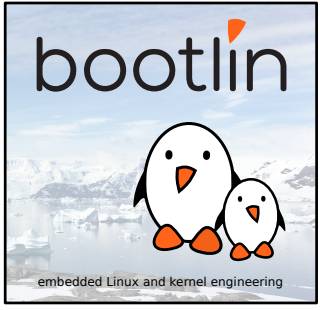
Embedded Linux system development training

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: April 10, 2024.

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

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

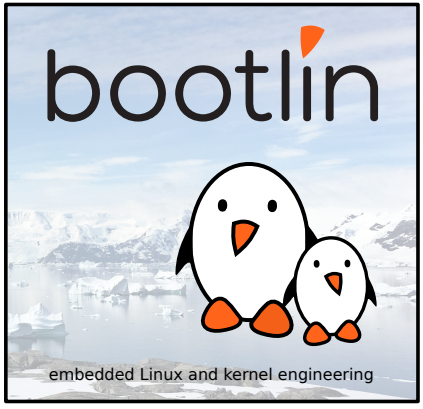


Embedded Linux system development training

- 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



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

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	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours
Embedded Linux audio On-site: 2 days Online: 4 * 4 hours			

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

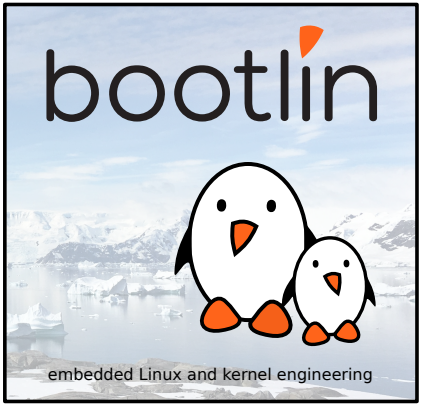
- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 8000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- 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



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Supported hardware

Discovery Kits from STMicroelectronics: STM32MP157A-DK1, STM32MP157D-DK1, STM32MP157C-DK2 or STM32MP157F-DK2

- STM32MP157 (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
- LCD touchscreen (DK2 only)

DK1 Discovery Kit

Board and CPU documentation, design files, software: A-DK1, D-DK1, C-DK2, F-DK2

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
- Nintendo Nunchuk with UEXT connector [1]
- Breadboard jumper wires - Male ends (to connect the Nunchuk) [2]
- A standard USB audio headset.
- A micro SD card with at least 1 GB of capacity

[1] https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/

[2] https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/

Labs proposed on another platform

After this course, you can also run all labs on the Beaglebone Black or Beaglebone Black Wireless ARM board.

Lab instructions are available at
https://bootlin.com/doc/training/embedded-linux-bbb/

open source hardware

Labs proposed on another platform

In addition to the BeagleBone and the STM32MP157D-DK1 you can also run most labs on the Beagleplay board.

Lab instructions are available at
https://bootlin.com/doc/training/embedded-linux-beagleplay/

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!

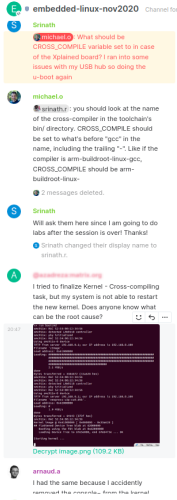
During the lectures...

- 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 always keep your camera on!
  - Also make sure your name is properly filled.
  - 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.

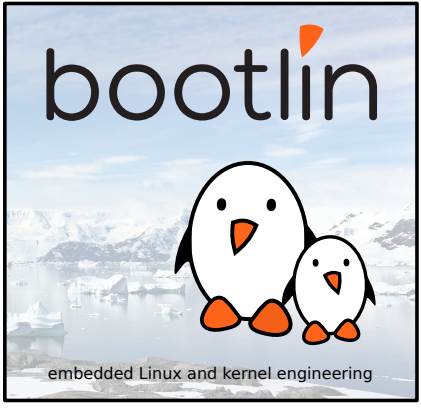
Collaborate!

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.



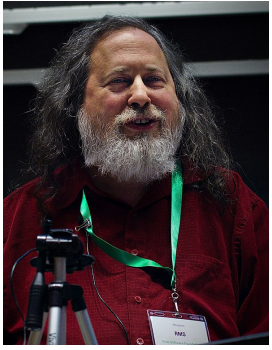
Introduction to Embedded Linux



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Birth of Free Software

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



Richard Stallman in 2019
https://commons.wikimedia.org/wiki/File:Richard_Stallman_at_LibrePlanet_2019.jpg

Free software?

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

What is embedded Linux?

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

Advantages of Linux and Open-Source 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.
- **Security**
  You can trace the sources of all system components and perform independent vulnerability assessments.
- **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

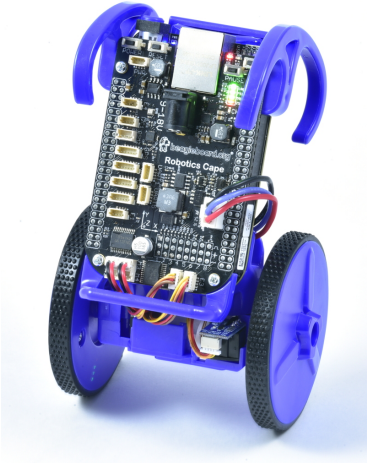
Image credits: Evan Amos (https://bit.ly/2JzDIkv)

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)

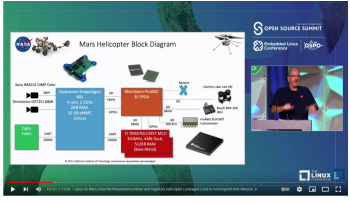
In space

SpaceX Starlink satellites

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 increase performance and re-use as many existing components as possible.

Communication

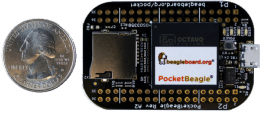
- The Linux kernel has support for many common communication buses
  - I2C
  - SPI
  - 1-wire
  - SDIO
  - PCI
  - USB
  - CAN (mainly used in automotive)
- And also extensive networking support
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced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

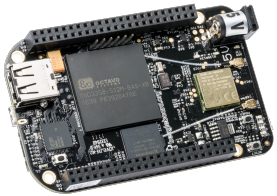
Image credits



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

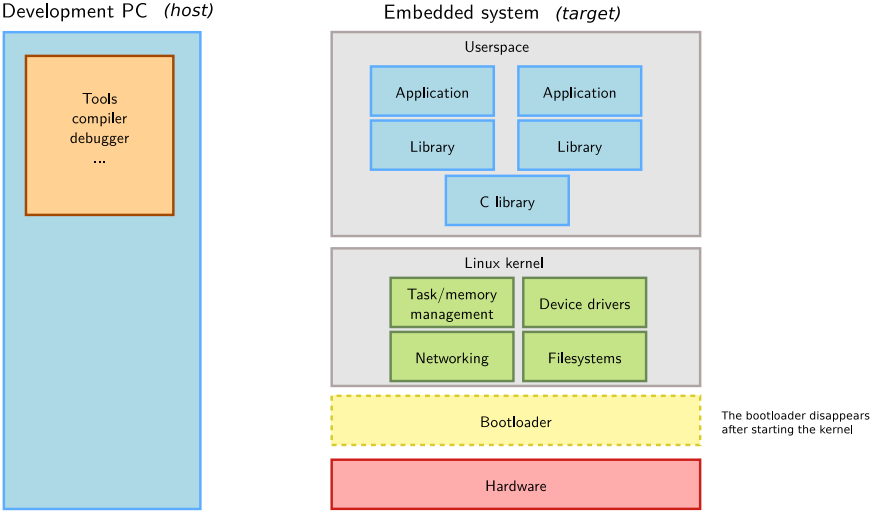
Olimex Open hardware ARM laptop main board
Image credits (Olimex): https://www.olimex.com/Products/DIY-Laptop/

Criteria for choosing the hardware

- Most SoCs are delivered with support for the Linux kernel and for an open-source bootloader.
- Having support for your SoC 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

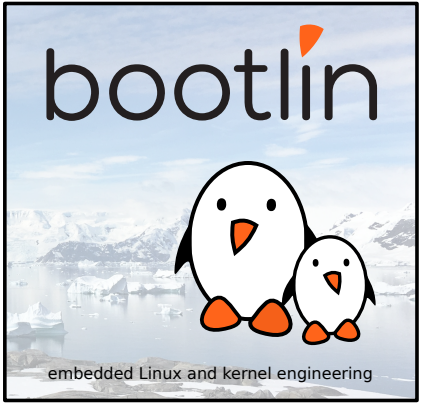
- Cross-compilation toolchain
  - Compiler that runs on the development machine, but generates code for the target
- Bootloader
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- Linux Kernel
  - Contains the process and memory management, network stack, 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



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

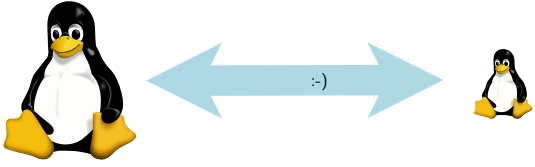
Embedded Linux solutions

- Two ways to switch to embedded Linux
  - Use **solutions provided and supported by vendors** like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
  - Use **community solutions**. They are completely open, supported by the community.
- In 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, macOS) 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 10/11, you can also run your favorite native Linux distro through Windows Subsystem for Linux (WSL2)



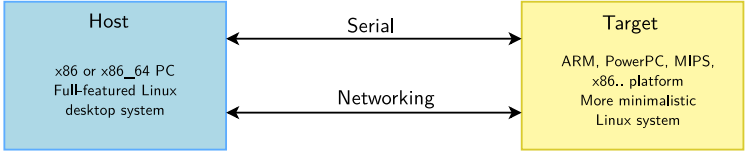
Desktop Linux distribution

- **Any good and sufficiently recent Linux desktop distribution** can be used for the development workstation
  - Ubuntu, Debian, Fedora, openSUSE, Arch Linux, 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

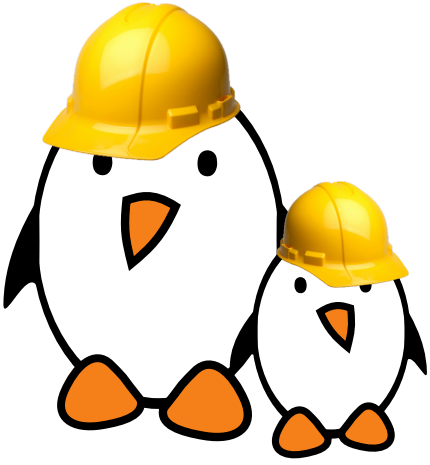
- When doing embedded development, there is always a split between
  - The *host*, the development workstation, which is typically a powerful PC
  - The *target*, which is the embedded system under development
- They are connected by various means: almost always a serial line for debugging purposes, 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*, *tmux* 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`

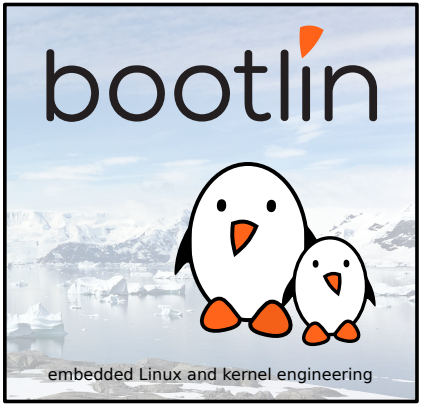
Practical lab - Training Setup



Prepare your lab environment

- Download and extract the lab archive

Cross-compiling toolchains



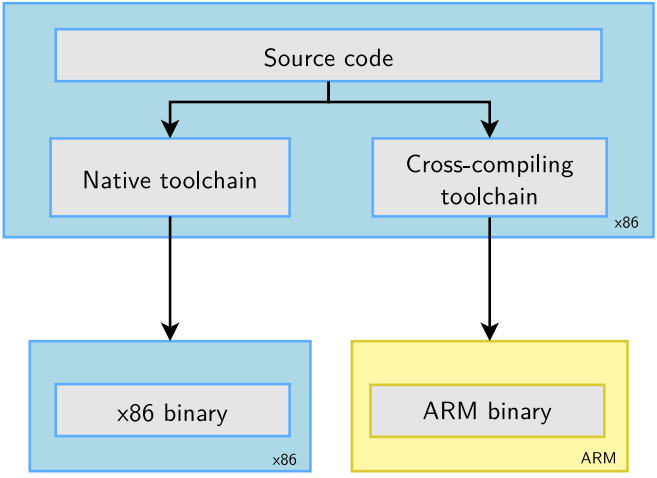
© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Definition and Components

Toolchain definition (1)

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

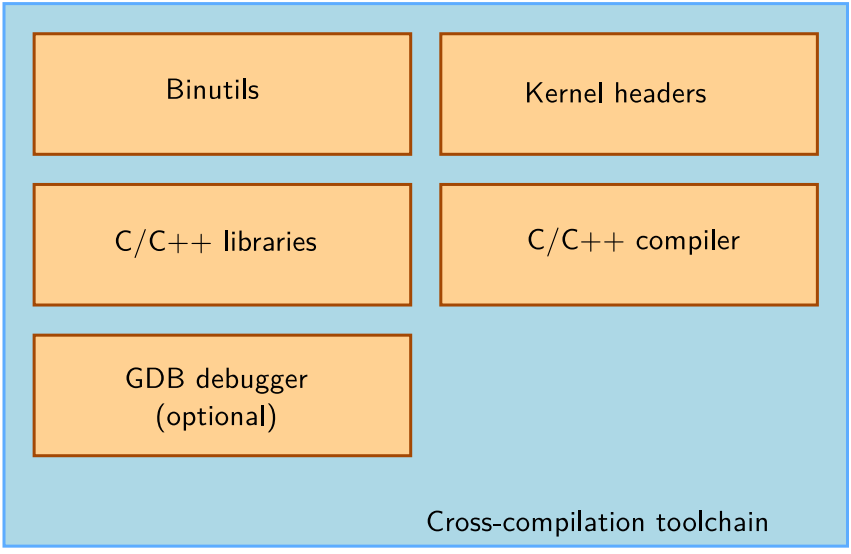
Toolchain definition (2)



Architecture tuple and toolchain prefix

- Many UNIX/Linux build mechanisms rely on *architecture tuple* names to identify machines.
- Examples: `arm-linux-gnueabihf`, `mips64el-linux-gnu`, `arm-vendor-none-eabihf`
- These tuples are 3 or 4 parts:
  1. The architecture name: `arm`, `riscv`, `mips64el`, etc.
  2. Optionally, a vendor name, which is a free-form string
  3. An operating system name, or `none` when not targeting an operating system
  4. The ABI/C library (see later)
- This tuple is used to:
  - configure/build software for a given platform
  - as a prefix of cross-compilation tools, to differentiate them from the native toolchain
    - `gcc` → native compiler
    - `arm-linux-gnueabihf-gcc` → cross-compiler

Components of gcc toolchains



Binutils

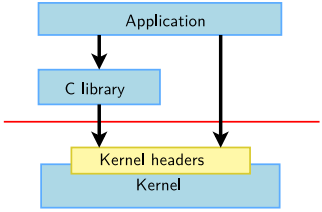
- **Binutils** is a set of tools to generate and manipulate binaries (usually with the ELF format) for a given CPU architecture
  - `as`, the assembler, that generates binary code from assembler source code
  - `ld`, the linker
  - `ar`, `ranlib`, to generate `.a` archives (static libraries)
  - `objdump`, `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

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.

Kernel headers (1)

- The C standard library and compiled programs need to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C standard 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)

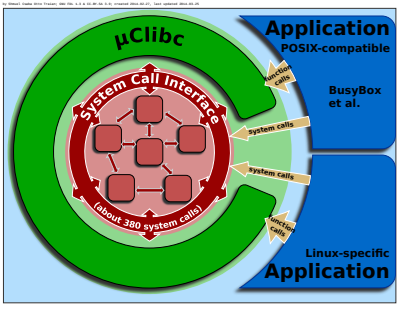
The kernel to user space interface is **backward compatible**

- 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 standard library

- The C standard 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 standard libraries are available: *glibc*, *uClibc*, *musl*, *klibc*, *newlib*...
- The choice of the C standard library must be made at cross-compiling toolchain generation time, as the GCC compiler is compiled against a specific C standard library.



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

glibc

- License: LGPL
- C standard 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
- https://www.gnu.org/software/libc/

Image source

uClibc-ng

- https://uclibc-ng.org/
- A continuation of the old uClibc project, license: LGPL
- Lightweight C standard 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 standard 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, supported by Buildroot but not by Yocto Project.

musl C standard library

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

- A lightweight, fast and simple standard library for embedded systems
- Created while uClibc's development was stalled
- In particular, great at making small static executables, which can run anywhere, even on a system built from another C standard library.
- 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

Other smaller C libraries

- Several other smaller C libraries exist, but they do not implement the full POSIX interface required by most Linux applications
- They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- Therefore not commonly used in most embedded Linux systems
- 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.

Advice for choosing the C standard library

- Advice to start developing and debugging your applications with *glibc*, which is the most standard solution
- If you have size constraints, try to compile your app and then the entire filesystem with *uClibc* or *musl*
  - The size advantage of *uClibc* or *musl*, which used to be a significant argument, is less relevant with today's storage capacities.
  - Smaller binaries and filesystems remain useful when optimizing boot time, though, typically booting on a filesystem loaded in RAM, and to reduce the size of container and virtual machine images (one of the use cases of Alpine Linux).
- If you run into trouble, it could be because of missing features in the C standard library.
- In case you wish to make static executables, *musl* will be an easier choice in terms of licensing constraints.

Linux vs. bare-metal toolchain

- A **Linux toolchain**
  - is a toolchain that includes a Linux-ready C standard library, which uses the Linux system calls to implement system services
  - can be used to build Linux user-space applications, but also bare-metal code (firmware, bootloader, Linux kernel)
  - is identified by the `linux` OS identifier in the toolchain tuple: `arm-linux`, `arm-none-linux-gnueabihf`
- A **bare metal toolchain**
  - is a toolchain that does not include a C standard library, or a very minimal one that isn't tied to a particular operating system
  - can be used to build only bare-metal code (firmware, bootloader, Linux kernel)
  - is identified by the `none` OS identifier in the toolchain tuple: `arm-none-eabi`, `arm-none-none-eabi` (vendor is `none`, OS is `none`)

An alternate compiler suite: LLVM

- Most Embedded Linux projects use toolchains based on GNU project: GCC compiler, binutils, GDB debugger
- The LLVM project has been developing an alternative compiler suite:
  - Clang, C/C++ compiler, https://clang.llvm.org/
  - LLDB, debugger, https://lldb.llvm.org/
  - LLD, linker, https://lld.llvm.org/
  - and more, see https://llvm.org/
- While they are used by several high-profile projects, they are not yet in widespread use in most Embedded Linux projects.
- Initially had better code optimization and diagnostics than GCC, but thanks to having competition, GCC has improved significantly in this area.
- Available under MIT/BSD licenses
- https://en.wikipedia.org/wiki/LLVM

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 32-bit, two main ABIs: *EABI* and *EABIhf*
  - *EABIhf* passes floating-point arguments in floating-point registers → needs an ARM processor with a FPU
- On RISC-V, several ABIs: *ilp32*, *ilp32f*, *ilp32d*, *lp64*, *lp64f*, and *lp64d*
- https://en.wikipedia.org/wiki/Application_Binary_Interface

Floating point support

- All ARMv7-A (32-bit) and ARMv8-A (64-bit) processors have a floating point unit
- RISC-V cores with the F extension have a floating point unit
- Some older ARM cores (ARMv4/ARMv5) or some RISC-V cores may not have a floating point unit
- For processors without a floating point unit, two solutions for floating point computation:
  - 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
- For processors with a floating point unit, sometimes different FPU are possible. For example on ARM: VFPv3, VFPv3-D16, VFPv4, VFPv4-D16, etc.

CPU optimization flags

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

Obtaining a Toolchain

Building a toolchain manually

- Building a cross-compiling toolchain manually is a fairly difficult process
- Lots of details to learn: many components to build with complicated configuration
- Typical process is:
  - Build dependencies of binutils/gcc (GMP, MPFR, ISL, etc.)
  - Build *binutils*
  - Build a baremetal, first stage *GCC*
  - Extract kernel headers from the Linux source code
  - Build the C library using the first stage GCC
  - Build the second stage and final *GCC* supporting the Linux OS and the C library.
- Many decisions to make about the components: C library, gcc and binutils versions, ABI, floating point mechanisms, etc. Not trivial to find correct combinations of these possibilities
- See the Crosstool-NG documentation for details on how toolchains are built.
- Talk: *Anatomy of Cross-Compilation Toolchains*, by Thomas Petazzoni, ELCE 2017, video and slides

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, version of the kernel headers, ABI, soft float or hard float, etc.
- Some possibilities:
  - Toolchains packaged by your distribution, for example Ubuntu package gcc-arm-linux-gnueabihf or Fedora gcc-arm-linux-gnu. Often limited to ARM/ARM64 with glibc.
  - Bootlin's GNU toolchains, most CPU architectures, with glibc/uClibc/musl, https://toolchains.bootlin.com
  - ARM and ARM64 toolchains released by ARM

Example of toolchains from ARM: downloading

x86_64 Linux hosted cross compilers

AArch32 bare-metal target (arm-none-eabi)

- gcc-arm-10.3-2021.07-x86_64-arm-none-eabi.tar.xz
- gcc-arm-10.3-2021.07-x86_64-arm-none-eabi.tar.xz.asc

AArch32 target with hard float (arm-none-linux-gnueabihf)

- gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz
- gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz.asc

AArch64 ELF bare-metal target (aarch64-none-elf)

- gcc-arm-10.3-2021.07-x86_64-aarch64-none-elf.tar.xz
- gcc-arm-10.3-2021.07-x86_64-aarch64-none-elf.tar.xz.asc

AArch64 GNU/Linux target (aarch64-none-linux-gnu)

- gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu.tar.xz
- gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu.tar.xz.asc

AArch64 GNU/Linux target (aarch64_be-none-linux-gnu)

- gcc-arm-10.3-2021.07-x86_64-aarch64_be-none-linux-gnu.tar.xz
- gcc-arm-10.3-2021.07-x86_64-aarch64_be-none-linux-gnu.tar.xz.asc

From Arm GNU Toolchains

Example of toolchains from ARM: using

```
$ wget https://developer.arm.com/-/media/Files/downloads/gnu-a/10.3-2021.07/binrel/[...]
    [...]gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz

$ tar xf gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz

$ cd gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf/

$ ./bin/arm-none-linux-gnueabihf-gcc -o test test.c

$ file test
test: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf.so.3, [...]
   for GNU/Linux 3.2.0, with debug_info, not stripped
```

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.
- Allows to rebuild the toolchain if needed to fix a bug or security issue.
- 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*, supporting only glibc and uClibc (version 2023.01 status)
  - https://www.ptxdist.org/
- **OpenEmbedded / Yocto Project**
  - A featureful, but more complicated build system, supporting only glibc and musl.
  - https://www.openembedded.org/
  - https://www.yoctoproject.org/

Crosstool-NG: download

- Getting Crosstool-NG

```
$ git clone https://github.com/crosstool-ng/crosstool-ng.git
```

- Using a well-known stable version

```
$ cd crosstool-ng
$ git checkout crosstool-ng-1.25.0
```

- As we're fetching from Git, the `configure` script needs to be generated:

```
$ ./bootstrap
```

Crosstool-NG: installation

- Installation can be done:
  - system-wide, for example in `/usr/local`, the `ct-ng` command is then available globally

    ```
    $ ./configure
    $ make
    $ sudo make install
    ```

  - or just locally in the source directory, the `ct-ng` command will be invoked from this directory

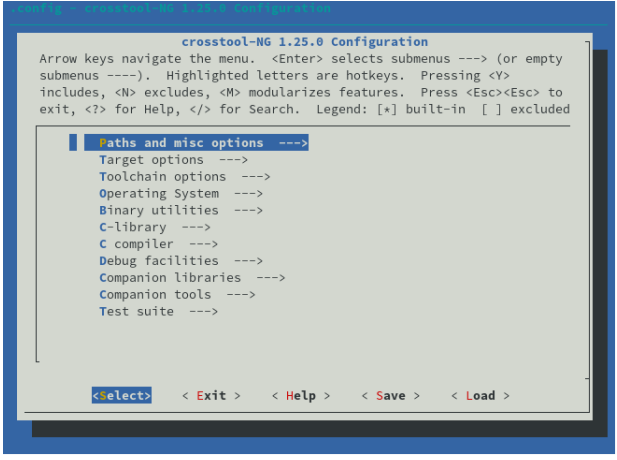
    ```
    $ ./configure --enable-local
    $ make
    ```

- In our labs, we will use the second method
- Note: the `make` invocation doesn't build any toolchain, it builds the `ct-ng` executable.

Crosstool-NG: toolchain configuration

- Once installed, the `ct-ng` tool allows to configure and build an arbitrary number of toolchains
- Its configuration system is based on *kconfig*, like the Linux kernel configuration system
- Configuration of the toolchain to build stored in a `.config` file
- Example configurations provided with Crosstool-NG
  - List: `./ct-ng list-samples`
  - Load an example: `./ct-ng <sample-name>`, replaces `.config`
  - For example `./ct-ng aarch64-unknown-linux-gnu`
  - No sample loaded → default Crosstool-NG configuration is a bare-metal toolchain for the *Alpha* CPU architecture!
- The configuration can then be refined using either:
  - `./ct-ng menuconfig`
  - `./ct-ng nconfig`

Crosstool-NG: toolchain configuration



./ct-ng menuconfig

Crosstool-NG: toolchain building

- To build the toolchain

  `./ct-ng build`

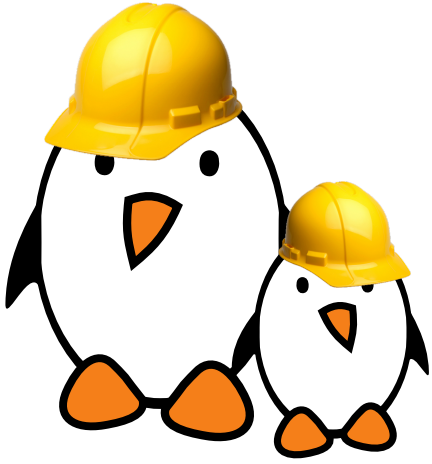
  This will automatically download all the needed dependencies, and build all toolchain components in the right order, with the specified configuration.
- By default the results go in `$HOME/x-tools/<architecture-tuple>`, as defined by the option `CT_PREFIX_DIR` in *Paths and misc options*

Important toolchain contents

- `bin/`: cross compilation tool binaries
  - This directory can be added to your `PATH` to ease usage of the toolchain
  - Sometimes with symlinks for shorter names

    `arm-linux-gcc -> arm-cortexa7-linux-uclibcgnueabihf-gcc`
- `<arch-tuple>/sysroot`: *sysroot* directory
  - `<arch-tuple>/sysroot/lib`: C library, GCC runtime, C++ standard library compiled for the target
  - `<arch-tuple>/sysroot/usr/include`: C library headers and kernel headers

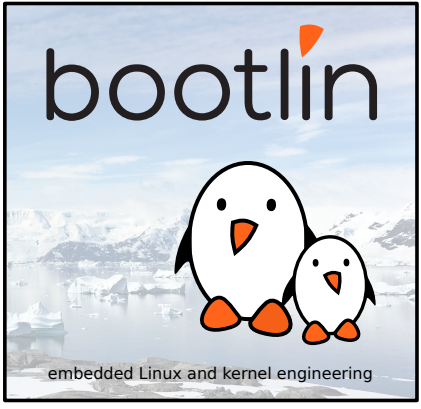
Practical lab - Using Crosstool-NG



Time to build your toolchain

- Getting and configuring Crosstool-NG
- Executing it to build a custom cross-compilation toolchain
- Exploring the contents of the toolchain

Bootloaders and firmware



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

Bootloader role

- 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 or menu
  - Menu to select the operating system to load
  - Shell with commands to load data from storage or network, inspect memory, perform hardware testing/diagnostics
- The first piece of code running by the processor that can be modified by us developers.

Booting on x86 platforms

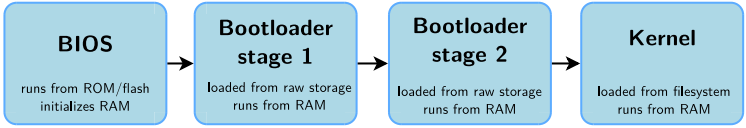
Legacy BIOS booting (1)

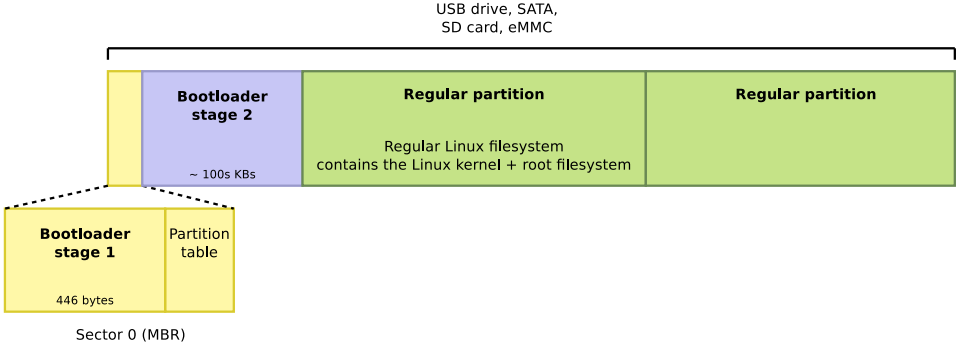
- x86 platforms shipped before 2005-2006 include a firmware called *BIOS*
  - BIOS = Basic Input Output System
  - Part of the hardware platform, closed-source, rarely modifiable
  - Implements the booting process
  - Provides runtime services that can be invoked - not commonly used
  - Stored in some flash memory, outside of regular user-accessible storage devices
- To be bootable, the first sector of a storage device is “special”
  - MBR = Master Boot Record
  - Contains the partition table
  - Contains up to 446 bytes of bootloader code, loaded into RAM and executed
  - The BIOS is responsible for the RAM initialization
- https://en.wikipedia.org/wiki/BIOS

Legacy BIOS booting (2)

- Due to the limitation in size of the bootloader, bootloaders are split into two stages
  - Stage 1, which fits within the 446 bytes constraint
  - Stage 2, which is loaded by stage 1, and can therefore be bigger
- Stage 2 is typically stored outside of any filesystem, at a fixed offset → simpler to load by stage 1
- Stage 2 generally has filesystem support, so it can load the kernel image from a filesystem

Legacy BIOS booting: sequence and storage

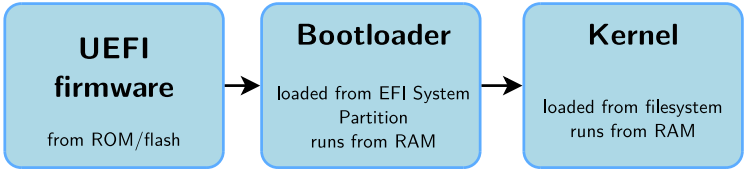




UEFI booting

- Starting from 2005-2006, UEFI is the new firmware interface on x86 platforms
  - Unified Extensible Firmware Interface
  - Describes the interface between the operating system and the firmware
  - Firmware in charge of booting
  - Firmware also provides runtime services to the operating system
  - Stored in some flash memory, outside of regular user-accessible storage devices
- Loads EFI binaries from the *EFI System Partition*
  - Generally a bootloader
  - Can also be directly the Linux kernel, with an *EFI Boot Stub*
- Special partition, formatted with the *FAT* filesystem
  - MBR: identified by type `0xEF`
  - GPT: identified with a specific *globally unique identifier*
- File `/efi/boot/bootx32.efi`, `/efi/boot/bootx64.efi`
- https://en.wikipedia.org/wiki/UEFI

UEFI booting: sequence and storage



ACPI

- Advanced Configuration and Power Interface
- *Open standard that operating systems can use to discover and configure computer hardware components, to perform power management, to perform auto configuration, and to perform status monitoring*
- *Tables* with descriptions of the hardware that cannot be dynamically discovered at runtime
- Tables provided by the firmware (UEFI or legacy) and used by the operating system (Linux kernel in our case)
- https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface

UEFI and ACPI on ARM

- Historically UEFI and ACPI are technologies coming from the Intel/x86 world
- ARM is also pushing for the adoption of UEFI and ACPI as part of its *ARM System Ready* certification
  - Mainly for servers/workstations SoCs
  - Does not impact embedded SoCs
  - Currently not common in embedded Linux projects on ARM
  - https://www.arm.com/architecture/system-architectures/systemready-certification-program
- Also some on-going effort to use UEFI on RISC-V, but not the de-facto standard
- When an embedded platform uses UEFI → its booting process is similar to an *x86* platform

Booting on embedded platforms

Booting on embedded platforms: ROM code

- Most embedded processors include a **ROM code** that implements the initial step of the boot process
- The ROM code is written by the processor vendor and directly built into the processor
  - Cannot be changed or updated
  - Its behavior is described in the processor datasheet
- Responsible for finding a suitable bootloader, loading it and running it
  - From NAND/NOR flash, from USB, from SD card, from eMMC, etc.
  - Well defined location/format
- *Generally* runs with the external RAM not initialized, so it can only load the bootloader into an internal SRAM
  - Limited size of the bootloader, due to the size of the SRAM
  - Forces the boot process to be split in two steps: first stage bootloader (small, runs from SRAM, initializes external DRAM), second stage bootloader (larger, runs from external DRAM)

7 Boot configuration

7.1 Boot mode selection

In the STM32MP15x lines devices, different boot modes can be selected by means of the BOOT[2:0] pins. the reserved configuration is highlighted in gray in the table.

