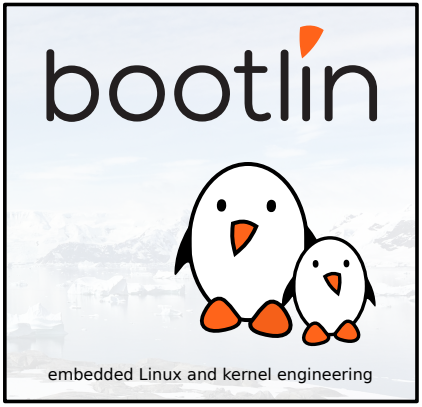
Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.

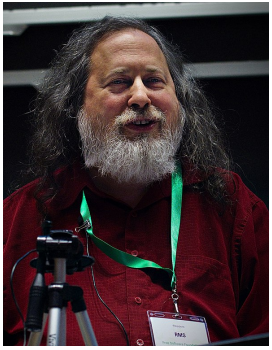


Introduction to Embedded Linux



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!**

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

Advantages of Linux and Open-Source in embedded systems

- **Ability to reuse components**
  Many features, protocols and hardware are supported. Allows to focus on the added value of your product.
- **Low cost**
  No per-unit royalties. Development tools free too. But of course deploying Linux costs time and effort.
- **Full control**
  You decide when to update components in your system. No vendor lock-in. This secures your investment.
- **Easy testing of new features**
  No need to negotiate with third-party vendors. Just explore new solutions released by the community.
- **Quality**
  Your system is built on high-quality foundations (kernel, compiler, C-library, base utilities...). Many Open-Source applications have good quality too.
- **Community support**
  Can get very good support from the community if you approach it with a constructive attitude.
- **Participation in community work**
  Possibility to collaborate with peers and get opportunities beyond corporate barriers.

A few examples of embedded systems running Linux

Wireless routers

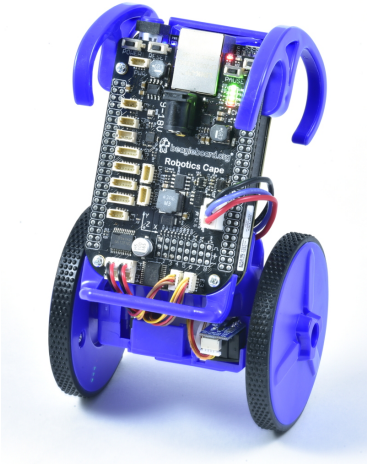
Image credits: Evan Amos (https://bit.ly/2JzDIkv)

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.

Robots



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

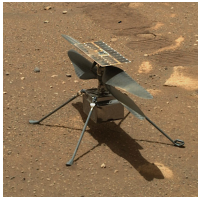
In space

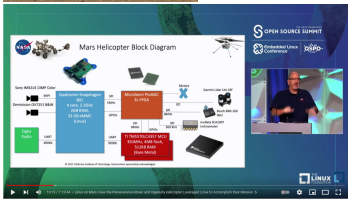
SpaceX Starlink satelites

SpaceX Falcon 9 and Falcon Heavy rockets

Image credits: Wikipedia

Mars Ingenuity Helicopter





See the *Linux on Mars: How the Perseverance Rover and Ingenuity Helicopter Leveraged Linux to Accomplish their Mission* presentation from Tim Canham (JPL, NASA): https://youtu.be/0_GfMcBmbCg?t=111

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 Chip*s (*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.

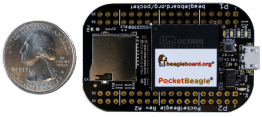
Communication

- 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.

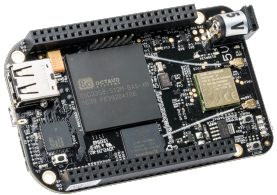
STM32MP157C-EV1 evaluation board
Image credits (st.com):
https://frama.link/NySnaxuV



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 Wireless board

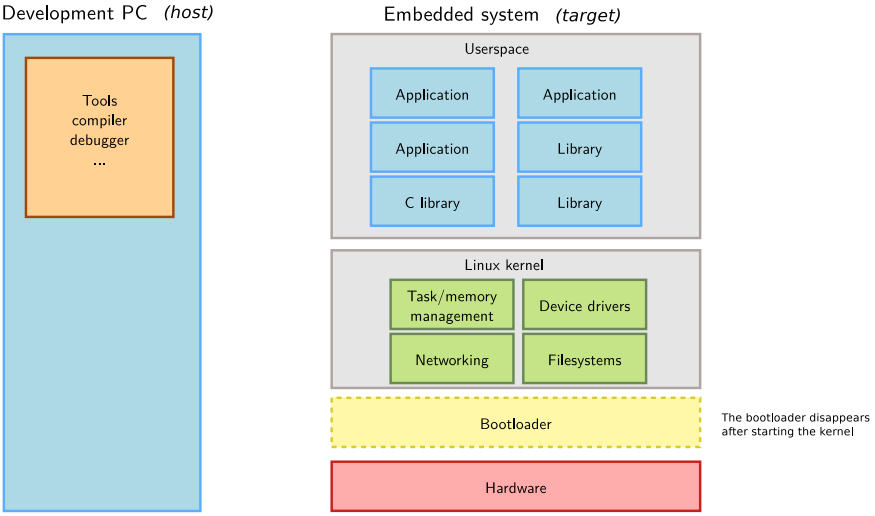
Olimex Open hardware ARM laptop main board
Image credits (Olimex): https://www.olimex.com/Products/DIY-Laptop/

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



Software components

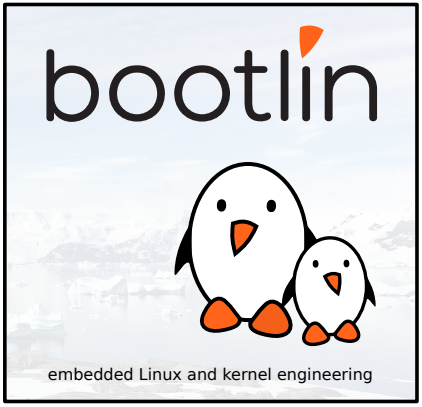
- Cross-compilation toolchain
  - Compiler that runs on the development machine, but generates code for the target
- Bootloader
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- Linux Kernel
  - Contains the process and memory management, network stack,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

Embedded Linux work

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

- **Board Support Package development**
  - A BSP contains a bootloader and kernel with the suitable device drivers for the targeted hardware
  - 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



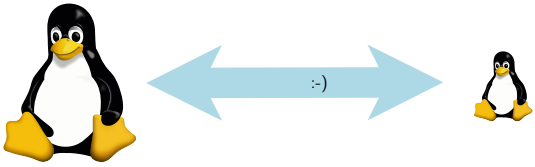
Embedded Linux solutions

- Two ways to switch to embedded Linux
  - Use **solutions provided and supported by vendors** like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
  - Use **community solutions**. They are completely open, supported by the community.
- In 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)



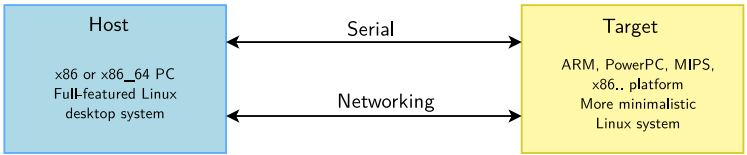
Desktop Linux distribution

- **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, derived from Debian, 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

Host vs. target

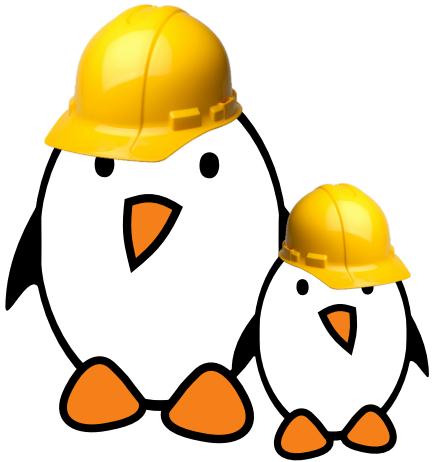
- When doing embedded development, there is always a split between
  - The *host*, the development workstation, which is typically a powerful PC
  - The *target*, which is the embedded system under development
- They are connected by various means: almost always a serial line for debugging purposes, frequently a networking 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
  - `ttyUSBx` for USB to serial converters
  - `ttySx` for real serial ports
- Most frequent command: `picocom -b 115200 /dev/ttyUSB0`

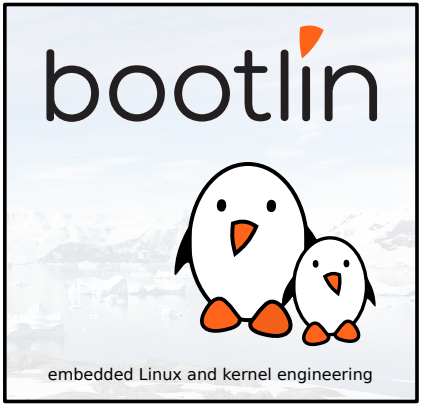
Practical lab - Training Setup



Prepare your lab environment

- Download and extract the lab archive

Cross-compiling toolchains

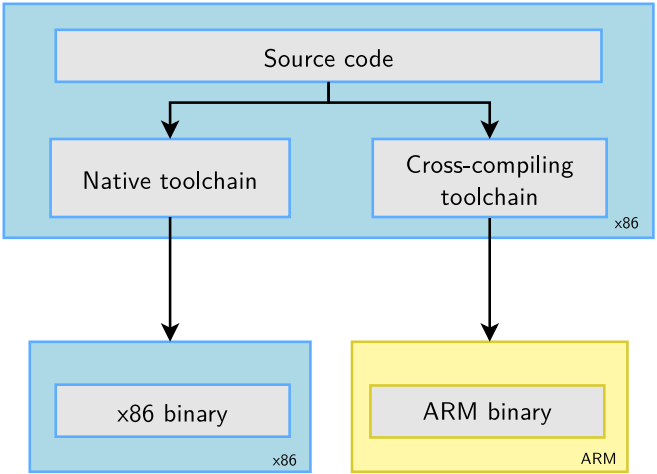


Definition and Components

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)



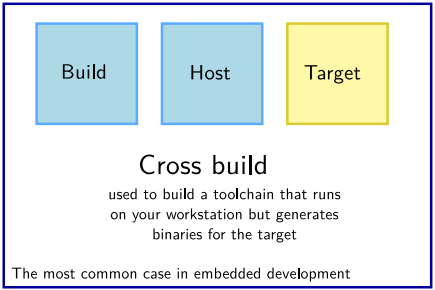
Machines in build procedures

- 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 build

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

The most common case in embedded development

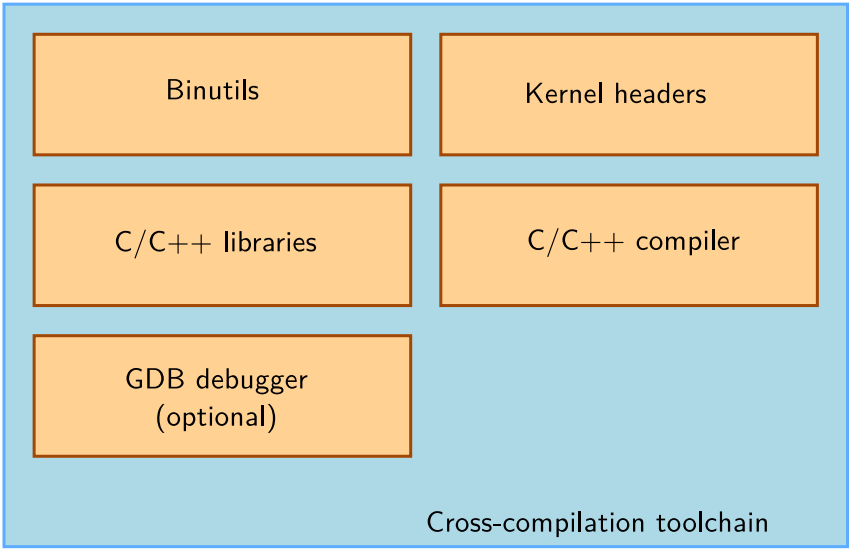
Cross-native build

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

Canadian 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

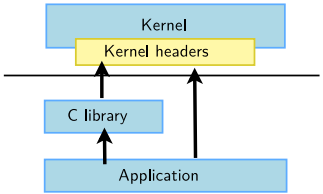


Binutils

- **Binutils** is a set of tools to generate and manipulate binaries (usually with the ELF format) 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 needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in `<linux/...>` and `<asm/...>` and a few other directories corresponding to the ones visible in `include/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.

