



Nasser Afshin

Afshin.Nasser@Gmail.com

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: November 07, 2025.

Training details:

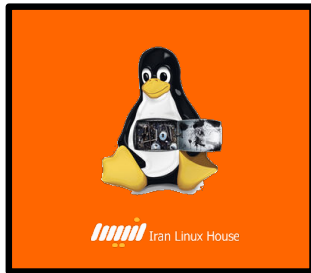
<https://anisa.co.ir/courses/embedded/linux-system-integration.html>

Document updates:

<https://t.me/+zPRUOuFxmSjZmY0>

Corrections, suggestions, contributions and translations are welcome!

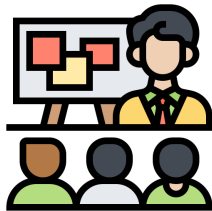
Send them to afshin.nasser@gmail.com





Advanced Embedded Linux system integration and development training

- ▶ These slides are the training materials for Iran Linux House's *Advanced Embedded Linux system integration and development* training course.
- ▶ If you are interested in following this course with me, 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.
- ▶ Details and registrations:
<https://anisa.co.ir/courses/embedded/linux-system-integration.html>
- ▶ Contact: afshin.nasser@gmail.com



Icon by Eucalyp, Flaticon



My Training Courses

Embedded I:
Embedded Linux
from Scratch
Rpi/QEMU

Online: 7 * 5 hours

Embedded II:
Advanced
Embedded Linux
System Integration
and Development

Online: 7 * 5 hours

Embedded III:
Buildroot
System
Development

Online: 7 * 5 hours

Yocto Project
System
Development

Online: 7 * 5 hours

Embedded
Linux Kernel
Device Driver
Development

Online: 9 * 5 hours

x86
Linux Kernel
Device Driver
Development

Online: 9 * 5 hours

Understanding
the Linux
Graphics Stack

Online: 7 * 5 hours

Embedded Linux
Boot Time
Optimization

Online: 7 * 5 hours

Real-Time Linux
with
PREEMPT_RT

Online: 4 * 5 hours

Linux Debugging,
Tracing, Profiling
and Performance
Analysis

Online: 7 * 5 hours



My Training Workshops

- ▶ W1001: How to Write/Use Libraries in C/C++ (6 hours)
- ▶ W2001: Kernel-space USB Device Drivers, It's not That Hard (12 hours)
- ▶ W2002: User-space USB Device Drivers, How Do They Work? (6 hours)
- ▶ W2003: Writing a USB Gadget Driver in its Easiest Form (3 hours)
- ▶ W2011: Implementing I2C Client Device Drivers (6 hours)
- ▶ W2021: Implementing SPI Client Device Drivers (6 hours)
- ▶ W2031: How to Avoid Writing Kernel Space Device Drivers? (12 hours)
- ▶ W2032: UIO: A Quick and Easy Way to Write Device Drivers for (Embedded) Linux (12 hours)
- ▶ W3001: Build Your Own (Embedded) Linux Distribution Using Yocto! (6 hours)
- ▶ W3002: Embedded Linux Distribution From Scratch! (3 hours)
- ▶ W3003: An Introduction to Buildroot on QEMU (3 hours)



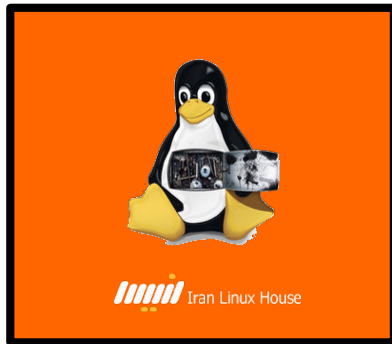
Generic course information

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





What about our test bench?

We use Qemu, the FAST! processor emulator

- ▶ A generic and opensource machine emulator and virtualizer
- ▶ Can be used for:
 - Full-system emulation
 - User mode emulation
 - Virtualization





Labs on QEMU virtual hardware

Practical labs on the QEMU emulated ARM
Versatile Express Cortex A9 or ARM virt board

- ▶ Most features as on a real embedded board
- ▶ Can boot the Linux kernel from U-Boot
- ▶ Supports mainline Linux and U-Boot
- ▶ Networking possible with the emulated board
- ▶ Emulated SD card



Lab instructions available on

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



Labs proposed on another platform

During this course, you can also run all labs on the Raspberry Pi 4 board.

- ▶ Quad-core Cortex-A72 (64-bit SoC) ARM CPU from Broadcom
- ▶ 2GB LPDDR4 RAM
- ▶ 1 Ethernet ports (1000 Mbit) + 802.11ac Wi-Fi + Bluetooth 5
- ▶ 2 USB 2.0 and 2 USB 3.0 ports
- ▶ 1 MMC/SD slot
- ▶ 6 serial ports + I2C, SPI ports
- ▶ 2 MIPI DSI display and 2 MIPI CSI camera ports
- ▶ Stereo audio and composite video ports
- ▶ Dual monitor support (micro HDMI) up to 4Kp60
- ▶ 5V DC via USB-C or GPIO header
- ▶ Misc: JTAG, GPIOs





Participate!

During the lectures...