Table 13. Boot modes

BOOT2	BOOT1	BOOT0	Initial boot mode	Comments
0	0	0	UART and USB(1)	Wait incoming connection on: – USART2/3/6 and UART4/5/7/8 on default pins – USB High-Speed device on OTG_HS_DP/DM pins(2)
0	0	1	Serial NOR-Flash(3)	Serial NOR-Flash on QUADSPI(5)
0	1	0	eMMC™(3)	eMMC™ on SDMMC2 (default)(5)(6)
0	1	1	NAND-Flash(3)	SLC NAND-Flash on FMC
1	0	0	Engineering boot (No Flash boot)	Used to get debug access without boot from Flash(4)
1	0	1	SD-Card(3)	SD-Card on SDMMC1 (default)(5)(6)
1	1	0	UART and USB(1)(3)	Wait incoming connection on: – USART2/3/6 and UART4/5/7/8 on default pins – USB High-speed device on OTG_HS_DP/DM pins(2)
1	1	1	Serial NAND-Flash(3)	Serial NAND-Flash on QUADSPI(5)

Source: https://www.st.com/resource/en/application_note/dm00389996-getting-started-with-stm32mp151-stm32mp153-and-stm32mp157-line-hardware-development-stmicroelectronics.pdf

Useful details:

https://wiki.st.com/stm32mpu/wiki/

STM32_MPU_ROM_code_overview

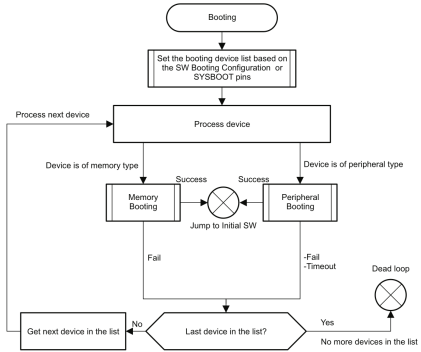
Booting on AM335x (32 bit BeagleBone): datasheet

26.1.5 Booting

26.1.5.1 Overview

Figure 26-6 shows the booting procedure. First a **booting device list** is created. The list consists of all devices which will be searched for a booting image. The list is filled in based on the **SYSBOOT**.

Figure 26-6. ROM Code Booting Procedure

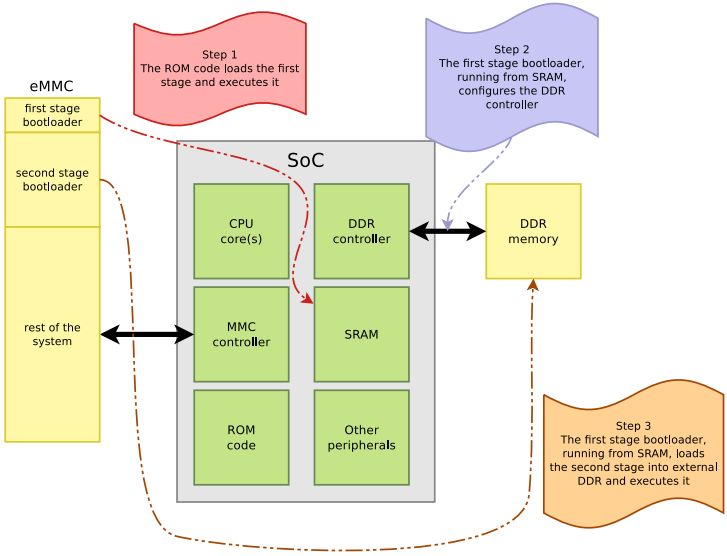


Once the booting device list is set up, the booting routine examines the devices enumerated in the list sequentially and either executes the memory booting or peripheral booting procedure depending on the booting device type. The memory booting procedure is executed when the booting device type is one of NOR, NAND, MMC or SPI-EEPROM. The peripheral booting is executed when the booting device type is Ethernet, USB or UART.

Source:
https://www.mouser.com/pdfdocs/spruh73h.pdf,

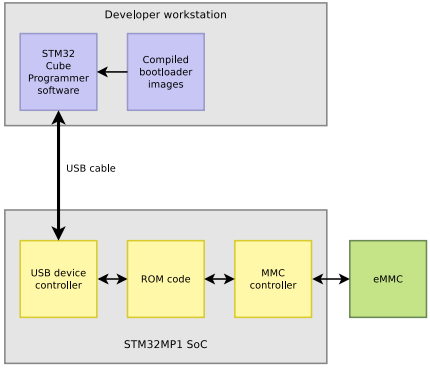
chapter 26

Two stage booting sequence



ROM code recovery mechanism

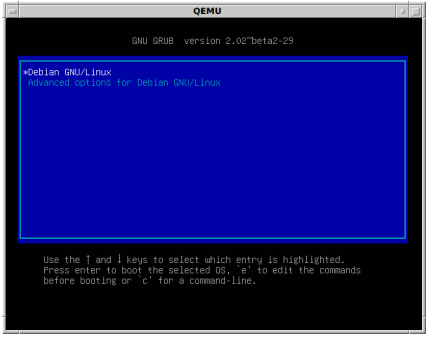
- Most ROM code also provide some sort of *recovery* mechanism, allowing to flash a board with no bootloader or a broken one, usually with a vendor-specific protocol over UART or USB.
- Often allows to push a new bootloader into RAM, making it possible to reflash the bootloader.
- Vendor-specific tools to run on the workstation
  - STM32MP1: STM32 Cube Programmer
  - NXP i.MX: uuu
  - Microchip AT91/SAM: SAM-BA
  - Allwinner: sunxi-fel
  - Some open-source, some proprietary
- Snagboot: new vendor agnostic tool replacing the above ones: https://github.com/bootlin/snagboot



Bootloaders

GRUB

- *Grand Unified Bootloader*, from the GNU project
- De-facto standard in most Linux distributions for x86 platforms
- Supports x86 legacy and UEFI systems
- Can read many filesystem formats to load the kernel image, modules and configuration
- Provides a menu and powerful shell with various commands
- Can load kernel images over the network
- Also supports ARM, ARM64, RISC-V, PowerPC, but less popular than other bootloaders on those platforms
- https://www.gnu.org/software/grub/



Syslinux

- For network and removable media booting (USB key, SD card, CD-ROM)
- `syslinux`: booting from FAT filesystem
- `pxelinux`: booting from the network
- `isolinux`: booting from CD-ROM
- `extlinux`: booting from numerous filesystem types
- A bit rustic to build and configure, not very actively maintained, but still useful for specific use-cases
- https://wiki.syslinux.org/
- https://kernel.org/pub/linux/utils/boot/syslinux/

systemd-boot

- Simple UEFI boot manager
- Useful alternative to GRUB for UEFI systems: simpler than GRUB
- Configured using files stored in the *EFI System Partition*
- Part of the *systemd* project, even though obviously distinct from *systemd* itself
  - See our slides later in this course for more details on *systemd*
- https://www.freedesktop.org/wiki/Software/systemd/systemd-boot/

shim

- Minimal UEFI bootloader
- Mainly used in secure boot scenario: it is signed by Microsoft and therefore successfully verified by UEFI firmware in the field
- Allows to chainload another bootloader (GRUB) or directly the Linux kernel, with signature checking
- https://github.com/rhboot/shim

- The de-facto standard and most widely used bootloader on embedded architectures: ARM, ARM64, RISC-V, PowerPC, MIPS, and more.
- Also supports x86 with UEFI firmware.
- Very likely the one provided by your SoC vendor, SoM vendor or board vendor for your hardware.
- We will study it in detail in the next section, and use it in all practical labs of this course.
- https://www.denx.de/wiki/U-Boot

Barebox

- Another bootloader for most embedded CPU architectures: ARM/ARM64, MIPS, PowerPC, RISC-V, x86, etc.
- Initially developed as an alternative to U-Boot to address some U-Boot shortcomings
  - *kconfig* for the configuration like the Linux kernel
  - well-defined *device model* internally
  - More Linux-style shell interface
  - Cleaner code base
- Actively maintained and developed, but
  - Less widely used than U-Boot
  - Less platform support than in U-Boot
- https://www.barebox.org/
- Talk *barebox Bells and Whistles*, by Ahmad Fatoum, ELCE 2020, video and slides

Trusted firmware

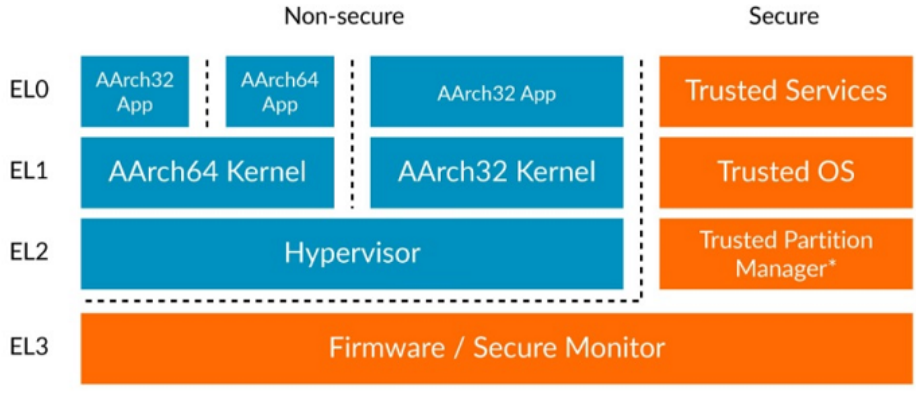
Concept

- Traditionally, bootloaders are only used during the booting process
  - Bootloader loads operating system, jumps to it, and is discarded
- Modern SoCs have advanced security mechanisms that require running some sort of *trusted firmware*
- This firmware is loaded by the bootloader, or part of the boot chain itself
- This *trusted firmware* **stays resident** after control has been passed to the OS
  - It is stored in a dedicated portion of the DDR, or some specific SRAM, inaccessible from the OS
  - It provides services to the OS, which the OS cannot perform directly
  - Can also be responsible for running a secure OS alongside the regular OS (Linux in our case)

ARM

- Modern ARMv7 and ARMv8 processors have
  - 4 privilege levels (*Exception Levels*)
    - EL3, the most priviledged, runs secure firmware
    - EL2, typically used by hypervisors, for virtualization
    - EL1, used to run the Linux kernel
    - EL0, used to run Linux user-space applications
  - 2 *worlds*
    - Normal world, used to run a general purpose OS, like Linux
    - Secure world, to run a separate, isolated, secure operating system and applications. Also called *TrustZone* by ARM.
- EL3 only exists in the secure world
- EL2 exists in both secure and normal worlds since ARMv8.4, before that EL2 was only in the normal world
- EL1 and EL0 exist in both secure and normal worlds

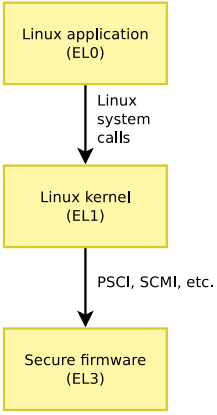
ARM exception levels and worlds



Source: ARM documentation

Interfaces with secure firmware

- Standardized by ARM
- Services
  - implemented by the secure firmware
  - called by the operating system
- Prevents the operating system running in normal world from directly accessing critical hardware resources
- PSCI, Power State Coordination Interface
  - Power management related: turn CPUs on/off, CPU idle state, platform shutdown/reset
- SCMI, System Control and Management Interface
  - Power domain, clocks, sensor, performance
- Secure firmware implementing these interfaces is
  - Mandatory to run Linux on ARMv8
  - Mandatory to run Linux on some ARMv7 platforms, but not all



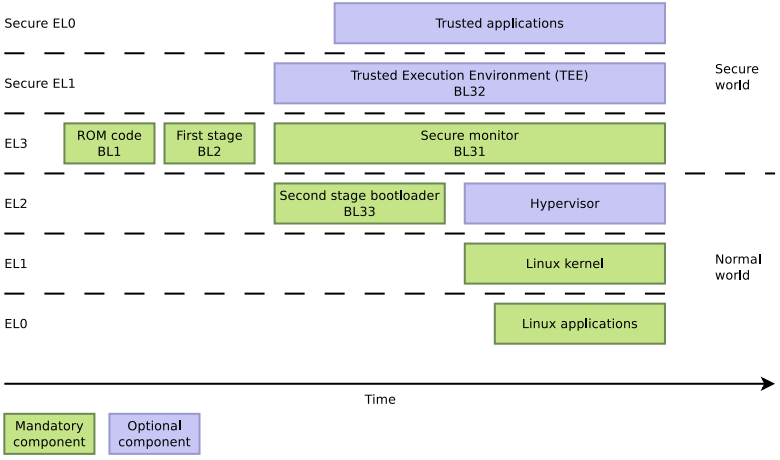
TF-A

- *Trusted Firmware-A (TF-A) provides a reference implementation of secure world software for Armv7-A and Armv8-A, including a Secure Monitor executing at Exception Level 3 (EL3)*
- Formerly known as *ATF*, for ARM Trusted Firmware
- Implements the various standard interfaces that operating systems need from the secure firmware
- Has drivers for the hardware blocks that are not accessed directly by Linux
- Needs to be ported for each SoC
- Depending on the platform, may also need to be ported per board: DDR initialization
- Used on the vast majority of ARMv8 platforms, and on a few recent ARMv7 platforms
- https://www.trustedfirmware.org/projects/tf-a/

Trusted OS, OP-TEE

- A trusted operating system can run in the *secure world*, also called *Trusted Execution Environment* or *TEE*
- Hardware partitioning between *secure world* and *normal world*
  - Some hardware resources only available in the *secure world*, by the trusted OS
- Allows to run trusted applications/services
  - isolated from Linux
  - can provide services to Linux applications
- Most common open-source implementation: *OP-TEE*
  - Supported by most silicon vendors
  - https://www.op-tee.org/

ARM: summary

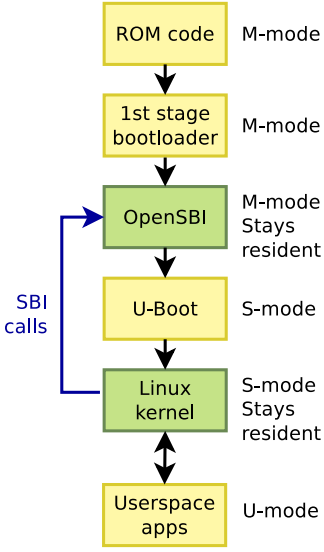


Largely inspired from *Ahmad Fatoum* presentation *From Reset Vector to Kernel*, slides, video

See also details about the ARM terms: BL1, BL2...

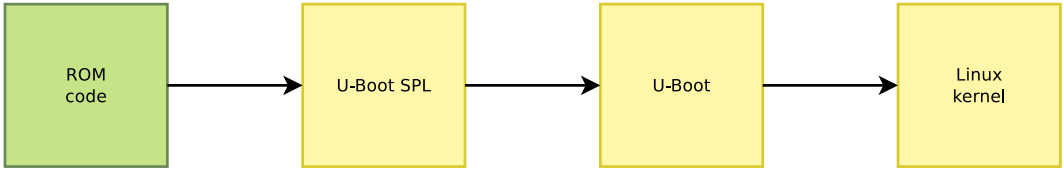
RISC-V

- Linux-class RISC-V processors have several privilege levels
  - M-mode: machine mode
  - S-mode: level at which the Linux kernel runs
  - U-mode: level at which Linux user-space applications run
- Some specific HW resources are not accessible in S-mode
- A more priviledged firmware runs in M-mode
- RISC-V has defined SBI, *Supervisor Binary Interface*
  - Standardized interface between the OS and the firmware
  - https://github.com/riscv-non-isa/riscv-sbi-doc
- OpenSBI is a reference, open-source implementation of SBI
  - https://github.com/riscv-software-src/opensbi

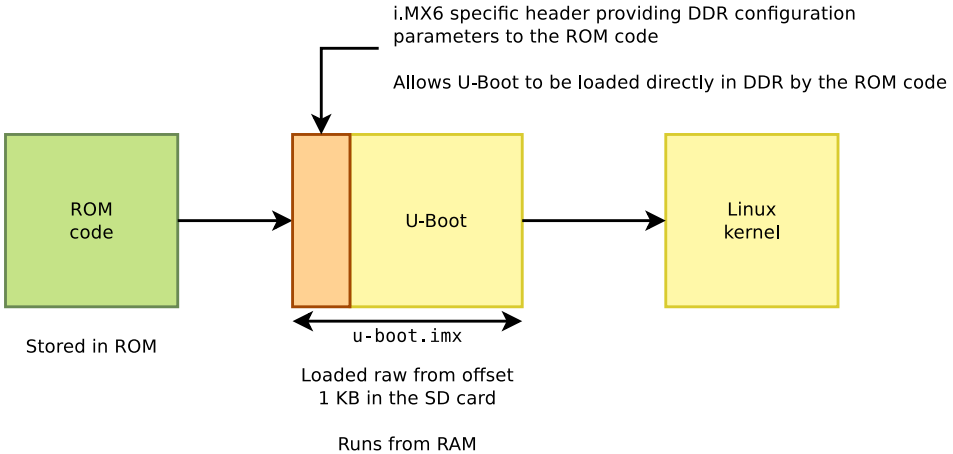


Example boot sequences on ARM

TI AM335x (32 bit BeagleBone): ARMv7

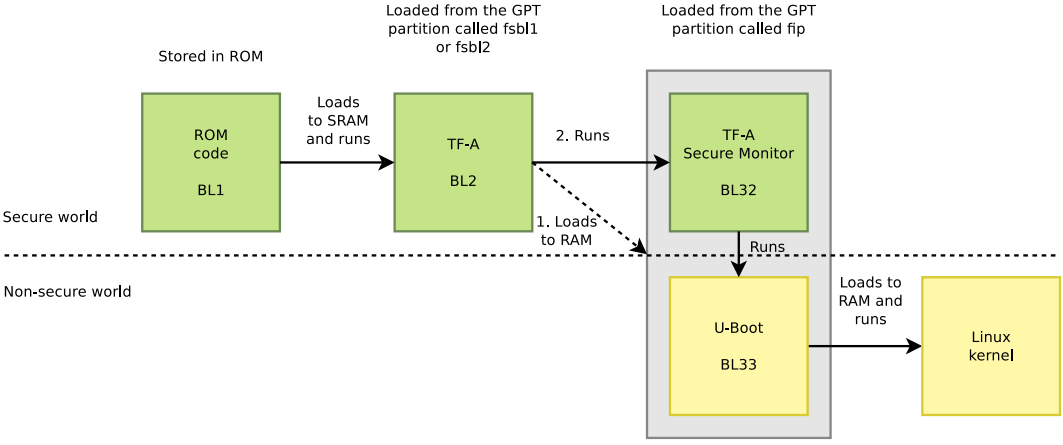


NXP i.MX6: ARMv7



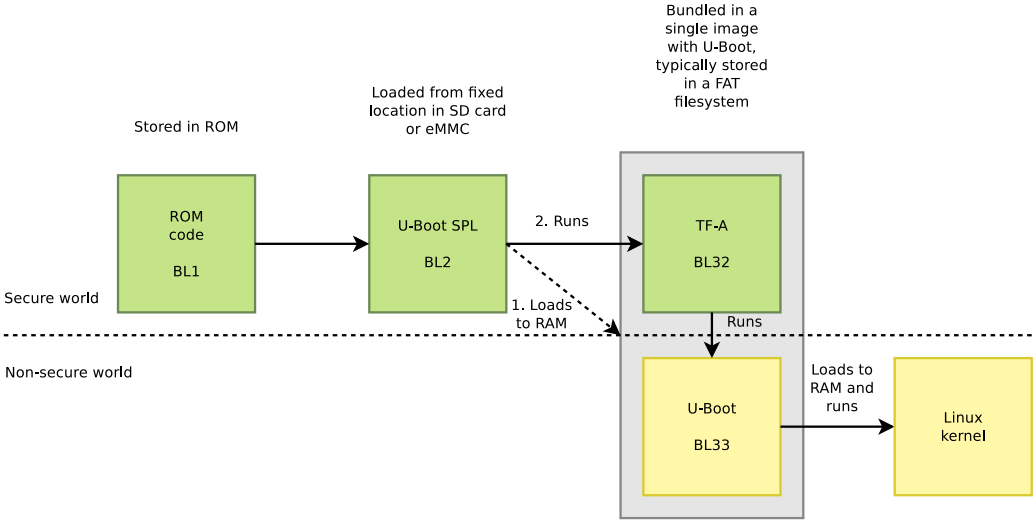
Note: this diagram shows one possible boot flow on NXP i.MX6, but it is also possible to use the U-Boot SPL → U-Boot boot flow on i.MX6.

STM32MP1: ARMv7

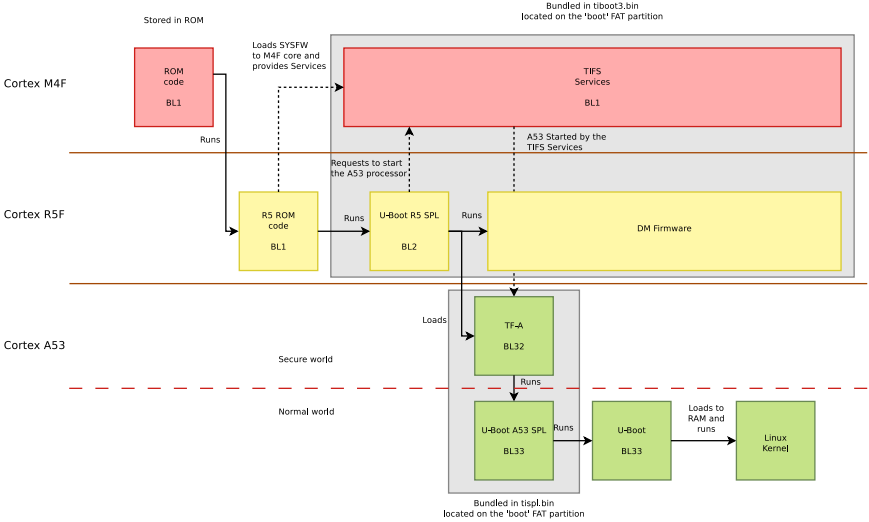


Note: booting with U-Boot SPL and U-Boot is also possible.

Allwinner ARMv8 cores



TI AM62x (BeaglePlay): ARMv7 and ARMv8 cores



See https://u-boot.readthedocs.io/en/latest/board/ti/am62x_sk.html for details.

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 source

Where to get U-Boot from?

- **Ideal:** your platform is supported directly by **upstream** U-Boot
  - Best quality → code reviewed and approved by the community
  - Long-term maintenance
  - Use directly U-Boot from https://gitlab.denx.de/u-boot/u-boot Git repository
- **Less ideal:** use a **fork** of U-Boot by your silicon vendor, system-on-module vendor or board vendor
  - Generally older, does not follow all upstream U-Boot updates
  - Changes not reviewed by the community → quality is often dubious
  - Check your HW vendor documentation/SDK
- If designing your own custom board
  - You will have to port U-Boot
  - If good support for your SoC in upstream U-Boot → use upstream U-Boot
  - If not → use the U-Boot fork from your SoC vendor

U-Boot configuration

- Configuration system based on *kconfig* from the Linux kernel
- The `configs/` directory contains configuration files for supported boards or platforms
  - There may be a single configuration supporting multiple boards based on the same processor
  - The configuration files defines all relevant options: CPU type, drivers needed, U-Boot features to compile in
  - Examples:
    - `configs/stm32mp15_basic_defconfig`
    - `configs/stm32mp15_trusted_defconfig`
- Note: migration to *kconfig* is still on-going
  - Not all boards have been converted to the new configuration system.
  - Many boards still have a combination of configuration settings in `include/configs/` header files, and configuration settings in `defconfig` files

U-Boot configuration file: stm32mp15_trusted_defconfig

```
CONFIG_ARM=y
CONFIG_ARCH_STM32MP=y
CONFIG_TFABOOT=y
CONFIG_SYS_MALLOC_F_LEN=0x3000
CONFIG_ENV_OFFSET=0x280000
CONFIG_ENV_SECT_SIZE=0x40000
CONFIG_DEFAULT_DEVICE_TREE="stm32mp157c-ev1"
[..]
CONFIG_CMD_ADTIMG=y
CONFIG_CMD_ERASEENV=y
CONFIG_CMD_NVEDIT_EFI=y
CONFIG_CMD_MEMINFO=y
CONFIG_CMD_MEMTEST=y
CONFIG_CMD_UNZIP=y
CONFIG_CMD_ADC=y
CONFIG_CMD_CLK=y
CONFIG_CMD_DFU=y
CONFIG_CMD_FUSE=y
CONFIG_CMD_GPIO=y
[...]
CONFIG_SPI=y
CONFIG_DM_SPI=y
CONFIG_STM32_QSPI=y
CONFIG_STM32_SPI=y
[...]
```

See the full file: configs/stm32mp15_trusted_defconfig

U-Boot configuration

- U-Boot must be configured before being compiled
- Configuration stored in a `.config` file
- Load a pre-defined configuration

  `$ 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.

U-Boot compilation

- The path to the cross-compiler must be specified in the `CROSS_COMPILE` variable
- `CROSS_COMPILE` contains the prefix common to all cross-compilation tools, e.g `arm-linux-`
- Common to add the cross-compiler location in `PATH` to keep the `CROSS_COMPILE` value short

```
$ export PATH=/path/to/toolchain/bin:$PATH
$ 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`...

Concept of U-Boot SPL

- To meet the need of a two-stage boot process, U-Boot has the concept of *U-Boot SPL*
- SPL = *Secondary Program Loader*
- The SPL is a stripped-down version of U-Boot, made small enough to meet the size constraints of a first stage bootloader
- Configured through *menuconfig*, one can define the subset of drivers to include
- No U-Boot shell/commands: the behavior is hardcoded in C code
- For some platforms: TPL, *Tertiary Program Loader*, an even more minimal first stage bootloader to do TPL → SPL → main U-Boot.

Device Tree in U-Boot

- The *Device Tree* is a data structure that describes the topology of the hardware
- Allows software to know which hardware peripherals are available and how they are connected to the system
- Initially mainly used by Linux, now also used by U-Boot, Barebox, TF-A, etc.
- Used by U-Boot on most platforms.
- Device Tree files located in `arch/ARCH/dts`
- One `.dts` for each board: need to create one if you build a custom board
- U-Boot *defconfigs* usually specify a default Device Tree, but it can be overridden using the `DEVICE_TREE` variable
- More details on the *Device Tree* later in this course.

U-Boot build example: TI AM335x BeagleBoneBlack wireless

- One *defconfig* file suitable for all AM335x platforms: `configs/am335x_evm_defconfig`
  - Yes its name looks like it supports only the EVM (EValuation Module) board
  - Contains `CONFIG_DEFAULT_DEVICE_TREE="am335x-evm"` → uses `arch/arm/dts/am335x-evm.dts` by default
- One *Device Tree* file describing the BeagleBoneBlack Wireless: `arch/arm/dts/am335x-boneblack-wireless.dts`
- Configure and build U-Boot

```
$ export PATH=/path/to/toolchain/bin:$PATH
$ make am335x_evm_defconfig
$ make DEVICE_TREE=am335x-boneblack-wireless CROSS_COMPILE=arm-linux-
```

- Produces:
  - `MLO`, the SPL, first-stage bootloader. Called MLO (*Mmc LOad*) as required on TI platforms.
  - `u-boot.img`, full U-Boot, second-stage bootloader

Installing U-Boot

1. If U-Boot is loaded from external storage, just update the binaries on such storage.
2. If U-Boot is loaded from internal storage (eMMC or NAND), you can update it using *Snagboot* (https://github.com/bootlin/snagboot) if it supports your SoC, or with the custom solution from the SoC vendor.
3. An alternative is to reflash internal storage with JTAG (if available), but that's more complicated and requires a JTAG probe.

U-boot shell 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.
- The U-Boot shell is not a Linux shell: commands are completely different from Linux ones.

```
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
=>
```

U-Boot help command

- `help` command to list all available commands
- The set of available commands depend on the U-Boot configuration
  - Many `CONFIG_CMD_*` options to enable commands at compile time
  - See *Command line interface* submenu in `menuconfig`
- `help <command>` for the complete help of one command

```
STM32MP> help
?         - alias for 'help'
adc       - ADC sub-system
adtimg    - manipulate dtb/dtbo Android image
base      - print or set address offset
[...]
usb       - USB sub-system
[...]

STM32MP> help usb
usb - USB sub-system

Usage:
usb start - start (scan) USB controller
usb reset - reset (rescan) USB controller
usb stop [f] - stop USB [f]=force stop
usb tree - show USB device tree
usb info [dev] - show available USB devices
[...]
```

U-Boot information commands

Version details: `version`

```
=> 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`

```
=> 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
```

MMC information: `mmc info`

```
=> mmc info
Device: STM32 SD/MMC
Manufacturer ID: 3
[...]
Capacity: 14.8 GiB
Bus Width: 4-bit
```

Board information: `bdinfo`

```
=> bdinfo
boot_params = 0x00000000
DRAM bank   = 0x00000000
-> start    = 0xc0000000
-> size     = 0x20000000
flashstart  = 0x00000000
flashsize   = 0x00000000
flashoffset = 0x00000000
baudrate    = 115200 bps
relocaddr   = 0xddb21000
reloc off   = 0x1da21000
[...]
fdt_blob    = 0xdbb01950
new_fdt     = 0xdbb01950
fdt_size    = 0x0001d540
Video       = display-controller@5a001000 inactive
[...]
```

- DRAM starts at `0xc0000000`, for a size of 512 MB (`0x20000000`).
- The end of the memory is used by U-Boot itself: `relocaddr` is the location of U-Boot in RAM.

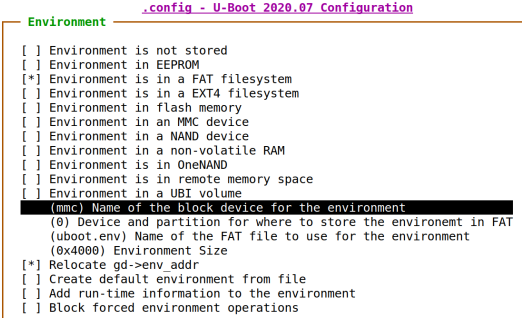
Concept of U-Boot environment

- A significant part of the U-Boot configuration happens at compile time: `menuconfig`
- U-Boot also has runtime configuration, through the concept of *environment variables*
- Environment variables are key/value pairs
  - Some specific environment variables impact the behavior of different U-Boot commands
  - Additional custom environment variables can be added, and used in *scripts*
- U-Boot environment variables are loaded and modified in RAM
- U-Boot has a default environment built into its binary
  - used when no other environment is found
  - defined in the configuration
  - the default environment is sometimes quite complex in some existing configurations
- The environment can be persistently stored in non-volatile storage

U-Boot environment persistent storage

Depending on the configuration, the U-Boot environment can be:

- 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 on a FAT or ext4 partition
- In a UBI volume
- Not stored at all, only the built-in environment in the U-Boot binary is used



U-Boot environment configuration menu

U-Boot environment commands

- `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>`
  Interactively edits the value of a variable, only in RAM
- After an `editenv` or `setenv`, changes in the environment are lost if they are not saved persistently
- `saveenv`
  Saves the current state of the environment to storage for persistence.
- `env` command, with many sub-commands: `env default`, `env info`, `env erase`, `env set`, `env save`, etc.

U-Boot environment 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
```

U-Boot memory allocation

- Many commands in U-Boot loading data into memory, or using data from memory, expect a RAM address as argument
- No built-in memory allocation mechanism → up to the user to know usable memory areas to load/use data
- Use the output of `bdinfo` to know the start address and size of RAM
- Avoid the end of the RAM, which is used by the U-Boot code and dynamic memory allocations
- Not the best part of the U-Boot design, sadly

U-Boot memory manipulation commands

- Commands to inspect or modify any memory location, useful for debugging, poking into hardware registers, etc.
- Addresses manipulated in U-Boot are directly physical addresses
- Memory display
  `md [.b, .w, .l, .q] address [# of objects]`
- Memory write
  `mw [.b, .w, .l, .q] address value [count]`
- Memory modify (modify memory contents interactively starting from address)
  `mm [.b, .w, .l, .q] address`

U-Boot raw storage commands

U-Boot can manipulate raw storage devices:

- NAND flash
  - `nand info`
  - `nand read <addr> <off|partition> <size>`
  - `nand erase [<off> [<size>]]`
  - `nand write <addr> <off|partition> <size>`
  - More: `help nand`
- MMC
  - `mmc info`
  - `mmc read <addr> <blk#> <cnt>`
  - `mmc write <addr> <blk#> <cnt>`
  - `mmc part` to show partition table
  - `mmc dev` to show/set current MMC device
  - More: `help mmc`
- USB storage
  - `usb info`
  - `usb read <addr> <blk#> <cnt>`
  - `usb write <addr> <blk#> <cnt>`
  - `usb part`
  - `usb dev`
  - More: `help usb`

Note: `<addr>` are addresses in RAM where data is stored

U-Boot commands example

List partitions on MMC

```
STM32MP> mmc part
Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000022      0x000001d3      "fsbl1"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   72c63477-c475-4cf7-988e-b763bce4604e
  2     0x000001d4      0x00000385      "fsbl2"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   66d616db-de56-4a1e-9b13-9b1a5a6e360f
  3     0x00000386      0x00001385      "fip"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   6251ecf7-d985-4d81-a396-7a6b6fab8b7c
  [...]
```

Read block 0x22 from MMC to RAM 0xc0000000

```
STM32MP> mmc read c0000000 22 1

MMC read: dev # 0, block # 34, count 1 ... 1 blocks read: OK
```

Dump memory at 0xc00000000

```
STM32MP> md c0000000
c0000000: 324d5453 00000000 00000000 00000000  STM2............
c0000010: 00000000 00000000 00000000 00000000  ................
c0000020: 00000000 00000000 00000000 00000000  ................
c0000030: 00000000 00000000 00000000 00000000  ................
```

U-Boot filesystem storage commands

- U-Boot has support for many filesystems
  - The exact list of supported filesystems depends on the U-Boot configuration
- Per-filesystem commands
  - FAT: `fatinfo`, `fatls`, `fatsize`, `fatload`, `fatwrite`
  - ext2/3/4: `ext2ls`, `ext4ls`, `ext2load`, `ext4load`, `ext4size`, `ext4write`
  - Squashfs: `sqfsls`, `sqfsload`
- "New" generic commands, working for all filesystem types
  - Load a file:
    `load <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]]]]]`
  - List files: `ls <interface> [<dev[:part]> [directory]]`
  - Get the size of a file: `size <interface> <dev[:part]> <filename>`
    (result stored in `filesize` environment variable)
  - `interface`: `mmc`, `usb`
  - `dev`: device number, `0` for first device, `1` for second device
  - `part`: partition number

U-Boot filesystem command example

List files

```
STM32MP> ls mmc 0:4
<DIR>       1024 .
<DIR>       1024 ..
<DIR>      12288 lost+found
<DIR>       2048 bin
<DIR>       1024 boot
<DIR>       1024 dev
<DIR>       1024 etc
[...]