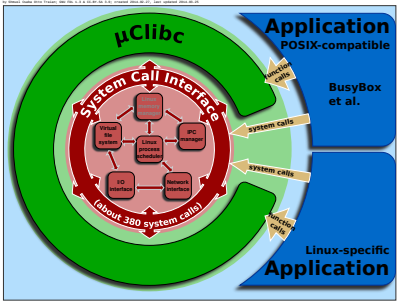
What to remember: updating your kernel shouldn't break your programs; it's usually fine to keep an old toolchain as long is it works fine for your project.

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.



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

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

C Libraries

glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- 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

uClibc-ng

- 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 (RAM and storage) 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.

musl C library

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-eabihf`
- 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, and is best supported by debugging tools (*ltrace* not supported by *musl* in Buildroot, for example).
- 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

ABI

- When building a toolchain, the ABI used to generate binaries needs to be defined
- ABI, for *Application Binary Interface*, defines the calling conventions (how function arguments are passed, how the return value is passed, how system calls are made) and the organization of structures (alignment, etc.)
- All binaries in a system 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 RISC-V, several ABIs: *ilp32*, *ilp32f*, *ilp32d*, *lp64*, *lp64f*, and *lp64d*
- 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

- GNU tools (gcc, binutils) can only be compiled for a specific target architecture at a time (ARM, x86, RiscV...)
- 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 GNU 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.
- Note: LLVM (clang, lld...) utilities support multiple target architectures at once.

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-gnueabihf`
  - Bootlin's GNU toolchains (for most architectures):
    https://toolchains.bootlin.com
  - ARM GNU toolchains released by ARM (previously shipped by Linaro):
    https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

Toolchain building utilities

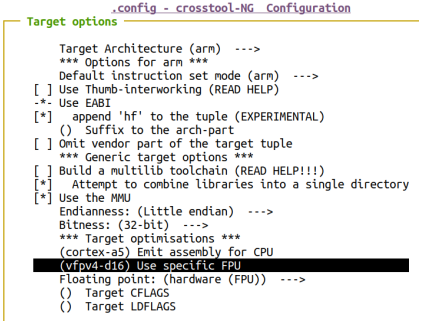
Another solution is to use utilities that **automate the process of building the toolchain**

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

Toolchain building utilities (2)

Crosstool-ng

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



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
  - https://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
```

- 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)

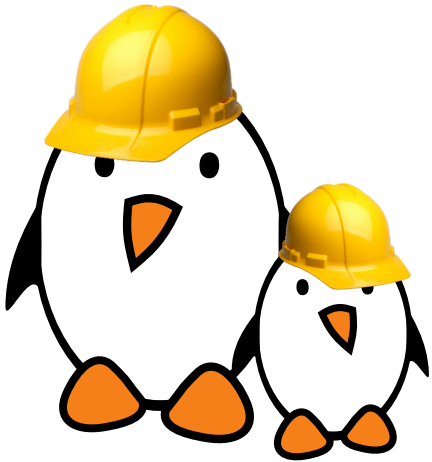
- To build the toolchain

```
./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)

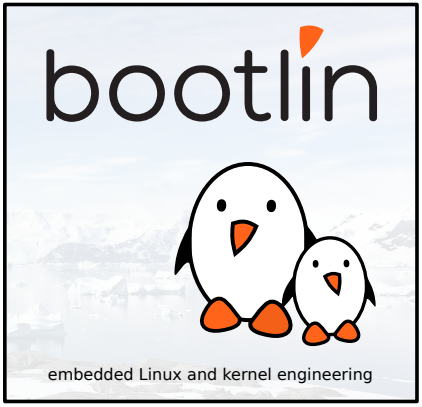
Practical lab - Using Crosstool-NG



Time to build your toolchain

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

Bootloaders



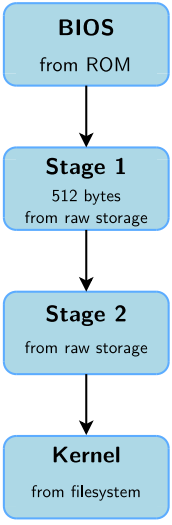
Boot Sequence

Bootloaders

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

Bootloaders on 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.

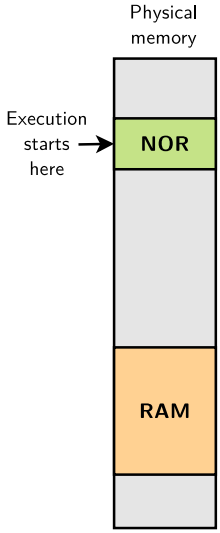


Bootloaders on x86 (2)

- *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.
- *Syslinux*, for network and removable media booting (USB key, CD-ROM)
  https://kernel.org/pub/linux/utils/boot/syslinux/
- *Systemd-boot*, a very simple UEFI boot manager (formerly *Gummiboot*)
  - Of course, not based on *Systemd*, but hosted by this project.

Booting on embedded CPUs: case 1

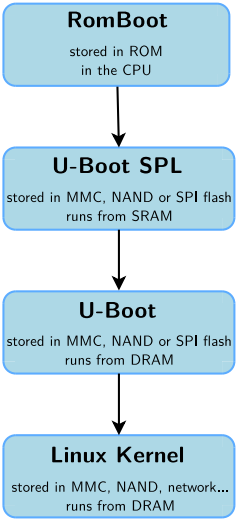
- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows direct access from the CPU (just like RAM), which NAND doesn't allow (external storage that needs to be copied to RAM before executing).
- **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 (usually open-source).
- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into DRAM

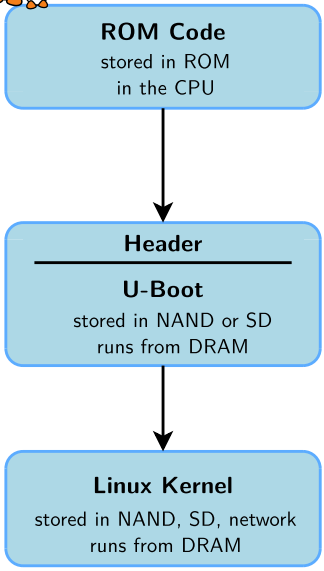
Booting on Microchip ARM SAMA5D3



- **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 DRAM and starts it. No user interaction possible.
- **U-Boot**: runs from DRAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to DRAM and starts it. Shell with commands provided.
- **Linux Kernel**: runs from DRAM. 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

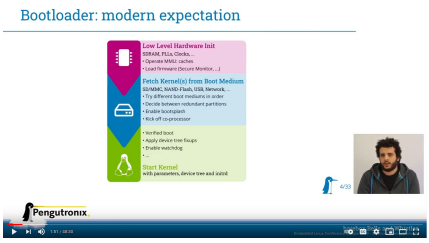


- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into DRAM. The DRAM configuration is described in a CPU-specific header, prepended to the bootloader image.
- **U-Boot**: runs from DRAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to DRAM and starts it. Shell with commands provided. File called `u-boot.kwb`.
- **Linux Kernel**: runs from DRAM. 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



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

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://u-boot.readthedocs.io/en/latest/
- 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.

Image credits:
https://frama.link/rwCUFc-T

U-Boot configuration

- Get the source code from the website or from git
- 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 overriden 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 should see U-Boot starting up.
- 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.

```
U-Boot SPL 2022.01 (Mar 31 2022 - 14:58:17 +0200)
Trying to boot from MMC1


U-Boot 2022.01 (Mar 31 2022 - 14:58:17 +0200)

CPU  : AM335X-GP rev 2.1
Model: TI AM335x BeagleBone Black
DRAM:  512 MiB
WDT:   Started wdt@44e35000 with servicing (60s timeout)
NAND:  0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Loading Environment from FAT... OK
Net:   Could not get PHY for ethernet@4a100000: addr 0
eth2: ethernet@4a100000, eth3: usb_ether [PRIME]
Hit any key to stop autoboot:  0
=>
```

Information commands

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
```

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
```

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`...
- Similar commands for other filesystems: `ext2load`, `ext2ls`, `ext4load`, `ext4ls`, `sqfsload`, `sqfsls`... (SquashFS support contributed by Bootlin)
- Note that filesystem independent commands such as `load`, `ls`, and `size` exist. Examples:
  - `load usb 0:1 0x21000000 zImage`
  - `ls mmc 0:2 boot/`
  - `size mmc 0:1 dtb` (result stored in `filesize` environment variable)
- `loadb`, `loads`, `loady`, load a file from the serial line to RAM
- `tftp`, loads a file from the network to RAM (example given later)

Important commands (2)

- `ping`, to test the network
- `bootd` (can be abbreviated as `boot`), runs the default boot command, stored in the `bootcmd` environment variable (explained later)
- `bootz <address>`, starts a compressed kernel image loaded at the given address in 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
- `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 bdinfo command

```
=> bdinfo
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 environemt 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

```
=> 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
=> setenv serverip 10.0.0.100
=> printenv serverip
serverip=10.0.0.100
=> 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'`

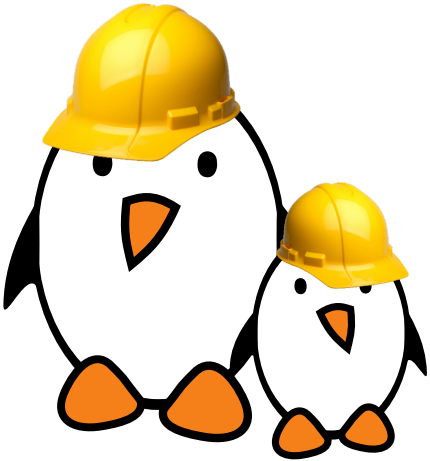
Transferring files to the target

- 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)

TFTP

- Network transfer from the development workstation to U-Boot on the target takes place through TFTP
  - *Trivial File Transfer Protocol*
  - Somewhat similar to FTP, but without authentication and over UDP
- A TFTP server is needed on the development workstation
  - `sudo apt 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`

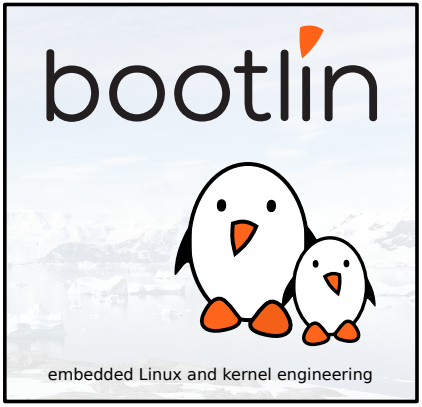
Practical lab - U-Boot



Time to start the practical lab!