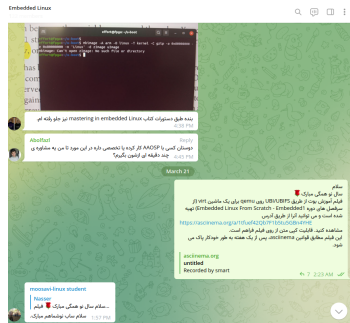
- ▶ Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- ▶ Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- ▶ Your point of view is most valuable, because it can be similar to your colleagues' 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 Telegram channel 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 Telegram channel is also a good place to ask questions outside of training hours, and after the course is over.
- ▶ Our Telegram channel: <https://t.me/elugiran>





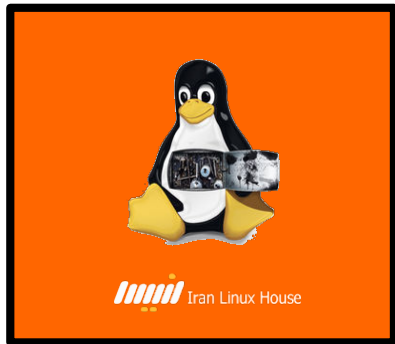
Accessing hardware devices

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





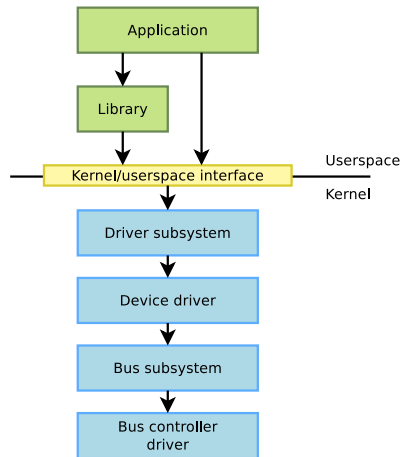
Kernel drivers



Typical software stack for hardware access

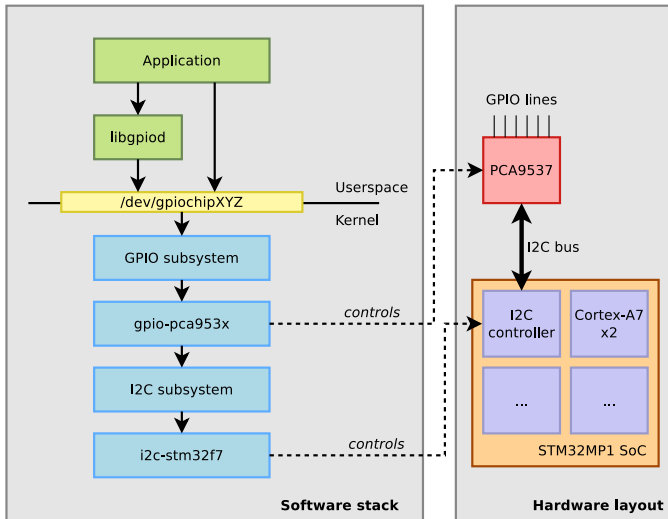
From the bottom to the top:

- ▶ 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 auxiliary processors
- ▶ crypto for cryptographic accelerators
- ▶ hwmon for hardware monitoring sensors
- ▶ block layer for block storage

and many more



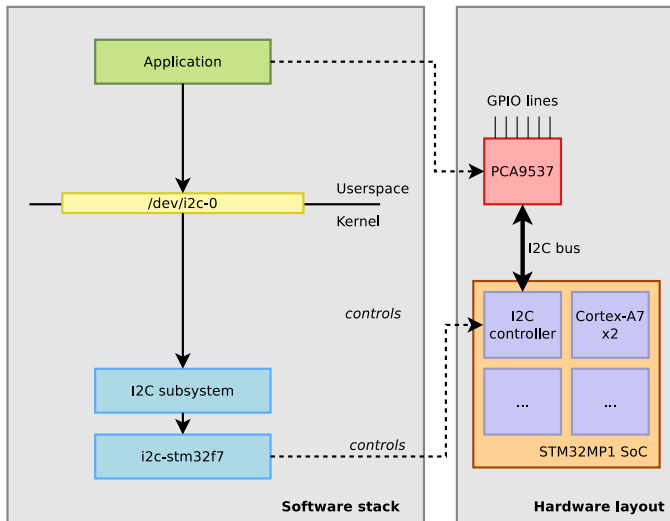
Accessing devices directly from user-space