STM32MP> ls mmc 0:4 /etc
<DIR>       1024 .
<DIR>       1024 ..
             209 asound.conf
<DIR>       1024 fonts
             334 fstab
             347 group
[...]
```

Load file

```
STM32MP> load mmc 0:4 c0000000 /etc/fstab
334 bytes read in 143 ms (2 KiB/s)
```

Show file contents

```
STM32MP> md c0000000
c0000000: 663c2023 20656c69 74737973 093e6d65  # <file system>.
c0000010: 756f6d3c 7020746e 3c093e74 65707974  <mount pt>.<type
c0000020: 6f3c093e 6f697470 093e736e 6d75643c  >.<options>.<dum
c0000030: 3c093e70 73736170 642f0a3e 722f7665  p>.<pass>./dev/r
c0000040: 09746f6f 6509092f 09327478 6e2c7772  oot./..ext2.rw,n
[...]
```

U-Boot networking

- Environment variables
  - `ethaddr`: MAC address
  - `ipaddr`: IP address of the board
  - `serverip`: IP address of the server for network related commands
- Important commands
  - `ping`: ping a destination machine. Note: U-Boot is not an operating system with multitasking/interrupts, so ping from another machine to U-Boot cannot work.
  - `tftp`: load a file using the TFTP protocol
  - `dhcp`: get an IP address using DHCP

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 `/srv/tftp` 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`

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'`

U-Boot booting commands

- Commands to boot a Linux kernel image
  - `bootz` → boot a compressed ARM32 `zImage`
  - `booti` → boot an uncompressed ARM64 or RISC-V `Image`
  - `bootm` → boot a kernel image with legacy U-Boot headers
  - `zboot` → boot a compressed x86 `bzImage`
- `bootz [addr [initrd[:size]] [fdt]]`
  - `addr`: address of the kernel image in RAM
  - `initrd`: address of the *initrd* or *initramfs*, if any. Otherwise, must pass -
  - `fdt`: address of the *Device Tree* passed to the Linux kernel
- Important environment variables
  - `bootcmd`: list of commands executed automatically by U-Boot after the count down
  - `bootargs`: Linux kernel command line

U-Boot booting example

Load kernel image and Device Tree

```
STM32MP> ls mmc 0:4 /boot
<DIR>       1024 .
<DIR>       1024 ..
         117969 stm32mp157c-dk2.dtb
        7538376 zImage

STM32MP> load mmc 0:4 c2000000 /boot/zImage
7538376 bytes read in 463 ms (15.5 MiB/s)

STM32MP> load mmc 0:4 c4000000 /boot/stm32mp157c-dk2.dtb
117969 bytes read in 148 ms (778.3 KiB/s)
```

Set kernel command line and boot

```
STM32MP> setenv bootargs root=/dev/mmcblk0p4 rootwait

STM32MP> bootz c2000000 - c4000000
Kernel image @ 0xc2000000 [ 0x000000 - 0x7306c8 ]
## Flattened Device Tree blob at c4000000
   Booting using the fdt blob at 0xc4000000
   Loading Device Tree to cffe0000, end cffffcd0 ... OK
[...]
```

FIT image

- U-Boot has a concept of **FIT** image
- FIT = *Flat Image Tree*
- Container format that allows to bundle multiple images into one
  - Multiple kernel images
  - Multiple Device Trees
  - Multiple initramfs
  - Any other image: FPGA bitstream, etc.
- Typically useful for secure booting and to ensure binaries don't overlap in memory.
- Interestingly, relies on the *Device Tree Compiler*
  - `.its` file describes the contents of the image
  - Device Tree Compiler compiles it into an `.itb`
- U-Boot can load an `.itb` image and use its different elements
- https://www.thegoodpenguin.co.uk/blog/u-boot-fit-image-overview/

Generic Distro boot (1)

- Each board/platform used to have its own U-Boot environment, with custom variables/commands
- Wish to standardize the behavior of bootloaders, including U-Boot
- *Generic Distro boot* concept
- If enabled, at boot time, U-Boot:
  - Can be asked to locate a bootable partition (`part list` command), as defined by the bootable flag of the partition table
  - With the `sysboot` command, will look for a `/extlinux/extlinux.conf` or `/boot/extlinux/extlinux.conf` file describing how to boot, and will offer a prompt in the console to choose between available configurations.
  - Once a configuration is selected, will load and boot the corresponding kernel, device tree and initramfs images.
  - Example `bootcmd`:
    `part list mmc 0 -bootable bootpart; sysboot mmc 0:${bootpart} any`
- https://u-boot.readthedocs.io/en/latest/develop/distro.html

Generic Distro boot (2)

Several environment variables need to be set:

- `kernel_addr_r`: address in RAM to load the kernel image
- `ramdisk_addr_r`: address in RAM to load the initramfs image (if any)
- `fdt_addr_r`: address in RAM to load the DTB (Flattened Device Tree)
- `pxefile_addr_r`: address in RAM to load the configuration file (usually `extlinux.conf`)
- `bootfile`: the path to the configuration file, for example `/boot/extlinux/extlinux.conf`

Example /boot/extlinux/extlinux.conf

```
label stm32mp157c-dk2-buildroot
  kernel /boot/zImage
  devicetree /boot/stm32mp157c-dk2.dtb
  append root=/dev/mmcblk0p4 rootwait
```

U-Boot boot log

```
Hit any key to stop autoboot:  0
Boot over mmc0!
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:4...
Found /boot/extlinux/extlinux.conf
Retrieving file: /boot/extlinux/extlinux.conf
131 bytes read in 143 ms (0 Bytes/s)
1:       stm32mp157c-dk2-buildroot
Retrieving file: /boot/zImage
7538376 bytes read in 462 ms (15.6 MiB/s)
append: root=/dev/mmcblk0p4 rootwait
Retrieving file: /boot/stm32mp157c-dk2.dtb
117969 bytes read in 148 ms (778.3 KiB/s)
Kernel image @ 0xc2000000 [ 0x000000 - 0x7306c8 ]
## Flattened Device Tree blob at c4000000
   Booting using the fdt blob at 0xc4000000
   Loading Device Tree to cffe0000, end cffffcd0 ... OK

Starting kernel ...
```

TF-A: Trusted Firmware

Concept of FIP

- FIP = *Firmware Image Package*
- Concept specific to TF-A
- *packaging format used by TF-A to package firmware images in a single binary*
- Typically used to bundle the BL33, i.e. the U-Boot bootloader that will be loaded by TF-A.
- https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/tools-build.html
- https://wiki.st.com/stm32mpu/wiki/How_to_configure_TF-A_FIP

Configuring TF-A

- TF-A does not use *Kconfig* for configuration
- All the configuration is based on variables passed on the `make` command line
- Most variables are documented at: https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/build-options.html

Configure TF-A: important variables

- `CROSS_COMPILE`, cross-compiler prefix
- `ARCH`, CPU architecture: `aarch32` or `aarch64`
- `ARM_ARCH_MAJOR`, `7` for ARMv7, `8` for ARMv8
- `PLAT`, SoC family, any directory name in `plat` that contains `platform.mk`
- `AARCH32_SP`, the Secure Payload, specific to ARMv7. Either OP-TEE or the built-in *SP-MIN* provided by TF-A
- `DTB_FILE_NAME`, path to the Device Tree describing our board
- `BL33`, path to the second stage bootloader, usually U-Boot, to include in the FIP image
- Specific to STM32MP1
  - `BL33_CFG`, path to the U-Boot Device Tree
  - `STM32MP_SDMMC=1`, enable support for SD card/eMMC in TF-A

Building TF-A for STM32MP1

```
$ make CROSS_COMPILE=arm-linux- \
       ARM_ARCH_MAJOR=7 \
       ARCH=aarch32 \
       PLAT=stm32mp1 \
       AARCH32_SP=sp_min \
       DTB_FILE_NAME=stm32mp157a-dk1.dtb \
       BL33=/path/to/u-boot/u-boot-nodtb.bin \
       BL33_CFG=/path/to/u-boot/u-boot.dtb \
       STM32MP_SDMMC=1 \
       fip all
```

Build results in `build/stm32mp1/release`. Important files:

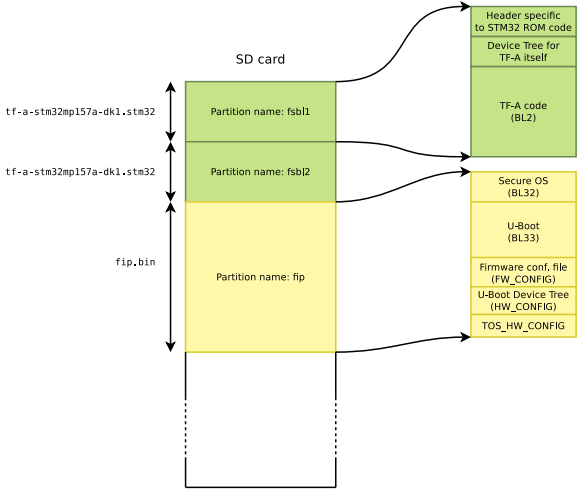
- `tf-a-stm32mp157a-dk1.stm32`, TF-A itself
- `fip.bin`, the FIP image, containing U-Boot and other elements

FIP image contents

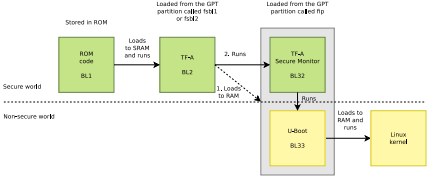
fiptool info

```
$ ./tools/fiptool/fiptool info build/stm32mp1/release/fip.bin
Secure Payload BL32 (Trusted OS): offset=0x100, size=0x8AEC, cmdline="--tos-fw"
Non-Trusted Firmware BL33: offset=0x8BEC, size=0xECE6C, cmdline="--nt-fw"
FW_CONFIG: offset=0xF5A58, size=0x226, cmdline="--fw-config"
HW_CONFIG: offset=0xF5C7E, size=0x16A98, cmdline="--hw-config"
TOS_FW_CONFIG: offset=0x10C716, size=0x3CF6, cmdline="--tos-fw-config"
```

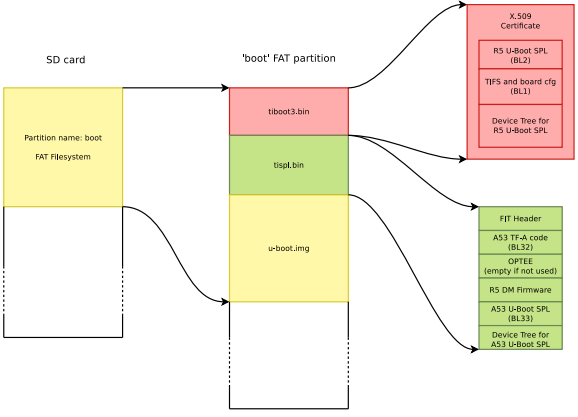
STM32MP1 partition layout



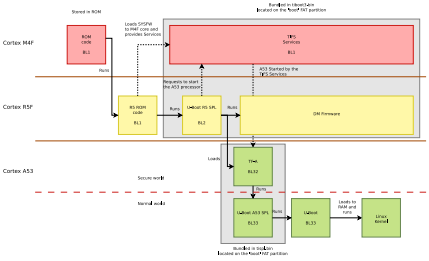
Reminder: boot sequence with TF-A on STM32MP1



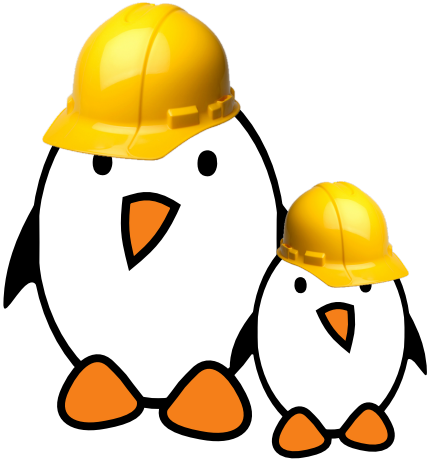
AM62x (BeaglePlay) partition layout



Reminder: boot sequence with TF-A on AM62x



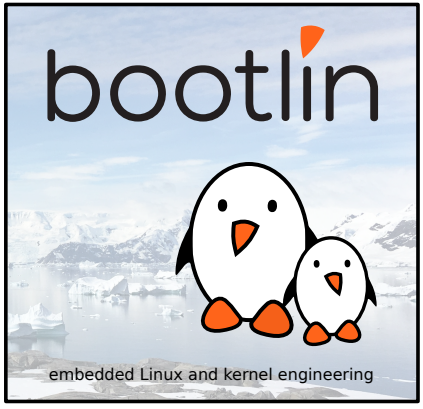
Practical lab - U-Boot / TF-A



Time to start the practical lab!

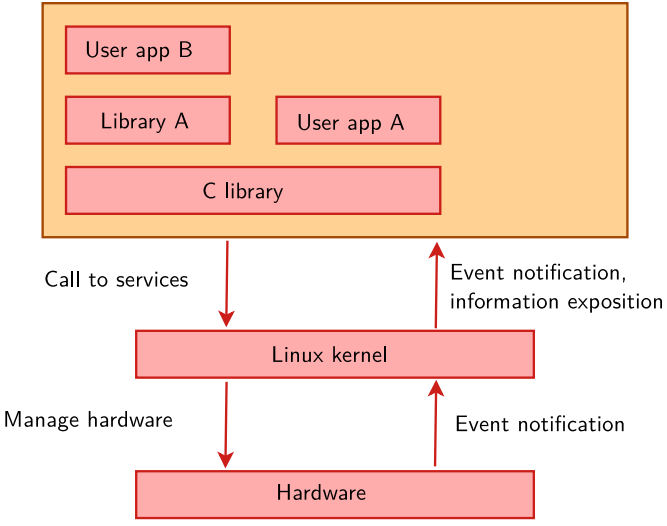
- Communicate with the board using a serial console
- Configure, build and install the bootloader stages:
  - *TF-A* and *U-Boot* on STM32MP1 and Beagleplay
  - *U-Boot SPL* and *U-Boot* on BeagleBoneBlack and QEMU
- Learn *U-Boot* commands
- Set up *TFTP* communication with the host

Linux kernel introduction



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

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

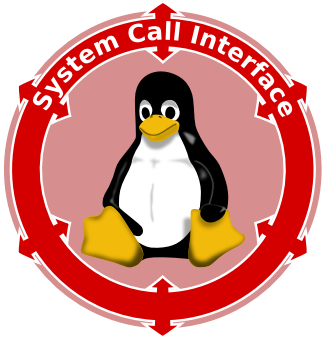


Image credits (Wikipedia): https://bit.ly/2U2rdGB

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.

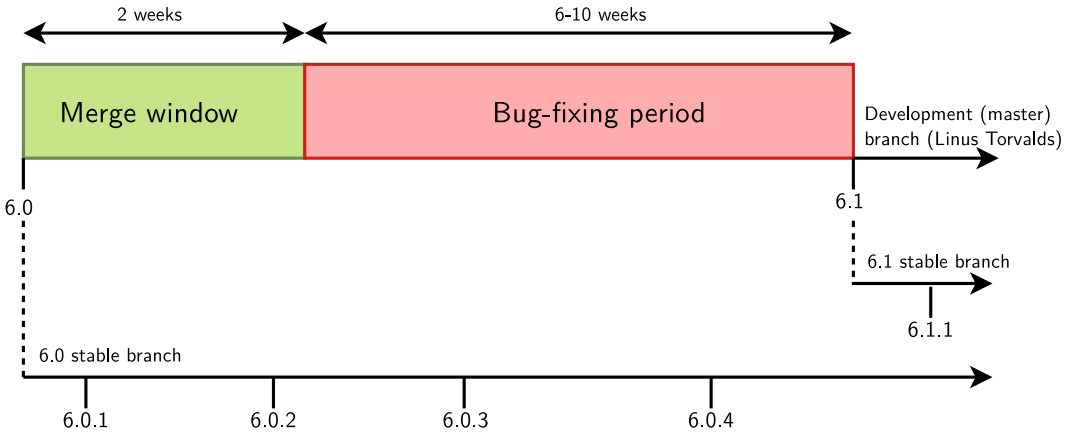
Linux versioning scheme and development process

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)
  - Versions `5.0` (Mar. 2019) to `5.19` (July 2022)
  - Version `6.0` was released in Oct. 2022.
- 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: 6.0.1, 6.0.2, etc.

Linux development model

Using merge and bug fixing windows



Need for long term support (1)

- Issue: bug and security fixes only released for most recent 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 at least 2 years.

Longterm release kernels

Version	Maintainer	Released	Projected EOL
6.6	Greg Kroah-Hartman & Sasha Levin	2023-10-29	Dec, 2026
6.1	Greg Kroah-Hartman & Sasha Levin	2022-12-11	Dec, 2026
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Dec, 2026
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

Captured on `https://kernel.org` in Nov. 2023, following the *Releases* link.

- Example at Google: starting from *Android O (2017)*, all new Android devices 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 and 6.1) on selected architectures. See https://wiki.linuxfoundation.org/civilinfrastructureplatform/start.

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 (`master` branch)
  - 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 updated 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

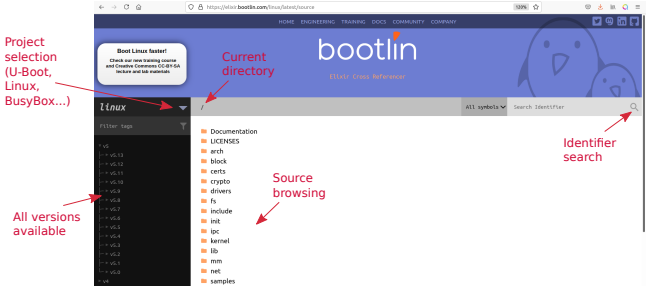
Getting Linux sources

- 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 has 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!

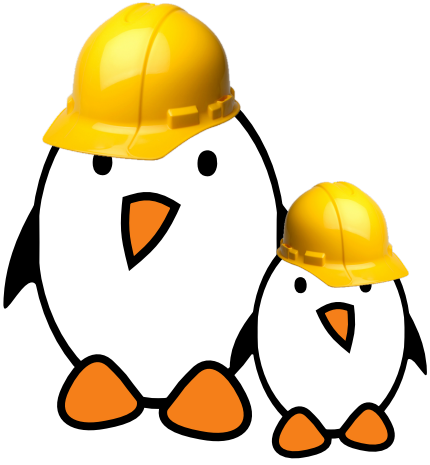


Linux kernel size and structure

- Linux v5.18 sources: close to 80k files, 35M lines, 1.3GiB
- But a compressed Linux kernel just sizes a few megabytes.
- So, why are these sources so big?
  Because they include numerous device drivers, network protocols, architectures, filesystems... The core is pretty small!
- As of kernel version v5.18 (in percentage of total number of lines):

- drivers/: 61.1%
- arch/: 11.6%
- fs/: 4.4%
- sound/: 4.1%
- tools/: 3.9%
- net/: 3.7%
- include/: 3.5%
- Documentation/: 3.4%
- kernel/: 1.3%
- lib/: 0.7%
- usr/: 0.6%
- mm/: 0.5%
- scripts/, security/, crypto/, block/, samples/, ipc/, virt/, init/, certs/: <0.5%
- Build system files: Kbuild, Kconfig, Makefile
- Other files: COPYING, CREDITS, MAINTAINERS, README

Practical lab - Fetching Linux kernel sources



- Clone the mainline Linux tree
- Accessing stable releases

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/`:
  `ARCH=arm` or `ARCH=arm64` or `ARCH=riscv`, etc
- 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
  - Specify the prefix of your cross-compiler executable, for example for `arm-linux-gnueabi-gcc`: `CROSS_COMPILE=arm-linux-gnueabi-`

Set `LLVM` to `1` to compile your kernel with Clang.
See our LLVM tools for the Linux kernel presentation.

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, see also the https://direnv.net/ project.

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

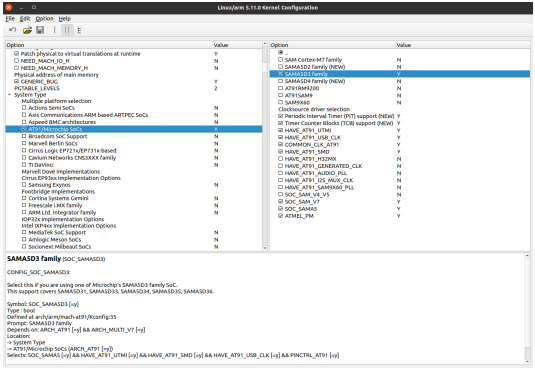
- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `CONFIG_PARAM=value`
  - Options are grouped by sections and are prefixed with `CONFIG_`
  - Included by the top-level kernel Makefile
  - Typically not edited by hand because of the dependencies

```
#
# 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`

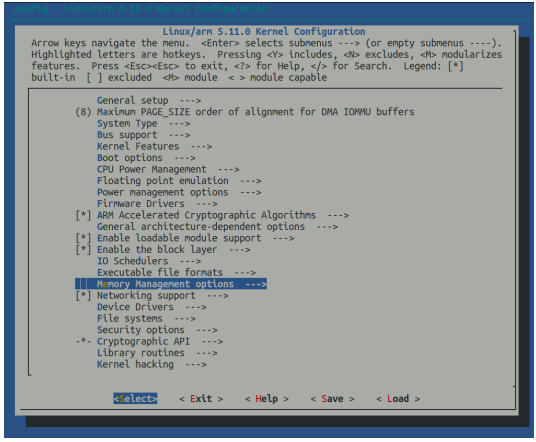
- A graphical interface to configure the kernel.
- File browser: easy to load configuration files
- Search interface to look for parameters (`[Ctrl] + [f]`)
- Required Debian/Ubuntu packages: `qtbase5-dev` on Ubuntu 22.04



menuconfig

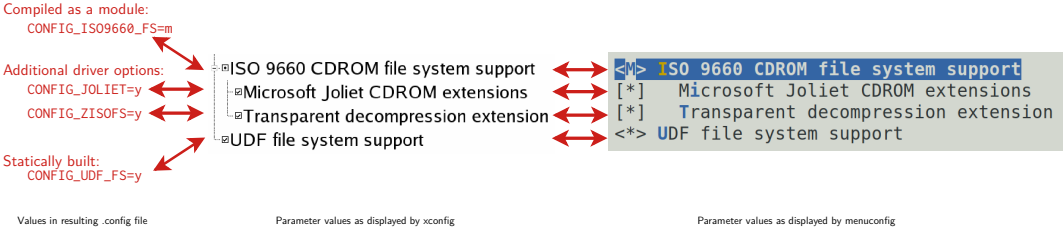
`make menuconfig`

- Useful when no graphics are available. Very efficient interface.
- Same interface found in other tools: BusyBox, Buildroot...
- Convenient number shortcuts to jump directly to search results.
- Required Debian/Ubuntu packages: `libncurses-dev`
- Alternative: `make nconfig` (now also has the number shortcuts)



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.

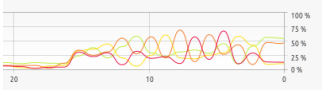
Compiling and installing the kernel

Kernel compilation

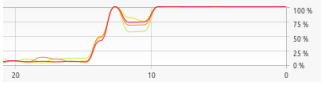
`make`

- Only works from the top kernel source directory
- Should not be performed as a priviledged user
- Run several **j**obs 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-"`

Tests on Linux 5.11 on arm
configuration
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
  - `modules.dep`, `modules.dep.bin`
    Module dependencies. Kernel modules can depend on other modules, based on the symbols (functions and data structures) they use.
  - `modules.symbols`, `modules.symbols.bin`
    Tells which module a given symbol belongs to (related to module dependencies).
  - `modules.builtin`
    List of built-in modules of the kernel.

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`

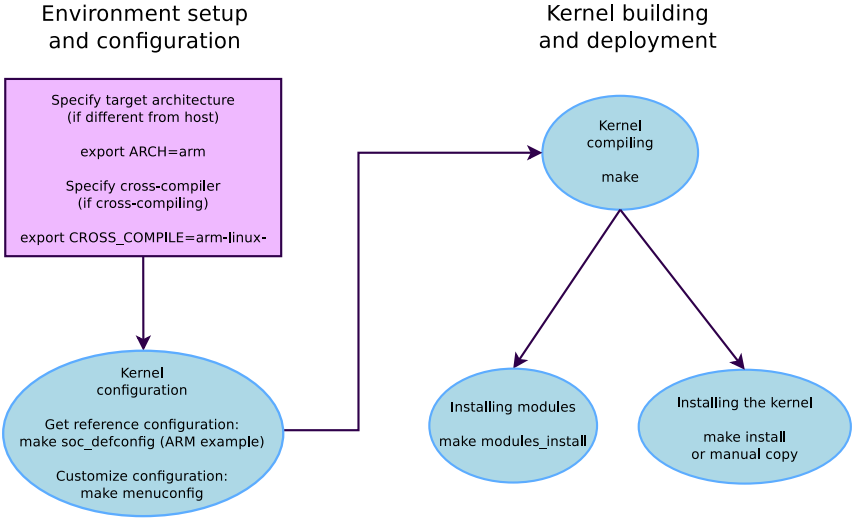
Kernel cleanup targets

- 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

Hardware description

- Many embedded architectures have a lot of non-discoverable hardware (serial, Ethernet, I2C, Nand flash, USB controllers...)
- This hardware needs to be described and passed to the Linux kernel.
- Using C code directly within the kernel is legacy, nowadays the bootloader/firmware is expected to provide this description when starting the kernel:
  - On x86: using ACPI tables
  - On most embedded devices: using an OpenFirmware Device Tree (DT)
- 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!

- Kernel developers write *Device Tree Sources (DTS)*, which become *Device Tree Blobs (DTB)* once compiled.
- There is one different Device Tree for each board/platform supported by the kernel, available in `arch/<arch>/boot/dts/<vendor>/<board>.dtb` (`arch/arm/boot/dts/<board>.dtb` on ARM 32 before Linux 6.5).
- As a board user, you may have legitimate needs to customize your board device tree:
  - To describe external devices attached to non-discoverable busses 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)

Booting with U-Boot

- On ARM32, U-Boot can boot `zImage` (`bootz` command)
- On ARM64 or RISC-V, it boots the `Image` file (`booti` command)
- 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 `boot[z|i] X - Y`
     The `-` in the middle indicates no *initramfs*

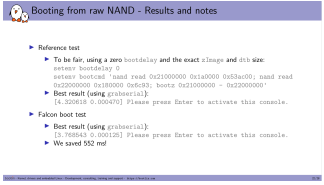
Kernel command line

- 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 contents 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.

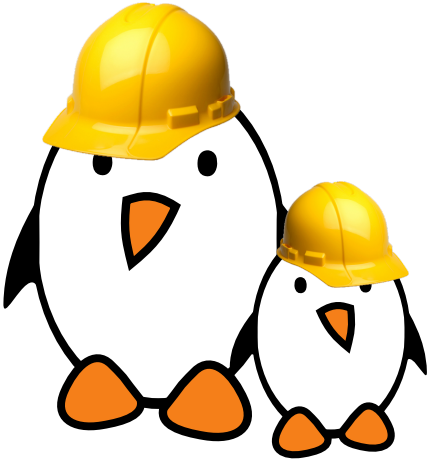


Slides: https://bootlin.com/pub/conferences/2021/lee/
Video: https://www.youtube.com/watch?v=LFe3x2QMhSo

Kernel log

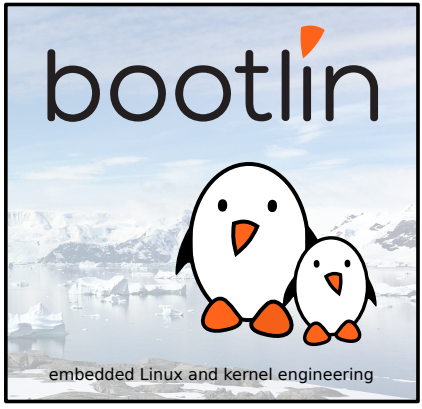
- The kernel keeps its messages in a circular buffer in memory
  - The size is configurable using `CONFIG_LOG_BUF_SHIFT`
- When a module is loaded, related information is available in the kernel log.
- Kernel log messages are available through the `dmesg` command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed on the console pointed by the `console=` kernel command line argument
  - Console messages can be filtered by level using the `loglevel` parameter:
    - `loglevel=` allows to filter messages displayed on the console based on priority
    - `ignore_loglevel` (same as `loglevel=8`) will lead to all messages being printed
    - `quiet` (same as `loglevel=0`) prevents any message from being displayed on the console
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 loglevel=5`
- It is possible to write to the kernel log from user space:
  `echo "<n>Debug info" > /dev/kmsg`

Practical lab - Kernel cross-compiling



- Configuring the Linux kernel and cross-compiling it for the embedded hardware platform.
- Downloading your kernel on the board through U-boot's TFTP client.
- Booting your kernel.
- Automating the kernel boot process with U-Boot scripts.

Linux Root Filesystem



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

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=`

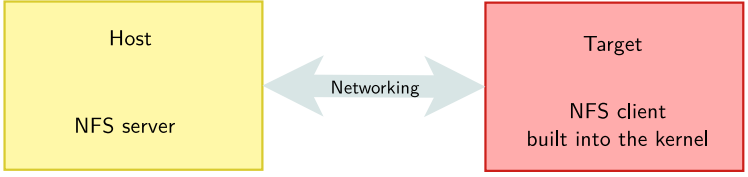
Mounting rootfs from storage devices

- 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)

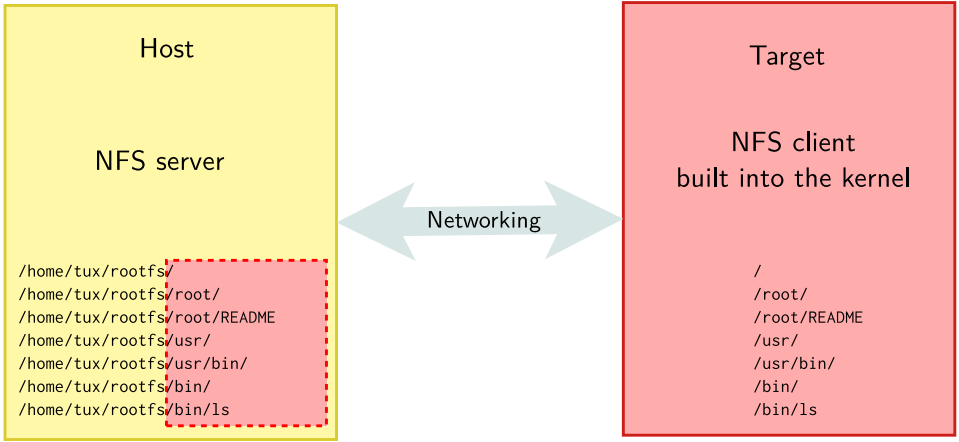
On the development workstation side, a NFS server is needed

- 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_ROOT_NFS=y` (support for NFS as rootfs)
  - `CONFIG_IP_PNP=y` (configure IP at boot time)
- 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 are not part of the kernel image (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
```

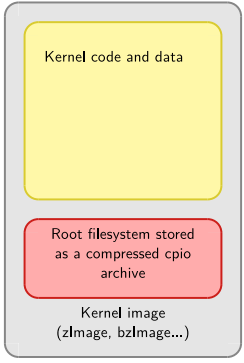
Built-in initramfs

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
- /run Run-time variable data (previously /var/run)
- /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/.

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/iomem`, `/proc/cpuinfo` 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

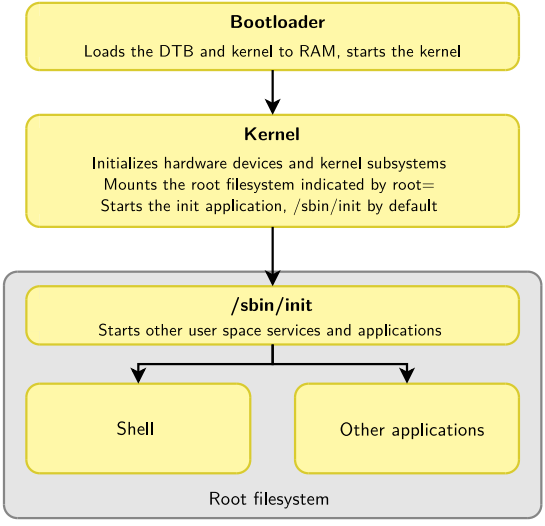
- It allows to represent in user space the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various user space applications that need to list and query the available hardware, for example `udev` or `mdev` (see later)
- All applications using sysfs expect it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

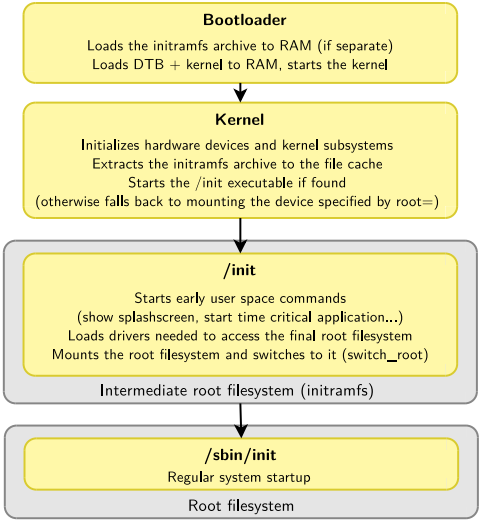
Basic applications

- 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`, `/etc/init`, `/bin/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 whose parent terminate 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

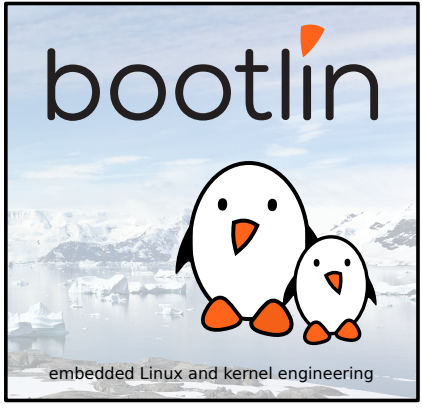
Overall booting process



Overall booting process with initramfs



BusyBox



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

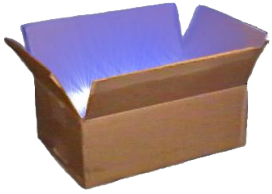
Why BusyBox?

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

General purpose toolbox: BusyBox

https://www.busybox.net/

- Rewrite of many useful UNIX command line utilities
  - Created in 1995 to implement a rescue and installer system for Debian, fitting in a single floppy disk (1.44 MB)
  - Integrated into a single project, which makes it easy to work with
  - Great for embedded systems: 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

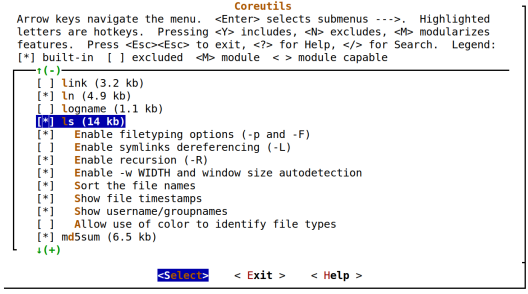
Configuring BusyBox

- 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`
  `-> Destination path for 'make install'`
- 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

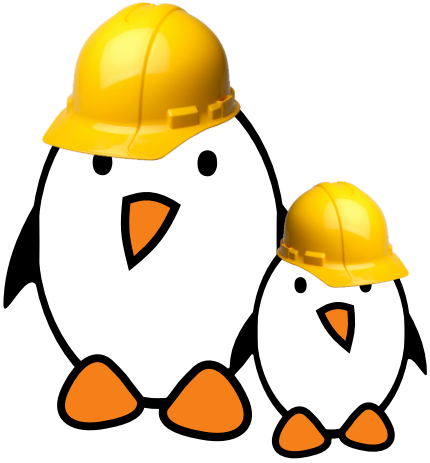
- BusyBox provides an implementation of an `init` program
- Simpler than the init implementation found on desktop/server systems (*SysV init* or *systemd*)
- A single configuration file: `/etc/inittab`
  - Each line has the form `<id>::<action>:<process>`
- Allows to start system services at startup,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

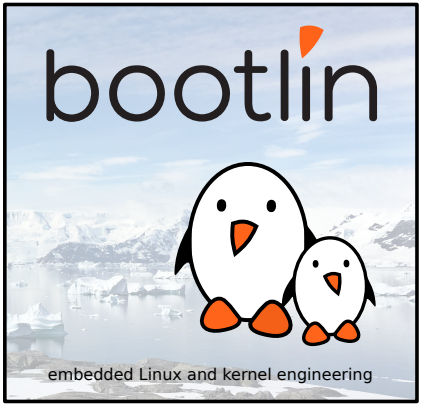
```
[ ] cmp (4.9 kb)
[ ] diff (13 kb)
[ ] ed (21 kb)
[ ] patch (9.4 kb)
[ ] sed (12 kb)
[*] vi (23 kb)
(4096) Maximum screen width
[*]   Allow to display 8-bit chars (otherwise shows dots)
[*]   Enable ":" colon commands (no "ex" mode)
[*]   Enable yank/put commands and mark cmds
[*]   Enable search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



- Setting up a kernel to boot your system on a workstation directory exported by NFS
- Passing kernel command line parameters to boot on NFS
- Creating the full root filesystem from scratch. Populating it with BusyBox based utilities.
- System startup using BusyBox `init`
- Using the BusyBox HTTP server.
- Controlling the target from a web browser on the PC host.
- Setting up shared libraries on the target and compiling a sample executable.