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

Linux kernel introduction



Linux kernel features

History

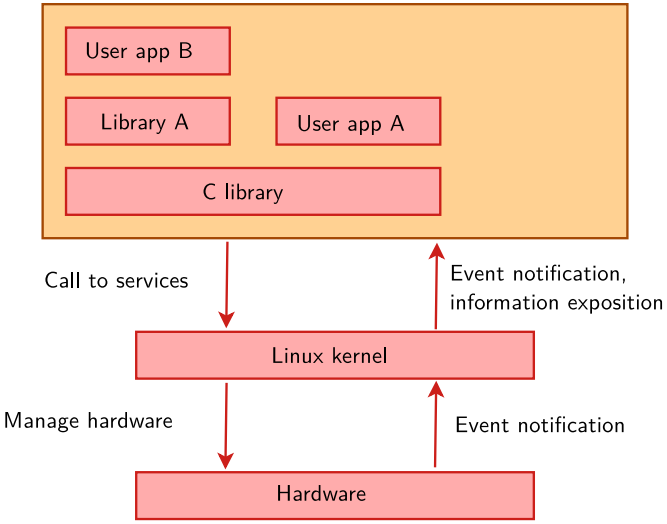
- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- As of 2022, about 2,000 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

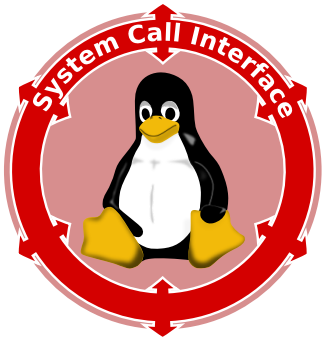
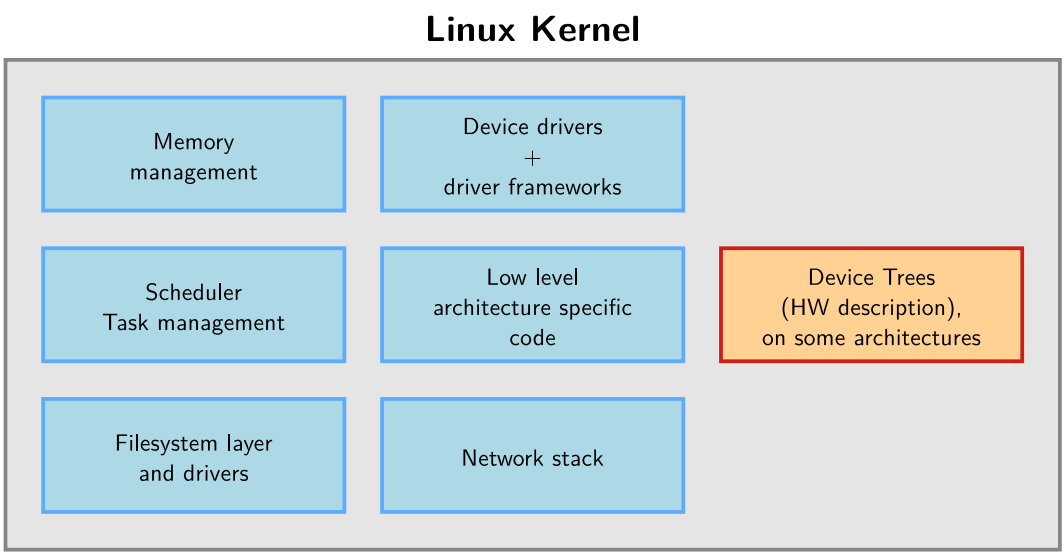


Image credits (Wikipedia): https://bit.ly/2U2rdGB

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
Memory management
Device drivers + driver frameworks
Scheduler Task management
Low level architecture specific code
Device Trees (HW description), on some architectures
Filesystem layer and drivers
Network stack

Linux license

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

Supported hardware architectures

See the arch/ directory in the kernel sources

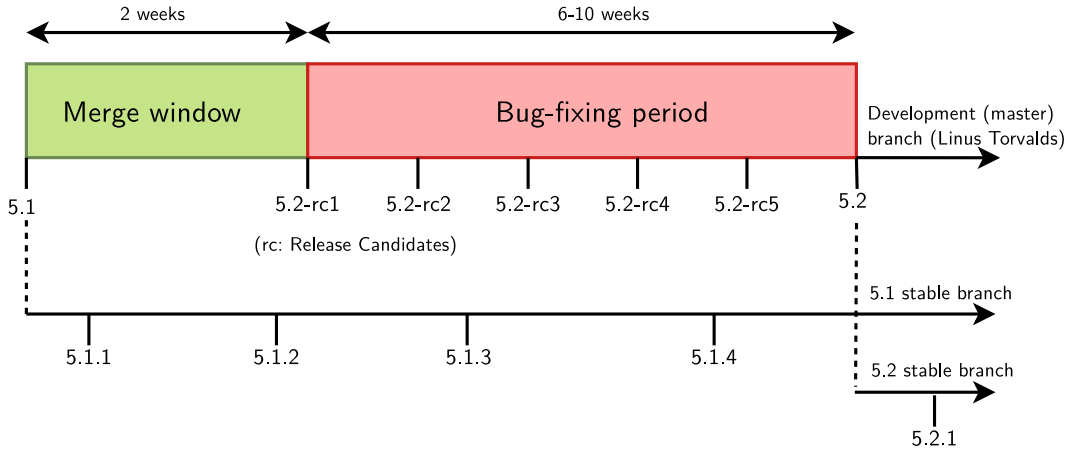
- Minimum: 32 bit processors, with or without MMU, supported by gcc or clang
- 32 bit architectures (arch/ subdirectories)
  Examples: arm, arc, m68k, microblaze (soft core on FPGA)...
- 64 bit architectures:
  Examples: alpha, arm64, ia64...
- 32/64 bit architectures
  Examples: mips, powerpc, riscv, 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 “stabilized” release branch of Linux every 2 or 3 years (2.0, 2.2, 2.4). Development branches took 2-3 years to be merged (too slow!).
- Since 2003, there is a new official 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 called stable releases: 5.0.1, 5.0.2, etc.

Using merge and bug fixing windows



Need for long term support (1)

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

Version	Maintainer	Released	Projected EOL
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Oct, 2023
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 Nov. 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 much longer (at least 10 years) selected LTS versions (currently 4.4, 4.19, 5.10) on selected architectures. See https://wiki.linuxfoundation.org/civilinfrastructureplatform/cipkernelmaintenance.

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 official kernel sources

- The mainline versions of the Linux kernel, as released by Torvalds
  - These versions follow the development model of the kernel
  - They may not contain the latest developments from a specific area yet
  - A good pick for products development phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- The stable versions of the Linux kernel, as maintained by a maintainers group
  - These versions do not bring new features compared to Linus' tree
  - Only bug fixes and security fixes are pulled there
  - Each version is stabilized during the development period of the next mainline kernel
  - Certain versions can be maintained for much longer, 2+ years
  - A good pick for products commercialization phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

Location of non-official kernel sources

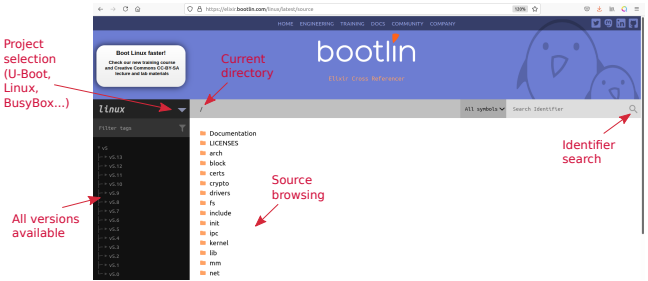
- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Sometimes they break support for other platforms/devices without caring
  - Useful in early phases only when mainline hasn't caught up yet (many vendors invest in the mainline kernel at the same time)
  - Suitable for PoC, not suitable for products on the long term as usually no updates are provided to these kernels
  - Getting stuck with a deprecated system with broken software that cannot be udpated has a real cost in the end
- Many kernel sub-communities maintain their own kernel, with usually newer but fewer stable features, only for cutting-edge development
  - Architecture communities (ARM, MIPS, PowerPC, etc)
  - Device drivers communities (I2C, SPI, USB, PCI, network, etc)
  - Other communities (real-time, etc)
  - Not suitable to be used in products

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).
- But today the entire open source community as settled in favor of Git
  - Fast, efficient with huge code bases, reliable, open source
  - Incidentally written by Torvalds

Going through Linux sources

- Development tools:
  - Any text editor will work
  - Vim and Emacs support ctags and cscope and therefore can help with symbol lookup and auto-completion.
  - It's also possible to use more elaborate IDEs to develop kernel code, like Visual Studio Code.
- Powerful web browsing: Elixir
  - Generic source indexing tool and code browser for C and C++.
  - Very easy to find symbols declaration/implementation/usage
  - Try out https://elixir.bootlin.com!



Linux kernel size

- Linux v5.18 sources:
  - 75,878 files (`git ls-files | wc -l`)
  - 33,242,942 lines (`git ls-files | xargs cat | wc -l`)
  - 1,154,591,060 bytes (`git ls-files | xargs cat | wc -c`)
- But a compressed Linux kernel just sizes a few megabytes.
- So, 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 sources structure

As of kernel version v5.18 (in percentage of total number of lines).

Source code:

- drivers/: 61.1%
- arch/: 11.6%
- fs/: 4.4%
- sound/: 4.1%
- tools/: 3.9%
- net/: 3.7%
- include/: 3.5%
- kernel/: 1.3%
- lib/: 0.7%
- usr/: 0.6%
- mm/: 0.5%
- scripts/: 0.4%
- security/: 0.3%
- crypto/: 0.3%
- block/: 0.2%
- samples/: 0.1%
- ipc/: 0.0%
- virt/: 0.0%
- init/: 0.0%
- certs/: 0.0%

Doc and bindings:

- Documentation/: 3.4%

Build system files:

- Kbuild
- Kconfig
- Makefile

Other files:

- COPYING
- CREDITS
- MAINTAINTERS
- README

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:

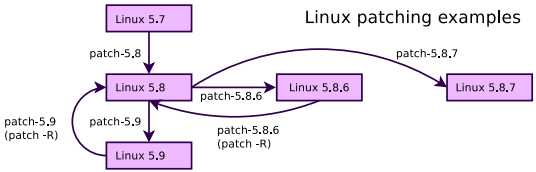
- Takes the patch contents on its standard input
- Applies the modifications described by the patch into the current directory

`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 (`-p`: *prune*) in the file paths
  - You can reverse apply a patch with the `-R` option
  - You can test a patch with `--dry-run` option

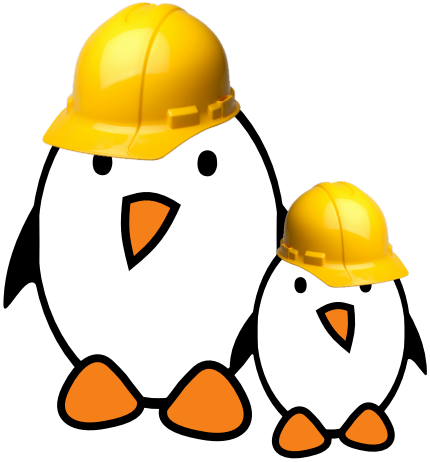
Applying a Linux patch

- 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
```

Practical lab - Kernel sources



Time to start the practical lab!

- Get the Linux kernel sources
- Apply patches

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/`
  - `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.

Choosing 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.

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: `cp /boot/config-`uname -r` .config`
- Embedded platform case:
  - Default configurations stored in-tree as minimal configuration files (only listing settings that are different with the defaults) in `arch/<arch>/configs/`
  - `make help` will list the available configurations for your platform
  - To load a default configuration file, just run `make foo_defconfig` (will erase your current `.config`!)
    - On ARM 32-bit, there is usually one default configuration per CPU family
    - On ARM 64-bit, there is only one big default configuration to customize

Create your own default configuration

- Use a tool such as `make menuconfig` to make changes to the configuration
- Saving your changes will overwrite your `.config` (not tracked by Git)
- When happy with it, create your own default configuration file:
  - Create a minimal configuration (non-default settings) file:
    `make savedefconfig`
  - Save this default configuration in the right directory:
    `mv defconfig arch/<arch>/configs/myown_defconfig`
- This way, you can share a reference configuration inside the kernel sources and other developers can now get the same `.config` as you by running
  `make myown_defconfig`

Built-in 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 built-in 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

Enabling a network driver requires the network stack to be enabled, therefore configuration symbols have two ways to express dependencies:

- `depends on` dependency:

```
config B
    depends on A
```

  - B is not visible until A is enabled
  - Works well for dependency chains

- `select` dependency:

```
config A
    select B
```

  - When A is enabled, B is enabled too (and cannot be disabled manually)
  - Should preferably not select symbols with `depends on` dependencies
  - Used to declare hardware features or select libraries

```
config SPI_ATH79
        tristate "Atheros AR71XX/AR724X/AR913X SPI controller driver"
        depends on ATH79 || COMPILE_TEST
        select SPI_BITBANG
        help
          This enables support for the SPI controller present on the
          Atheros AR71XX/AR724X/AR913X SoCs.
```

Kernel configuration details

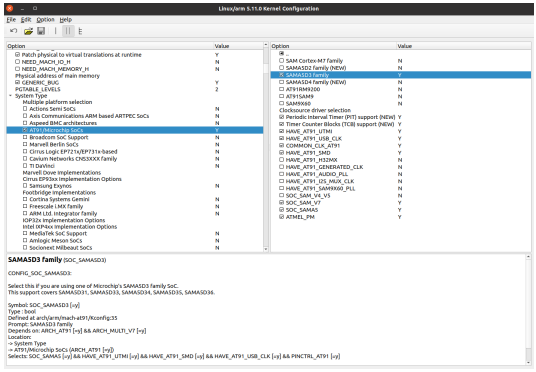
- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `CONFIG_PARAM=value`
  - Options are grouped by sections and are prefixed with `CONFIG_`
  - Included by the top-level kernel Makefile
  - Typically not edited by hand because of the dependencies

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

#
# DOS/FAT/EXFAT/NT Filesystems
#
CONFIG_FAT_FS=y
CONFIG_MSDOS_FS=y
# CONFIG_VFAT_FS is not set
CONFIG_FAT_DEFAULT_CODEPAGE=437
# CONFIG_EXFAT_FS is not set
```

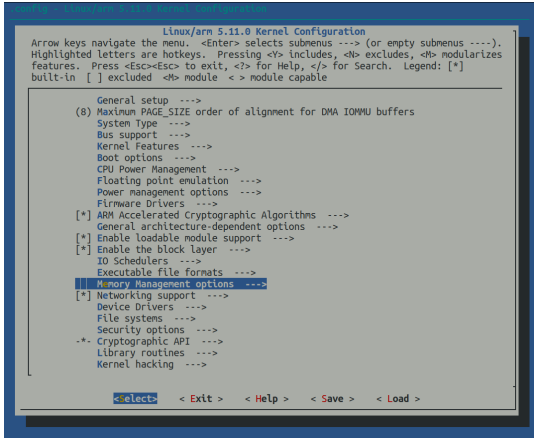
`make xconfig`

- The most common graphical interface to configure the kernel.
- File browser: easy to load configuration files
- Search interface to look for parameters (`[Ctrl] + [f]`)
- Required Debian/Ubuntu packages: `qt5-default` (`qtbase5-dev` on Ubuntu 22.04)



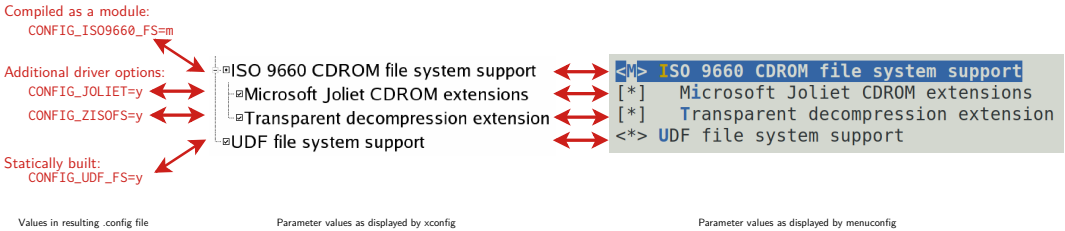
`make menuconfig`

- 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/Ubuntu packages: `libncurses-dev`



Kernel configuration options

You can switch from one tool to another, they all load/save the same `.config` file, and show the same set of options



make oldconfig

`make oldconfig`

- 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 tools keep this `.config.old` backup copy.

Compiling and installing the kernel

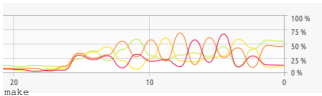
Kernel compilation

`make`

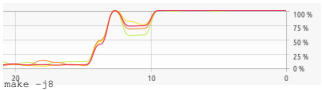
- Only works from the top kernel source directory
- Should not be performed as a priviledged user
- Run several jobs in parallel. Our advice: `ncpus * 2` to fully load the CPU and I/Os at all times.
  Example: `make -j 8`
- To **re**compile faster (7x according to some benchmarks), use the `ccache` compiler cache:
  `export CROSS_COMPILE="ccache arm-linux-"`

Benefits of parallel compile jobs (make -j<n>)

Tests on Linux 5.11 on arm
`make allnoconfig` configuration
`gnome-system-monitor` showing the load on 4 threads / 2 CPUs



Command: make
Total time: 129 s



Command: make -j8
Total time: 67 s

Kernel compilation results

- `arch/<arch>/boot/Image`, uncompressed kernel image that can be booted
- `arch/<arch>/boot/*Image*`, compressed kernel images that can also 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 Blobs
- All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.
- `vmlinux`, a raw uncompressed kernel image in the ELF format, useful for debugging purposes but generally not used for booting purposes

Kernel installation: native case

- `sudo make install`
  - Does the installation for the host system by default
- 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

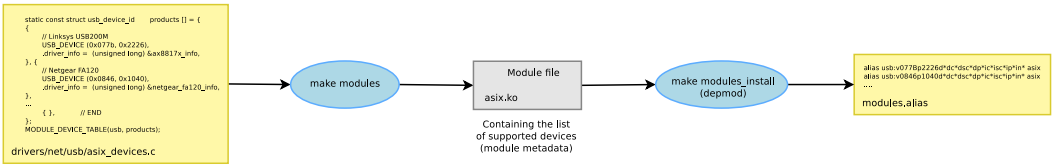
- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle.
- Another reason is that there is no standard way to deploy and use the kernel image.
- Therefore making the kernel image available to the target is usually manual or done through scripts in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

Module installation: native case

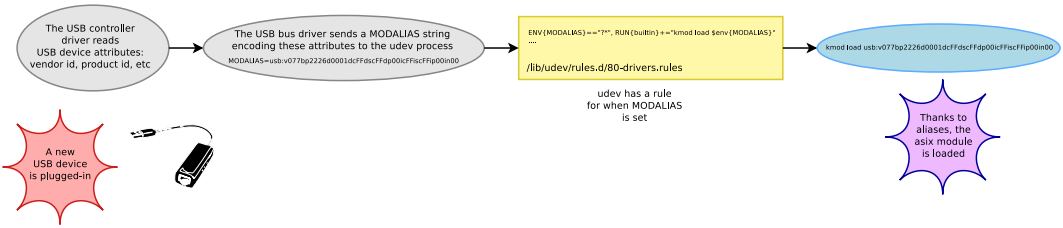
- `sudo make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs all modules in `/lib/modules/<version>/`
  - `kernel/`
    Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias`, `modules.alias.bin`
    Aliases for module loading utilities. Further explanations on the next slide.
  - `modules.dep`, `modules.dep.bin`
    Module dependencies
  - `modules.symbols`, `modules.symbols.bin`
    Tells which module a given symbol belongs to (related to module dependencies).
  - `modules.builtin`
    List of modules that are builtin the kernel.

Automatic module loading with module aliases

Kernel compiling



System operation



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 (no need to be root):
  `make INSTALL_MOD_PATH=<dir>/ modules_install`

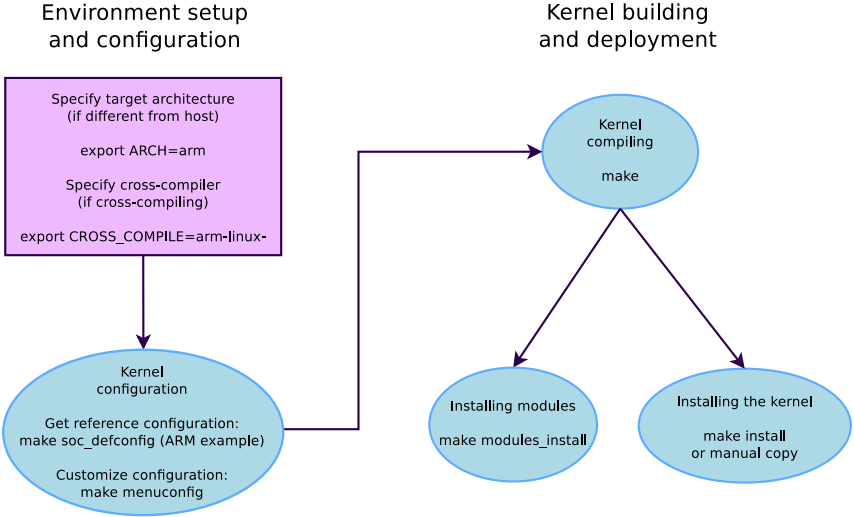
Kernel cleanup targets

- From `make help`:

```
Cleaning targets:
  clean           - Remove most generated files but keep the config and
                    enough build support to build external modules
  mrproper        - Remove all generated files + config + various backup files
  distclean       - mrproper + remove editor backup and patch files
```

- If you are in a git tree, remove all files not tracked (and ignored) by git: `git clean -fdx`

Kernel building overview



Booting the kernel

Device Tree 1/2

- 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 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.
- Its main purpose is to describe the hardware and its integration: non-discoverable devices, clocks, interrupts, DMA channels, pin muxing, etc.

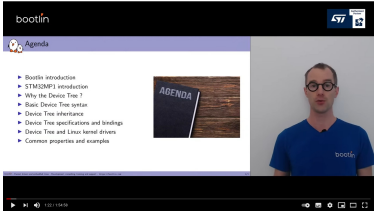
Device Tree 2/2

- A *Device Tree Source (DTS)*, written by kernel developers, is compiled into a binary *Device Tree Blob (DTB)*, 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 DTB in memory before starting the kernel.
- This way, a kernel supporting different SoCs knows which SoC and device initialization hooks to run on the current board.

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/a9CZ1Uk3OYQ



Booting with U-Boot

- U-Boot can directly boot the `zImage` binary.
- In addition to the kernel image, U-Boot should also pass a DTB to the kernel.
- The typical boot process is therefore:
  1. Load `zImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `bootz X - Y`
     The `-` in the middle indicates no *initramfs*

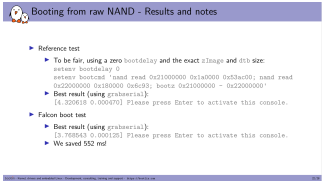
Kernel command line

- 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.