- ▶ 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

```
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



sysfs filesystem example

- ▶ `/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/bus/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/gpio`
 - 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.rst](#)



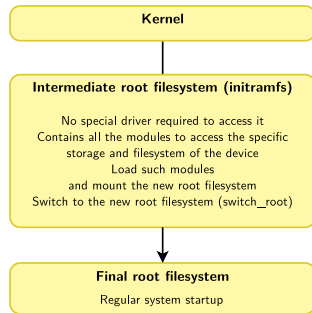
Using kernel modules



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

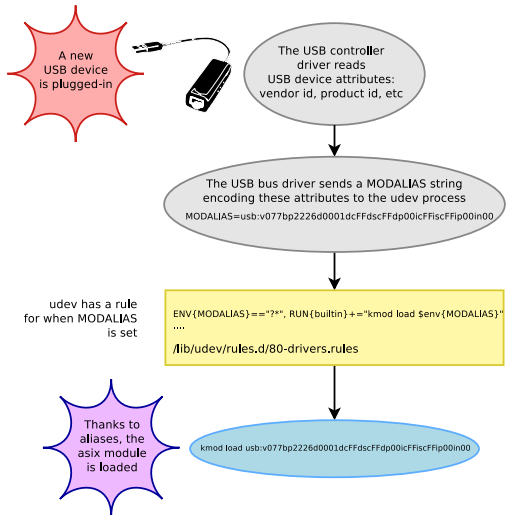
Module file
asix.ko

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





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



Describing non-discoverable hardware

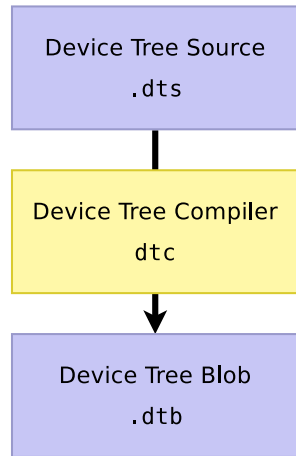
3. Using a **Device Tree**

- ▶ 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` $\approx$ few dozens of kilobytes
- ▶ DTB also called **FDT**, *Flattened Device Tree*, once loaded into memory.
 - `fdt` command in U-Boot
 - `fdt_` APIs





dts example

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

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



dts 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/;

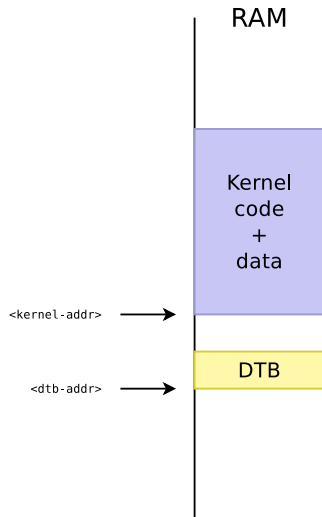
/ {
    welcome = <0xbadcafe>;

    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)
 - $\approx$ 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 $\approx$ a device or IP block
- ▶ Properties $\approx$ device characteristics
- ▶ Notion of **cells** in property values
- ▶ Notion of **phandle** to point to other nodes
- ▶ dtc only does syntax checking, no semantic validation

```
/ {  
    node@0 {  
        a-string-property = "A string";  
        a-string-list-property = "first string", "second string";  
        a-byte-data-property = [0x01 0x23 0x34 0x56];  
  
        child-node@0 {  
            first-child-property;  
            second-child-property = <1>;  
            a-reference-to-something = <&node1>;  
        };  
  
        child-node@1 {  
        };  
    };  
  
    node1: node@1 {  
        an-empty-property;  
        a-cell-property = <1 2 3 4>;  
  
        child-node@0 {  
        };  
    };  
};
```

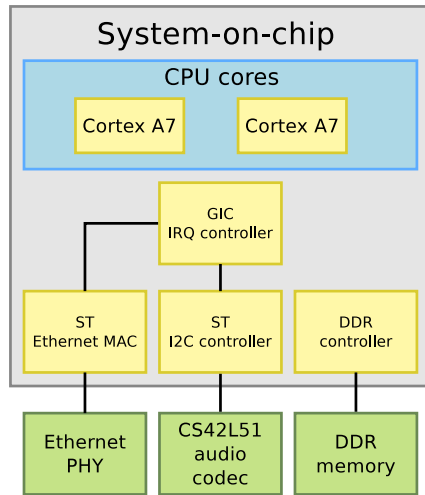
Diagram annotations:

- Node name: `node@0`
- Unit address: `0`
- Property name: `a-string-property`
- Property value: `"A string"`
- Properties of node@0: `a-string-property`, `a-string-list-property`, `a-byte-data-property`
- Bytestring: `[0x01 0x23 0x34 0x56]`
- A handle (reference to another node): `&node1`
- Label: `node1:`
- Four cells (32 bits values): `<1 2 3 4>`



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 { ... };  
    };  
};
```



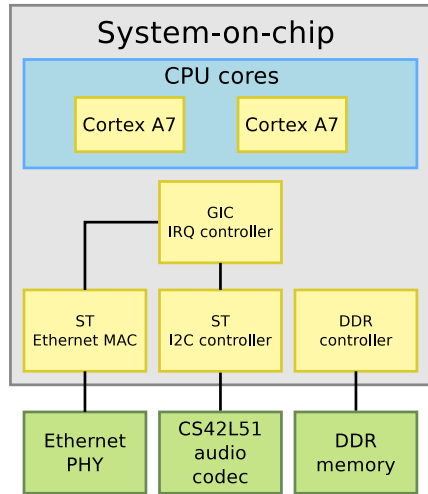


DT overall structure: simplified example

```
/ {
  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>;
    };
  };

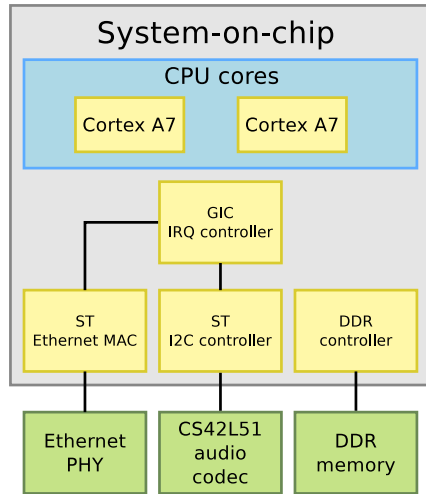
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```





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 { ... };  
  };  
};
```