Accessing hardware devices



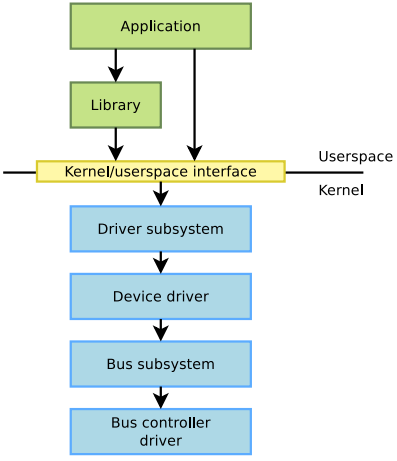
© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Kernel drivers

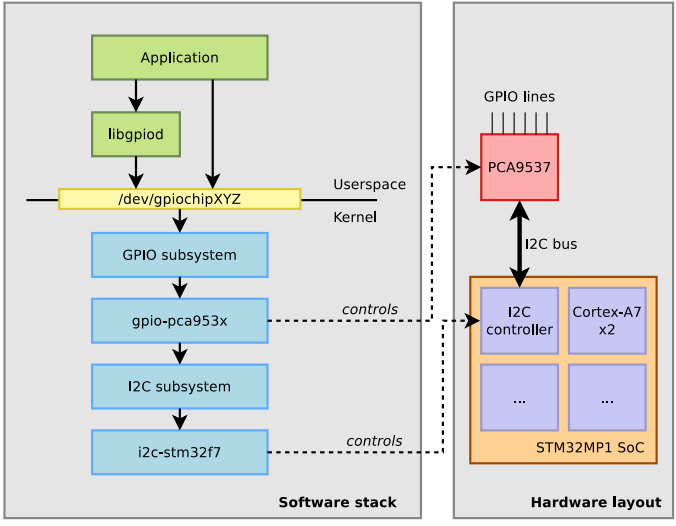
Typical software stack for hardware access

From the bottom to the top:

- A *bus controller driver* in the kernel drives an I2C, SPI, USB, PCI controller
- A *bus subsystem* provides an API for drivers to access a particular type of bus: I2C, SPI, PCI, USB, etc.
- A *device driver* in the kernel drives a particular device connected to a given bus
- A *driver subsystem* exposes features of certain class of devices, through a standard *kernel/user-space interface*
- An application can access the device through this standard *kernel/user-space interface* either directly or through a library.



Stack illustrated with a GPIO expander



Standardized user-space interface

- Strong advantage of kernel drivers: they expose a standard *kernel to user-space interface*
- All devices of the same class (e.g GPIO controllers) will expose the same *kernel to user-space interface*
- Applications don't have to know the details of the GPIO controller, they just need to know the standard user-space interface valid for all GPIO controllers
- Applications can use existing open-source libraries that leverage this standard user-space interface
- Such kernel drivers can also be used internally inside the kernel, for example if one driver needs to control a GPIO directly (reset signal, interrupt signal, etc.)

Numerous kernel subsystems for device classes

- Networking stack for Ethernet, WiFi, CAN, 802.15.4, etc.
- GPIO
- Video4Linux for camera, video encoders/decoders
- DRM for display controllers, GPU
- ALSA for audio
- IIO for ADC, DAC, gyroscopes, sensors, and more
- MTD for flash memory
- PWM
- Input for keyboard, mouse, touchscreen, joystick
- Watchdog
- RTC for real-time clocks
- remoteproc for auxilliary processors
- crypto for cryptographic accelerators
- hwmon for hardware monitoring sensors
- block layer for block storage

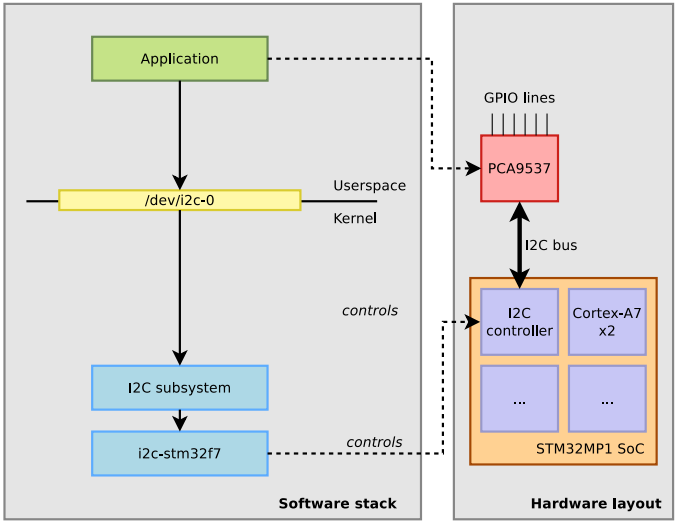
and many more

Accessing devices directly from user-space

- Even though device drivers in the kernel are preferred, it is also possible to access devices directly from user-space
- Especially useful for very specific devices that do not fit in any existing kernel subsystems
- The kernel provides the following mechanisms, depending on the bus:
  - I2C: i2c-dev
  - SPI: spidev
  - Memory-mapped: UIO
  - USB: `/dev/bus/usb`, through libusb
  - PCI: sysfs entries for PCI

Accessing devices directly from user-space: GPIO example

This diagram shows what's not recommended to do → for a GPIO controller, a kernel driver is preferred



What can go wrong with a user-space driver?

- You write your GPIO driver in user-space: other kernel drivers cannot use GPIOs from this GPIO controller
  - Other devices that use GPIO signals from this controller for reset, interrupt, etc. cannot control/configure those signals
  - Your application is less portable: it will take many changes to support another type of GPIO controller.
- You write your touchscreen driver in user-space: the standard Linux graphics stack components cannot use your touchscreen
- You write your network driver in user-space
  - You can probably send/receive packets
  - But you cannot leverage the Linux kernel networking stack for IP, TCP, UDP, etc.
  - And none of the Linux networking applications can use your network device

Upstream drivers vs. out-of-tree drivers

- The *upstream* Linux kernel contains thousands of drivers
  - This is the best place to look for drivers
  - Drivers have been reviewed and approved by the community
  - They comply with standard interfaces
- Vendor kernels often include additional drivers, directly in the kernel tree
- Device vendors sometimes also provide *out of tree drivers*
  - Their source code is provided separately from the Linux kernel tree
  - Quality is often dubious
  - Compatibility issues when updating to newer kernel releases
  - Not always use standard user-space interfaces
  - Example: https://github.com/lwfinger/rtl8723ds
  - Avoid them when possible!

Finding Linux kernel drivers

- `grep` in the Linux kernel tree is your *best friend*
  - For I2C, SPI and memory-mapped devices, matching of the driver is done based on the device name → *grep* for variants of the device name and vendor
  - For USB, PCI, matching is done either on the vendor ID/product ID, or the class → *grep* for these
- Driver file names are sometimes named in a "generic" way, not necessarily reflecting all devices they support.
  - Example: `drivers/gpio/gpio-pca953x.c` supports much more than just PCA953x. See the full list of devices supported by this driver

Finding Linux kernel drivers: an example

- You have a Maxim Integrated MAX7313 GPIO expander on I2C
- Search in the Linux kernel

git grep -i max7313

```
drivers/gpio/gpio-pca953x.c:    { "max7313", 16 | PCA953X_TYPE | PCA_INT, },
drivers/gpio/gpio-pca953x.c:    { .compatible = "maxim,max7313", .data = OF_953X(16, PCA_INT), },
```

- drivers/gpio/gpio-pca953x.c seems to support it
- Read drivers/gpio/Makefile to learn which kernel configuration option enables this driver

drivers/gpio/Makefile

```
obj-$(CONFIG_GPIO_PCA953X)              += gpio-pca953x.o
```

- Conclusion: you need to enable CONFIG_GPIO_PCA953X in your kernel configuration

User-space interfaces to drivers

User-space interfaces for hardware devices

For a high-level perspective: three main interfaces to access hardware devices exposed by the Linux kernel

- Device nodes in `/dev`
- Entries in the *sysfs* filesystem
- Network sockets and related APIs

Devices in */dev/*

- 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)
- See `Documentation/admin-guide/devices.txt` for the official list of reserved type/major/minor numbers.

Block vs. character 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

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

Creating device files

- Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needs 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` → the kernel automatically creates/removes device files
  - `CONFIG_DEVTMPFS_MOUNT` → asks the kernel to mount *devtmpfs* automatically at boot time (except when booting on an initramfs).

Better handling of device files: *udev* and *mdev*

- *devtmpfs* is great, but its capabilities are limited, so complementary solutions exist
- **udev**
  - daemon that receives events from the kernel about devices appearing/disappearing
  - can create/remove device files (but that's done by *devtmpfs* now), adjust permission/ownership, load kernel modules automatically, create symbolic links to devices
  - according to rules files in `/lib/udev/rules.d` and `/etc/udev/rules.d`
  - used in almost all desktop Linux distributions
  - https://en.wikipedia.org/wiki/Udev
- **mdev**
  - lightweight implementation of *udev*, part of Busybox
  - https://wiki.gentoo.org/wiki/Mdev

Examples of user-space interfaces in /dev

- Serial-ports: `/dev/ttyS*`, `/dev/ttyUSB*`, `/dev/ttyACM*`, etc.
- GPIO controllers (modern interface): `/dev/gpiochipX`
- Block storage devices: `/dev/sd*`, `/dev/mmcblk*`, `/dev/nvme*`
- Flash storage devices: `/dev/mtd*`
- Display controllers and GPUs: `/dev/dri/*`
- Audio devices: `/dev/snd/*`
- Camera devices: `/dev/video*`
- Watchdog devices: `/dev/watchdog*`
- Input devices: `/dev/input/*`
- and many more...

sysfs filesystem

- `block/`, symlinks to all block devices, in `/sys/devices`
- `bus/`, one sub-folder by type of bus
- `class/`, one sub-folder per class (category of devices): input, leds, pwm, etc.
- `dev/`
  - `block/`, one symlink per block device, named after major/minor
  - `char/`, one symlink per character device, named after major/minor
- `devices/`, all devices in the system, organized in a slightly chaotic way, see this article
- `firmware/`, representation of firmware data
  - `devicetree/`, directory and file representation of Device Tree nodes and properties
- `fs/`, properties related to filesystem drivers
- `kernel/`, properties related to various kernel subsystems
- `module/`, properties about kernel modules
- `power/`, power-management related properties

- `/sys/bus/i2c/drivers`: all device drivers for devices connected on I2C busses

```
[...]
edt_ft5x06
stpmic1
[...]
```

- `/sys/bus/i2c/devices`: all devices in the system connected to I2C busses

```
0-002a -> ../../../devices/platform/soc/40012000.i2c/i2c-0/0-002a
0-0039 -> ../../../devices/platform/soc/40012000.i2c/i2c-0/0-0039
0-004a -> ../../../devices/platform/soc/40012000.i2c/i2c-0/0-004a
1-0028 -> ../../../devices/platform/soc/5c002000.i2c/i2c-1/1-0028
1-0033 -> ../../../devices/platform/soc/5c002000.i2c/i2c-1/1-0033
i2c-0 -> ../../../devices/platform/soc/40012000.i2c/i2c-0
i2c-1 -> ../../../devices/platform/soc/5c002000.i2c/i2c-1
i2c-2 -> ../../../devices/platform/soc/40012000.i2c/i2c-0/i2c-2
```

sysfs filesystem example

/sys/bus/i2c/devices/0-002a/

```
lrwxrwxrwx    driver -> ../../../../../../bus/i2c/drivers/edt_ft5x06
-rw-r--r--    gain
drwxr-xr-x    input
-r--r--r--    modalias
-r--r--r--    name
lrwxrwxrwx    of_node -> ../../../../../../firmware/devicetree/base/soc/i2c@40012000/touchscreen@2a
-rw-r--r--    offset
-rw-r--r--    offset_x
-rw-r--r--    offset_y
drwxr-xr-x    power
-rw-r--r--    report_rate
lrwxrwxrwx    subsystem -> ../../../../../../bus/i2c
-rw-r--r--    threshold
-rw-r--r--    uevent
```

- `driver`, symlink to the driver directory in `/sys/bus/i2c/drivers`
- `of_node`, symlink to the directory for the Device Tree node describing this device

Example of driver interfaces in *sysfs*

- All devices are visible in *sysfs*, whether they have an interface in `/dev` or not
  - Usually `/dev` is to access the device
  - `/sys` is more about properties of the devices
- However, some devices only have a *sysfs* interface
  - LED: `/sys/class/leds`, see documentation
  - PWM: `/sys/class/pwm`, see documentation
  - IIO: `/sys/class/iio`, see documentation
  - etc.

Accessing GPIOs

A class of devices worth mentioning is GPIOs (*General Purpose Input Output*)

- The GPIOs can be accessed through a legacy interface in `/sys/class/gpios`
  - You will find many instructions on the Internet about how to drive GPIOs through this interface.
  - However, this interface is deprecated and has multiple shortcomings:
    - GPIOs remain exported if the process using them crashes
    - Need to compute the GPIO numbers, such numbers are not stable
- A new interface recommended: libgpiod
  - Based on `/dev/gpiochipx` character devices
  - Implementing advanced features not possible with the legacy interface
  - Of course, this is a C library
  - But it also provides command line utilities: `gpiodetect`, `gpioset`, `gpioget`...
  - The only constraint is to cross-compile them for your target (the legacy interface could be used without any additional software).

Other virtual filesystems

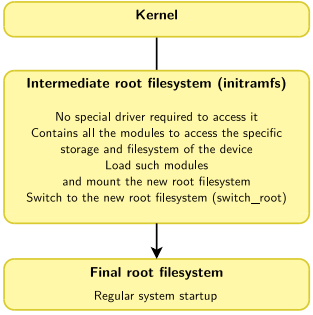
- *debugfs*
  - Conventionally mounted in `/sys/kernel/debug`
  - Contains lots of debug information from the kernel, including device related
  - `/sys/kernel/debug/pinctrl` for pin-mux debugging, `/sys/kernel/debug/gpio` for GPIO debugging, `/sys/kernel/debug/pwm` for PWM debugging, etc.
  - https://www.kernel.org/doc/html/latest/filesystems/debugfs.html
- *configfs*
  - Conventionally mounted in `/sys/kernel/config`
  - Allows to manage configuration of advanced kernel mechanisms
  - Example: configuration of USB gadget functionalities
  - Documentation/filesystems/configfs/configfs.txt

Using kernel modules

Why kernel modules?

- Primary reason: keep the kernel image minimal, and load drivers on-demand depending on the hardware detected
  - Needed to create a generic kernel configuration that works on many platforms
  - Used by all desktop/server Linux distributions
- But also useful for
  - Driver development: allows to modify, build and test a driver without rebooting
  - Boot time reduction: allows to defer the initialization of a driver after user-space has started critical applications

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 installation and metadata

- As discussed earlier, modules are installed in `/lib/modules/<kernel-version>/`
- Compiled kernel modules are stored in `.ko` (*Kernel Object*) files
- Metadata files:
  - `modules.dep`
  - `modules.alias`
  - `modules.symbols`
  - `modules.builtin`
- Each file has a corresponding `.bin` version, which is an optimized version of the corresponding text file

Module dependencies: *modules.dep*

- Some kernel modules can depend on other modules, based on the symbols (functions and data structures) that they use.
- Example: the `ubifs` module depends on the `ubi` and `mtd` modules.
  - `mtd` and `ubi` need to be loaded before `ubifs`
- These dependencies are described both in `/lib/modules/<kernel-version>/modules.dep` and in `/lib/modules/<kernel-version>/modules.dep.bin`
- Will be used by module loading tools.

Module alias: *modules.alias*

Kernel compiling

```
static const struct usb_device_id       products [] = {
{
        // Linksys USB200M
        USB_DEVICE (0x077b, 0x2226),
        .driver_info =  (unsigned long) &ax8817x_info,
}, {
        // Netgear FA120
        USB_DEVICE (0x0846, 0x1040),
        .driver_info =  (unsigned long) &netgear_fa120_info,
},
...
        { },            // END
};
MODULE_DEVICE_TABLE(usb, products);
```

drivers/net/usb/asix_devices.c

The device driver source code lists which devices it supports

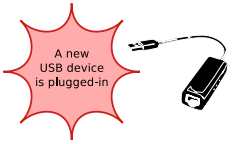
make modules → Module file: asix.ko

Containing the list of supported devices (module metadata)

make modules_install (depmod) → alias usb:v077Bp2226d*dc*dsc*dp*ic*isc*ip*in* asix
alias usb:v0846p1040d*dc*dsc*dp*ic*isc*ip*in* asix
....

modules.alias

System operation



The USB controller driver reads USB device attributes: vendor id, product id, etc

The USB bus driver sends a MODALIAS string encoding these attributes to the udev process

MODALIAS=usb:v077bp2226d0001dcFFdscFFdp00icFFiscFFip00in00

udev has a rule for when MODALIAS is set

ENV{MODALIAS}=="?*", RUN{builtin}+="kmod load $env{MODALIAS}"
....

/lib/udev/rules.d/80-drivers.rules

Thanks to aliases, the asix module is loaded

kmod load usb:v077bp2226d0001dcFFdscFFdp00icFFiscFFip00in00

Module utilities: *modinfo*

- `modinfo <module_name>`, for modules in `/lib/modules`
- `modinfo /path/to/module.ko`

```
# modinfo usb_storage
filename:       /lib/modules/5.18.13-200.fc36.x86_64/kernel/drivers/usb/storage/usb-storage.ko.xz
license:        GPL
description:    USB Mass Storage driver for Linux
author:         Matthew Dharm <mdharm-usb@one-eyed-alien.net>
alias:          usb:v*p*d*dc*dsc*dp*ic08isc06ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc05ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc04ip50in*
[...]
intree:         Y
name:           usb_storage
[...]
parm:           option_zero_cd:ZeroCD mode (1=Force Modem (default), 2=Allow CD-Rom (uint)
parm:           swi_tru_install:TRU-Install mode (1=Full Logic (def), 2=Force CD-Rom, 3=Force Modem) (uint)
parm:           delay_use:seconds to delay before using a new device (uint)
parm:           quirks:supplemental list of device IDs and their quirks (string)
```

Module utilities: *lsmod*

- Lists currently loaded kernel modules
- Includes
  - The reference count: incremented when the module is used by another module or by a user-space process, prevents from unloading modules that are in-use
  - Dependant modules: modules that depend on us
- Information retrieved through `/proc/modules`

```
$ lsmod
Module                  Size  Used by
tun                    61440  2
tls                   118784  0
rfcomm                 90112  4
snd_seq_dummy          16384  0
snd_hrtimer            16384  1
wireguard              94208  0
curve25519_x86_64      36864  1 wireguard
libcurve25519_generic    49152  2 curve25519_x86_64,wireguard
ip6_udp_tunnel         16384  1 wireguard
```

Module utilities: *insmod* and *rmmod*

- Basic tools to:
  - *load* a module: `insmod`
  - *unload* a module: `rmmod`
- Basic because:
  - Need a full path to the module `.ko` file
  - Do not handle module dependencies

```
# insmod /lib/modules/`uname -r`/kernel/fs/fuse/cuse.ko.xz
# rmmod cuse
```

Module utilities: *modprobe*

- *modprobe* is the more advanced tool for loading/unloading modules
- Takes just a module name as argument: `modprobe <module-name>`
- Takes care of dependencies automatically, using the `modules.dep` file
- Supports removing modules using `modprobe -r`, including its no longer used dependencies

```
# modinfo fat_test | grep depends
depends:        kunit,fat
# lsmod | grep -E "^(kunit|fat|fat_test)"
fat                    86016  1 vfat
# modprobe fat_test
# lsmod | grep -E "^(kunit|fat|fat_test)"
fat_test               24576  0
kunit                  36864  1 fat_test
fat                    86016  2 fat_test,vfat
# sudo modprobe -r fat_test
# lsmod | grep -E "^(kunit|fat|fat_test)"
fat                    86016  1 vfat
```

Passing parameters to modules

- Some modules have parameters to adjust their behavior
- Mostly for debugging/tweaking, as parameters are global to the module, not per-device managed by the module
- Through `insmod` or `modprobe`:
  `insmod ./usb-storage.ko delay_use=0`
  `modprobe usb-storage delay_use=0`
- `modprobe` supports configuration files: `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options usb-storage delay_use=0`
- Through the kernel command line, when the module is built statically into the kernel:
  `usb-storage.delay_use=0`
  - `usb-storage` is the *module name*
  - `delay_use` is the *module parameter name*. It specifies a delay before accessing a USB storage device (useful for rotating devices).
  - `0` is the *module parameter value*

Modules in *sysfs*

- All modules are visible in *sysfs*, under `/sys/module/<name>`
- Lots of information available about each module
- For example, the `/sys/module/<name>/parameters` directory contains one file per module parameter
- Can read the current value of module parameters
- Some of them can even be changed at runtime (determined by the module code)
- Example:
  `echo 0 > /sys/module/usb_storage/parameters/delay_use`

Describing non-discoverable hardware: Device Tree

Describing non-discoverable hardware

1. Directly in the **OS/bootloader code**
   - Using compiled data structures, typically in C
   - How it was done on most embedded platforms in Linux, U-Boot.
   - Considered not maintainable/sustainable on ARM32, which motivated the move to another solution.

Describing non-discoverable hardware

2. Using **ACPI** tables

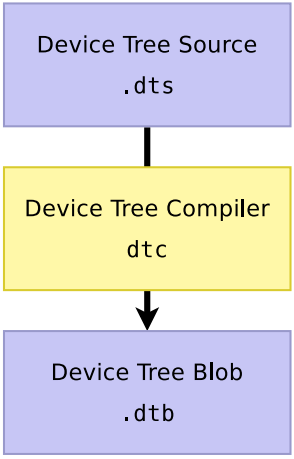
- On *x86* systems, but also on a subset of ARM64 platforms
- Tables provided by the firmware

3. Using a **Device Tree**

- Originates from **OpenFirmware**, defined by Sun, used on SPARC and PowerPC
  - That's why many Linux/U-Boot functions related to DT have a `of_` prefix
- Now used by most embedded-oriented CPU architectures that run Linux: ARC, ARM64, RISC-V, ARM32, PowerPC, Xtensa, MIPS, etc.
- Writing/tweaking a DT is necessary when porting Linux to a new board, or when connecting additional peripherals

Device Tree: from source to blob

- A tree data structure describing the hardware is written by a developer in a **Device Tree Source** file, `.dts`
- Processed by the **Device Tree Compiler**, `dtc`
- Produces a more efficient representation: **Device Tree Blob**, `.dtb`
- Additional C preprocessor pass
- `.dtb` → accurately describes the hardware platform in an **OS-agnostic** way.
- `.dtb` ≈ few dozens of kilobytes
- DTB also called **FDT**, *Flattened Device Tree*, once loaded into memory.
  - `fdt` command in U-Boot
  - `fdt_` APIs



dtc example

```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

dtc example

```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

```
$ dtc -I dts -O dtb -o foo.dtb foo.dts
$ ls -l foo.dt*
-rw-r--r-- 1 thomas thomas 169 ... foo.dtb
-rw-r--r-- 1 thomas thomas 102 ... foo.dts
```

dtc example

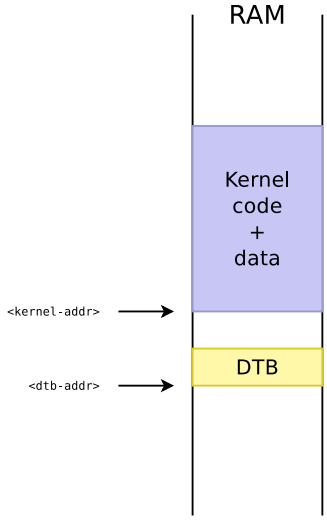
```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

```
$ dtc -I dts -O dtb -o foo.dtb foo.dts
$ ls -l foo.dt*
-rw-r--r-- 1 thomas thomas 169 ... foo.dtb
-rw-r--r-- 1 thomas thomas 102 ... foo.dts
```

```
$ dtc -I dtb -O dts foo.dtb
/dts-v1/;

/ {
        welcome = <0xbadcafe>;

        bootlin {
                webinar = "great";
                demo = <0x01 0x02 0x03>;
        };
};
```

Device Tree: using the blob

- Can be **linked directly** inside a bootloader binary
  - For example: U-Boot, Barebox
- Can be **passed** to the operating system by the bootloader
  - Most common mechanism for the Linux kernel
  - U-Boot:
    `boot[z,i,m] <kernel-addr> - <dtb-addr>`
  - The bootloader can adjust the DTB before passing it to the kernel
- The DTB parsing can be done using `libfdt`, or ad-hoc code

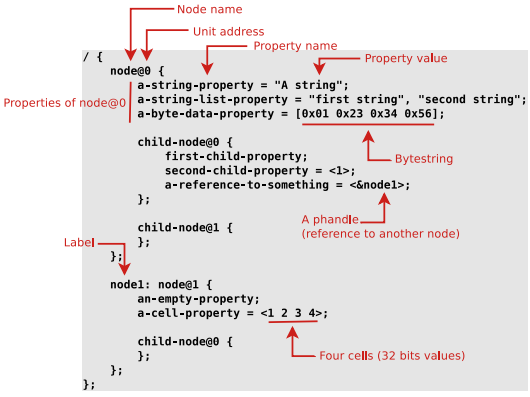


Where are Device Tree Sources located?

- Even though they are OS-agnostic, **no central and OS-neutral** place to host Device Tree sources and share them between projects
  - Often discussed, never done
- In practice, the Linux kernel sources can be considered as the **canonical location** for Device Tree Source files
  - `arch/<ARCH>/boot/dts/<vendor>/`
  - `arch/arm/boot/dts` (on ARM 32 architecture before Linux 6.5)
  - ≈ 4500 Device Tree Source files (`.dts` and `.dtsi`) in Linux as of 6.0.
- Duplicated/synced in various projects
  - U-Boot, Barebox, TF-A

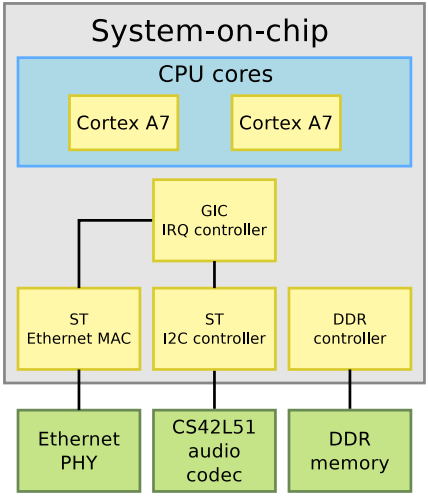
Device Tree base syntax

- Tree of **nodes**
- Nodes with **properties**
- Node ≈ a device or IP block
- Properties ≈ device characteristics
- Notion of **cells** in property values
- Notion of **phandle** to point to other nodes
- `dtc` only does syntax checking, no semantic validation



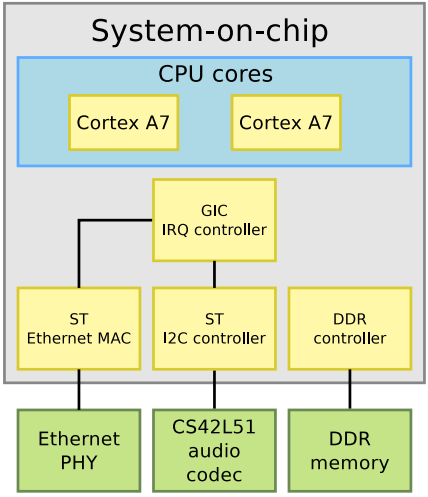
DT overall structure: simplified example

```
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  model = "STMicroelectronics STM32MP157C-DK2 Discovery Board";
  compatible = "st,stm32mp157c-dk2", "st,stm32mp157";

  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



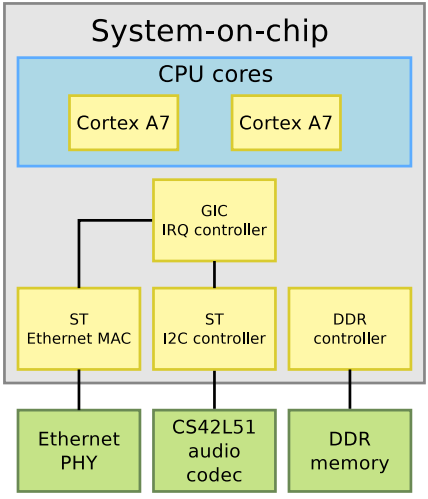
DT overall structure: simplified example

```
/ {
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu0: cpu@0 {
      compatible = "arm,cortex-a7";
      clock-frequency = <650000000>;
      device_type = "cpu";
      reg = <0>;
    };

    cpu1: cpu@1 {
      compatible = "arm,cortex-a7";
      clock-frequency = <650000000>;
      device_type = "cpu";
      reg = <1>;
    };
  };

  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



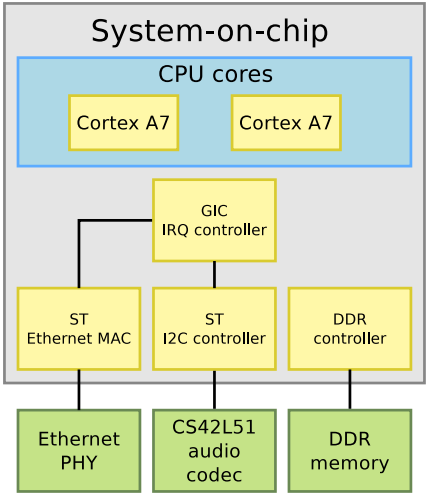
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 {
    device_type = "memory";
    reg = <0x0 0x20000000>;
  };

  chosen {
    bootargs = "";
    stdout-path = "serial0:115200n8";
  };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



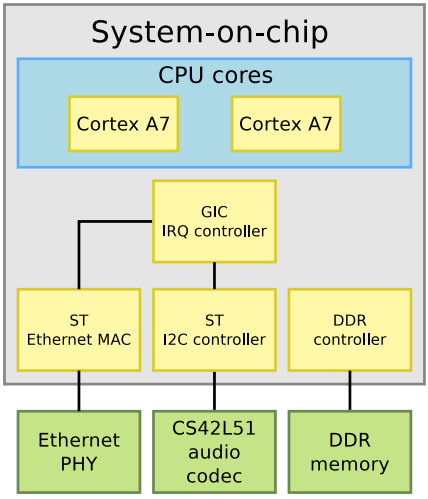
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };

  intc: interrupt-controller@a0021000 {
    compatible = "arm,cortex-a7-gic";
    #interrupt-cells = <3>;
    interrupt-controller;
    reg = <0xa0021000 0x1000>,
          <0xa0022000 0x2000>;
  };

  soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



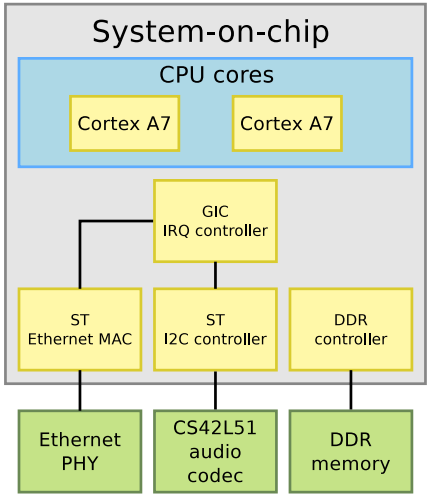
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 {
      compatible = "st,stm32mp15-i2c";
      reg = <0x40012000 0x400>;
      interrupts = <GIC_SPI 31 IRQ_TYPE_LEVEL_HIGH>,
                   <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>;
      #address-cells = <1>;
      #size-cells = <0>;
      status = "okay";

      cs42l51: cs42l51@4a {
        compatible = "cirrus,cs42l51";
        reg = <0x4a>;
        reset-gpios = <&gpiog 9 GPIO_ACTIVE_LOW>;
        status = "okay";
      };
    };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    compatible = "simple-bus";
    ...
    interrupt-parent = <&intc>;
    i2c1: i2c@40012000 { ... };

    ethernet0: ethernet@5800a000 {
      compatible = "st,stm32mp1-dwmac", "snps,dwmac-4.20a";
      reg = <0x5800a000 0x2000>;
      interrupts-extended = <&intc GIC_SPI 61 IRQ_TYPE_LEVEL_HIGH>;
      status = "okay";

      mdio0 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "snps,dwmac-mdio";
        phy0: ethernet-phy@0 {
          reg = <0>;
        };
      };
    };
  };
};
```



Device Tree inheritance

- Device Tree files are not monolithic, they can be split in several files, including each other.
- `.dtsi` files are included files, while `.dts` files are *final* Device Trees
  - Only `.dts` files are accepted as input to `dtc`
- Typically, `.dtsi` will contain
  - definitions of SoC-level information
  - definitions common to several boards
- The `.dts` file contains the board-level information
- The inclusion works by **overlaying** the tree of the including file over the tree of the included file, according to the order of the `#include` directives.
- Allows an including file to **override** values specified by an included file
- Uses the C pre-processor `#include` directive

Device Tree inheritance example

Definition of the STM32MP157A SoC

```
/ {
    soc {
        i2c1: i2c@40012000 {
            compatible = "st,stm32mp15-i2c";
            reg = <0x40012000 0x400>;
            interrupts = <GIC_SPI 31 IRQ...HIGH>,
                         <GIC_SPI 32 IRQ...HIGH>;
            status = "disabled";
        };
    };
};
```

stm32mp157.dtsi

+

Definition of the STM32MP157A-DK1 board

```
#include "stm32mp157.dtsi"

/ {
  soc {
      i2c1: i2c@40012000 {
          pinctrl-names = "default", "sleep";
          pinctrl-0 = <&i2c1_pins_a>;
          pinctrl-1 = <&i2c1_sleep_pins_a>;
          status = "okay";
          cs42l51: cs42l51@4a {
              compatible = "cirrus,cs42l51";
              reg = <0x4a>;
          };
      };
  };
};
```

stm32mp157a-dk1.dts

Note 1

The actual Device Trees for this platform are more complicated. This example is highly simplified.