Passing the kernel command line

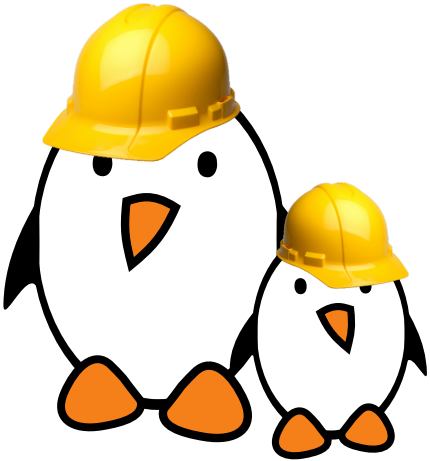
- U-Boot carries the Linux kernel command line string in its `bootargs` environment variable
- Right before starting the kernel, it will store the content of `bootargs` in the `chosen` section of the Device Tree
- The kernel will behave differently depending on its configuration:
  - If `CONFIG_CMDLINE_FROM_BOOTLOADER` is set:
    The kernel will use only the string from the bootloader
  - If `CONFIG_CMDLINE_FORCE` is set:
    The kernel will only use the string received at configuration time in `CONFIG_CMDLINE`
  - If `CONFIG_CMDLINE_EXTEND` is set:
    The kernel will concatenate both strings

See the "Understanding U-Boot Falcon Mode" presentation from Michael Opdenacker, for details about how U-Boot boots Linux.



Slides: https://bootlin.com/pub/conferences/2021/lee/
Video: https://www.youtube.com/watch?v=LFe3x2QMhSo

Practical lab - Kernel cross-compiling



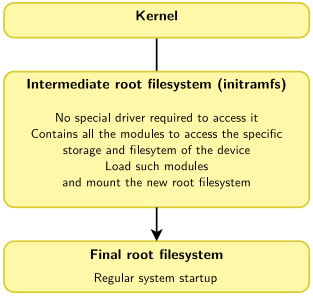
- Set up the cross-compiling environment
- Configure and cross-compile the kernel for an `arm` platform
- On this platform, interact with the bootloader and boot your kernel

Using kernel modules

Advantages of modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the `root` user can load and unload modules.
- 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**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel 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. That's useful when your device's serial console is being monitored for critical messages:
  `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 <top_module_name>`
  Most common usage of `modprobe`: tries to load all the dependencies of the given top module, 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 <top_module_name>`
  Tries to remove the given top module and all its no longer needed dependencies

Passing parameters to modules

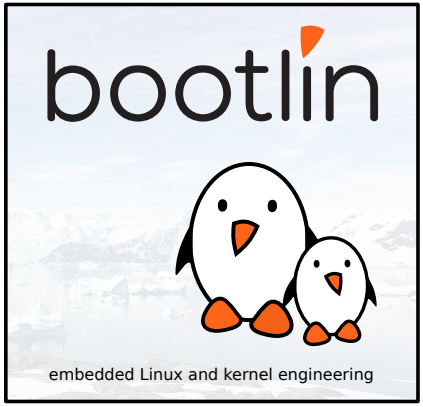
- Find available parameters:
  `modinfo usb-storage`
- Through `insmod`:
  `sudo insmod ./usb-storage.ko delay_use=0`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options usb-storage delay_use=0`
- Through the kernel command line, when the driver 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 Root Filesystem



Principle and solutions

Filesystems

- Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- In UNIX systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- Filesystems are **mounted** in a specific location in this hierarchy of directories
  - When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflect the contents of this filesystem.
  - 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=`

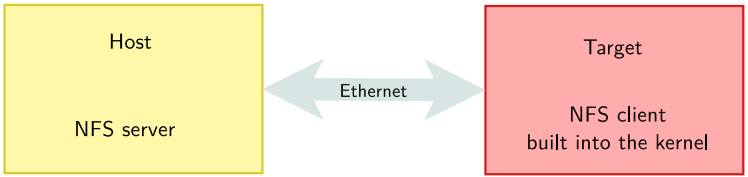
Mounting rootfs from storage devices

- 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 enumerated 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)

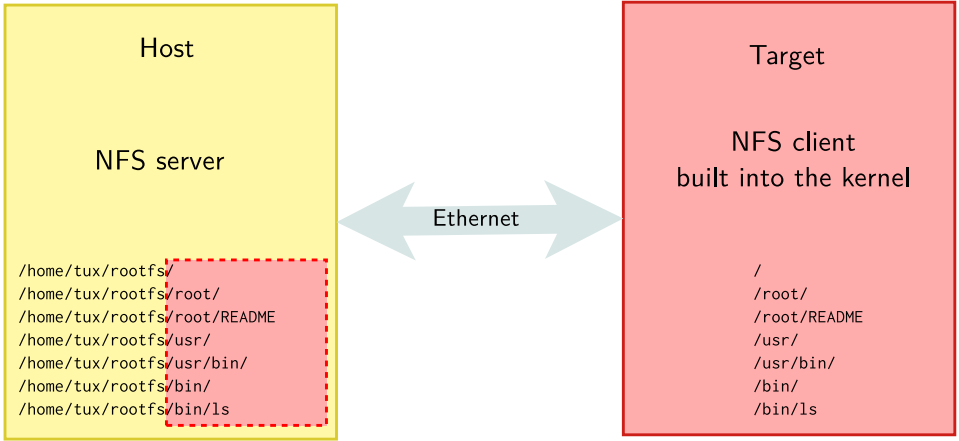
On the development workstation side, a NFS server is needed

- 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.
- Ask your NFS server to reload this file:
  `sudo exportfs -r`

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 documentation): `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
```

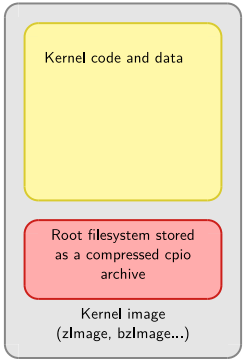
Built-in initramfs

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`

WARNING: only binaries from GPLv2 compatible code are allowed to be included in the kernel binary using this technique. Otherwise, use an external initramfs.



Contents

Root filesystem organization

- The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- https://refspecs.linuxfoundation.org/fhs.shtml
- 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 point for a temporarily mounted filesystem
/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, for system services. 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`
- Update: distributions are now making `/bin` link to `/usr/bin`, `/lib` to `/usr/lib` and `/sbin` to `/usr/sbin`. Details on https://www.freedesktop.org/wiki/Software/systemd/TheCaseForTheUsrMerge/.

Device Files

Devices

- 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`
- See `filesystems/proc` in kernel documentation or `man proc`

proc contents

- One directory for each running process in the system
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - It contains details about the files opened by the process, the CPU and memory usage, etc.
- `/proc/interrupts`, `/proc/devices`, `/proc/iomem`, 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 documentation.
  - Example (free the page cache and slab objects):
    `echo 3 > /proc/sys/vm/drop_caches`

sysfs filesystem

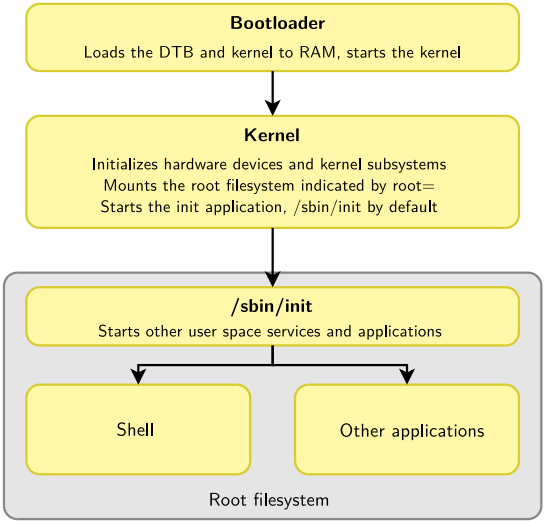
- It allows to represent in user space the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various user space applications that need to list and query the available hardware, for example `udev` or `mdev`.
- All applications using sysfs expect it to be mounted 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

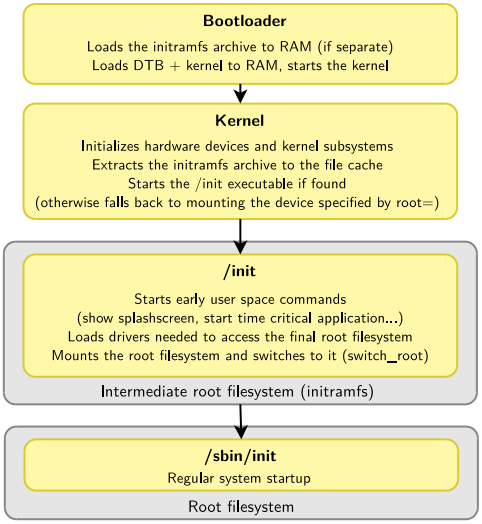
Basic applications

- 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, and for acting as a universal parent for processes which parent terminated before they do.
- 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

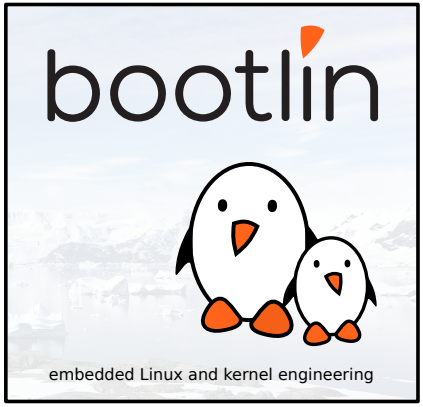
Overall booting process



Overall booting process with initramfs



BusyBox



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 - Most commands in one binary

[, [[, acpid, add-shell, addgroup, adduser, adjtimex, arch, arp, arping, ash, awk, base64, basename, bc, beep, blkdiscard, blkid, blockdev, bootchartd, brctl, bunzip2, bzcat, bzip2, cal, cat, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, conspy, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand, expr, factor, fakeidentd, fallocate, false, fatattr, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole, fgrep, find, findfs, flock, fold, free, freeramdisk, fsck, fsck.minix, fsfreeze, fstrim, fsync, ftpd, ftpget, ftpput, fuser, getopt, getty, grep, groups, gunzip, gzip, halt, hd, hdparm, head, hexdump, hexedit, hostid, hostname, httpd, hush, hwclock, i2cdetect, i2cdump, i2cget, i2cset, i2ctransfer, 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, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lsof, lspci, lsscsi, lsusb, lzcat, lzma, lzop, makedevs, makemime, man, md5sum, mdev, mesg, microcom, mim, mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.vfat, mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount, mountpoint, mpstat, mt, mv, nameif, nanddump, nandwrite, nbd-client, nc, 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, powertop, printenv, printf, ps, pscan, pstree, pwd, pwdx, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, remove-shell, renice, reset, resize, resume, rev, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-init, run-parts, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole, setfattr, setfont, setkeycodes, setlogcons, setpriv, 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, tunctl, ubiattach, ubidetach, ubimkvol, ubirename, ubirmvol, ubirsvol, ubiupdatevol, udhcpc, udhcpc6, udhcpd, udpsvd, uevent, umount, uname, unexpand, uniq, unix2dos, unlink, unlzma, unshare, 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` - July 2021 status

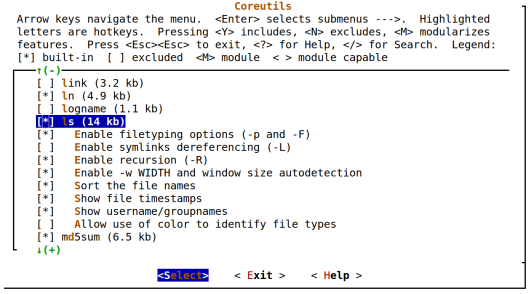
Configuring BusyBox

- 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!



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

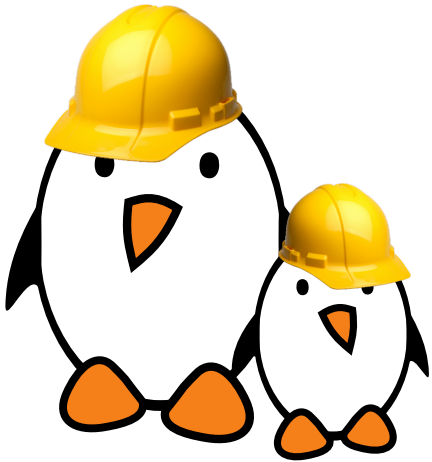
- BusyBox provides an implementation of an `init` program
- Simpler than the init implementation found on desktop/server systems (*SysV init* or *systemd*)
- A single configuration file: `/etc/inittab`
  - Each line has the form `<id>::<action>:<process>`
- Allows to start system services at startup, to control system shutdown, 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.

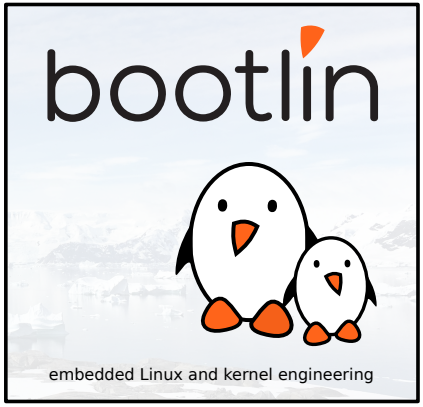
```
[ ] cmp (4.9 kb)
[ ] diff (13 kb)
[ ] ed (21 kb)
[ ] patch (9.4 kb)
[ ] sed (12 kb)
[*] vi (23 kb)
(4096) Maximum screen width
[*]   Allow to display 8-bit chars (otherwise shows dots)
[*]   Enable ":" colon commands (no "ex" mode)
[*]   Enable yank/put commands and mark cmds
[*]   Enable search 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