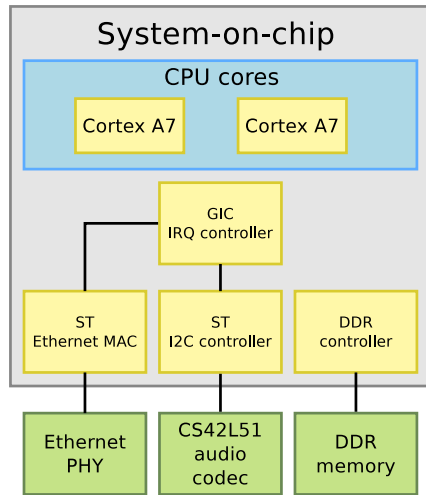
DT overall structure: simplified example

```
/ {
  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 { ... };
  };
};
```

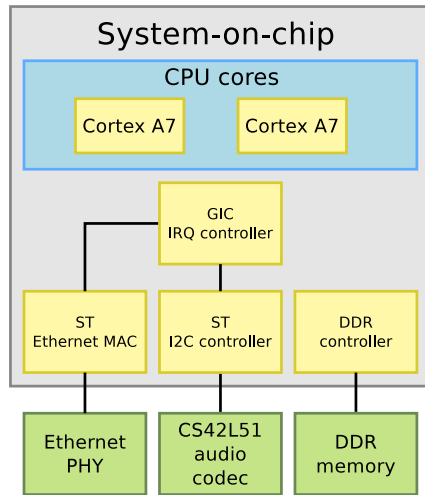




DT overall structure: simplified example

```
/ {
  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 = <&gpio9 9 GPIO_ACTIVE_LOW>;
        status = "okay";
      };
    };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



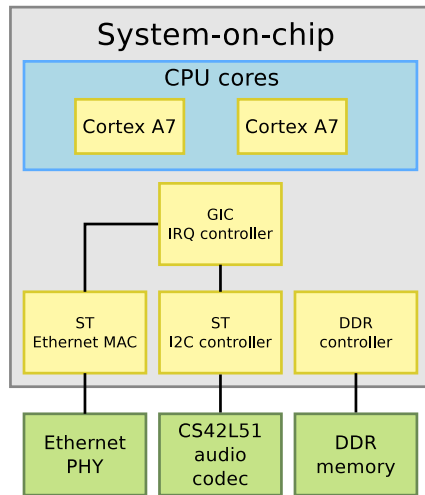


DT overall structure: simplified example

```
/ {
    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



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 1

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

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

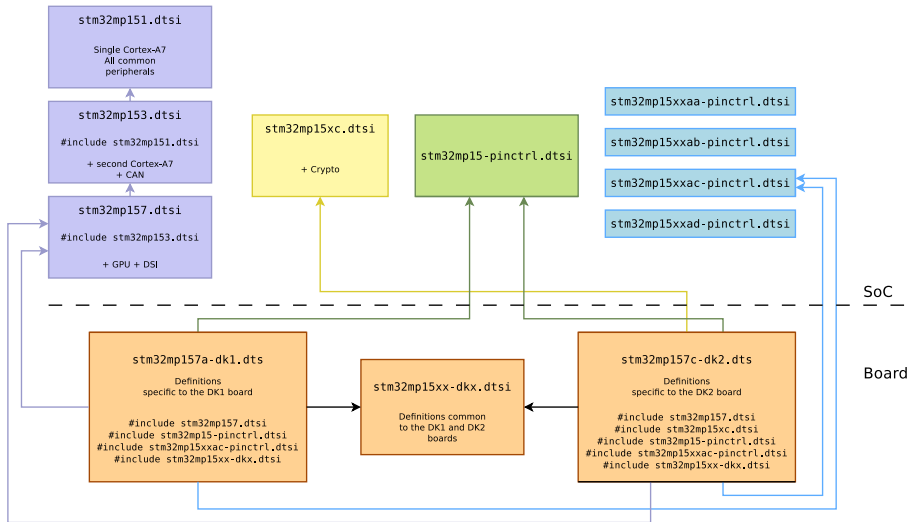
```
#include "soc.dtsi"

&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 ?
- ▶ **Device Tree 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](https://www.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 {  
        compatible = "...";  
    };  
    soc {  
        compatible = "simple-bus";  
        uart@1000 {  
            compatible = "...";  
        };  
        i2c@2000 {  
            compatible = "...";  
            eeprom@65 {  
                compatible = "...";  
            };  
        };  
    };  
};
```

Platform device

Platform device

Platform device

I2C device



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>;
};