=

Compiled DTB

```
/ {
    soc {
      i2c1: i2c@40012000 {
          compatible = "st,stm32mp15-i2c";
          reg = <0x40012000 0x400>;
          interrupts = <GIC_SPI 31 IRQ...HIGH>,
                       <GIC_SPI 32 IRQ...HIGH>;
          pinctrl-names = "default", "sleep";
          pinctrl-0 = <&i2c1_pins_a>;
          pinctrl-1 = <&i2c1_sleep_pins_a>;
          status = "okay";
          cs42l51: cs42l51@4a {
              compatible = "cirrus,cs42l51";
              reg = <0x4a>;
          };
      };
    };
};
```

stm32mp157a-dk1.dtb

Note 2

The real DTB is in binary format. Here we show the text equivalent of the DTB contents.

Inheritance and labels

Doing:

soc.dtsi

```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  soc {
    serial@5c000000 {
      status = "okay";
    };
  };
};
```

Inheritance and labels

Doing:

soc.dtsi

```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  soc {
    serial@5c000000 {
      status = "okay";
    };
  };
};
```

Is exactly equivalent to:

soc.dtsi

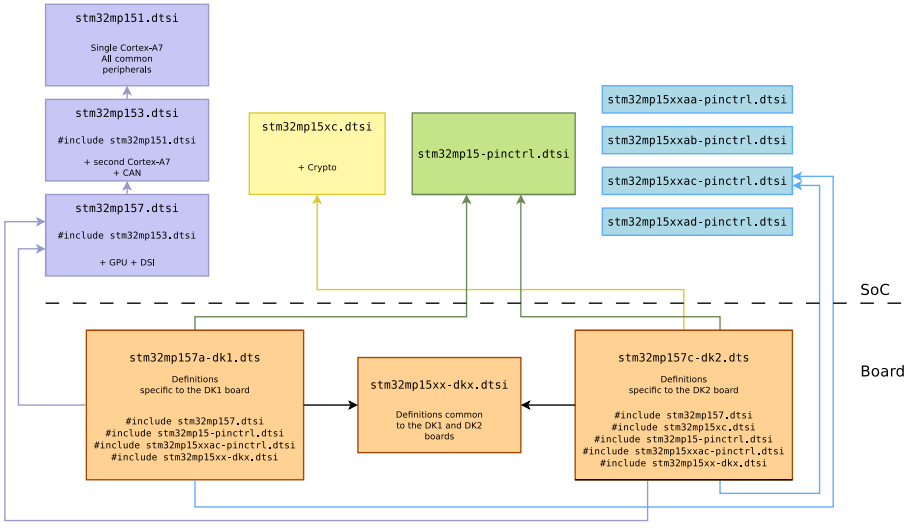
```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

&usart1 {
  status = "okay";
};
```

→ this solution is now often preferred

DT inheritance in STM32MP1 support



Device Tree design principles

- **Describe hardware** (how the hardware is), not configuration (how I choose to use the hardware)
- **OS-agnostic**
  - For a given piece of HW, Device Tree should be the same for U-Boot, FreeBSD or Linux
  - There should be no need to change the Device Tree when updating the OS
- Describe **integration of hardware components**, not the internals of hardware components
  - The details of how a specific device/IP block is working is handled by code in device drivers
  - The Device Tree describes how the device/IP block is connected/integrated with the rest of the system: IRQ lines, DMA channels, clocks, reset lines, etc.
- Like all beautiful design principles, these principles are sometimes violated.

Device Tree specifications

- How to write the correct nodes/properties to describe a given hardware platform ?
- **DeviceTree Specifications** → base Device Tree syntax + number of standard properties.
  - https://www.devicetree.org/specifications/
  - Not sufficient to describe the wide variety of hardware.
- **Device Tree Bindings** → documents that each specify how a piece of HW should be described
  - Documentation/devicetree/bindings/ in Linux kernel sources
  - Reviewed by DT bindings maintainer team
  - Legacy: human readable documents
  - New norm: YAML-written specifications

Devicetree Specification

Release v0.3

devicetree.org

13 February 2020

Device Tree binding: old style

Documentation/devicetree/bindings/mtd/spear_smi.txt

This IP is *not* used on STM32MP1.

```
* SPEAr SMI

Required properties:
- compatible : "st,spear600-smi"
- reg : Address range of the mtd chip
- #address-cells, #size-cells : Must be present if the device has sub-nodes
  representing partitions.
- interrupts: Should contain the STMMAC interrupts
- clock-rate : Functional clock rate of SMI in Hz

Optional properties:
- st,smi-fast-mode : Flash supports read in fast mode
```

```
Example:

        smi: flash@fc000000 {
                compatible = "st,spear600-smi";
                #address-cells = <1>;
                #size-cells = <1>;
                reg = <0xfc000000 0x1000>;
                interrupt-parent = <&vic1>;
                interrupts = <12>;
                clock-rate = <50000000>;        /* 50MHz */

                flash@f8000000 {
                        st,smi-fast-mode;
                        ...
                };
        };
```

Device Tree binding: YAML style

Documentation/devicetree/bindings/i2c/st,stm32-i2c.yaml

```
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
$id: http://devicetree.org/schemas/i2c/st,stm32-i2c.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: I2C controller embedded in STMicroelectronics STM32 I2C platform

maintainers:
  - Pierre-Yves MORDRET <pierre-yves.mordret@st.com>

properties:
  compatible:
    enum:
      - st,stm32f4-i2c
      - st,stm32f7-i2c
      - st,stm32mp15-i2c

  reg:
    maxItems: 1

  interrupts:
    items:
      - description: interrupt ID for I2C event
      - description: interrupt ID for I2C error

  resets:
    maxItems: 1

  clocks:
    maxItems: 1

  dmas:
    items:
      - description: RX DMA Channel phandle
      - description: TX DMA Channel phandle

  ...

  clock-frequency:
    description: Desired I2C bus clock frequency in Hz. If not specified,
                 the default 100 kHz frequency will be used.
                 For STM32F7, STM32H7 and STM32MP1 SoCs, if timing
                 parameters match, the bus clock frequency can be from
                 1Hz to 1MHz.
    default: 100000
    minimum: 1
    maximum: 1000000

required:
  - compatible
  - reg
  - interrupts
  - resets
  - clocks
```

Device Tree binding: YAML style example

```
examples:
  - |
    //Example 3 (with st,stm32mp15-i2c compatible on stm32mp)
    #include <dt-bindings/interrupt-controller/arm-gic.h>
    #include <dt-bindings/clock/stm32mp1-clks.h>
    #include <dt-bindings/reset/stm32mp1-resets.h>
      i2c@40013000 {
          compatible = "st,stm32mp15-i2c";
          #address-cells = <1>;
          #size-cells = <0>;
          reg = <0x40013000 0x400>;
          interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>,
                       <GIC_SPI 34 IRQ_TYPE_LEVEL_HIGH>;
          clocks = <&rcc I2C2_K>;
          resets = <&rcc I2C2_R>;
          i2c-scl-rising-time-ns = <185>;
          i2c-scl-falling-time-ns = <20>;
          st,syscfg-fmp = <&syscfg 0x4 0x2>;
      };
```

Validating Device Tree in Linux

- `dtc` only does syntactic validation
- YAML bindings allow to do semantic validation
- Linux kernel `make` rules:
  - `make dt_binding_check`
    verify that YAML bindings are valid
  - `make dtbs_check`
    validate DTs currently enabled against YAML bindings
  - `make DT_SCHEMA_FILES=Documentation/devicetree/bindings/trivial-devices.yaml dtbs_check`
    validate DTs against a specific YAML binding

The compatible property

- Is a list of strings
  - From the most specific to the least specific
- Describes the specific **binding** to which the node complies.
- It uniquely identifies the **programming model** of the device.
- Practically speaking, it is used by the operating system to find the **appropriate driver** for this device.
- When describing real hardware, the typical form is `vendor,model`
- Examples:
  - `compatible = "arm,armv7-timer";`
  - `compatible = "st,stm32mp1-dwmac", "snps,dwmac-4.20a";`
  - `compatible = "regulator-fixed";`
  - `compatible = "gpio-keys";`
- Special value: `simple-bus` → bus where all sub-nodes are memory-mapped devices

compatible property and Linux kernel drivers

- Linux identifies as **platform devices**:
  - Top-level DT nodes with a `compatible` string
  - Sub-nodes of `simple-bus`
    - Instantiated automatically at boot time
- Sub-nodes of I2C controllers → *I2C devices*
- Sub-nodes of SPI controllers → *SPI devices*
- Each Linux driver has a table of compatible strings it supports
  - `struct of_device_id[]`
- When a DT node compatible string matches a given driver, the device is *bound* to that driver.

```
/ {
    timer {                                  → Platform device
        compatible = "...";
    };
    soc {
        compatible = "simple-bus";
        uart@1000 {                          → Platform device
            compatible = "...";
        };
        i2c@2000 {                           → Platform device
            compatible = "...";
            eeprom@65 {                      → I2C device
                compatible = "...";
            };
        };
    };
};
```

Matching with drivers in Linux: platform driver

drivers/tty/serial/stm32-usart.c

```
static const struct of_device_id stm32_match[] = {
        { .compatible = "st,stm32-uart", .data = &stm32f4_info},
        { .compatible = "st,stm32f7-uart", .data = &stm32f7_info},
        { .compatible = "st,stm32h7-uart", .data = &stm32h7_info},
        {},
};
MODULE_DEVICE_TABLE(of, stm32_match);

...

static struct platform_driver stm32_serial_driver = {
        .probe          = stm32_serial_probe,
        .remove         = stm32_serial_remove,
        .driver = {
                .name   = DRIVER_NAME,
                .pm     = &stm32_serial_pm_ops,
                .of_match_table = of_match_ptr(stm32_match),
        },
};
```

Matching with drivers in Linux: I2C driver

sound/soc/codecs/cs42l51.c

```
const struct of_device_id cs42l51_of_match[] = {
        { .compatible = "cirrus,cs42l51", },
        { }
};
MODULE_DEVICE_TABLE(of, cs42l51_of_match);
```

sound/soc/codecs/cs42l51-i2c.c

```
static struct i2c_driver cs42l51_i2c_driver = {
        .driver = {
                .name = "cs42l51",
                .of_match_table = cs42l51_of_match,
                .pm = &cs42l51_pm_ops,
        },
        .probe = cs42l51_i2c_probe,
        .remove = cs42l51_i2c_remove,
        .id_table = cs42l51_i2c_id,
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.

```
sai4: sai@50027000 {
    reg = <0x50027000 0x4>, <0x500273f0 0x10>;
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.

```
&i2c1 {
   hdmi-transmitter@39 {
      reg = <0x39>;
   };
   cs42l51: cs42l51@4a {
      reg = <0x4a>;
   };
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.
- **SPI** devices: chip select number

```
&qspi {
        flash0: mx66l51235l@0 {
                reg = <0>;
        };
        flash1: mx66l51235l@1 {
                reg = <1>;
        };
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.
- **SPI** devices: chip select number
- The unit address must be the address of the first `reg` entry.

```
sai4: sai@50027000 {
    reg = <0x50027000 0x4>, <0x500273f0 0x10>;
};
```

Status property

- The `status` property indicates if the device is really in use or not
  - `okay` or `ok` → the device is really in use
  - any other value, by convention `disabled` → the device is not in use
- In Linux, controls if a device is instantiated
- In `.dtsi` files describing SoCs: all devices that interface to the outside world have `status = "disabled";`
- Enabled on a per-device basis in the board `.dts`

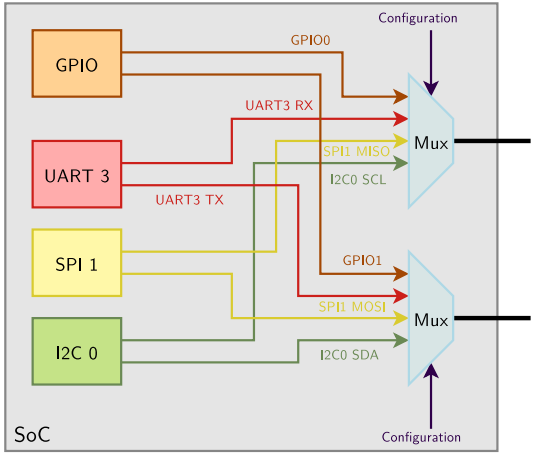
Resources: interrupts, clocks, DMA, reset lines, ...

- Common pattern for resources shared by multiple hardware blocks
  - Interrupt lines
  - Clock controllers
  - DMA controllers
  - Reset controllers
  - ...
- A Device Tree node describing the *controller* as a device
- References from other nodes that use resources provided by this *controller*

```
intc: interrupt-controller@a0021000 {
   compatible = "arm,cortex-a7-gic";
   #interrupt-cells = <3>;
   interrupt-controller;
   reg = <0xa0021000 0x1000>, <0xa0022000 0x2000>;
};

rcc: rcc@50000000 {
   compatible = "st,stm32mp1-rcc", "syscon";
   reg = <0x50000000 0x1000>;
   #clock-cells = <1>;
   #reset-cells = <1>;
};

dmamux1: dma-router@48002000 {
   compatible = "st,stm32h7-dmamux";
   reg = <0x48002000 0x1c>;
   #dma-cells = <3>;
   clocks = <&rcc DMAMUX>;
   resets = <&rcc DMAMUX_R>;
};

spi3: spi@4000c000 {
   interrupts = <GIC_SPI 51 IRQ_TYPE_LEVEL_HIGH>;
   clocks = <&rcc SPI3_K>;
   resets = <&rcc SPI3_R>;
   dmas = <&dmamux1 61 0x400 0x05>,  <&dmamux1 62 0x400 0x05>;
};
```

Pin-muxing description

- Most modern SoCs, including the STM32MP1, have more features than they have pins to expose those features to the outside world.
- Pins are muxed: a given pin can be used for one function **or** another
- A specific IP block in the SoC controls the muxing of pins: the **pinmux controller**
- The Device Tree describes which pin configurations are possible, and which configurations are used by the different devices.



Pin-muxing controllers on STM32MP1

arch/arm/boot/dts/stm32mp151.dtsi

```
pinctrl: pin-controller@50002000 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "st,stm32mp157-pinctrl";
        ...
        gpioa: gpio@50002000 { ... };
        gpiob: gpio@50003000 { ... };
        gpioc: gpio@50004000 { ... };
        gpiod: gpio@50005000 { ... };
        gpioe: gpio@50006000 { ... };
        gpiof: gpio@50007000 { ... };
        ...
};

pinctrl_z: pin-controller-z@54004000 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "st,stm32mp157-z-pinctrl";
        ranges = <0 0x54004000 0x400>;
        ...
        gpioz: gpio@54004000 { .... };
        ...
};
```

Pin-muxing configuration

arch/arm/boot/dts/stm32mp15-pinctrl.dtsi

```
&pinctrl {
        ...
        i2c1_pins_a: i2c1-0 {
                pins {
                        pinmux = <STM32_PINMUX('D', 12, AF5)>, /* I2C1_SCL */
                                 <STM32_PINMUX('F', 15, AF5)>; /* I2C1_SDA */
                        bias-disable;
                        drive-open-drain;
                        slew-rate = <0>;
                };
        };
        ...
        m_can1_pins_a: m-can1-0 {
                pins1 {
                        pinmux = <STM32_PINMUX('H', 13, AF9)>; /* CAN1_TX */
                        slew-rate = <1>;
                        drive-push-pull;
                        bias-disable;
                };
                pins2 {
                        pinmux = <STM32_PINMUX('I', 9, AF9)>; /* CAN1_RX */
                        bias-disable;
                };
        };
        ...
};
```

Pin-muxing configuration

Table 8. Alternate function AF0 to AF7[1] (continued)

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
		HDP/SYS/RTC	TIM1/2/16/17/ LPTIM1/SYS/ RTC	SAI1/4/I2C6/ TIM3/4/5/12/ HDP/SYS	SAI4/I2C2/ TIM8/ LPTIM2/3/4/5/ DFSDM1 /SDMMC1	SAI4/ I2C1/2/3/4/5/ USART1/ TIM15/LPTIM2/ DFSDM1/CEC	SPI1/I2S1/ SPI2/I2S2/ SPI3/I2S3/ SPI4/5/6/I2C1/ SDMMC1/3/ CEC	SPI3/I2S3/ SAI1/3/4/ I2C4/UART4/ DFSDM1	SPI2/I2S2/ SPI3/I2S3/ SPI6/ USART1/2/3/6/ UART7/ SDMMC2
Port D	PD6	-	TIM16_CH1N	SAI1_D1	DFSDM1_ CKIN4	DFSDM1_ DATIN1	SPI3_MOSI/ I2S3_SDO	SAI1_SD_A	USART2_RX
	PD7	TRACED6	-	-	DFSDM1_ DATIN4	I2C2_SCL	-	DFSDM1_ CKIN1	USART2_CK
	PD8	-	-	-	DFSDM1_ CKIN3	-	-	SAI3_SCK_B	USART3_TX
	PD9	-	-	-	DFSDM1_ DATIN3	-	-	SAI3_SD_B	USART3_RX
	PD10	RTC_REFIN	TIM16_BKIN	-	DFSDM1_ CKOUT	I2C5_SMBA	SPI3_MISO/ I2S3_SDI	SAI3_FS_B	USART3_CK
	PD11	-	-	-	LPTIM2_IN2	I2C4_SMBA	I2C1_SMBA	-	USART3_CTS/ USART3_NSS
	PD12	-	LPTIM1_IN1	TIM4_CH1	LPTIM2_IN1	I2C4_SCL	I2C1_SCL	-	USART3_RTS/ USART3_DE
	PD13	-	LPTIM1_OUT	TIM4_CH2	-	I2C4_SDA	I2C1_SDA	I2S3_MCK	-
	PD14	-	-	TIM4_CH3	-	-	-	SAI3_MCLK_B	-
	PD15	-	-	TIM4_CH4	-	-	-	SAI3_MCLK_A	-
Port E	PE0	-	LPTIM1_ETR	TIM4_ETR	-	LPTIM2_ETR	SPI3_SCK/ I2S3_CK	SAI4_MCLK_B	-
	PE1	-	LPTIM1_IN2	-	-	-	I2S2_MCK	SAI3_SD_B	-
	PE2	TRACECLK	-	SAI1_CK1	-	I2C4_SCL	SPI4_SCK	SAI1_MCLK_A	-
	PE3	TRACED0	-	-	-	TIM15_BKIN	-	SAI1_SD_B	-

Source: STM32MP157C datasheet. Note that I2C1_SDA is also available on pin PF15 (not shown here).

Pin-muxing consumer

```
&i2c1 {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&i2c1_pins_a>;
        pinctrl-1 = <&i2c1_sleep_pins_a>;
        ...
};
```

- Typically board-specific, in `.dts`
- `pinctrl-0`, `pinctrl-1`, `pinctrl-X` provides the pin mux configurations for the different **states**
- `pinctrl-names` gives a name to each state, mandatory even if only one state
- States are mutually exclusive
- The driver is responsible for switching between states
- `default` state is automatically set up when the device is *probed*

Example: LED and I2C device

- Let's see how to describe an LED and an I2C device connected to the DK1 platform.
- Create `arch/arm/boot/dts/stm32mp157a-dk1-custom.dts` which includes `stm32mp157a-dk1.dts`

```
#include "stm32mp157a-dk1.dts"
```

- Make sure `stm32mp157a-dk1-custom.dts` gets compiled to a DTB by changing `arch/arm/boot/dts/Makefile`

```
dtb-$(CONFIG_ARCH_STM32) += \
        ...
        stm32mp157a-dk1.dtb \
        stm32mp157a-dk1-custom.dtb \
```

- `make dtbs`

```
  DTC     arch/arm/boot/dts/stm32mp157a-dk1-custom.dtb
```

Example: describe an LED

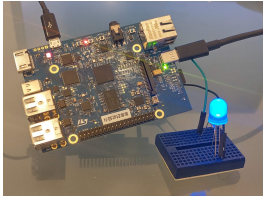
stm32mp157a-dk1-custom.dts

```
#include "stm32mp157a-dk1.dts"

/ {
        leds {
                compatible = "gpio-leds";
                webinar {
                        label = "webinar";
                        gpios = <&gpioe 1 GPIO_ACTIVE_HIGH>;
                };
        };
};
```

shell

```
# echo 255 > /sys/class/leds/webinar/brightness
```

CN14	1	ARD_D0	PE7	USART7_RX
	2	ARD_D1	PE8	USART7_TX
	3	ARD_D2	PE1	IO
	4	ARD_D3	PD14	TIM4_CH3
	5	ARD_D4	PE10	IO
	6	ARD_D5	PD15	TIM4_CH4
	7	ARD_D6	PE9	TIM1_CH1
	8	ARD_D7	PD1	IO



Example: connect I2C temperature, humidity and pressure sensor

stm32mp157a-dk1-custom.dts

```
&i2c5 {
        status = "okay";
        clock-frequency = <100000>;
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&i2c5_pins_a>;
        pinctrl-1 = <&i2c5_pins_sleep_a>;

        pressure@76 {
                compatible = "bosch,bme280";
                reg = <0x76>;
        };
};
```

shell

```
# cat /sys/bus/iio/devices/iio\:device2/in_humidityrelative_input
49147
# cat /sys/bus/iio/devices/iio\:device2/in_pressure_input
101.567167968
# cat /sys/bus/iio/devices/iio\:device2/in_temp_input
24380
```

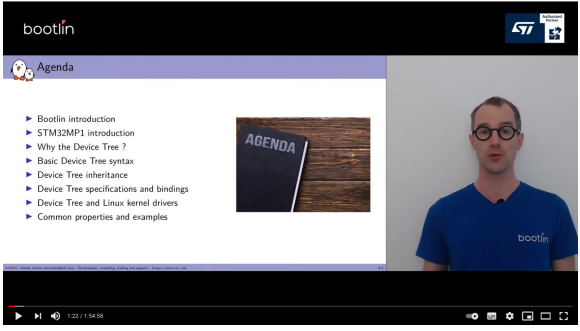
CN13	1	ARD_D8	PG3	IO
	2	ARD_D9	PH6	TIM12_CH1
	3	ARD_D10	PE11	SPI4_NSS and TIM1_CH2
	4	ARD_D11	PE14	SPI4_MOSI and TIM1_CH4
	5	ARD_D12	PE13	SPI4_MISO
	6	ARD_D13	PE12	SPI4_SCK
	7	GND	-	GND
	8	VREFP	-	VREF+
	9	ARD_D14	PA12	I2C5_SDA
	10	ARD_D15	PA11	I2C5_SCL

Details at https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-connecting-an-i2c-sensor/

Further details about the Device Tree

Check out our *Device Tree 101 webinar*, by Thomas Petazzoni (2021)

- Slides: https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/
- Video: https://youtu.be/a9CZ1Uk3OYQ

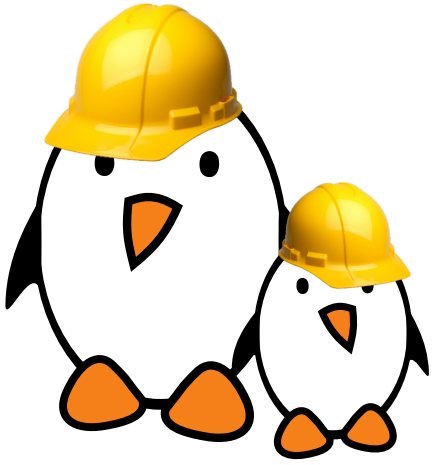


Discoverable hardware: USB and PCI

Discoverable hardware

- Some busses have built-in hardware discoverability mechanisms
- Most common busses: USB and PCI
- Hardware devices can be enumerated, and their characteristics retrieved with just a driver or the bus controller
- Useful Linux commands
  - `lsusb`, lists all USB devices detected
  - `lspci`, lists all PCI devices detected
  - A detected device does not mean it has a kernel driver associated to it!
- Association with kernel drivers done based on product ID/vendor ID, or some other characteristics of the device: device class, device sub-class, etc.

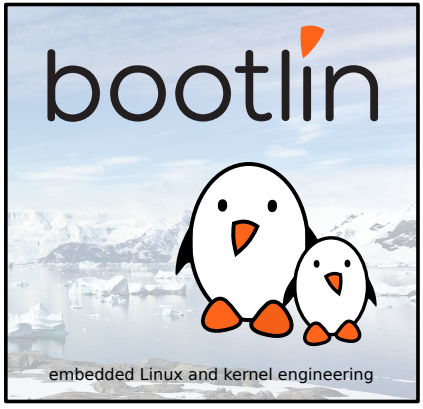
Practical lab - Accessing hardware devices



Time to start the practical lab!

- Exploring the contents of `/dev` and `/sys` and the devices available on the embedded hardware platform.
- Using GPIOs and LEDs.
- Modifying the Device Tree to control pin multiplexing and declare an I2C-connected joystick.
- Adding support for a USB audio card using Linux kernel modules
- Adding support for the I2C-connected joystick through an out-of-tree module.

Block filesystems



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

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

Ext2

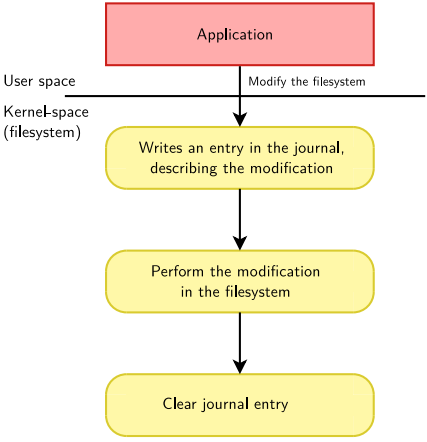
One of the earliest Linux filesystem, introduced in 1993

- `filesystems/ext2`
- Still actively supported. Low metadata overhead, module size and RAM usage
- But risk of metadata corruption after an unclean shutdown. You then need to run `e2fsck`, which takes time and may need operator intervention. Can't reboot autonomously.
- First successor: `ext3` (2001), addressing this limitation with *Journaling* (see next slides) but wasn't scaling well. Now deprecated.
- It supports all features Linux needs in a root filesystem: permissions, ownership, device files, symbolic links, etc.
- Date range: December 14, 1901 – January 18, 2038, because of 32 bit dates!

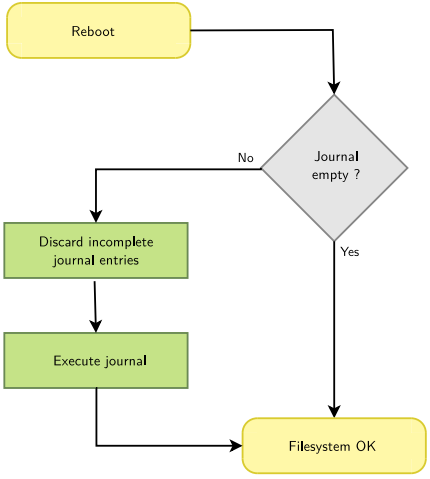
Not recommended for embedded systems today!

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.

Ext4

The modern successor of Ext2

- First introduced in 2006, filesystem with Journaling, without `ext3` limitations.
- Still actively developed (new features added). However, considered in 2008 by Ted Ts'o as a "stop-gap" based on old technologies.
- The default filesystem choice for many GNU/Linux distributions (Debian, Ubuntu)
- The `ext4` driver also supports `ext2` and `ext3` (one driver is sufficient).
- Noteworthy feature: transparent encryption (but compression not available).
- Minimum partition size to have a journal: 8MiB.
- Minimum partition size without a journal: 256KiB (only 32 inodes!).

https://en.wikipedia.org/wiki/Ext4

XFS

A Journaling filesystem

- Since 1994 (started by Silicon Graphics for the IRIX OS)
- Actively maintained and developed by Red Hat now
- Features: variable block size, direct I/O, online growth...
- Minimum partition size: 16MiB (9.7MiB of free space)

https://en.wikipedia.org/wiki/XFS

Btrfs

A copy-on-write filesystem

- Pronounced as "better F S", "butter F S" or "b-tree F S", since 2009.
- A modern filesystem with many advanced features: volumes, snapshots, transparent compression... Looks great for storage experts.
- Minimum partition size: 109MiB (only 32MiB of free space).
- However, big module size and long initialization time (bad for boot time)

https://en.wikipedia.org/wiki/Btrfs

F2FS — Flash-Friendly File System

A log-structured filesystem

- Since 2012 (started by Samsung, actively maintained)
- Designed from the start to take into account the characteristics of solid-state based storage (eMMC, SD, SSD)
- In particular, trying to make most writes sequential (best on SSD)
- Support for transparent encryption and compression (LZO, LZ4, Zstd), possible on a file by file (or file type) basis, through extended file attributes.
- Maximum partition size: 16TB, maximum file size: 3.94TB
- Minimum partition size: 52MiB (8MiB free space)

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

SquashFS — A Read-Only and Compressed File System

The most popular choice for this usage

- Started by Phillip Lougher, since 2009 in the mainline kernel, actively maintained.
- 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 (Gzip, LZO, XZ, LZ4, Zstd)
- Supposed to give priority to compression ratio vs read performance
- Suitable for very small partitions

https://en.wikipedia.org/wiki/SquashFS

EROFS — Enhanced Read-Only File System

A more recent read-only, compressed solution

- Started by Gao Xiang (Huawei), since 2019 in the mainline kernel.
- Used in particular in Android phones (Huawei, Xiaomi, Oppo...)
- Supposed to give priority to read performance vs compression ratio
- EROFS implements compression into fixed 4KB blocks (better for read performance), while SquashFS uses fixed-sized blocks of uncompressed data.
- Unlike Squashfs, EROFS also allows for random access to files in directories.
- Development seems more active than on SquashFS.
- Suitable for very small partitions

https://en.wikipedia.org/wiki/EROFS

Our advice for choosing the best filesystem

- Some filesystems will work better than others depending on how you use them.
- 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.
- If you haven't done benchmarks yet, a good default choice is `ext4` for read/write partitions.