Block devices

Block vs. raw flash

- Storage devices are classified in two main types: **block devices** and **raw 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, now used by all modern operating systems, supporting disks bigger than 2 TB.
- Numerous tools to create and modify the partitions on a block device: `fdisk`, `cfdisk`, `sfdisk`, `parted`, etc.

Transfering 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 (which is not a filesystem image) 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/
    ```

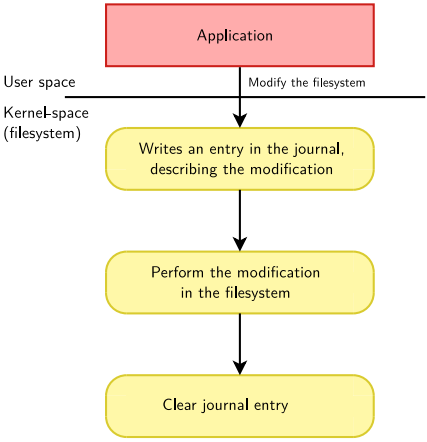
Available block filesystems

Standard Linux filesystem format: ext2, ext3, ext4

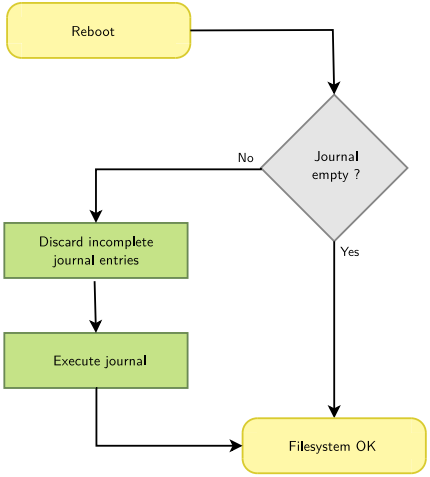
- The standard filesystem used on Linux systems is the series of `ext{2,3,4}` filesystems
  - `ext2` (`CONFIG_EXT2_FS`)
  - `ext3`, brought *journaling* (explained next slide) compared to `ext2`, now obsoleted by `ext4`.
  - `ext4` (`CONFIG_EXT4_FS`), mainly brought performance improvements and support for very big partitions.
- It supports all features Linux needs in a root filesystem: permissions, ownership, device files, symbolic links, etc.

Journaled 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)



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` (`CONFIG_BTRFS_FS`), the most actively developed filesystem for Linux. It integrates numerous features: data checksuming, integrated volume management, snapshots, etc.
- `XFS` (`CONFIG_XFS_FS`), high-performance filesystem inherited from SGI IRIX, still actively developed.
- `JFS` (`CONFIG_JFS_FS`), inherited from IBM AIX. No longer actively developed, provided mainly for compatibility.
- `reiserFS` (`CONFIG_REISERFS_FS`), 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: Flash-Friendly Filesystem

`CONFIG_F2FS_FS`, 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/
- Not as widely used as `ext4` and `btrfs`, even on flash-based storage.

Read-only filesystems

SquashFS: CONFIG_SQUASHFS

- Read-only, compressed filesystem for block devices. Fine for parts of a filesystem which can be read-only (kernel, binaries...)
- Used in most live CDs and live USB distributions
- Supports several compression algorithms (LZO, XZ, etc.)
- Gives priority to compression ratio vs read performance

EROFS: CONFIG_EROFS_FS

- https://en.wikipedia.org/wiki/EROFS
- Gives priority to read performance vs compression

See a comparison at https://blog.sigma-star.at/post/2022/07/squashfs-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.
- `ext2` is great in small partitions and on systems with little RAM.
- 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, a good default choice would be `ext4`, and then try `btrfs` and `f2fs` if you need extra performance.

Compatibility filesystems

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

- `vfat` (`CONFIG_VFAT_FS`) 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 (`CONFIG_EXFAT_FS`).
- `ntfs` (`CONFIG_NTFS_FS`) for compatibility with Windows NTFS filesystem.
- `hfs` (`CONFIG_HFS_FS`) for compatibility with the MacOS HFS filesystem.

tmpfs: filesystem in RAM

`CONFIG_TMPFS`

- 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 documentation.

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/sdb1`
  - `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!

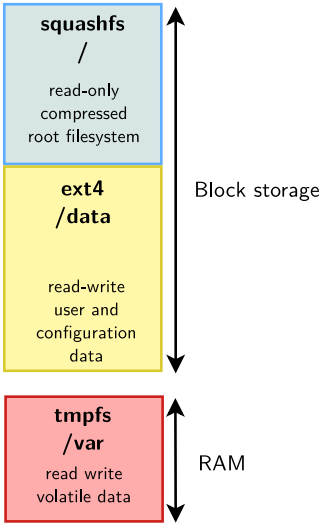
Creating squashfs filesystems

- 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 data/ data.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 data.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.
  Journaling guarantees filesystem integrity after power off or crashes.
- Ram storage for temporary files (tmpfs)

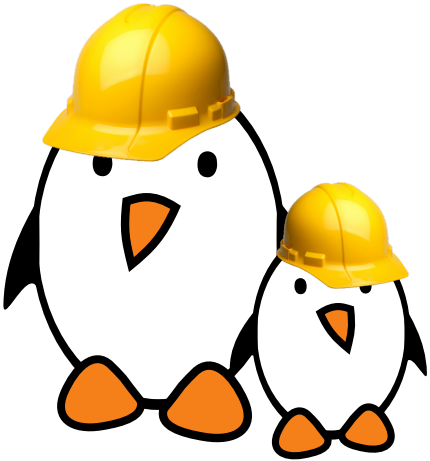


Issues with flash-based block storage

- 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.

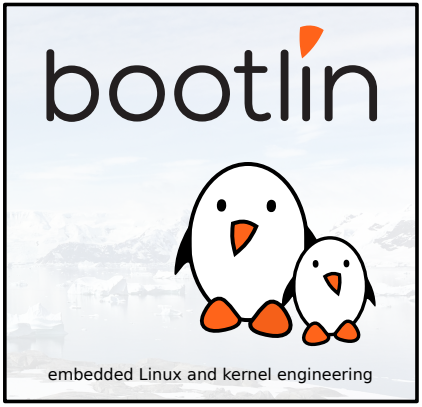
See the *Optimizing Linux with cheap flash drives* article from Arnd Bergmann and try his *flashbench* tool (`http://git.linaro.org/people/arnd/flashbench.git/about/`) for finding out the erase block and page size for your storage, and optimizing your partitions and filesystems for best performance. Note that some SD cards report their erase block size, available in `/sys/bus/mmc/devices/<dev>/preferred_erase_size`.

Practical lab - Block filesystems



- 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.

Embedded Linux system development



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 (if possible).
- 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:

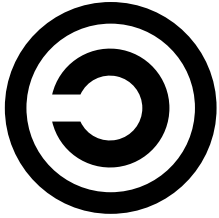
- **Vitality** of the developer and user communities. This vitality ensures long-term maintenance of the component, and relatively good support. It can be measured by looking at the mailing-list traffic and the version control system activity.
- **Quality** of the component. Typically, if a component 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



GPL

- **GNU General Public License**
- Covers around 55% of the free software projects
  - Including the Linux kernel, BusyBox and many applications
- Is a copyleft license
  - Requires derivative works to be released under the same license
  - Programs linked with a library released under the GPL must also be released under the GPL
- Some programs covered by version 2 (Linux kernel, BusyBox, 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

GPL: redistribution

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

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
- Check Free Software Foundation's opinion
  https://www.fsf.org/licensing/licenses/
- Check Open Source Initiative's opinion
  https://www.opensource.org/licenses
- Otherwise, read the license text

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

Respect free software licenses (1)

- 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.

Respect free software licenses (2)

- Free Software licenses have been enforced successfully in courts
- Risks:
  - Users complaining to copyright owners, who could sue you.
  - A competitor could look for copyright violations in your firmware (binary scanning tools exist) to try to have your product withdrawn from the market until this is fixed.
- Organizations which can help solving issues:
  - 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.