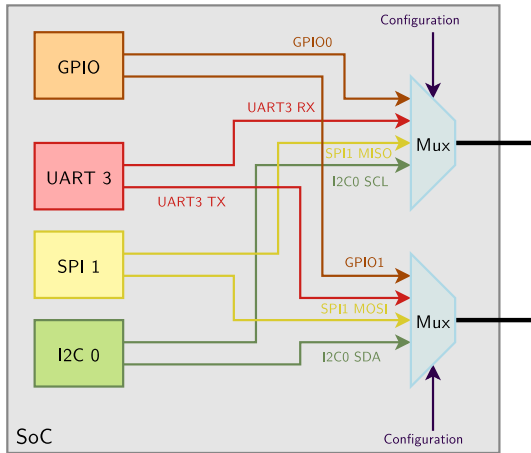
dmamux1: dma-router@48002000 {
    compatible = "st,stm32h7-dmamux";
    reg = <0x48002000 0x1c>;
    #dma-cells = <3>;
    clocks = <&rcc DMAMUX>;
    resets = <&rcc DMAMUX_R>;
};

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



DS12505 Rev 7

97/262

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/I2/ 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	-

STM32MP157C/F

Pinouts, pin description and alternate functions

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

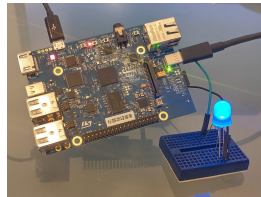
```
#include "stm32mp157a-dk1.dts"

/ {
    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

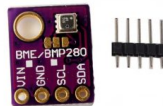
```
&i2c5 {
    status = "okay";
    clock-frequency = <100000>;
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;

    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/a9CZ1Uk30YQ>

bootlin

Agenda

- ▶ Bootlin introduction
- ▶ STM32MP1 introduction
- ▶ Why the Device Tree ?
- ▶ Basic Device Tree syntax
- ▶ Device Tree inheritance
- ▶ Device Tree specifications and bindings
- ▶ Device Tree and Linux kernel drivers
- ▶ Common properties and examples

1:22 / 1:54:58



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.



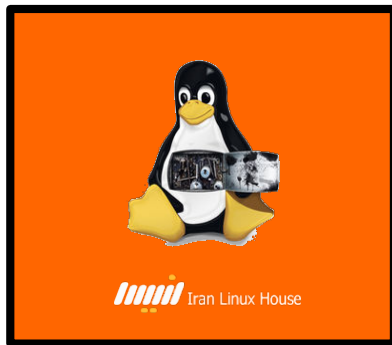
Cross-compiling user-space libraries and applications

Based on slides by **Bootlin**

© Copyright 2004-2026, 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



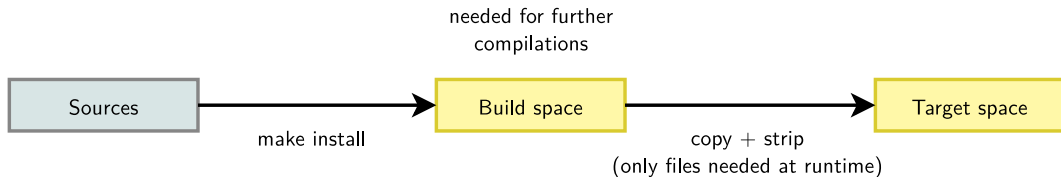
Concept of build system

- ▶ 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-gnueabi/bin:$PATH
$ make CC=arm-none-linux-gnueabi-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-gnueabi-gcc NO_ENCRYPTION=1
arm-none-linux-gnueabi-gcc -g -Wall -Wextra [...] -c server_announce.c
[...]
arm-none-linux-gnueabi-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-gnueabi-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-gnueabi-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-gnueabi-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-gnueabi-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-gnueabi-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.

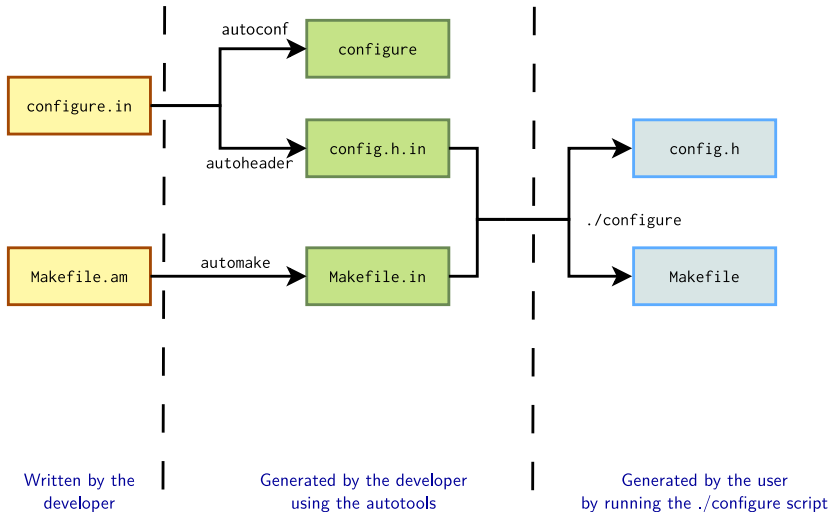


Autotools

- ▶ 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





- ▶ 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 -l configure* *makefile*
configure.ac
GNUmakefile.am
```

No configure and GNUmakefile.in,
autoreconf needed.

Autoreconf

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

Configuration