Filesystem benchmarks

	Boot time [1]	Mount time	Read	Seq. read	Write	Seq. write	read write delete	Delete	Space
ext4	very good	very good	fair	good	best	very good	good	good	good
xfs	bad	average	fair	good	very good	best	average	fair	fair
btrfs	worst	good	fair	good	good	good	fair	worst	good
f2fs	fair	average	good	good	fair	very good	very good	average	worst
squashfs	excel-lent	best	very good					very good	best
erofs	best	best	best					best	very good

1. Boot time = Module loading time + Mount time

See our presentation for more details and benchmarks (Linux 6.3, ARM32 BeagleBone Black):
https://bootlin.com/pub/conferences/2023/eoss/opdenacker-finding-best-block-filesystem/

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 /run
mount -t tmpfs shm /dev/shm
```

- See `filesystems/tmpfs` in kernel documentation.

Using block filesystems

Creating filesystems

- To create an empty ext4 filesystem on a block device or inside an already-existing image file
  - `mkfs.ext4 /dev/sda3`
  - `mkfs.ext4 disk.img`
- To create a filesystem image from a directory containing all your files and directories
  - For some filesystems, there are utilities to create a filesystem image from an existing directory:
    - *ext2*: `genext2fs -d rootfs/ rootfs.img`
    - *squashfs*: `mksquashfs rootfs/ rootfs.sqfs` (details later)
    - *erofs*: `mkfs.erofs rootfs.erofs rootfs/`
  - For other (read-write) filesystems: create a disk image, format it, mount it (see next slides), copy contents and umount.
  - 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 modify its contents from the development workstation, using the **loop** mechanism:
- Example:

```
mkdir /mnt/test
mount -t ext4 -o loop rootfs.img /mnt/test
```

- In the `/mnt/test` 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!

How to access partitions in a disk image

- You may have dumped a complete block device (with partitions) into a disk image.
- The `losetup` command allows to manually associate a loop device to a file, and offers a `--partscan` option allowing to also create extra block device files for the partitions inside the image:

```
$ sudo losetup -f --show --partscan disk.img
/dev/loop2

$ ls -la /dev/loop2*
brw-rw---- 1 root disk   7,  2 Jan 14 10:50 /dev/loop2
brw-rw---- 1 root disk 259, 11 Jan 14 10:50 /dev/loop2p1
brw-rw---- 1 root disk 259, 12 Jan 14 10:50 /dev/loop2p2
```

- Each partition can then be accessed individually, for example:

```
$ mount /dev/loop2p2 /mnt/rootfs
```

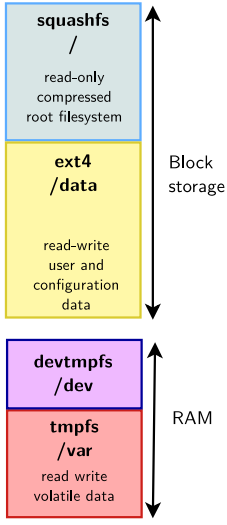
Creating squashfs filesystems

- 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)
- Similar commands exist for EROFS

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)

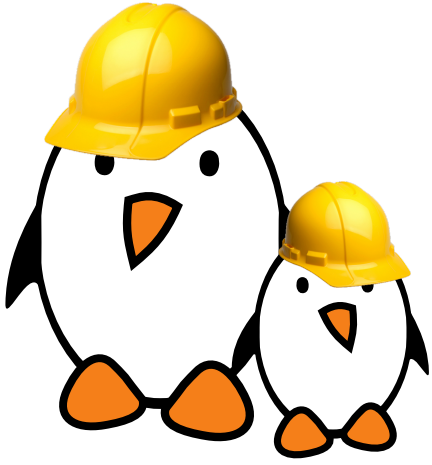


Issues with flash-based block storage

- 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.

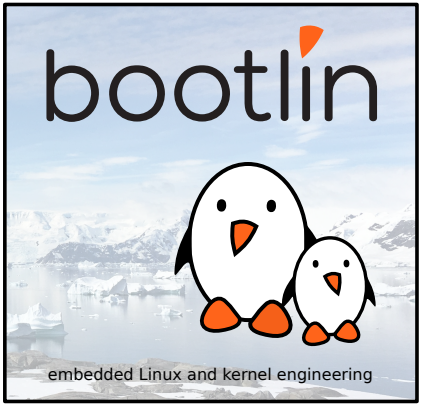
See the *Optimizing Linux with cheap flash drives* article from Arnd Bergmann and try his *flashbench* tool (`http://git.linaro.org/people/arnd/flashbench.git/about/`) for finding out the erase block and page size for your storage, and optimizing your partitions and filesystems for best performance. 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 further partitions on your SD card
- Booting a system with a mix of filesystems: *SquashFS* for the root filesystem, *ext4* for data, and *tmpfs* for temporary system files.
- Loading everything from the SD card, including the kernel and device tree.

Flash storage and filesystems



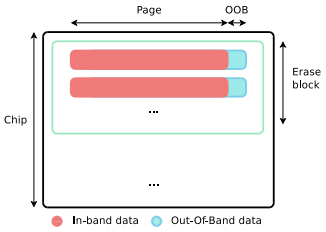
© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Block devices vs raw flash devices: reminder

- Block devices:
  - Allow for random data access using fixed size blocks
  - Do not require special care when writing on the media
  - Block size is relatively small (minimum 512 bytes, can be increased for performance reasons)
  - Considered as reliable (if the storage media is not, some hardware or software parts are supposed to make it reliable)
- Raw flash devices:
  - Flash chips directly driven by the flash controller on your SoC. You can control how they are managed.
  - Allow for random data access too, but require erasing before writing on the media.
  - Read and write (for example 4 KiB) don't use the same block size as erasing (for example 128 KiB).
  - Multiple flash technologies: NOR flash, NAND flash (Single Level Cell - SLC: 1 bit per cell, MLC: multiple bits per cell).

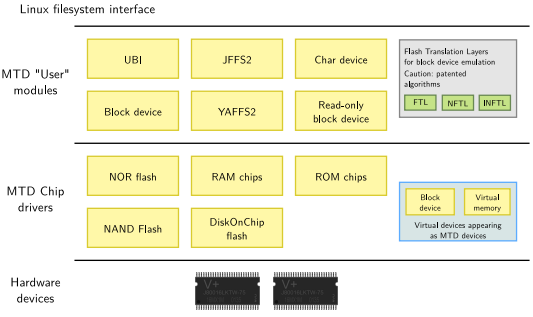
NAND flash storage: constraints

- Reliability
  - Reliability depends on flash technology (SLC, MLC)
  - Require mechanisms to recover from bit flips: ECC (Error Correcting Code), stored in the OOB (Out-Of-Band area)
- Lifetime
  - Relatively short lifetime: between 1,000,000 (SLC) and 1,000 (MLC) erase cycles per block
  - Wear leveling required to erase blocks evenly
  - Bad block detection/handling required too
- Widely used anyway in embedded systems for several reasons: low cost, high capacity, good read and write performance.



The MTD subsystem

- MTD stands for *Memory Technology Devices*
- Generic subsystem in Linux dealing with all types of storage media that are not fitting in the block subsystem
- Supported media types: RAM, ROM, NOR flash, NAND flash, Dataflash...
- Independent of the communication interface (drivers available for parallel, SPI, direct memory mapping, ...)
- Abstract storage media characteristics and provide a simple API to access MTD devices



MTD partitioning

- MTD devices are usually partitioned
  - It allows to use different areas of the flash for different purposes: read-only filesystem, read-write filesystem, backup areas, bootloader area, kernel area, etc.
- Unlike block devices, which contains their own partition table, the partitioning of MTD devices is described externally (don't want to put it in a flash sector which could become bad)
  - Specified in the board Device Tree (default partitions, not always relevant)
  - Specified through the kernel command line
- MTD partitions are defined through the `mtdparts` parameter in the kernel command line
- U-Boot understands the Linux syntax via the `mtdparts` and `mtdids` variables

MTD partitions on Linux

- Each partition becomes a separate MTD device
- Different from block device labeling (`sda3`, `mmcblk0p2`)
- `/dev/mtd0` is the first enumerated partition on the system
- `/dev/mtd1` is the second enumerated partition on the system (either from a single flash chip or from a different one).
- Note that the master MTD device (the device those partitions belong to) is not exposed in `/dev`

Commands to manage NAND devices

- From U-Boot
  - `help nand` to see all `nand` subcommands
  - `nand info`, `nand read`, `nand write`, `nand erase`...
- From Linux
  - **mtdchar** driver: one `/dev/mtdX` and `/dev/mtdXro` character device per partition.
  - Accessed through `ioctl()` operations to erase and flash the storage.
  - Used by these utilities: `flash_eraseall`, `nandwrite`
    Provided by the *mtd-utils* package, also available in BusyBox
  - There are also host commands in *mtd-utils*: `mkfs.jffs2`, `mkfs.ubifs`, `ubinize`...

Flash wear leveling

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

Flash file-systems

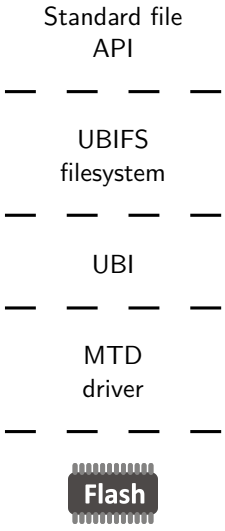
- 'Standard' file systems (*ext2*, *ext4*...) are meant to work on block devices
- Specific file systems have been developed to deal with flash constraints
- These file systems are relying on the MTD layer to access flash chips
- There are several legacy flash filesystems which might be useful for small partitions: JFFS2, YAFFS2.
- Nowadays, UBI/UBIFS is the de facto standard for medium to large capacity NANDs

UBI (1)

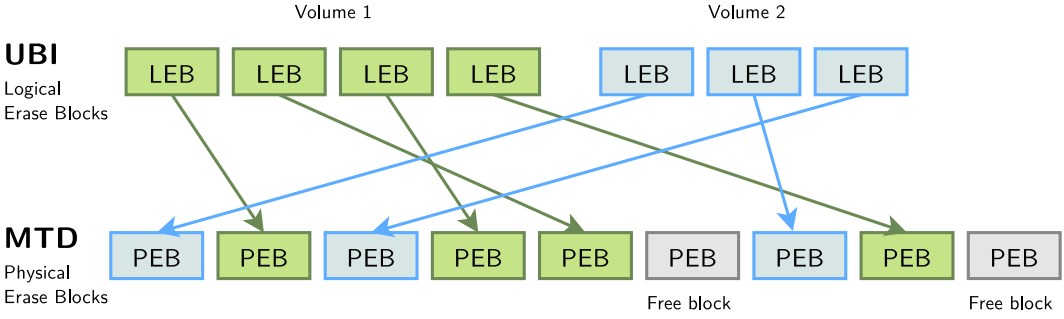
UBI: Unsorted Block Images

- Design choices:
  - Split the wear leveling and filesystem layers
  - Add some flexibility
  - Focus on scalability, performance and reliability
- Drawback: introduces noticeable space overhead, especially when used on small devices or partitions. JFFS2 still makes sense on small MTD partitions.
- Implements logical volumes on top of MTD devices (like LVM for block devices)
- Allows wear leveling to operate on the whole storage, not only on individual partitions.

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



UBI (2)

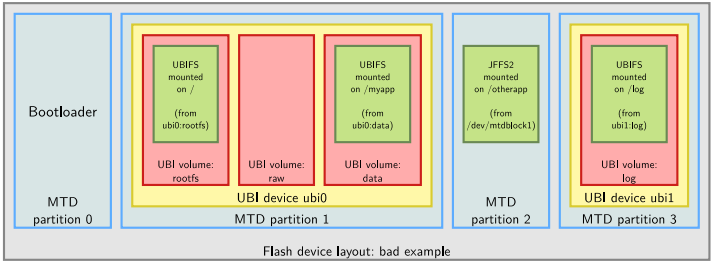


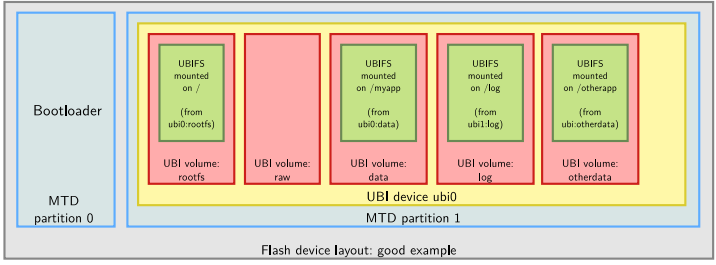
When there is too much activity on an LEB, UBI can decide to move it to another PEB with a lower erase count. Even read-only volumes participate to wear leveling!

UBI: good practice

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

UBI: bad and good practice



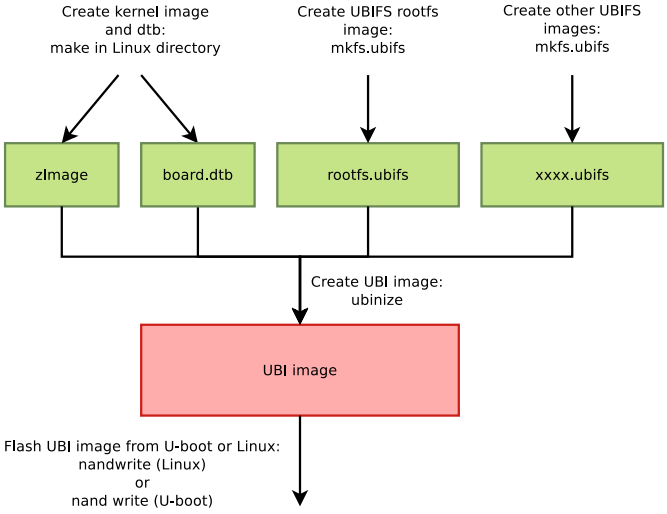


UBIFS

Unsorted Block Images File System

- http://www.linux-mtd.infradead.org/doc/ubifs.html
- Journaling file system providing better performance than its predecessor (JFFS2) and addressing its scalability issues
- Can be mounted as the root filesystem too
- UBIFS filesystem images can be created using `mkfs.ubifs` from *mtd-utils*
- This image can then be flashed on a volume or included in a UBI image (`ubinize` command).

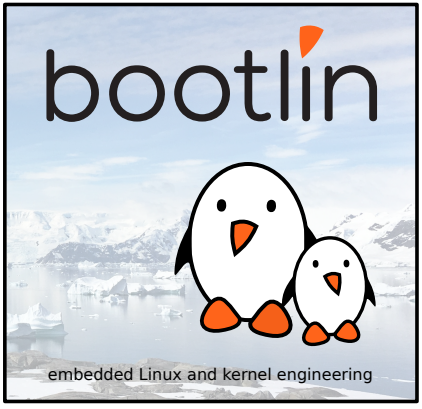
ubinize for UBI image creation



Linux: Block emulation layers

- Sometimes needed to use read-only block filesystems such as Squashfs and EROFS
- Linux provides two block emulation layers:
  - `mtdblock` (`CONFIG_MTD_BLOCK`): block devices emulated on top of MTD devices.
    - Named `/dev/mtdblockX`, one for each partition.
    - Originally the `mount` command wanted a block device to mount JFFS2 and YAFFS2.
    - Don't write to `mtdblock` devices: bad blocks are not handled!
  - `ubiblock` (`CONFIG_MTD_UBI_BLOCK`): **read-only** block devices emulated on top of UBI volumes
    - Used on static (read-only) volumes
    - Usually named `/dev/ubiblockX_Y`, where X is the UBI device id and Y is the UBI volume id (example: `/dev/ubiblock0_3`)

Cross-compiling user-space libraries and applications



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Integrating user-space libraries and applications

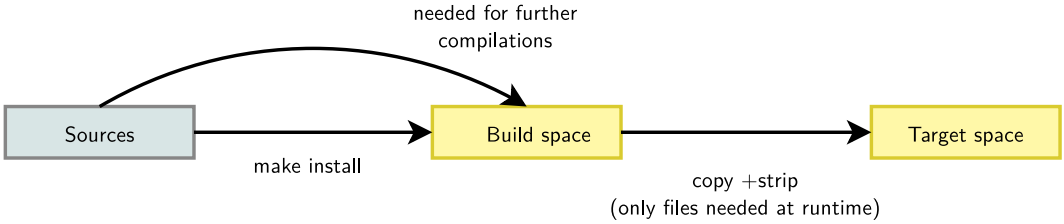
- One of the advantages of embedded Linux is the wide range of third-party libraries and applications that one can leverage in its product
- There's much more than U-Boot, Linux and Busybox that we can re-use from the open-source world
- Networking, graphics, multimedia, crypto, language interpreters, and more.
- Each of those additional software components needs to be cross-compiled and installed for our target
- Including all their dependencies
  - Which can be quite complex as open-source encourages code re-use

Concept of build system

- Each open-source software project comes with its own set of scripts/files to control its configuration/compilation: its *build system*
  - Detect if system requirements/dependencies are met
  - Compile all source files, to generate applications/libraries, as well as documentation
  - Installs build products
- Most common build systems:
  - Hand-written *Makefiles*
  - *Autotools*: *autoconf*, *automake*, *libtool*
    https://en.wikipedia.org/wiki/GNU_Autotools
  - *CMake*
    https://cmake.org/
  - *Meson*
    https://mesonbuild.com/
  - Language specific build systems for Python, Perl, Go, Rust, NodeJS, etc.

Target and staging spaces

- When manually cross-compiling software, we will distinguish two “copies” of the root filesystem
  1. The target root filesystem, which ends up on our embedded hardware, which contains only what is needed for *runtime*
  2. The staging space, which has a similar layout, but contains a lot more files than the *target* root filesystem: headers, static libraries, documentation, binaries with debugging symbols. Contains what's needed for *building* code.
- Indeed, we want the root filesystem on the target to be as minimal as possible.



Cross-compiling with hand-written Makefiles

- There is no general rule, as each project has a different set of Makefiles, that use a different set of variables
- Though it is common to use `make` standard variables: `CC` (C compiler path), `CXX` (C++ compiler path), `LD` (linker path), `CFLAGS` (C compiler flags), `CXXFLAGS` (C++ compiler flags), `LDFLAGS` (linker flags)
- `DESTDIR` for installation destination, sometimes `PREFIX` for execution location
- Common sequence

```
$ make CC=arm-linux-gcc CFLAGS=-I/path/to/headers \
       LDFLAGS=-L/path/to/libraries
$ make DESTDIR=/installation/path install
```

- Need to read the documentation (if any), read the Makefiles, and adapt to their behavior.

Example: *uftp* native compilation

Download and extract

```
$ wget http://sourceforge.net/projects/uftp-multicast/files/\
      source-tar/uftp-5.0.tar.gz
$ tar xf uftp-5.0.tar.gz
$ cd uftp-5.0
```

Build and install

```
$ make
cc  -g -Wall -Wextra [...]  -c server_announce.c
[...]
cc  -g -Wall -Wextra -o uftp uftp_common.o encrypt_openssl.o \
   server_announce.o [...] server_main.o \
   -lm -lcrypto  -lpthread
$ make DESTDIR=/tmp/test install
```

Look at installed files

```
$ tree /tmp/test
/tmp/test/
    usr
        bin
            uftp
            uftpd
            [...]
        share
            man
                man1
                    uftp.1
                    [...]

$ file /tmp/test/usr/bin/uftp
/tmp/test/usr/bin/uftp: ELF 64-bit LSB executable, x86-64
```

Example: *uftp* cross-compilation

First attempt

```
$ export PATH=/xtools/gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf/bin:$PATH
$ make CC=arm-none-linux-gnueabihf-gcc
[...]
encryption.h:87:10: fatal error: openssl/rsa.h: No such file or directory
```

- Build fails because *uftp* uses *OpenSSL*
- This is an optional dependency that can be disabled using the special `make` variable `NO_ENCRYPTION`

Second attempt

```
$ make CC=arm-none-linux-gnueabihf-gcc NO_ENCRYPTION=1
arm-none-linux-gnueabihf-gcc  -g -Wall -Wextra [...]  -c server_announce.c
[...]
arm-none-linux-gnueabihf-gcc  -g -Wall -Wextra -o uftp uftp_common.o \
   encrypt_none.o server_announce.o [...] -lm   -lpthread
$ make DESTDIR=/tmp/target NO_ENCRYPTION=1 install
$ file /tmp/target/usr/bin/uftp
/tmp/target/usr/bin/uftp: ELF 32-bit LSB executable, ARM
```

Example: *OpenSSL* cross-compilation

OpenSSL has a hand-written `Configure` shell script that needs to be invoked before the build.

Download/extract

```
$ wget https://www.openssl.org/source/openssl-1.1.1q.tar.gz
$ tar xf openssl-1.1.1q.tar.gz
$ cd openssl-1.1.1q
```

Configuration/build

```
$ CC=arm-none-linux-gnueabihf-gcc ./Configure --prefix=/usr \
    linux-generic32 no-asm
$ make
$ make DESTDIR=/tmp/staging install
```

Installed files

```
$ tree /tmp/staging
    usr
        bin
            openssl
        include
            openssl
                rsa.h
                [...]
        lib
            libcrypto.a
            libcrypto.so -> libcrypto.so.1.1
            libcrypto.so.1.1
            [...]
            pkgconfig
                libcrypto.pc
                [...]
        share
            doc
                openssl
            man
```

Example: *uftp* with *OpenSSL* support

```
$ make CC=arm-none-linux-gnueabihf-gcc
encryption.h:87:10: fatal error: openssl/rsa.h:
   No such file or directory
[...]
```

It cannot find the header, let's add `CFLAGS` pointing to where OpenSSL headers are installed.

```
$ make CC=arm-none-linux-gnueabihf-gcc \
       CFLAGS=-I/tmp/staging/usr/include
[... build OK, but at link time ...]
ld: cannot find -lcrypto
```

Compilation of object files work, but link fails as the linker cannot find the OpenSSL library. Let's add `LDFLAGS` pointing to where the OpenSSL libraries are installed.

```
$ make CC=arm-none-linux-gnueabihf-gcc \
       CFLAGS=-I/tmp/staging/usr/include \
       LDFLAGS=-L/tmp/staging/usr/lib
[... builds OK! ...]
$ make DESTDIR=/tmp/target install
```

Now it builds and installs fine!

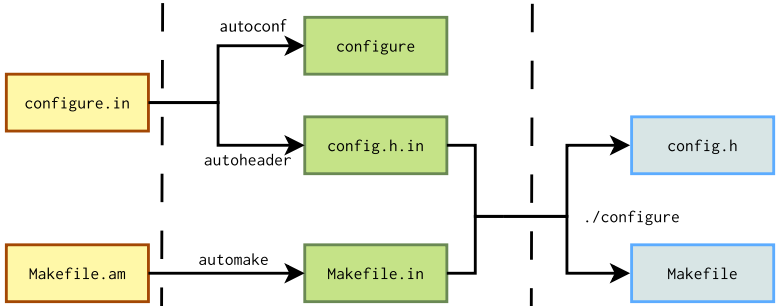
```
$ arm-none-linux-gnueabihf-readelf -d /tmp/target/usr/bin/uftp
[...]
 0x00000001 (NEEDED) Shared library: [libm.so.6]
 0x00000001 (NEEDED) Shared library: [libcrypto.so.1.1]
 0x00000001 (NEEDED) Shared library: [libpthread.so.0]
 0x00000001 (NEEDED) Shared library: [libc.so.6]
[...]
```

We can indeed see that `uftp` is linked against the `libcrypto.so.1.1` shared library.

Autotools

- 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
  - **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 large number of software components, even though *Meson* is gaining significant traction as a replacement today
- See also Bootlin Autotools training materials

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
  - Software downloaded as a tarball: usually shipped pre-generated in the tarball
  - Software downloaded from Git: no pre-generated files under version control, so they must be generated
- The `Makefile` files are generated at configure time, before compiling
  - They are never shipped in the software package.

autotools usage: four steps

1. Only if needed: generate `configure` and `Makefile.in`. Either using *autoreconf* tool, or sometimes an `autogen.sh` script is provided by the package
2. **Configuration:** `./configure`
   - `./configure --help` is very useful
   - `--prefix`: execution location
   - `--host`: target machine when cross-compiling, if not provided, auto-detected. Also used as the cross-compiler prefix.
   - Often `--enable-<foo>`, `--disable-<foo>`, `--with-<foo>`, `--without-<foo>` for optional features.
   - `CC`, `CXX`, `CFLAGS`, `CXXFLAGS`, `LDFLAGS` and many more variables
3. **Build:** `make`
4. **Installation:** `make install`
   - `DESTDIR` variable for *diverted installation*

Example: can-utils native compilation

Download

```
$ git clone https://github.com/linux-can/can-utils.git
$ cd can-utils/
$ git checkout v2021.08.0
$ ls -1 configure* *makefile*
configure.ac
GNUmakefile.am
```

No `configure` and `GNUmakefile.in`, *autoreconf needed*.

Autoreconf

```
$ autoreconf -i
$ ls -1 configure* *makefile*
configure
configure.ac
GNUmakefile.am
GNUmakefile.in
```

Configuration

```
$ ./configure --prefix=/usr
$ ls -1 *makefile*
GNUmakefile
GNUmakefile.am
GNUmakefile.in
```

We now have the `GNUmakefile`, we can build and install.

Build/install

```
$ make
$ make DESTDIR=/tmp/test install
$ file /tmp/test/usr/bin/candump
/tmp/test/usr/bin/candump: ELF 64-bit LSB executable, x86-64
```

Example: *can-utils* cross-compilation

```
$ export PATH=/xtools/gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf/bin:$PATH
$ ./configure --prefix=/usr --host=arm-none-linux-gnueabihf
$ make
$ make DESTDIR=/tmp/target install
$ file /tmp/target/usr/bin/candump
/tmp/target/usr/bin/candump: ELF 32-bit LSB executable, ARM
```

Note: This is a simple example, as *can-utils* does not have any dependency other than the C library, and has a simple `configure.ac` file.

CMake

https://en.wikipedia.org/wiki/CMake

- More modern build system, started in 1999, maintained by a company called *Kitware*
- Used by Qt 6, KDE, and many projects which didn't like *autotools*
- Perhaps losing traction these days in favor of *Meson*
- Needs `cmake` installed on your machine
- Based on:
  - `CMakeLists.txt` files that describe what the dependencies are and what to build and install
  - `cmake`, a tool that processes `CMakeLists.txt` to generate either Makefiles (default) or Ninja files (covered later)
- Typical sequence, when using the *Makefile* backend:
  1. `cmake .`
  2. `make`
  3. `make install`

Example: *cJSON* native compilation

Download

```
$ git clone https://github.com/DaveGamble/cJSON.git
$ cd cJSON
$ git checkout v1.7.15
```

Configure, build, install

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr .
$ make
$ make DESTDIR=/tmp/test install
```

Installed files

```
$ tree /tmp/test
/tmp/test/
    usr
        include
            cjson
                cJSON.h
        lib64
            cmake
                cJSON
                    cjson.cmake
                    cJSONConfig.cmake
                    cJSONConfigVersion.cmake
                    cjson-noconfig.cmake
            libcjson.so -> libcjson.so.1
            libcjson.so.1 -> libcjson.so.1.7.15
            libcjson.so.1.7.15
            pkgconfig
                libcjson.pc
```

Example: *cJSON* cross-compilation

cJSON has no dependency on any other library, so cross-compiling it is very easy as only the C cross-compiler needs to be specified:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_C_COMPILER=arm-none-linux-gnueabihf-gcc .
$ make
$ make DESTDIR=/tmp/target install
$ file /tmp/target/usr/lib/libcjson.so.1.7.15
/tmp/target/usr/lib/libcjson.so.1.7.15: ELF 32-bit LSB shared object, ARM
```

CMake *toolchain file*

- When cross-compiling with *CMake*, the number of arguments to pass to specify the paths to all cross-compiler tools, libraries, headers, and flags can become quite long.
- They can be grouped into a *toolchain file*, which defines *CMake* variables
- Can then be used with
  `cmake -DCMAKE_TOOLCHAIN_FILE=/path/to/toolchain-file.txt`
- Such a *toolchain file* is commonly provided by embedded Linux build systems: Buildroot, Yocto, etc.
- Facilitates cross-compilation using CMake
- https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html

Meson

https://en.wikipedia.org/wiki/Meson_(software)

- The most modern one, written in Python
- Gaining big traction in lots of major open-source projects
- Processes `meson.build` + `meson_options.txt` and generates *Ninja* files
- *Ninja* is an alternative to `make`, with much shorter build times
- Needs `meson` and `ninja` installed on your machine
- Meson requires an *out-of-tree* build: the build directory must be distinct from the source directory

```
1. mkdir build
2. cd build
3. meson ..
4. ninja
5. ninja install
```

Example: *ipcalc* native compilation

Download

```
$ git clone https://gitlab.com/ipcalc/ipcalc.git
$ cd ipcalc
$ git checkout 1.0.1
```

Configuration, build, installation

```
$ mkdir build
$ cd build
$ meson --prefix /usr ..
$ ninja
$ DESTDIR=/tmp/test ninja install
```

Installed files

```
$ tree /tmp/test
/tmp/test/
    usr
        bin
            ipcalc
```

Meson *cross file*

- In a similar manner to CMake's *toolchain file*, *Meson* has a concept of *cross file*
- Small text file that contains variable definitions telling *Meson* all details needed for cross-compilation
- Can be created manually, or may be provided by an embedded Linux build systems such as Buildroot or Yocto.
- `--cross-file` option of *Meson*

Cross file example

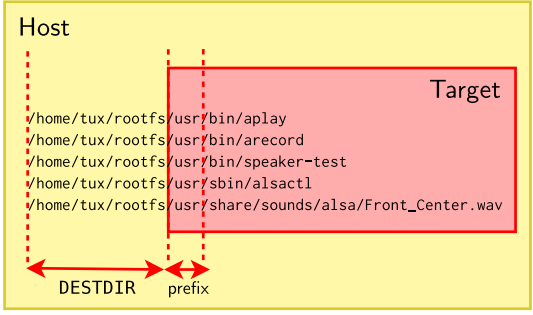
```
[binaries]
c = 'arm-none-linux-gnueabihf-gcc'
strip = 'arm-none-linux-gnueabihf-strip'

[host_machine]
system = 'linux'
cpu_family = 'arm'
cpu = 'cortex-a9'
endian = 'little'
```

Example: *ipcalc* cross-compilation

```
$ cat cross-file.txt
[binaries]
c = 'arm-none-linux-gnueabihf-gcc'
strip = 'arm-none-linux-gnueabihf-strip'

[host_machine]
system = 'linux'
cpu_family = 'arm'
cpu = 'cortex-a9'
endian = 'little'
$ mkdir build-cross
$ cd build-cross
$ meson --cross-file ../cross-file.txt --prefix /usr ..
$ ninja
$ DESTDIR=/tmp/target ninja install
$ file /tmp/target/usr/bin/ipcalc
/tmp/target/usr/bin/ipcalc: ELF 32-bit LSB executable, ARM
```

Distinction between *prefix* and *DESTDIR*

- There is often a confusion between *prefix* and *DESTDIR*
- Distinction is very important in cross-compilation context
- `prefix`: where the software will be executed from on the target
- `DESTDIR`: where the software is installed by the build system installation procedure. Allows to install in a different place than `prefix`, when creating a root filesystem for a different machine.



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
- https://people.freedesktop.org/~dbn/pkg-config-guide.html
- The database is made of `.pc` files, installed by default in `<prefix>/lib/pkgconfig/`.
- `pkg-config` is often used by *autotools*, *CMake*, *Meson* to find libraries
- 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.

pkg-config example for native compilation

```
$ pkg-config --list-all
openssl                         OpenSSL - Secure Sockets Layer and cryptography libraries and tools
zlib                            zlib - zlib compression library
blkid                           blkid - Block device id library
cairo-script                    cairo-script - script surface backend for cairo graphics library
cairo-pdf                       cairo-pdf - PDF surface backend for cairo graphics library
xcb-xinput                      XCB XInput - XCB XInput Extension (EXPERIMENTAL)
libcurl                         libcurl - Library to transfer files with ftp, http, etc.
[...]
$ pkg-config --cflags --libs openssl
-lssl -lcrypto
$ pkg-config --cflags --libs cairo-script
-I/usr/include/cairo -I/usr/include/libpng16 -I/usr/include/freetype2 -I/usr/include/harfbuzz
[...] -lcairo -lz
```

pkg-config example for cross-compilation

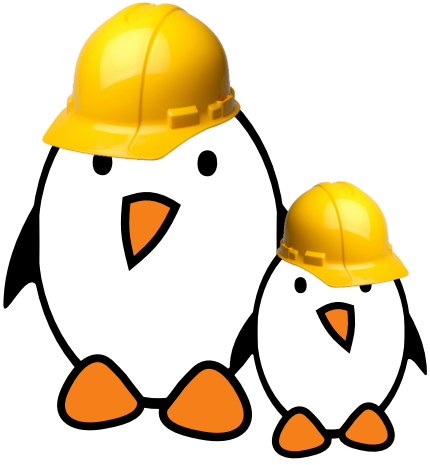
Use PKG_CONFIG_LIBDIR