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
- **netsnmp**, 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

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
- Supported languages
  - Shell (bash, sh...)
  - Lua, easy to embed in C applications
  - Python
  - Perl
  - Ruby
  - TCL
  - PHP

Tools for the target device: Audio, video and multimedia

Audio, video and multimedia

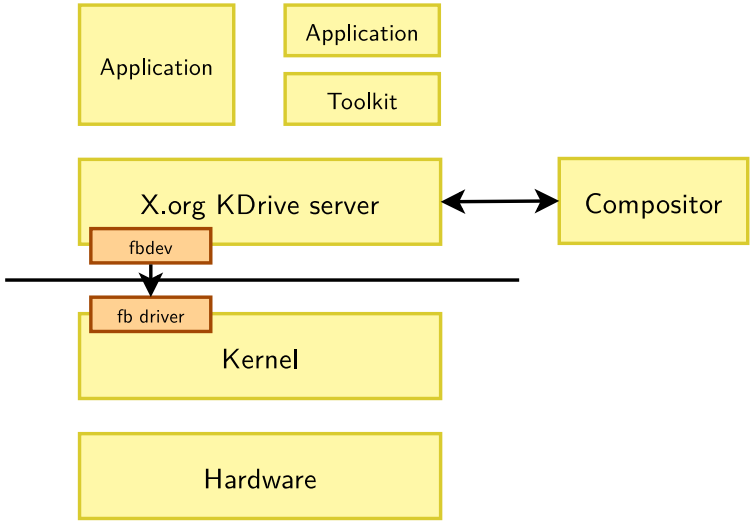
- **GStreamer**, a multimedia framework
  - Allows to decode/encode a wide variety of codecs.
  - Supports hardware encoders and decoders through plugins, proprietary/specific plugins are often provided by SoC vendors.
- **alsa-lib**, the user space library associated to the ALSA kernel sound subsystem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

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



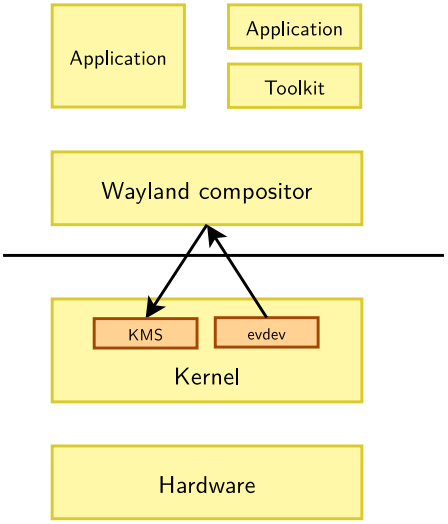
Kdrive: usage

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

Wayland

- 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)

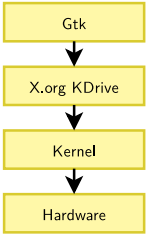
Wayland: architecture

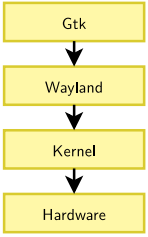


Graphical toolkits: “High-level” solutions

Gtk

- The famous toolkit, providing widget-based high-level APIs to develop graphical applications
- Standard API in C, but bindings exist for various languages: C++, Python, etc.
- Works on top of X.org 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

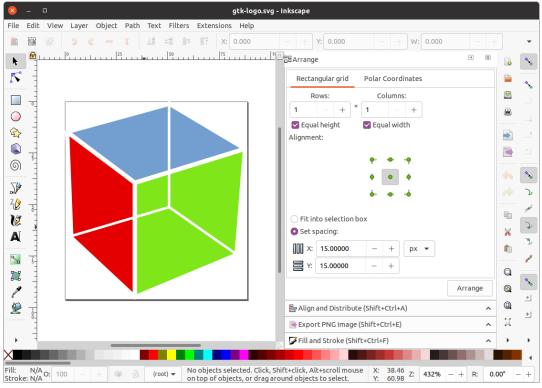




Gtk stack components

- **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
  - EGLFS
  - Linux 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/

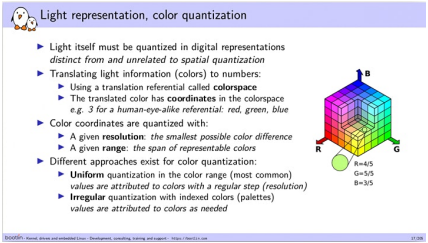
See https://en.wikipedia.org/wiki/List_of_widget_toolkits

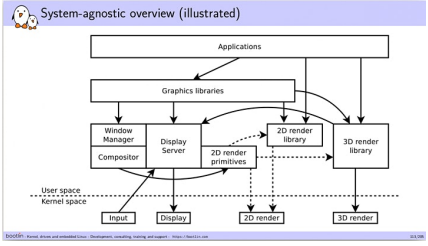
Further details on Linux graphics

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





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

WebKit

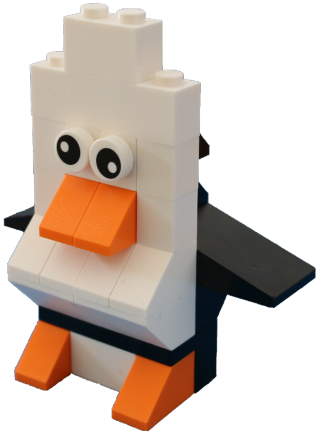
https://webkit.org/

- 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)

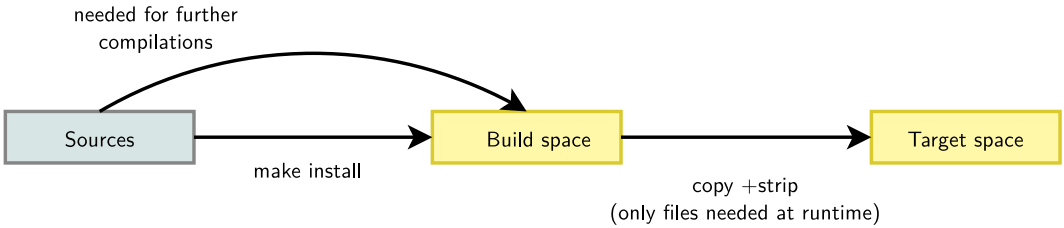
- Manual system building is not recommended for production projects
- However, using automated tools often requires the developer to dig into specific issues
- Having a basic understanding of how a system can be built manually is therefore very useful to fix issues encountered with automated tools
  - 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

System foundations

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

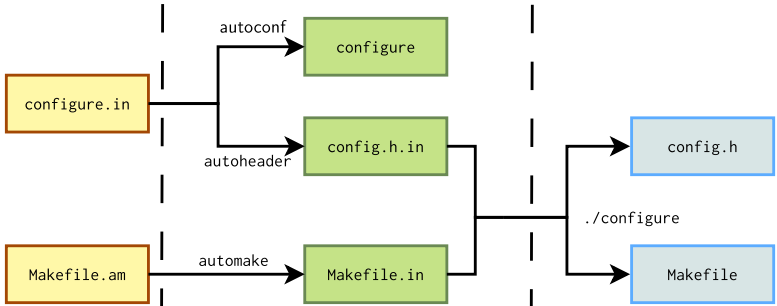
Each open-source component comes with a mechanism to configure, compile and install it

- 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/
  - More recent and simpler than the *autotools*. Used by (sometimes large) projects such as KDE, KiCad, LLVM / Clang, Scribus, OpenCV, Qt (since version 6).
- Meson, https://mesonbuild.com/
  - Even more recent. Faster and simple to use. Now used by projects such as GNOME (partially), GTK+, Gstreamer, Mesa, Systemd, Wayland (Weston).
- Many more exist

Autotools and friends

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

automake / autoconf / autoheader



automake / autoconf

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

Configuring and compiling: native case

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

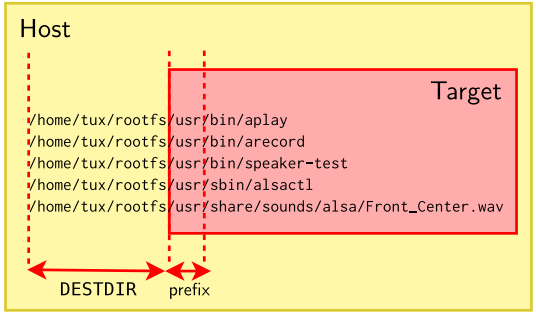
Configuring and compiling: cross case (1)

- For cross-compilation, things are a little bit more complicated.
- At least some of the environment variables `AR`, `AS`, `LD`, `NM`, `CC`, `GCC`, `CPP`, `CXX`, `STRIP`, `OBJCOPY` must be defined to point to the proper cross-compilation tools. The host tuple is also by default used as prefix.
- `configure` script arguments:
  - `--host`: mandatory but a bit confusing. Corresponds to the *target* platform the code will run on. Example: `--host=arm-linux`
  - `--build`: build system. Automatically detected.
  - `--target` is only for tools generating code.
- It is 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, to recompile existing ones, and to debug your applications.

pkg-config

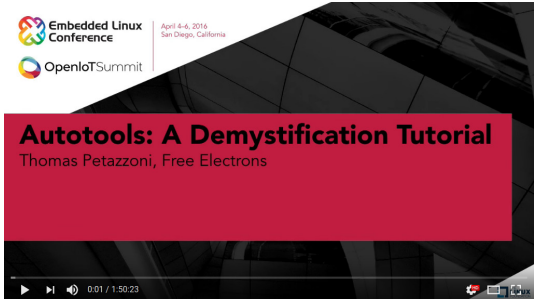
- `pkg-config` is a tool that allows to query a small database to get information on how to compile programs that depend on libraries
- The database is made of `.pc` files, installed by default in `<prefix>/lib/pkgconfig/`.
- `pkg-config` is used by the `configure` script to get the library configurations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

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, typically `/my/build/space/usr/lib/pkgconfig`
- The `PKG_CONFIG_SYSROOT_DIR` variable must be set to the build space directory.

Further details about autotools

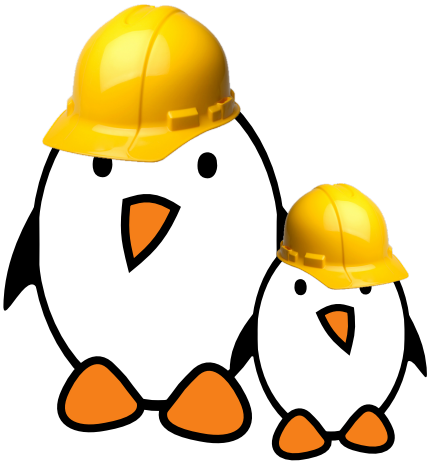
See our *Demystification tutorial* presentation about the GNU Autotools by Thomas Petazzoni, 2016: slides (101 pages!) [1], video [2]



[1] https://bootlin.com/pub/conferences/2016/elc/petazzoni-autotools-tutorial/petazzoni-autotools-tutorial.pdf

[2] https://youtu.be/a1NRxIA9ahA

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

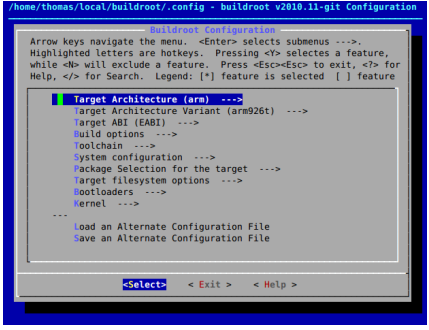
- **Buildroot**, developed by the community
  https://buildroot.org
  See our dedicated course and training materials:
  https://bootlin.com/training/buildroot/
- **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project
  https://openwrt.org
- **PTXdist**, developed 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
  https://www.openembedded.org and its industrialized version **Yocto Project**. 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 2800 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 size 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 `gqview.hash` file (optional, but recommended), containing hashes for the files to download, and for the license file.
  - 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.

OpenEmbedded / Yocto Project

- The most versatile and powerful embedded Linux build system
  - A collection of recipes (`.bb` files)
  - A tool that processes the recipes: `bitbake`
- Integrates 2000+ application and libraries, is highly configurable, can generate binary packages to make the system customizable, supports multiple versions/variants of the same package, no need for full rebuild when the configuration is changed.
- Configuration takes place by editing various configuration files
- 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.
- Active community, releases published every 6 months. LTS releases available (supported at least 2 years, 4 years for "Dunfell").

Distributions - Debian

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. See our blog post [1].

[1] https://bootlin.com/blog/elbe-automated-building-of-ubuntu-images-for-a-raspberry-pi-3b/

Distributions - Others

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!

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://bit.ly/2EzmJLF

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.