```
$ ./configure --prefix=/usr
$ ls -l *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-gnueabi/bin:$PATH
$ ./configure --prefix=/usr --host=arm-none-linux-gnueabi
$ 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-gnueabi-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>



[https://en.wikipedia.org/wiki/Meson_\(software\)](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-gnueabi-hf-gcc'
strip = 'arm-none-linux-gnueabi-hf-strip'

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



Example: *ipcalc* cross-compilation

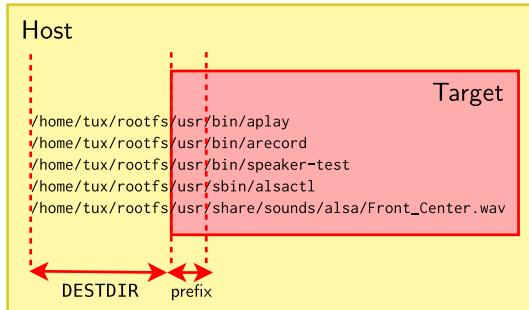
```
$ cat cross-file.txt
[binaries]
c = 'arm-none-linux-gnueabi-gcc'
strip = 'arm-none-linux-gnueabi-strip'

[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 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
```



Practical lab - Cross-compiling applications and libraries



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.



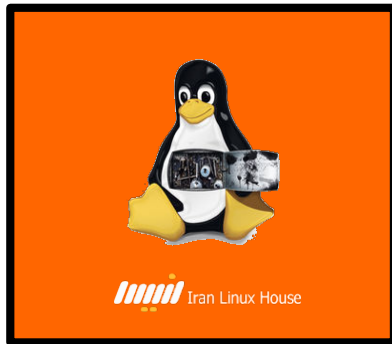
Embedded system building tools

Based on slides by **Bootlin**

© Copyright 2004-2026, 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 a binary distribution such as Debian, Ubuntu or Fedora
3. Use an *embedded Linux build system* that automates the cross-compilation process



Approaches pros and cons

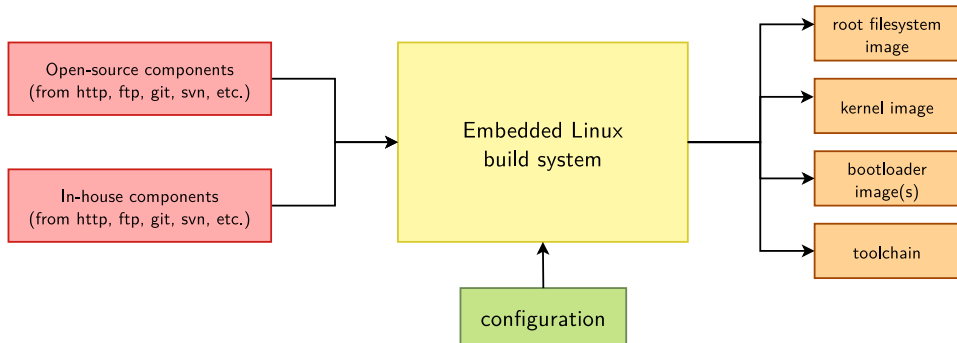
	Pros	Cons
Building everything manually	Full flexibility Learning experience	Dependency hell Need to understand a lot of details Version compatibility Lack of reproducibility
Binary distribution Debian, Ubuntu, Fedora, etc.	Easy to create and extend Extensive set of packages Usually excellent security 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
Embedded Linux 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

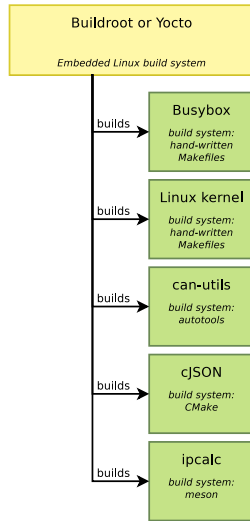


- ▶ Building from source → lot of flexibility
- ▶ Cross-compilation → leveraging fast build machines
- ▶ Recipes for building components → easy



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

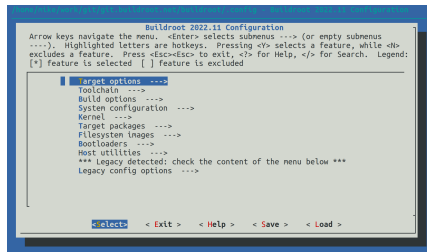


- ▶ Allows to build a toolchain, a root filesystem image with many applications and libraries, a bootloader and a kernel image
 - Or any combination of the previous items
- ▶ Supports using uClibc, glibc and musl toolchains, either built by Buildroot, or external
- ▶ Over 2800 applications or libraries 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

The image shows the Buildroot website. At the top, there's a navigation bar with links: Home, Documentation, Support, Contacts, Services, Associates, and a Download button. Below this is a large blue banner with a yellow hard hat icon. The text on the banner says "Buildroot Making Embedded Linux Easy" with "LEARN MORE" and "DOWNLOAD" buttons. Below the banner, a paragraph states: "Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation." There are three circular icons with text below them: 1. A penguin icon with the text "Can handle everything" and a description of cross-compilation. 2. A hammer icon with the text "Is very easy" and a description of how easy it is to use. 3. A gift icon with the text "Supports several thousand packages" and a description of the package support.