```
$ export PKG_CONFIG_LIBDIR=/tmp/staging/usr/lib/pkgconfig
$ pkg-config --list-all
openssl                         OpenSSL - Secure Sockets Layer and cryptography libraries and tools
libssl                          OpenSSL-libssl - Secure Sockets Layer and cryptography libraries
libcrypto                       OpenSSL-libcrypto - OpenSSL cryptography library
$ pkg-config --cflags --libs openssl
-I/usr/include -L/usr/lib -lssl -lcrypto
```

The `-L/usr/lib` is incorrect, we need to use `PKG_CONFIG_SYSROOT_DIR`.

Use PKG_CONFIG_SYSROOT_DIR

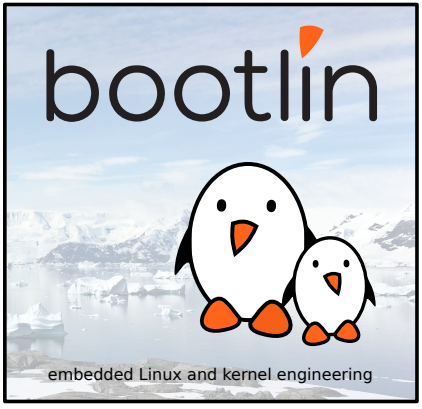
```
$ export PKG_CONFIG_SYSROOT_DIR=/tmp/staging/
$ pkg-config --cflags --libs openssl
-I/tmp/staging/usr/include -L/tmp/staging/usr/lib -lssl -lcrypto
```



Time to start the practical lab!

- Manual cross-compilation of several open-source libraries and applications for an embedded platform.
- Learning about common pitfalls and issues, and their solutions.
- This includes compiling *alsa-utils* package, and using its `speaker-test` program to test that audio works on the target.

Embedded system building tools



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Approaches

Three main approaches to build your embedded Linux system:

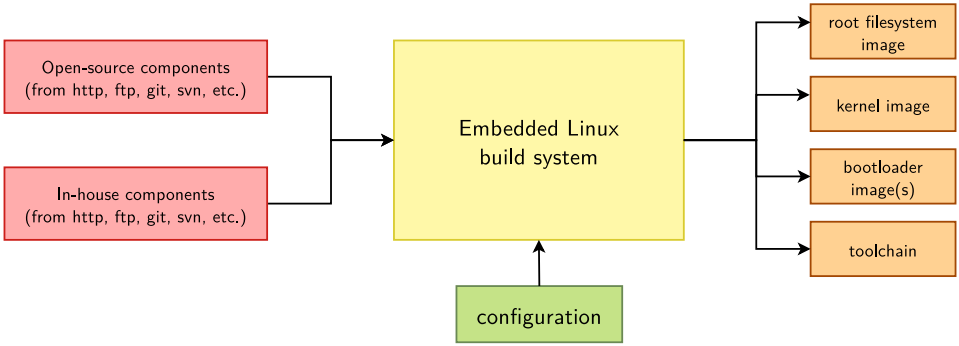
1. Cross-compile everything manually from source
2. Use an *embedded Linux build system* that automates the cross-compilation process
3. Use a binary distribution such as Debian, Ubuntu or Fedora

Approaches pros and cons

	Pros	Cons
Building everything manually	Full flexibility Learning experience	Dependency hell Need to understand a lot of details Version compatibility Lack of reproducibility
Binary distribution Debian, Ubuntu, Fedora, etc.	Easy to create and extend Extensive set of packages Usually excellent security maintenance	Hard to customize Hard to optimize (boot time, size) Hard to rebuild the full system from source Large system Uses native compilation (slow) No well-defined mechanism to generate an image Lots of mandatory dependencies Not available for all architectures
Build systems Buildroot, Yocto, PTXdist, OpenWrt, etc.	Nearly full flexibility Built from source: customization and optimization are easy Fully reproducible Uses cross-compilation Have embedded specific packages not necessarily in desktop distros Make more features optional	Not as easy as a binary distribution Build time

Embedded Linux build systems

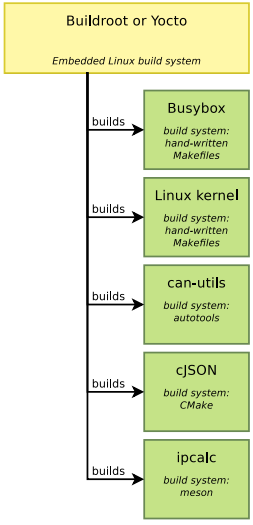
Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- Recipes for building components → easy

Build systems vs. Embedded Linux build systems

- Possible confusion between *build system* (Makefiles, autotools, CMake, Meson) and *embedded Linux build systems* (Buildroot, Yocto/OpenEmbedded, OpenWrt, etc.)
- *Build systems* are used by individual software components, to control the build process of each source file into a library, executable, documentation, etc.
- *Embedded Linux build systems* are tools that orchestrate the build of all software components one after the other. They invoke the *build system* of each software component.

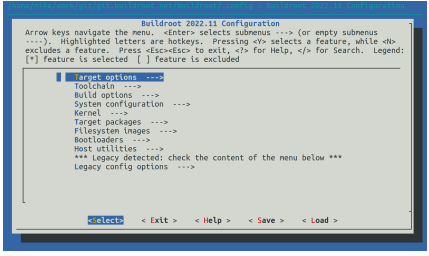


Buildroot: introduction

- 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 using 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, NodeJS, Go, Rust, 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: configuration and build

- 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:
  `make`
- Useful build results in `output/images/`



Buildroot: adding a new package

- A package allows to integrate a user application or library to Buildroot
- Can be used to integrate
  - Additional open-source libraries or applications
  - But also your own proprietary libraries and applications → fully integrated build process
- Each package has its own directory (such as `package/jose`). 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 `jose.mk` file (mandatory), describing how the package is built.
  - A `jose.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, Config.in

package/jose/Config.in

```
config BR2_PACKAGE_JOSE
        bool "jose"
        depends on BR2_TOOLCHAIN_HAS_THREADS
        select BR2_PACKAGE_ZLIB
        select BR2_PACKAGE_JANSSON
        select BR2_PACKAGE_OPENSSL
        help
          C-language implementation of Javascript Object Signing and
          Encryption.

          https://github.com/latchset/jose
```

package/Config.in

```
[...]
source "package/jose/Config.in"
[...]
```

Buildroot: adding new package, .mk file

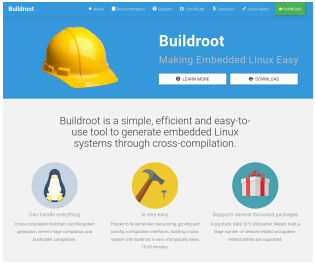
package/jose/jose.mk

```
JOSE_VERSION = 11
JOSE_SOURCE = jose-$(JOSE_VERSION).tar.xz
JOSE_SITE = https://github.com/latchset/jose/releases/download/v$(JOSE_VERSION)
JOSE_LICENSE = Apache-2.0
JOSE_LICENSE_FILES = COPYING
JOSE_INSTALL_STAGING = YES
JOSE_DEPENDENCIES = host-pkgconf zlib jansson openssl

$(eval $(meson-package))
```

- The package directory and the prefix of all variables must be identical to the suffix of the main configuration option BR2_PACKAGE_JOSE
- The meson-package infrastructure knows how to build *Meson* packages. Many other infrastructures exist, for different *build systems*

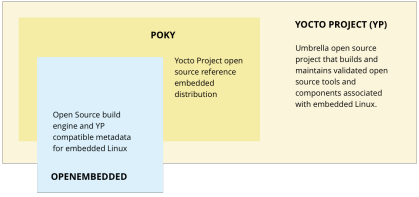
Buildroot resources

- Official site: https://buildroot.org/
- Buildroot manual: https://buildroot.org/downloads/manual/manual.html
- Complete *Buildroot system development* training course from Bootlin
  - https://bootlin.com/training/buildroot/
  - Freely available training materials

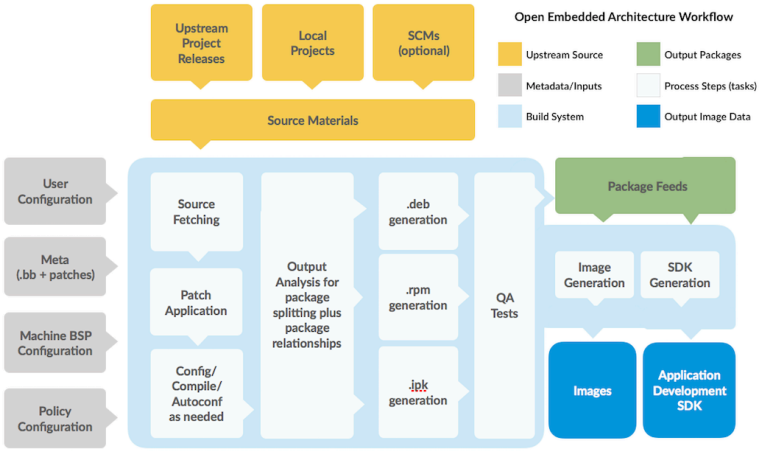


Yocto Project / OpenEmbedded

- OpenEmbedded
  - Started in 2003
  - Goal is to build custom Linux distributions for embedded devices
  - Back then, no stable releases, limited/no documentation, difficult to use for products
- Yocto Project
  - Started in 2011
  - By the Linux Foundation
  - Goal is to *industrialize* OpenEmbedded
  - Funds the development of OpenEmbedded, makes regular stable releases, QA effort, extensive documentation
  - One Long Term Support release every 2 years, supported for 4 years.



Yocto Project overview



Yocto Project concepts

- Terminology
  - **Layer**: Git repository containing a collection of recipes, machines, etc.
  - **Recipe**: metadata that describes how to build a particular software component, the contents of an image to generate
  - **Machine**: a specific hardware platform
  - **bitbake**: the orchestration tool that processes *recipes* to generate the final products
- Yocto/OpenEmbedded generate a *distribution*
  - For each recipe, it produces one or several binary packages (`deb`, `rpm`, `ipk`)
  - A selection of these binary packages are installed to create a *root filesystem image* that can be flashed
  - The other packages can be installed at runtime on the system using a package management system: `apt`, `dnf`, `opkg`

Public layers (1/2)

- Core layers
  - bitbake, not really a layer, but the core build orchestration tool
  - openembedded-core, the very core recipes, to build the most common software packages: Linux, BusyBox, toolchain, systemd, mesa3d, X.org, Wayland bootloaders. Supports only QEMU machines.
  - poky, a layer from the Yocto Project that combines *openembedded-core*, *bitbake*, that defines the *Poky* distribution, a reference distribution. Supports a few more machines. In practice not useful for real projects.
  - meta-openembedded, community maintained additional recipes from the OpenEmbedded project
- BSP layers, provided by HW vendors or the community, to support additional hardware platforms: recipes for building custom Linux kernel, bootloaders, for HW-related software components
  - meta-intel, meta-arm, meta-ti, meta-xilinx, meta-freescale, meta-atmel, meta-st-stm32mp, etc.

Public layers (2/2)

- Additional software layers: recipes for building additional software components, not in *openembedded-core*
  - meta-qt6, meta-virtualization, meta-rauc, meta-swupdate, etc.
- Layer index: `https://layers.openembedded.org/`
- Each layer normally has a branch matching the Yocto release you're using
- Not all layers have the same level of quality/maintenance: third-party layers are not necessarily reviewed by OpenEmbedded experts.

Combine layers

- For your project, you will typically combine a number of public layers
  - At least the *openembedded-core* layer
  - Possibly one or several *BSP layers*
  - Possibly one or several additional *software layers*
- And you will create your *own layer*, containing recipes for:
  - Machine definitions for your custom hardware platforms
  - Image/distro definitions for your custom system(s)
  - Recipes for your custom software
- A tool is often used to automate the retrieval of the necessary layers, at the right version
  - Google repo tool, the Yocto-specific Kas utility

Yocto quick start: STM32MP1 example

Download *bitbake* and layers

```
$ git clone https://git.openembedded.org/openembedded-core -b kirkstone
$ git clone https://git.openembedded.org/meta-openembedded -b kirkstone
$ git clone https://git.openembedded.org/bitbake -b 2.0
$ git clone https://github.com/STMicroelectronics/meta-st-stm32mp.git -b kirkstone
```

Note: we're not using a tool such as *repo* or *Kas* here, we are fetching each layer manually.

Enter the build environment

```
$ source openembedded-core/oe-init-build-env
```

This automatically enters a directory called `build/`, with a few files/directories already prepared.

Yocto quick start: STM32MP1 example

Configure layers: *conf/bblayers.conf*

```
BBLAYERS ?= " \
  /path/to/openembedded-core/meta \
  /path/to/meta-st-stm32mp \
  /path/to/meta-openembedded/meta-oe \
  /path/to/meta-openembedded/meta-python \
  "
```

Start the build

```
$ MACHINE=stm32mp1 bitbake core-image-minimal
```

- `MACHINE=stm32mp1` will build images usable on all STM32MP1 platforms
- `core-image-minimal` builds a minimal image

Build results

```
$ ls tmp/deploy/images/stm32mp1/
```

Yocto recipe example

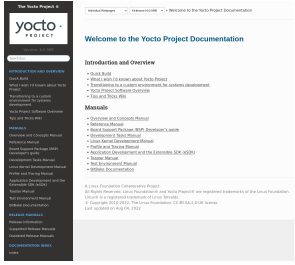
openembedded-core/tree/meta/recipes-extended/libmnl/libmnl_1.0.5.bb

```
SUMMARY = "Minimalistic user-space Netlink utility library"
DESCRIPTION = "Minimalistic user-space library oriented to Netlink developers, providing \
    functions for common tasks in parsing, validating, and constructing both the Netlink header and TLVs."
HOMEPAGE = "https://www.netfilter.org/projects/libmnl/index.html"
SECTION = "libs"
LICENSE = "LGPL-2.1-or-later"
LIC_FILES_CHKSUM = "file://COPYING;md5=4fbd65380cdd255951079008b364516c"

SRC_URI = "https://netfilter.org/projects/libmnl/files/libmnl-${PV}.tar.bz2"
SRC_URI[sha256sum] = "274b9b919ef3152bfb3da3a13c950dd60d6e2bcd54230ffeca298d03b40d0525"

inherit autotools pkgconfig

BBCLASSEXTEND = "native"
```

- Recipe to build libmnl
- Build system based on *autotools* → `inherit autotools`
- Available both for the target and the host → `BBCLASSEXTEND = "native"`

Yocto resources

- Official website: https://www.yoctoproject.org/
- Release information: https://wiki.yoctoproject.org/wiki/Releases
- Official documentation: https://docs.yoctoproject.org/
  - Maintained by Bootlin engineers!
- Complete *Yocto Project and OpenEmbedded system development* training course from Bootlin
  - https://bootlin.com/training/yocto/
  - Freely available training materials



Buildroot vs. Yocto: a few key differences

- What it builds
  - **Yocto**: builds a distribution, with binary packages and a package management system
  - **Buildroot**: builds a fixed functionality root filesystem, no binary packages
  - Note: binary packages are not necessarily a good thing for embedded!

Buildroot vs. Yocto: a few key differences

- What it builds
- Configuration
  - **Yocto**: flexible, powerful but complex configuration description
  - **Buildroot**: very simple configuration system, but sometimes limited

Buildroot vs. Yocto: a few key differences

- What it builds
- Configuration
- Build strategy
  - **Yocto**: complex and heavy logic, but with efficient caching of artifacts and "rebuild only what's needed" features
  - **Buildroot**: simple but somewhat dumb logic, no caching of built artifacts, full rebuilds needed for some config changes

Buildroot vs. Yocto: a few key differences

- What it builds
- Configuration
- Build strategy
- Ecosystem
  - **Yocto**: (relatively) small common base in OpenEmbedded, lots of features supported in third party layers → lots of things, but varying quality
  - **Buildroot**: everything in one tree → perhaps less things, but more consistent quality

Buildroot vs. Yocto: a few key differences

- What it builds
- Configuration
- Build strategy
- Ecosystem
- Complexity/learning curve
  - **Yocto**: admittedly steep learning curve, *bitbake* remains a magic black box for most people
  - **Buildroot**: much smoother and shorter learning curve, the tool is simple to approach, and reasonably simple to understand

Buildroot vs. Yocto: a few key differences

- What it builds
- Configuration
- Build strategy
- Ecosystem
- Complexity/learning curve
- And also a matter of personal taste/preference, as often when choosing tools

OpenWrt

- Another Embedded Linux build system
- Derived from Buildroot a *very* long time ago
  - Now completely different, except for the use of *Kconfig* and *make*
- Targeted at building firmware for WiFi routers and other networking equipments
- Unlike Buildroot or Yocto that leave a lot of flexibility to the user in defining the system architecture, OpenWrt makes a lot of set in stone decisions:
  - *musl* is the C library
  - an OpenWrt specific init system
  - an OpenWrt specific inter-process communication bus
  - a Web UI specific to OpenWrt
- The aim of OpenWrt is to build a final product out of the box, with support for popular networking products and development boards
- https://openwrt.org/

Working with distributions

Binary distributions

- Many popular Linux desktop/server distributions have support for embedded architectures
  - Debian: ARMv5, ARMv7, ARM64, i386, x86-64, MIPS, PowerPC, RISC-V in progress
  - Ubuntu: ARMv7, ARM64, x86-64, RISC-V (initial support), PowerPC64 little-endian
  - Fedora: ARMv7, ARM64, x86-64, MIPS little-endian, PowerPC64 little-endian, RISC-V
- Some more specialized Linux distributions as well
  - Raspberry Pi OS, a Debian derivative targeted at RaspberryPi platforms
  - Alpine Linux, a lightweight distribution, based on *musl* and *Busybox*, ARMv7, ARM64, i386, x86-64, PowerPC64 little-endian

Binary distributions pitfalls

- Be careful when using a binary distribution on how you create your system image, and how reproducible this process is
- We have seen projects use the following (bad) procedure:
  - Install a binary distribution manually on their target hardware
  - Install all necessary packages by hand
  - Compile the final applications on the target
  - Tweak configuration files directly on the target
  - Then duplicate the resulting SD card for all other boards
- This process is really bad as:
  - it is not reproducible
  - it requires installing many more things on the target than needed (development tools), increasing the footprint, the attack surface and the maintenance effort
- If you end up using a binary distribution in production, make sure you have an automated and reproducible process to generate the complete image, ready to flash on your target.

Debian/Ubuntu image building tools

ELBE

- **E.**mbedded **L.**inux **B.**uild **E.**nvironment
- Implemented in Python
- Uses an XML file as input to describe the system to generate
- Can use pre-built packages from Debian/Ubuntu repositories, but can also cross-compile and install additional packages
- https://elbe-rfs.org/
- Building Embedded Debian and Ubuntu Systems with ELBE talk
- ELBE: automated building of Ubuntu images for a Raspberry Pi 3B

DebOS

- Debian OS images builder
- Implemented in Go
- Uses a YAML file as input to describe the system to generate
- Creating Debian-Based Embedded Systems in the Cloud Using Debos talk
- https://github.com/go-debos/debos

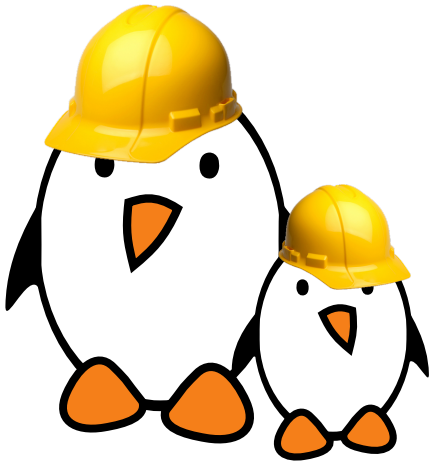
Android

- The obviously highly popular mobile operating system
- Uses the Linux kernel
- Most of the user-space is completely different from a normal embedded Linux system
  - Most components rewritten by Google
  - *bionic* C library
  - Custom *init* system and device management
  - Custom IPC mechanism, custom display stack, custom multimedia stack
  - Custom build system
- Android pitfalls for industrial embedded systems
  - Large footprint, and resource hungry
  - Complexity and build time
  - Maintenance issues: difficult to upgrade to newer releases due to increasing hardware requirements
- Embedded Android Training course from Opersys, with freely available training materials

Automotive Grade Linux, Tizen

- Industry groups collaborate around the creation of embedded Linux distributions targeting specific markets
  - These are regular embedded Linux systems, usually based on Yocto, with a selection of relevant open-source software components
  - Fund the development of missing features in existing components, or development of new software components
- Automotive Grade Linux
  - Linux Foundation project
  - *Collaborative open source project that is bringing together automakers, suppliers and technology companies to accelerate the development and adoption of a fully open software stack for the connected car*
  - https://www.automotivelinux.org/
- Tizen
  - Linux Foundation project too
  - Operating system targeting TVs, wearables, phones, in-vehicle infotainment, based on HTML5 applications.
  - https://www.tizen.org/

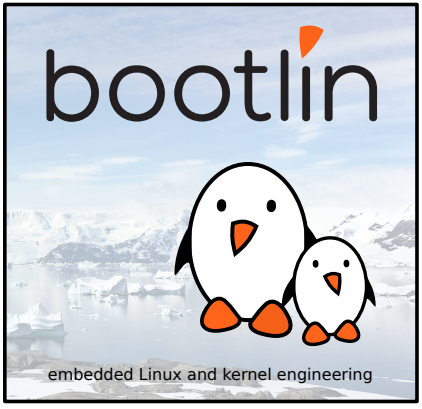
Practical lab - System build with Buildroot



Time to start the practical lab!

- Using Buildroot to rebuild the same basic system plus a sound playing server (*MPD*) and a client to control it (*mpc*).
- Overlaying the root filesystem built by Buildroot
- Driving music playback, directly from the target, and then remotely through an MPD client on the host machine.
- Analyzing dependencies between packages.
- Building *evtest* and using it to test the Nunchuk device driver.

Open source licenses and compliance



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

Free software vs. open-source

- **Free software**: term defined by the *Free Software Foundation*, grants 4 freedoms
  - 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
- **Open Source**: term defined by the *Open Source Initiative*, with 10 criterias
  - See https://www.opensource.org/docs/osd
- *Free Software* movement insists more on ethics, while *Open Source* insists more on the technical advantages
- From a freedom standpoint, they are similar.

Open source licenses

- All free software/open-source licenses rely on *copyright law*
- Those licenses fall in two main categories
  - The copyleft licenses
  - The non-copyleft licenses, also called *permissive* licenses

Non-Copyleft VS Copyleft licenses

Non-Copyleft
(BSD, MIT, Apache, X11...)

You can
Use
Modify
Redistribute

You must
Provide license text
Attribution

Copyleft
(GPL, LGPL, AGPL...)

You can
Use
Modify
Redistribute

You must
Provide license text
Attribution
Make source code available

What is *copyleft*

- The concept of *copyleft* is to ask for reciprocity in the freedoms given to a user.
- You receive software under a copyleft license and redistribute it, modified or not → you must do so under the same license
  - Same freedoms to the new users
  - Incentive, but no obligation, to contribute back your changes instead of keeping them secret
- Copyleft is *not* the opposite of copyright!
- Non-copyleft licenses have no such requirements: modified versions can be made proprietary, but they still require attribution
- https://en.wikipedia.org/wiki/Copyleft

Non-copyleft licenses

Most common non-copyleft licenses

- **MIT**, **BSD 2 CLAUSE**
  - Very simple
  - Require to preserve the copyright notice
- **BSD 3 CLAUSE**
  - Adds a non-endorsement clause
- **Apache**
  - More complex
  - Includes a *patent grant*, a mechanism to prevent users of the licensed project from suing others based on patents related to the project

Copyleft licenses

GPL: GNU General Public License

- The flagship license of the GNU project
- Used by Linux, BusyBox, U-Boot, Barebox, GRUB, many projects from GNU
- Is a copyleft license
  - Requires derivative works to be released under the same license
  - Source code must be redistributed, including modifications
  - If GPL code is integrated in your code, your code must now be GPL-licensed
  - Only applies when redistribution takes place
- Also called **strong** copyleft license
  - Programs linked with a library released under the GPL must also be released under the GPL
  - Does not prevent GPL programs and non-GPL programs from co-existing in the same system or to communicate
- https://www.gnu.org/licenses/gpl-2.0.en.html
- https://www.gnu.org/licenses/gpl-3.0.en.html
- https://en.wikipedia.org/wiki/GNU_General_Public_License

LGPL: GNU Lesser General Public License

- Used by *glibc*, *uClibc*, and many libraries
- Derived from the GPL license
- Also a copyleft license
- But a **weaker** copyleft license
  - 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.
  - Requires using dynamic linking, or in the case of static linking, to provide the object files to relink with the library
- https://www.gnu.org/licenses/lgpl-2.1.en.html
- https://www.gnu.org/licenses/lgpl-3.0.en.html
- https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License

GPL/LGPL: 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
- In all cases, the attribution and the license must be preserved

GPL/LGPL: version 2 vs version 3

- GPLv2/LGPLv2 published in 1991, widely used in the open-source world for major projects
- GPLv3/LGPLv3 published in 2007, and adopted by some projects
- Main differences
  - More *legalese* and definitions to clarify the license
  - Explicit patent grant
  - Grace period of 30 days to get back into compliance instead of immediate termination
  - Anti-Tivoization clause
- Anti-Tivoization
  - Requirement that the user must be able to **run** the modified versions on the device
  - Need to provide *installation instructions*
  - Only required for *User products*, i.e. consumer devices
  - On-going debate on how strong this requirement is, and how difficult it is to comply with

GPL: v2, v3, v2 or later, v3 or later

- Some projects are released under *GPLv2 only*
  - Examples: Linux kernel, U-Boot
- Some projects are released under *GPLv3 only*
- Some projects are released under *GPLv2 or later*
  - The recipient can chose to apply either the terms of GPLv2, GPLv3 or any later version
- Some projects are released under *GPLv3 or later*
  - The recipient can chose to apply the terms of GPLv3 or any later version (none of which exists today)
  - Examples: GCC, Samba, Bash, GRUB
- Note: this logic applies similarly to the LGPL license.

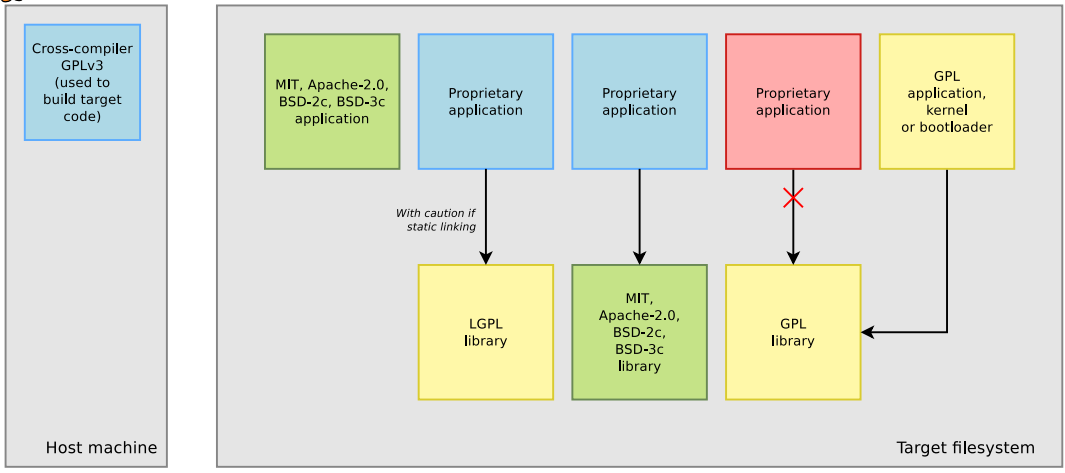
Dual licensing

- Some companies use a *dual licensing* business model, mainly for software libraries
- Their software is offered under two licenses:
  - A strong copyleft license, typically GPL, to encourage adoption of the software by the open-source world, allow the development and distribution of GPL licensed applications based on this library
  - A commercial license, offered against a fee, which allows to develop and distribute proprietary applications based on this library.
- Examples: Qt (only parts), MySQL, wolfSSL, Asterisk, etc.

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
- Check the simplified license description on tl;drLegal
  https://www.tldrlegal.com
- Otherwise, read the license text

Licensing: examples



Best practices

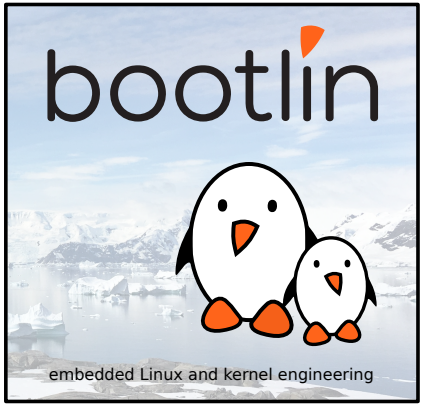
Respect free software licenses

- Free Software is not public domain software, the distributors have obligations due to the licenses
- **Before** using a free software component, make sure the license matches your project constraints
- Make sure to keep your modifications and adaptations well-separated from the original version.
- Make sure to keep a complete list of the free software packages you use, and the version in use
- Buildroot and Yocto Project can generate this list for you!
  - Buildroot: `make legal-info`
  - Yocto: see the project documentation
- Conform to the license requirements before shipping the product to the customers.

Keeping changes separate

- 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
- If possible, use the same version control system as the upstream project to maintain your changes.

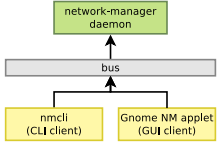
Overview of major embedded Linux software stacks



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

D-Bus

- *Message-oriented middleware mechanism that allows communication between multiple processes running concurrently on the same machine*
- Relies on a daemon to pass messages between applications
- Mainly used by system daemons to offer services to client applications
- Example: a network configuration daemon, running as *root*, offers a D-Bus API that CLI and GUI clients can use to configure networking
- Several busses
  - One system bus, accessible by all users, for system services
  - One session bus for each user logged in
- Object model: interfaces, objects, methods, signals
- https://www.freedesktop.org/wiki/Software/dbus/



systemd (1)

- Modern *init* system used by almost all Linux desktop/server distributions
- Much more complex than *Busybox init*, but also much more powerful
- Only supported with *glibc*, not with *uClibc* and *Musl*
- Provides features such as
  - Parallel startup of services, taking into account dependencies
  - Monitoring of services
  - On-demand startup of services, through *socket activation*
  - Resource-management of services: CPU limits, memory limits
- Configuration based on *unit files*
  - Declarative language, instead of shell scripts used in other init systems

systemd (2)

- Systemd also provides
  - *journald*, logging daemon, replacement for *syslogd*
  - *networkd*, network configuration management
  - *udevd*, hotplugging and `/dev` management
  - *logind*, login management
  - *systemctl*, tool to control/monitor systemd
  - And many, many other things
- https://systemd.io/

systemd service unit file example

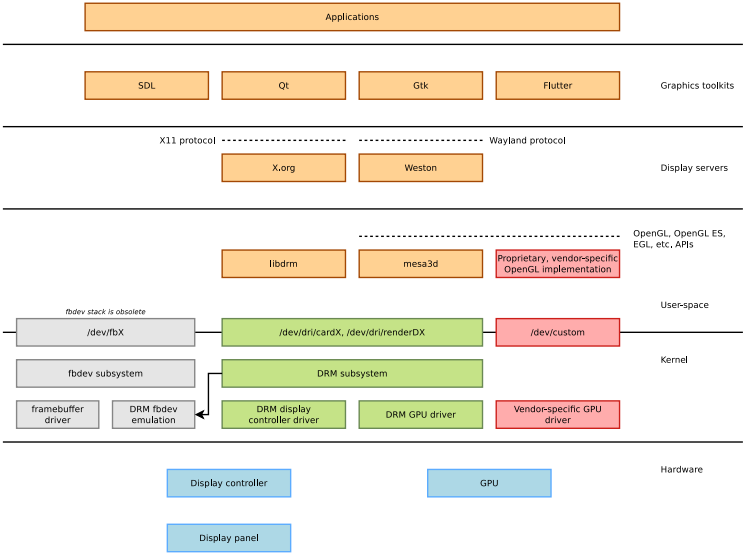
/usr/lib/systemd/system/sshd.service

```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.service
Wants=sshd-keygen.service