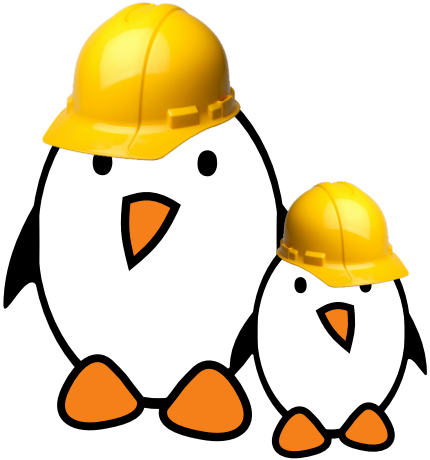
- **Tizen**: https://www.tizen.org/
  Targeting smartphones, wearables (watches), smart TVs and In Vehicle Infotainment devices.
  Supported by big phone manufacturers (mostly Samsung) and operators
  HTML5 base application framework.
  Wikipedia: 21% of the smart TVs market share in 2018
  See https://en.wikipedia.org/wiki/Tizen
- **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.

TIZEN™

android

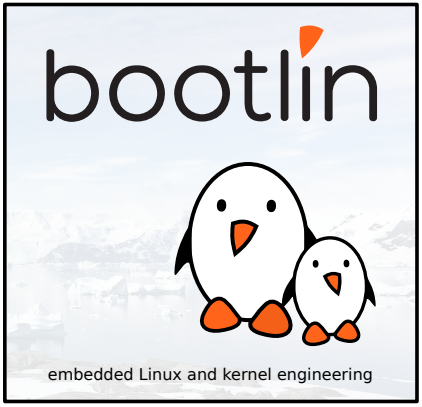
Image credits:
https://frama.link/yhPuj_oS

Practical lab - Buildroot



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

Embedded Linux application development



Contents

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 simpler and more modern tools instead, such as *CMake* and *Meson*.

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)
```

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

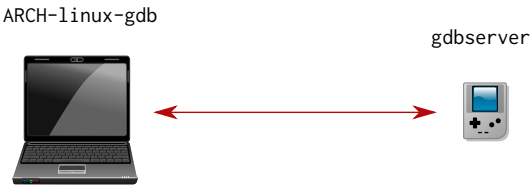
GDB crash course (2)

- `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

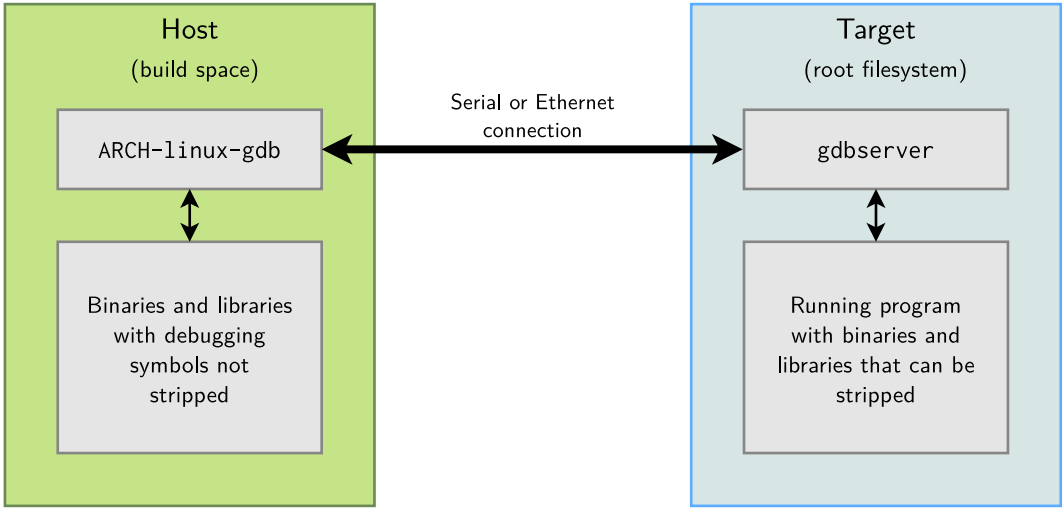
Remote debugging

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).



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/ttyUSB0` (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`

Profiling

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()`. This tells you what files are handled by further system calls.

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
...
```

ltrace

A tool to trace **shared** 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.
- Also offers a summary with its `-c` option.
- Manual page: https://linux.die.net/man/1/ltrace
- Works better with *glibc*. `ltrace` was broken with *uClibc* and may still be, and was not supported with *Musl* (Buildroot 2021.08 status).

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

ltrace example output

```
# ltrace  ffmpeg -f video4linux2 -video_size 544x288 -input_format mjpeg -i /dev
/video0 -pix_fmt rgb565le -f fbdev /dev/fb0
__libc_start_main([ "ffmpeg", "-f", "video4linux2", "-video_size"... ] <unfinished ...>
setvbuf(0xb6a0ec80, nil, 2, 0)                  = 0
av_log_set_flags(1, 0, 1, 0)                    = 1
strchr("f", ':')                                = nil
strlen("f")                                     = 1
strncmp("f", "L", 1)                            = 26
strncmp("f", "h", 1)                            = -2
strncmp("f", "?", 1)                            = 39
strncmp("f", "help", 1)                         = -2
strncmp("f", "-help", 1)                        = 57
strncmp("f", "version", 1)                      = -16
strncmp("f", "buildconf", 1)                    = 4
strncmp("f", "formats", 1)                      = 0
strlen("formats")                               = 7
strncmp("f", "muxers", 1)                       = -7
strncmp("f", "demuxers", 1)                     = 2
strncmp("f", "devices", 1)                      = 2
strncmp("f", "codecs", 1)                       = 3
...
```

ltrace summary

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

```
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 52.64    5.958660     5958660         1 __libc_start_main
 20.64    2.336331     2336331         1 avformat_find_stream_info
 14.87    1.682895         421      3995 strncmp
  7.17    0.811210      811210         1 avformat_open_input
  0.75    0.085290         584       146 av_freep
  0.49    0.055150         434       127 strlen
  0.29    0.033008         660        50 av_log
  0.22    0.025090         464        54 strcmp
  0.20    0.022836       22836         1 avformat_close_input
  0.16    0.017788         635        28 av_dict_free
  0.15    0.016819         646        26 av_dict_get
  0.15    0.016753         440        38 strchr
  0.13    0.014536         581        25 memset
  0.09    0.009762        9762         1 avcodec_send_packet
...
------ ----------- ----------- --------- --------------------
100.00   11.318773                  4762 total
```

Valgrind (1)

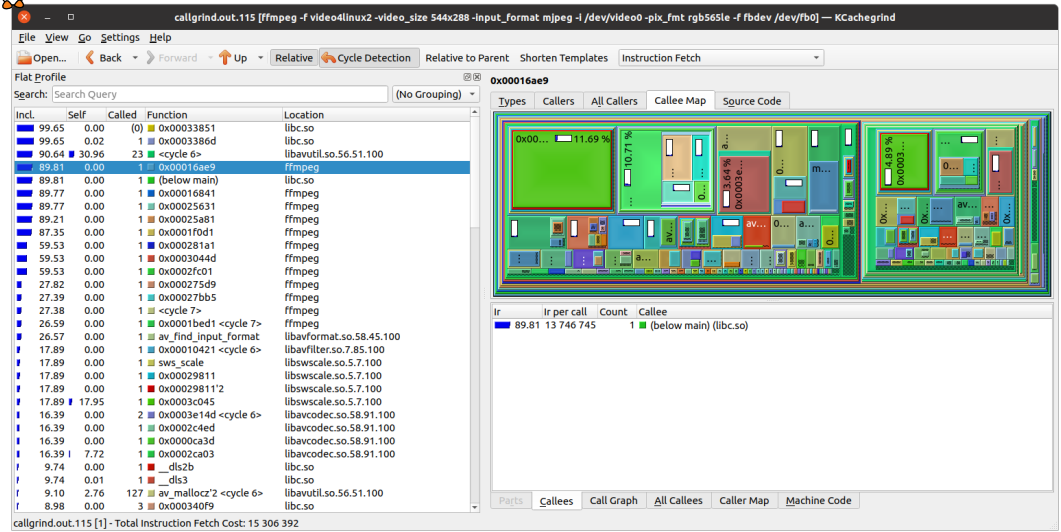
https://valgrind.org/

- 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)

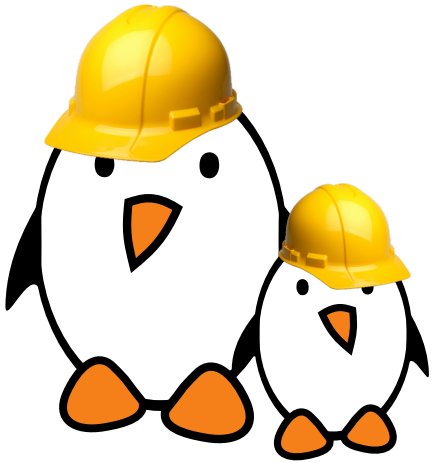
- Can be used to run any program, without the need to recompile it.
- Examples
  `valgrind --leak-check=yes <program>` (leak check mode)
  `valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes <program>` (profiling mode)
- Works by adding its own instrumentation to your code and then running in on its own virtual cpu core. Significantly slows down execution, but still fine for debugging and profiling!
- More details on https://valgrind.org/info/ and https://valgrind.org/docs/manual/manual.html
- The Valgrind tool suite is easy to add to your root filesystem with Buildroot.

Kcachegrind - Visualizing Valgrind profiling data



Directly run it on Callgrind's output file.

Practical lab - App. development and debugging



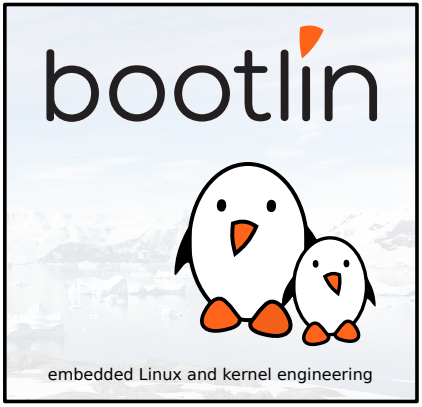
Application development

- 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

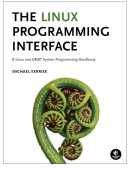
References



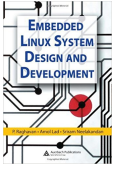
Books

- **Mastering Embedded Linux, 3rd Edition** [1]
  By Chris Simmonds, Packt Publishing, May 2021
  An up-to-date resource covering most aspects of embedded Linux development.

- **The Linux Programming Interface** [2]
  Michael Kerrisk (maintainer of Linux manual pages), 2010, No Starch Press
  A gold mine about Linux system programming



- **Embedded Linux System Design and Development** [3]
  P. Raghavan, A. Lad, S. Neelakandan, Auerbach, Dec. 2005.
  Very good coverage of the POSIX programming API (still up to date).



[1] https://www.packtpub.com/product/mastering-embedded-linux-programming-third-edition/9781789530384
[2] https://man7.org/tlpi/
[3] https://www.amazon.com/Embedded-Linux-System-Design-Development/dp/0849340586

Web sites

- **ELinux.org**, https://elinux.org, a Wiki entirely dedicated to embedded Linux. Lots of topics covered: real-time, filesystems, 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.

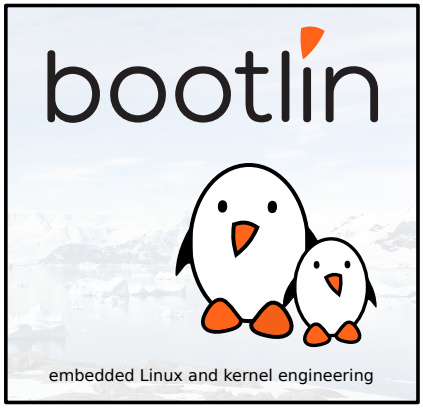
International conferences (1)

- 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



Last slide

Thank you!

And may the Source be with you

Rights to copy

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