Embedded Linux development with Buildroot training
On-line version, 2 versions of 4 lectures
Last updated: August 30, 2023

Title	Embedded Linux development with Buildroot training
Training objectives	• Be able to understand the role and principle of an embedded Linux build system and compare Buildroot to other tools offering similar functionality. • Be able to create a simple embedded Linux system with Buildroot: create a configuration, set the build, install the result on an embedded platform. • Be able to adapt the Buildroot configuration to build an embedded Linux system: architecture-specific, multi-chose of the cross-compilation toolchain, some options of the Linux kernel configuration, customization of the user libraries, etc. • Be able to create new packages in Buildroot to integrate additional applications and libraries into the embedded Linux system. • Be able to adapt the configuration of Buildroot to manage and optimize the build: security vulnerability tracking, feature compliance, etc. • Be able to develop and build Linux user space applications in the context of Buildroot. • Be able to interact with the Buildroot open-source community, and to understand the methods of Buildroot.
Duration	Five full days - 20 hours or four per half day
Prerequisites	• Lectures delivered by the trainer, over video-conference. Participants can ask questions at any time. • Practical demonstrations done by the trainer, based on practical labs, over video-conference. Participants can ask questions at any time. Optionally, participants who have access to the hardware accessories can reproduce the practical labs for Buildroot. • Issues managing, for questions between sessions (English only 24h, outside of work time, and French business hours). • Electronic copies of presentations, lab instructions and data files. They are freely available at https://www.bootlin.com/training/buildroot/.
Trainer	One of the engineers listed on: https://www.bootlin.com/training/buildroot/
Language	First language: English Second language: English

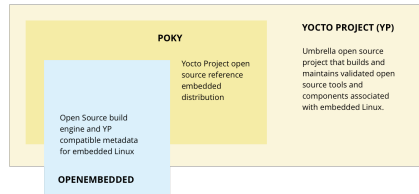


► OpenEmbedded

- 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

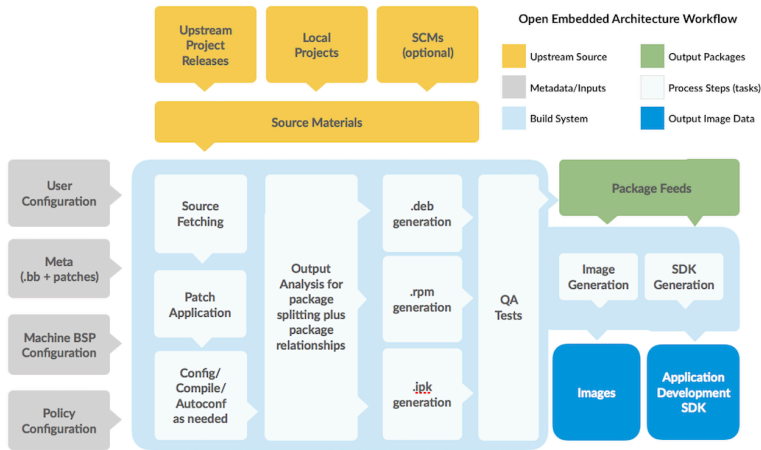
► Yocto Project

- 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
$ git -C openembedded-core checkout e67d659847af
$ git clone https://git.openembedded.org/meta-openembedded
$ git -C meta-openembedded checkout 4052c97dc83d
$ git clone https://git.openembedded.org/bitbake -b 2.0
$ git clone https://github.com/STMicroelectronics/meta-st-stm32mp.git \
    -b openstlinux-5.15-yocto-kirkstone-mp1-v23.07.26
```

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-glibc/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

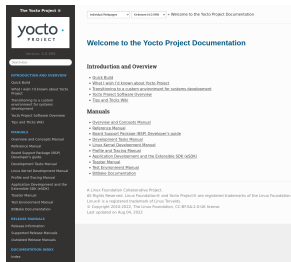
BBCLASSEXTEND = "native"
```

- ▶ 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



Yocto Project and OpenEmbedded development training	
Training objectives	• Be able to understand the role and principle of an embedded Linux build system, and compare Yocto Project/OpenEmbedded to other tools offering similar functionality. • Be able to configure and build basic embedded Linux system with Yocto, and install the result on an embedded platform. • Be able to write and extend recipes, for your own packages or customizations. • Be able to use existing layers of recipes, and create your own new layers. • Be able to integrate recipes for your own embedded board into a BSP layer. • Be able to create custom images. • Be able to use the tools and workflows available to develop applications with the Yocto Project SDK.
Duration	Four half days - 16 hours (4 hours per half day)
Prerequisites	• Linux familiarity for the entire, non collaborative, Participants can ask questions at any time. • Practical demonstrations done by the trainer, based on practical lab, over video conference. Participants can ask questions at any time. Optionally, participants who have access to the hardware accessories can spend the practical lab by themselves. • Content covering the questions between sessions (up to 24h, outside of work-time and bank holidays). • Electronic copies of presentations, lab instructions and data files. They are freely available at https://www.bootlin.com/training/yocto/.
Trainer	One of the engineers based on https://bootlin.com/training/yocto/
Language	Can be done: English, French, Mandarin, English
Audience	Engineers and engineers interested in using the Yocto Project to build their embedded Linux system.