[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
```

Example systemctl/journalctl commands

- `systemctl status`, status of all services
- `systemctl status <service>`, status of one service
- `systemctl [start|stop] <service>`, start or stop a service
- `systemctl [enable|disable] <service>`, enable or disable a service, i.e. whether it should start at boot time
- `systemctl list-units`, list all available units
- `journalctl -a`, all logs
- `journalctl -f`, show the last entries, and keep printing new entries as they arrive
- `journalctl -u`, logs from a particular service

Linux graphics stack overview



Display controller support

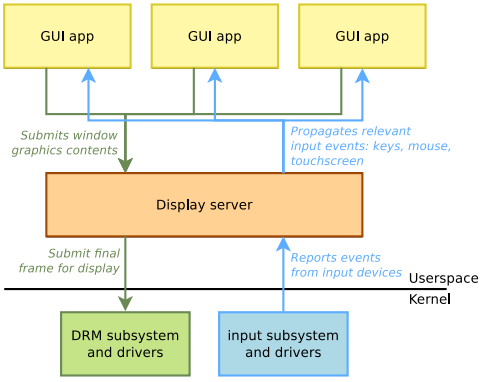
- Deprecated Linux kernel subsystem: *fbdev*
  - Still a few old graphics drivers only available in this subsystem
  - If possible, don't use!
  - https://en.wikipedia.org/wiki/Linux_framebuffer
- Modern Linux kernel subsystem: *DRM*
  - Supports display controllers of SoC or graphics cards, and all types of display panels and bridges: parallel, LVDS, DSI, HDMI, DisplayPort, etc.
  - Also supports small display panels connected over I2C or SPI
  - Devices exposed as `/dev/dri/cardX`
  - Companion user-space library: `libdrm`, includes a very handy test tool: `modetest`
  - https://en.wikipedia.org/wiki/Direct_Rendering_Manager

GPU support: OpenGL acceleration

- Open-source
  - A kernel driver in the DRM subsystem to send commands to the GPU and manage memory
  - `mesa3d` user-space library implementing the various OpenGL APIs, contains massive GPU-specific logic
  - More and more GPUs supported
  - https://www.mesa3d.org/
- Proprietary
  - Many embedded GPUs used to be supported only through proprietary blobs → long-term maintenance issues
  - A kernel driver provided out-of-tree by the vendor → they are not accepted upstream if the user-space is closed source
  - A (huge) closed-source user-space binary blob implementing the various OpenGL APIs

Concept of display servers

- The Linux kernel does not handle the *multiplexing* of the display and input devices between applications
  - Only one user-space application can use a display and a given set of input devices
- Display servers are special user-space applications that multiplex display/input by:
  - Allowing multiple client GUI applications to submit their window contents
  - Composing the final frame visible on the screen, based on contents submitted by applications, window visibility and layering
  - Propagating input events to the appropriate clients, based on focus



X11 and X.org

- *X.org* is the historical display server on UNIX systems, including Linux
- Implements the *X11* protocol, used between clients and the server
  - UNIX socket for local clients, TCP for remote clients
- On modern Linux, works on top of DRM or fbdev for graphics, input subsystem for input events
- Still maintained, but now legacy.
- X11 license
- https://www.x.org

Wayland

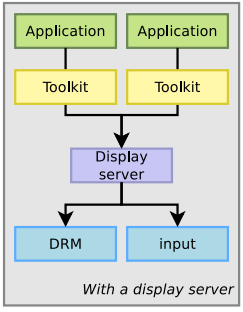
- *Communication* **protocol** *that specifies the communication between a display server and its clients, as well as a C library implementation of that protocol*
- A display server using the Wayland protocol is called a Wayland **compositor**
- Modern replacement for the aging X11 protocol
- More heavily based on OpenGL technologies
- https://wayland.freedesktop.org/
- https://en.wikipedia.org/wiki/Wayland_(display_server_protocol)

Wayland compositors

- Weston
  - The reference compositor
  - https://gitlab.freedesktop.org/wayland/weston
- Mutter, used by the GNOME desktop environment
  https://gitlab.gnome.org/GNOME/mutter
- wlroots, a Wayland compositor library, used by
  - Cage, a Wayland kiosk-style compositor
    https://github.com/Hjdskes/cage
  - swayWM, a tiling Wayland compositor
    https://swaywm.org/
- And many more
  https://wiki.archlinux.org/title/wayland#Compositors

Concept of graphics toolkits

- The X11 and Wayland protocols are very low-level protocols
- While possible, developing applications directly using those protocols or their corresponding client libraries would be painful
- Existence of *toolkits*
  - Some of them work only on top of a display server: X11 or Wayland
  - Some of them can work directly on top of DRM + input, for single full-screen applications
- Widget-oriented toolkits, with APIs to create windows, buttons, text fields, drop-down lists, etc.
- Game/multimedia-oriented toolkits, with no pre-defined widget API

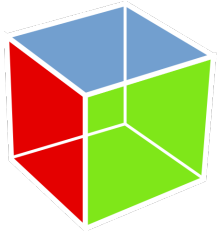


Qt

- Highly popular and well-documented development framework, providing:
  - Core libraries: data structures, event handling, XML, databases, networking, etc.
  - Graphics libraries: widgets and more
- Standard API is C++, but bindings to other languages available
- Works as
  - Single application with DRM with OpenGL, or *fbdev* with no acceleration
  - Multiple applications on top of X11 or Wayland
- Multiplatform: Linux, MacOS, Windows.
- Somewhat complex licensing, with a mix of LGPLv3, GPLv2, GPLv3, and an (expensive) commercial license
- https://www.qt.io/

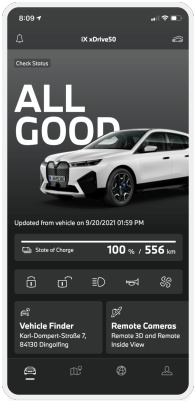
Gtk

- Toolkit used as the base for the GNOME desktop environment, the most popular desktop environment for Linux desktop distributions, but loosing traction in embedded projects.
- Composed of *glib* (core library), *pango* (text handling), *cairo* (vector graphics), *gtk* (widget library)
- Standard API in C, but bindings exist for many languages
- Requires a display server: X11 or Wayland
- License: LGPLv2
- Version 3.x the most deployed currently, 4.x is a new major release
- Multiplatform: Linux, MacOS, Windows.
- https://www.gtk.org



- Cross-platform UI application development: Linux, Android, iOS, Windows, MacOS
- Developed and maintained by Google
- Applications must be developed using the *Dart* programming language
- Applications can run in the Dart virtual machine, or be natively compiled for better performance.
- License: BSD-3-Clause
- https://flutter.dev

Read our blog post: https://bootlin.com/blog/flutter-nvidia-jetson-openembedded-yocto/



SDL

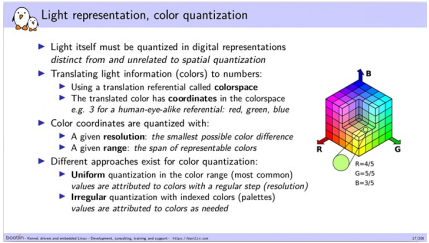
- *Cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware*
- Implemented in C, lightweight
- Does not provide a widget library
- Games, media players, custom UIs
- License: zlib license (simple permissive license)

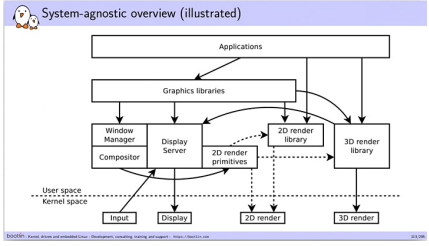
Other graphical toolkits

- Enlightenment Foundation Libraries (EFL) / Elementary
  - Lightweight and very powerful, but a lot less popular
  - Work on top of X or Wayland.
  - License: LGPLv2.1
  - https://www.enlightenment.org/about-efl.md
- LVGL
  - Very lightweight, mostly targeted at micro-controllers, but also runs on Linux
  - License: MIT
  - https://lvgl.io/
- See https://en.wikipedia.org/wiki/List_of_widget_toolkits

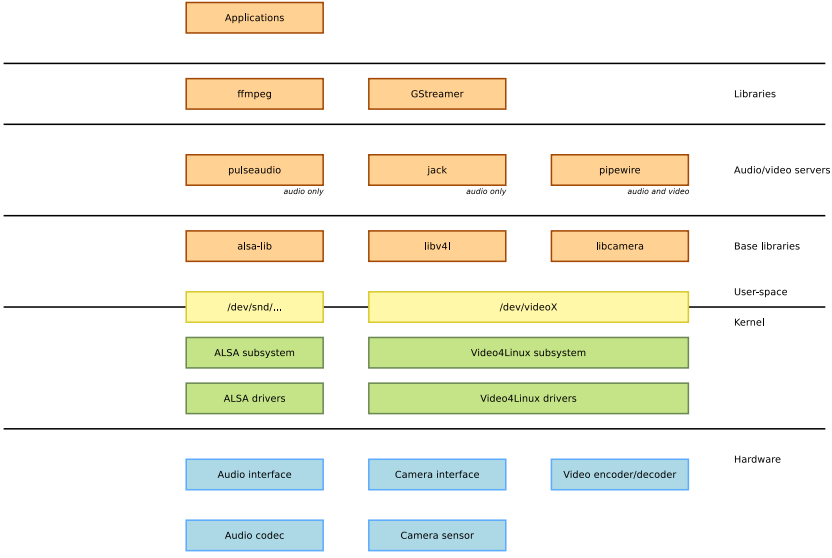
Further details on Linux graphics

- Bootlin *Understanding the Linux graphics stack*
- A complete course focused exclusively on this topic
- Freely available training materials
- https://bootlin.com/training/graphics





Linux multimedia stack overview



Audio stack

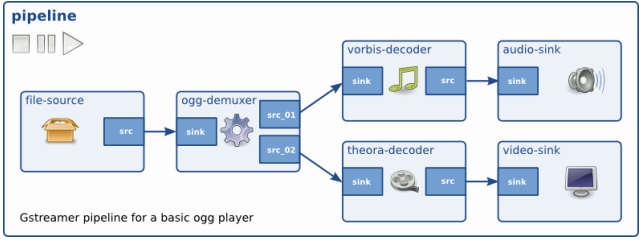
- Kernel-side: the ALSA subsystem, *Advanced Linux Sound Architecture*
  - Includes drivers for audio interfaces and audio codecs
  - Exposes audio devices in `/dev/snd/`
  - https://alsa-project.org
- Companion user-space library: *alsa-lib*
- Audio servers
  - Needed when multiple applications share audio devices: mix audio stream, route audio stream from specific applications to specific devices
  - *JACK*: mainly for professional audio
  - *pulseaudio*: mainly for regular desktop Linux audio
  - *pipewire*: modern replacement for both pulseaudio and JACK, already adopted by some Linux distributions
  - https://pipewire.org/

Video stack

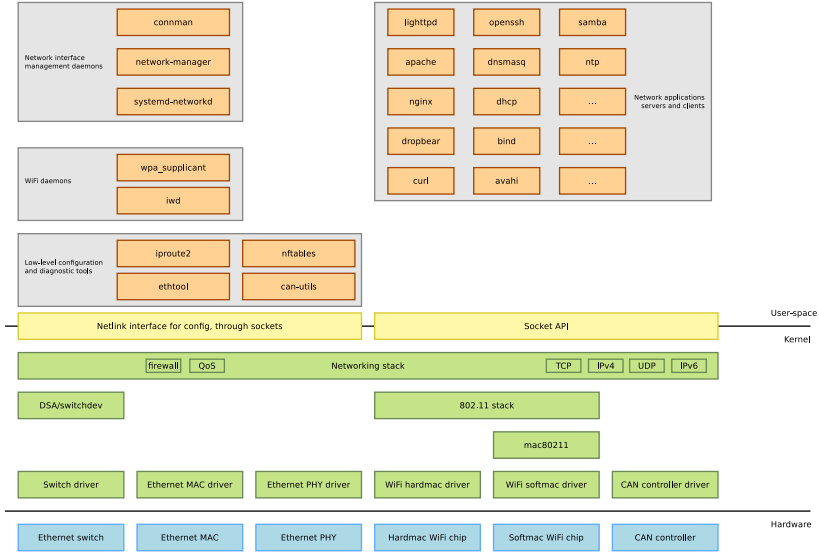
- Kernel-side: Video4Linux subsystem, or V4L in short
  - Supports camera devices: webcams as well as camera interfaces of SoCs and camera sensors (parallel, CSI, etc.)
  - Also used to support video encoding/decoding HW accelerators: H264, H265, etc.
  - Exposes video devices as `/dev/videoX`
  - https://www.linuxtv.org/
- Traditional user-space library: *libv4l*
- New user-space library, more modern, with many more features, under adoption: *libcamera*
- Supported in lots of multimedia stacks/software: GStreamer, ffmpeg, VLC, etc.

GStreamer

- *Library for constructing graphs of media-handling components*
- Allows to create *pipelines* to transform, convert, stream, display, capture multimedia streams, both audio and video
- Composed of a vast amounts of plugins: video capture/display, audio capture/playback, encoding/decoding, scaling, filtering, and more.
- https://gstreamer.freedesktop.org/
- An interesting alternative is *ffmpeg*



Linux networking stack



Web accessible UI

- Very common in embedded systems to use a Web interface for device configuration/monitoring
- Needs a web server: *Busybox httpd* for very simple needs, *lighttpd*, *nginx*, *apache* for more complex needs
- Can use PHP, NodeJS or other interpreted languages, or simple CGI shell scripts

Web browsers: rendering engines

To add HTML rendering capability to your device

- WebKit
  - Started by Apple, used in iOS, Safari
  - Open source project: LGPLv2.1 and BSD-2-Clause
  - https://webkit.org/
  - Integrated with Gtk: WebKitGTK
  - Integrated with Qt: QtWebKit
  - Port optimized for embedded devices: WPE WebKit
- Blink
  - Forked from WebKit
  - Developed by Google, used in Chrome
  - https://en.wikipedia.org/wiki/Blink_(browser_engine)
  - Integrated with Qt: QtWebEngine
  - Used by Electron

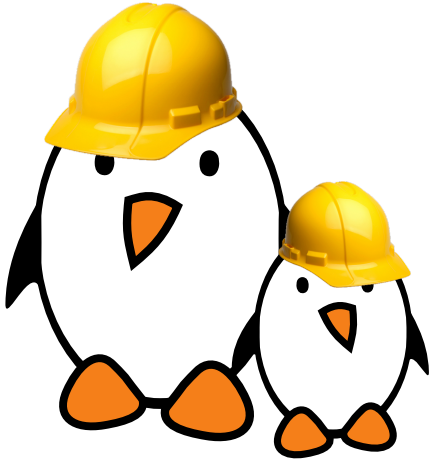
Web-based UIs

- An alternative to native GUI applications is to create a GUI based on Web technologies
- Run a Web browser full-screen, and use popular Web technologies to develop the application
- Some possible options
  - *Cog*, a simple launcher for the WPE Webkit port
  - *Electron*, a way to package a NodeJS application with a web rendering engine, into a self-contained application
- Beware of the footprint and performance impact: a web rendering engine is a massive and resource-consuming piece of software

Programming languages

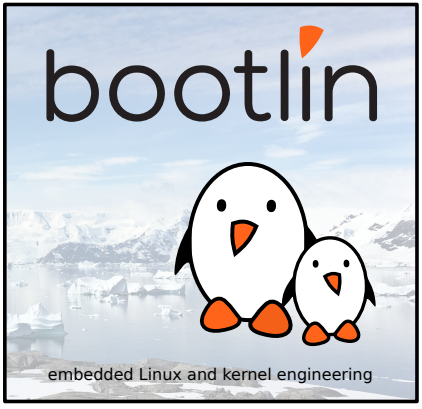
- Wide range of languages and frameworks available, not just C/C++
- Beware of footprint and performance implications
- Natively compiled languages
  - Rust
  - Go
  - Ada
  - Fortran
- Interpreted languages
  - Python
  - Javascript, NodeJS
  - Lua
  - Shell scripts
  - Perl, Ruby, PHP

Practical lab - Integration of additional software stacks



- Integration of *systemd* as an init system
- Use *udev* built in *systemd* for automatic module loading

Embedded Linux application development



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Contents

- Application development
  - Developing applications on embedded Linux
  - Building your applications
- Debugging and analysis tools
  - Debuggers
  - Remote debugging
  - Tracing and profiling

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.

Leverage existing libraries and languages

- Many developers getting started with embedded Linux limit themselves to C, sometimes C++, and the C/C++ standard library.
- However, there are a lot of libraries and languages that can help you accelerate and simplify your application development
  - Compiled languages like Rust and Go are increasingly popular
  - Interpreted languages, especially Python
  - Higher-level libraries: Qt, Glib, Boost, and many more
- Make sure to evaluate what is the right choice for your project, but pay attention to
  - Footprint and performance on low-end platforms
  - Use well-maintained and well-known technologies

Building your applications/libraries

- Even for simple applications or libraries, make use of a build system
  - CMake
  - Meson
- This will simplify
  - the build process of your application
  - the life of developers joining your project
  - the packaging of your application into an embedded Linux build system

Getting started with *meson*

Minimal `meson.build`

```
project('example', 'c')
executable('demo', 'main.c')
```

`meson.build` for multiple programs and source files

```
project('example', 'c')
src_demo1 = ['demo1.c', 'foo1.c']
executable('demo1', src_demo1)
src_demo2 = ['demo2.c', 'foo2.c']
executable('demo2', src_demo2)
```

Options with *meson*

meson_options.txt

```
option('demo-debug', type : 'feature', value : 'disabled')
```

meson.build

```
project('tutorial', 'c')
demo_c_args = []
if get_option('demo-debug').enabled()
   demo_c_args += '-DDEBUG'
endif
executable('demo', 'main.c', c_args: demo_c_args)
```

Library dependencies with *meson*

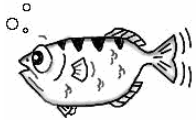
meson.build

```
project('tutorial', 'c')
gtkdep = dependency('gtk+-3.0')
executable('demo', 'main.c', dependencies : gtkdep)
```

The dependency `gtk+-3.0` is searched using `pkg-config`.

Debugging

GDB: GNU Project Debugger



- The debugger on GNU/Linux, available for most embedded architectures.
- Supported languages: C, C++, Pascal, Objective-C, Fortran, Ada...
- Command-line interface
- Integration in many graphical IDEs
- Can be used to
  - control the execution of a running program, set breakpoints or change internal variables
  - to see what a program was doing when it crashed: post mortem analysis
- https://www.gnu.org/software/gdb/
- https://en.wikipedia.org/wiki/Gdb
- New alternative: *lldb* (https://lldb.llvm.org/) from the LLVM project.

GDB crash course (1/3)

- GDB is used mainly to debug a process by starting it with *gdb*
  - `$ gdb <program>`
- GDB can also be attached to running processes using the program PID
  - `$ gdb -p <pid>`
- When using GDB to start a program, the program needs to be run with
  - `(gdb) run [prog_arg1 [prog_arg2] ...]`

GDB crash course (2/3)

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`, `print $reg` or `print task->files[0].fd` (`p`)
  Print the variable `var`, the register `$reg` or a more complicated reference. GDB can also nicely display structures with all their members
- `info registers`
  Display architecture registers

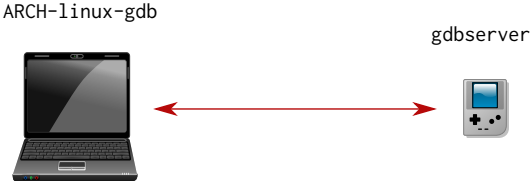
GDB crash course (3/3)

- `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
- `stepi` (`si`)
  Continue to the next instruction
- `finish`
  Execute up to function return
- `backtrace` (`bt`)
  Display the program stack

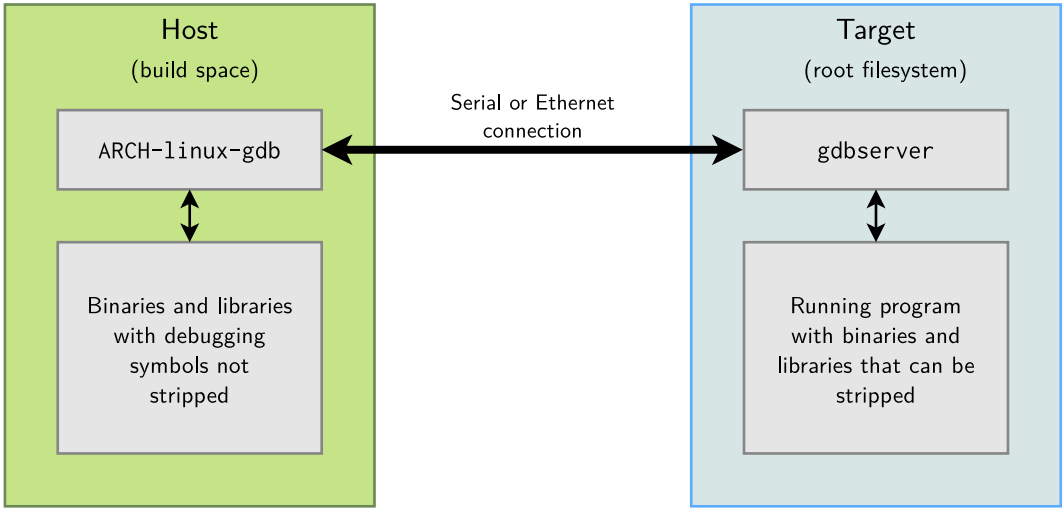
Remote debugging

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 400 KB on arm).



Remote debugging: architecture



Remote debugging: target setup

- On the target, run a program through `gdbserver`.
  Program execution will not start immediately.
  `gdbserver :<port> <executable> <args>`
  `gdbserver /dev/ttyS0 <executable> <args>`
- Otherwise, attach `gdbserver` to an already running program:
  `gdbserver --attach :<port> <pid>`
- You can also start gdbserver without passing any program to start or attach (and set the target program later, on client side):
  `gdbserver --multi :<port>`

Remote debugging: host setup

- Then, on the host, start `ARCH-linux-gdb <executable>`, and use the following `gdb` commands:
  - To tell `gdb` where shared libraries are:
    `gdb> set sysroot <library-path>` (typically path to build space without `lib/`)
  - To connect to the target:
    `gdb> target remote <ip-addr>:<port>` (networking)
    `gdb> target remote /dev/ttyUSB0` (serial link)
    - Make sure to replace `target remote` with `target extended-remote` if you have started gdbserver with the `--multi` option
  - If you did not set the program to debug on gdbserver commandline:
    `gdb> set remote exec-file <path_to_program_on_target>`

Coredumps for 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 in the ELF format. 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
  - The output name for the coredump file can be modified using `/proc/sys/kernel/core_pattern`.
  - See `man core(5)`
- 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`

minicoredumper

- Coredumps can be huge for complex applications
- minicoredumper is a userspace tool based on the standard core dump feature
  - Based on the possibility to redirect the core dump output to a user space program via a pipe
- Based on a JSON configuration file, it can:
  - save only the relevant sections (stack, heap, selected ELF sections)
  - compress the output file
  - save additional information from `/proc`
- https://github.com/diamon/minicoredumper
- "Efficient and Practical Capturing of Crash Data on Embedded Systems"
  - Presentation by minicoredumper author John Ogness
  - Video: https://www.youtube.com/watch?v=q2zmwrgLJGs
  - Slides: elinux.org/images/8/81/Eoss2023_ogness_minicoredumper.pdf

Tracing and 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 -f <command>` (**f**ollow child processes too)
  `strace -p <pid>` (tracing an existing process)
  `strace -c <command>` (time statistics per system call)
  `strace -e <expr> <command>` (use **e**xpression for advanced filtering)

See the strace manual for details.

Image credits: https://strace.io/

strace example output

```
> strace cat Makefile
[...]
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(0xb7f66000, 9840, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3) = 0
[...]
openat(AT_FDCWD, "Makefile", O_RDONLY)  = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=173, ...}, AT_EMPTY_PATH) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7290d28000
read(3, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 131072) = 173
write(1, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 173ifneq ($(KERNELRELEASE),)
```

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.
- 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.
- With the `-S` option it shows system calls too!
- Also offers a summary with its `-c` option.
- Manual page: https://linux.die.net/man/1/ltrace
- Works better with *glibc*. `ltrace` used to be broken with *uClibc* (now fixed), and is not supported with *Musl* (Buildroot 2022.11 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

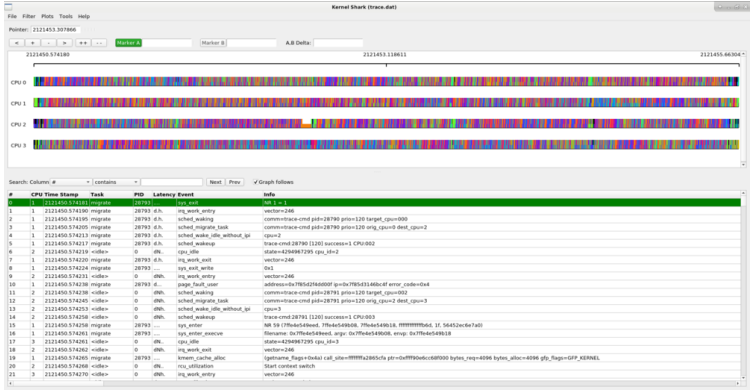
Example summary at the end of the ltrace output (`-c` option)

```
% 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
...
------ ----------- ----------- --------- --------------------
100.00   11.318773                  4762 total
```

ftrace

- In-kernel *tracing* functionality
- Can trace
  - Well-defined trace locations in the kernel, called *tracepoints*, identifying important events in the kernel: scheduling, interrupts, etc.
  - Arbitrary functions in the kernel
  - Arbitrary functions in user-space applications
- Low-overhead and optimized tracing
- Accessible using the dedicated *tracefs* filesystem
- `trace-cmd` is a higher-level CLI tool to use *ftrace*
- Can be used to understand overall system activity (what is my system doing?) as well as narrow down specific performance issues
- https://www.kernel.org/doc/Documentation/trace/ftrace.txt
- https://www.trace-cmd.org/

- Visualization tool for *ftrace* traces
- https://kernelshark.org/



perf

- *instrument CPU performance counters, tracepoints, kprobes, and uprobes*
- Directly included in the Linux kernel source code: `tools/perf`
- Began as a tool for using the performance counters in Linux, and has had various enhancements to add tracing capabilities
- Supports a list of measurable events: hardware events (cycle count, L1 cache hits/miss, page faults), software events (tracepoints)
- https://perf.wiki.kernel.org

perf examples

- List all currently known events
  `perf list`
- List scheduler tracepoints
  `perf list 'sched:*'`
- CPU counter statistics for the specified command
  `perf stat <command>`
- CPU counter statistics for the entire system, for 5 seconds
  `perf stat -a sleep 5`
- Profiling: sample on-CPU functions for the specified command, at 99 Hertz
  `perf record -F 99 <command>`
- Tracing: trace all context-switches via sched tracepoint, until Ctrl-C
  `perf record -e sched:sched_switch -a`
- Many more at https://www.brendangregg.com/perf.html

perf GUI: hotspot

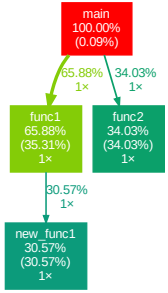
- Hotspot - the Linux perf GUI for performance analysis
- The main feature of hotspot is visualizing a `perf.data` file graphically
- github.com/KDAB/hotspot

gprof

- Application-level profiler
- Part of *binutils*
- Requires passing gcc `-pg` option at build/link time
- Run your program normally, it automatically generates a `gmon.out` file when exiting
- Use the `gprof` tool on `gmon.out` to extract profiling data
- http://sourceware.org/binutils/docs/gprof/

gprof example

```
$ ./test-gprof
$ gprof test-gprof gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 35.31      7.46     7.46        1     7.46    13.92  func1
 34.03     14.65     7.19        1     7.19     7.19  func2
 30.57     21.11     6.46        1     6.46     6.46  new_func1
  0.09     21.13     0.02                             main
[...]
```



Generated with gprof2dot

Memory debugging

Valgrind

https://valgrind.org/

- *instrumentation framework for building dynamic analysis tools*
  - detect many memory management and threading bugs
  - profile programs
- Supported architectures: x86, x86-64, ARMv7, ARMv8, mips32, s390, ppc32 and ppc64
- Very popular tool especially for debugging memory issues
- Runs your program on a synthetic CPU → significant performance impact (100 x slower on SAMA5D3!), but very detailed instrumentation
- Runs on the target. Easy to build with Yocto Project or Buildroot.

Valgrind tools

- *Memcheck*: detects memory-management problems
- *Cachegrind*: cache profiler, detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code
- *Callgrind*: extension to Cachegrind, provides extra information about call graphs
- *Massif*: performs detailed heap profiling by taking regular snapshots of a program's heap
- *Helgrind*: thread debugger which finds data races in multithreaded programs. Looks for memory locations accessed by multiple threads without locking.
- More at https://valgrind.org/info/tools.html

Valgrind examples

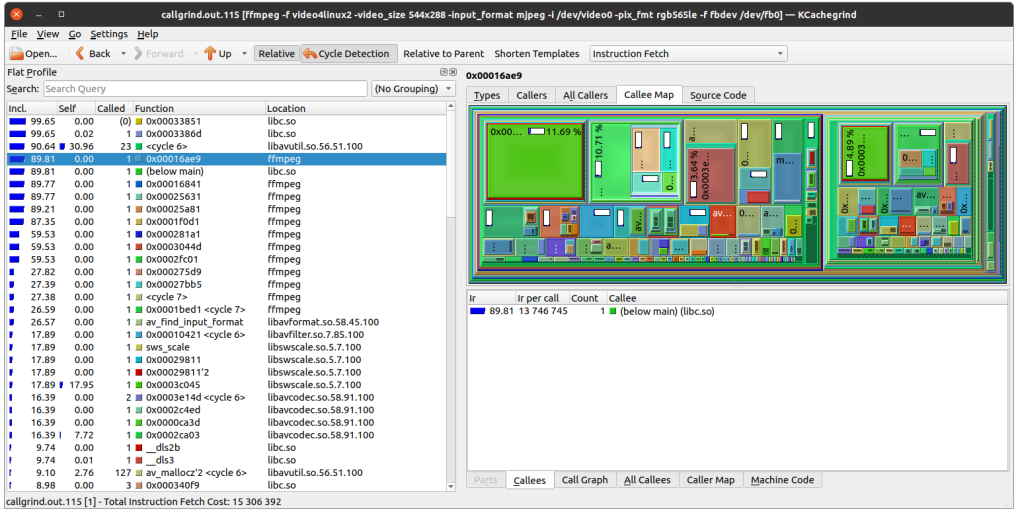
- *Memcheck*

```
$ valgrind --leak-check=yes <program>
  ==19182== Invalid write of size 4
  ==19182==    at 0x804838F: f (example.c:6)
  ==19182==    by 0x80483AB: main (example.c:11)
  ==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
  ==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
  ==19182==    by 0x8048385: f (example.c:5)
  ==19182==    by 0x80483AB: main (example.c:11)
```

- *Callgrind*

```
$ valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes <program>
$ ls callgrind.out.*
callgrind.out.1234
$ callgrind_annotate callgrind.out.1234
```

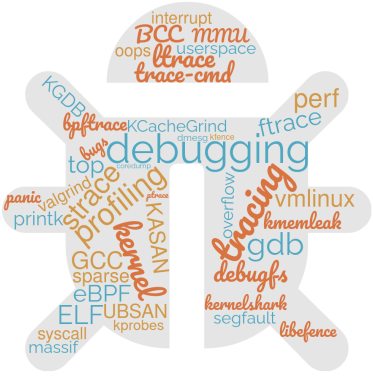
Kcachegrind - Visualizing Valgrind profiling data

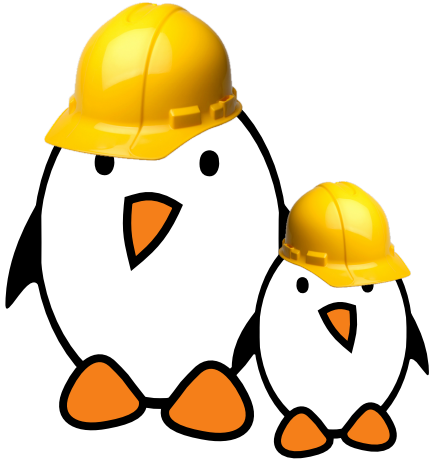


https://github.com/KDE/kcachegrind

Debugging resources

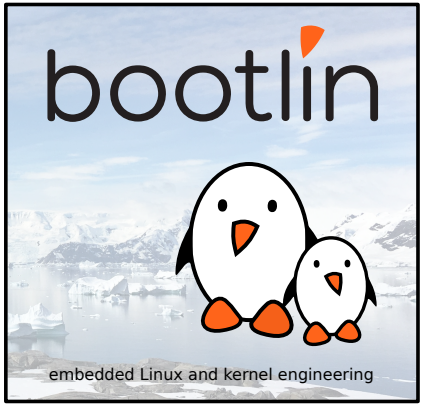
- Brendan Gregg Systems performance book
- Brendan Gregg Linux Performance page
- Bootlin's "Linux debugging, profiling, tracing and performance analysis" training course and free training materials (250 pages): https://bootlin.com/training/debugging/.





- Creating an application that uses an I2C-connected joystick to control an audio player.
- Setting up an IDE to develop and remotely debug an application.
- Using *strace*, *ltrace*, *gdbserver* and *perf* to debug/investigate buggy applications on the embedded board.

Useful resources

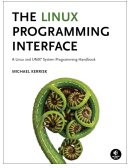


© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Books

- **Mastering Embedded Linux Programming, 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



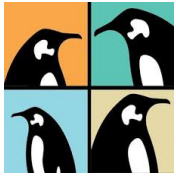
[1] https://www.packtpub.com/product/mastering-embedded-linux-programming-third-edition/9781789530384
[2] https://man7.org/tlpi/

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.

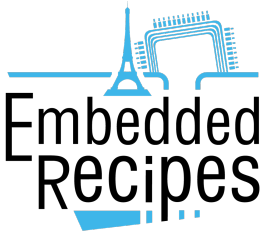
International conferences (1)

- Embedded Linux Conference:
  - https://embeddedlinuxconference.com/
  - Organized by the Linux Foundation
  - Once per year, alternating North America/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.

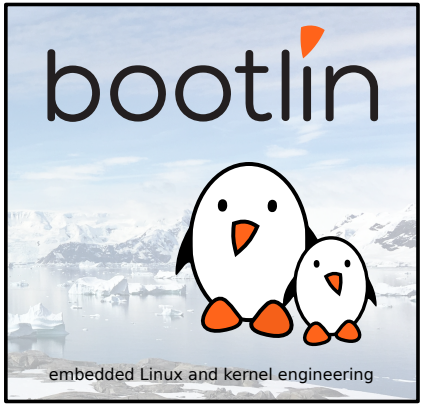


International conferences (2)

- FOSDEM: https://fosdem.org
  - Brussels (Belgium), February
  - Community-oriented conference, free, during the week-end
  - Many *developer rooms*, including on low-level, embedded and hardware topics
- Embedded Recipes: https://embedded-recipes.org
  - Paris (France), September
  - 2-day conference about all embedded Linux topics
  - Well attended by known contributors
  - Very affordable conference, thanks to sponsors (like Bootlin).
- Most conferences are now also accessible on-line, which makes them much more affordable.



Last slides



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Thank you!

And may the Source be with you

Rights to copy

© Copyright 2004-2024, Bootlin
License: Creative Commons Attribution - Share Alike 3.0
https://creativecommons.org/licenses/by-sa/3.0/legalcode
You are free:

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

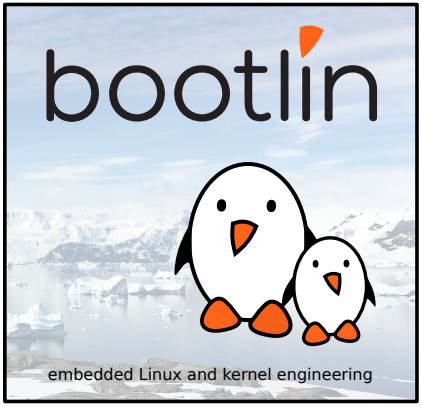
Under the following conditions:

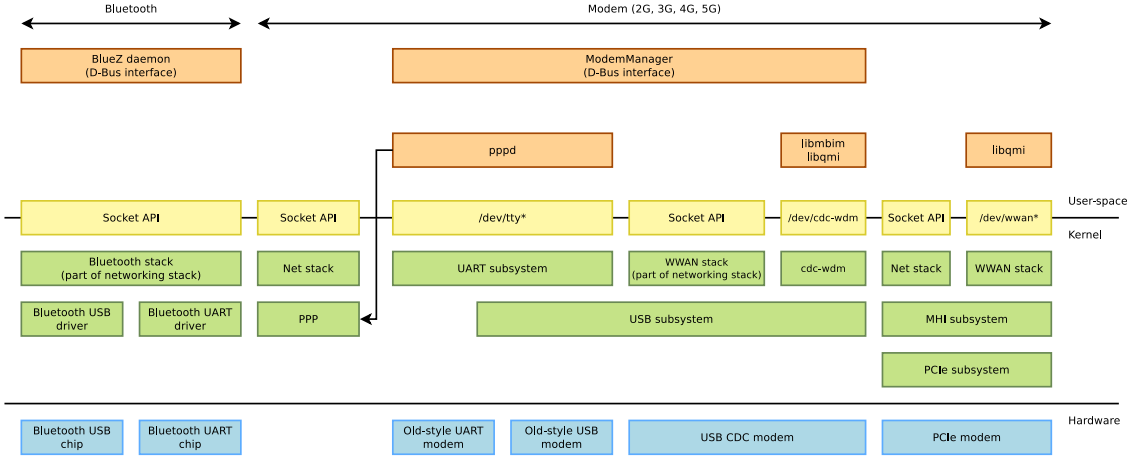
- **Attribution**. You must give the original author credit.
- **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this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

Your fair use and other rights are in no way affected by the above.

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

Extra slides



© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Linux connectivity stack