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



- ▶ 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 [Opsys](#), 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 MPU6050 device driver.



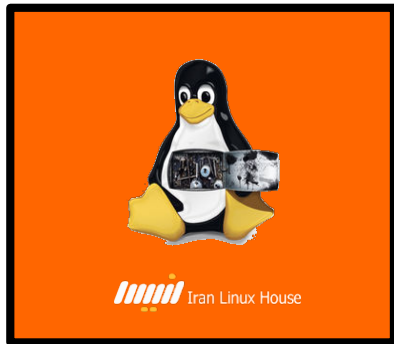
Open source licenses and compliance

Based on slides by **Bootlin**

© Copyright 2004-2026, 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.

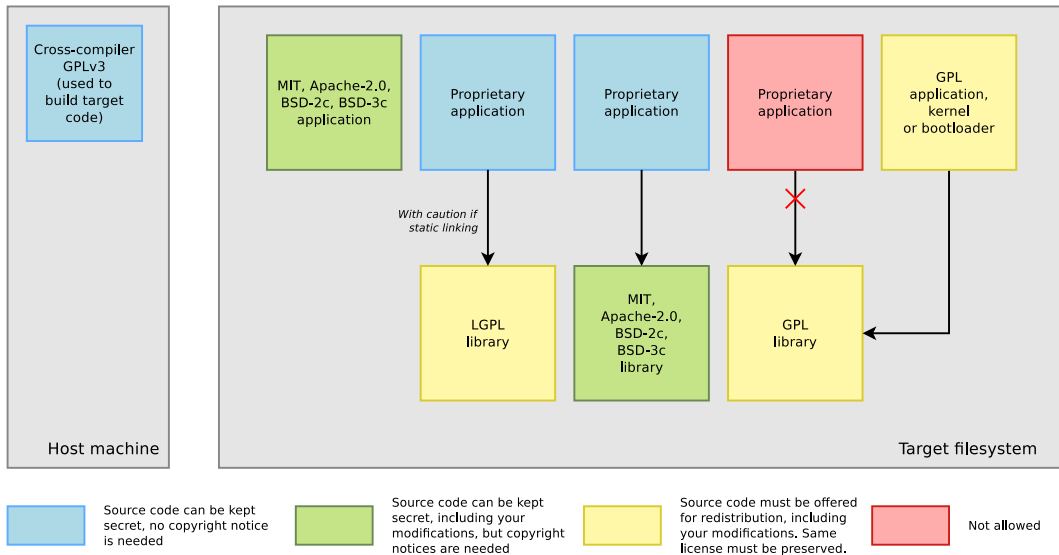


Is this free software?

- ▶ 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.



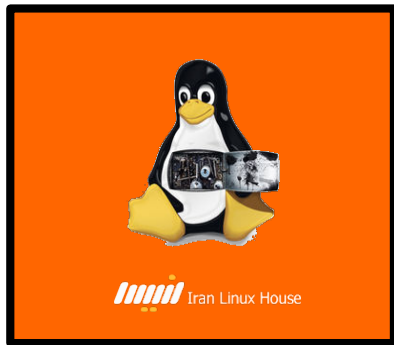
Overview of major embedded Linux software stacks

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.

Creative Commons BY-SA 3.0 license.

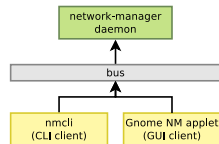
Corrections, suggestions, contributions and translations are welcome!





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
 - *udev*, 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/ssh
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

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

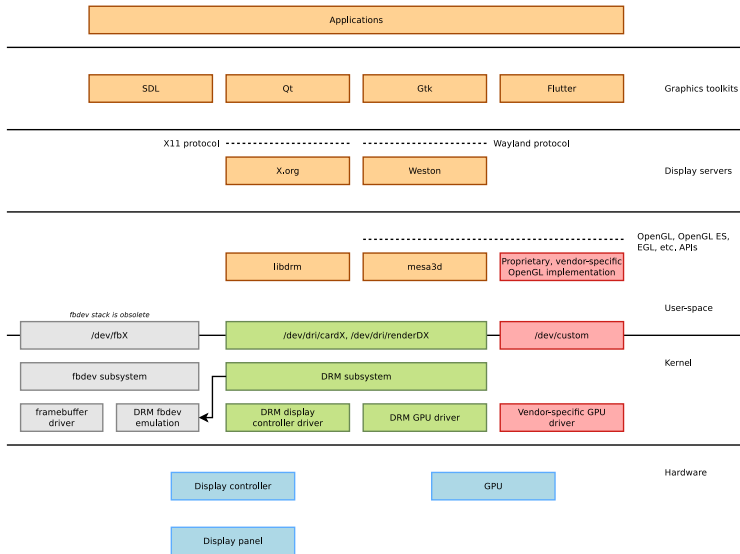


Example systemctl/journalctl commands

- ▶ `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/>

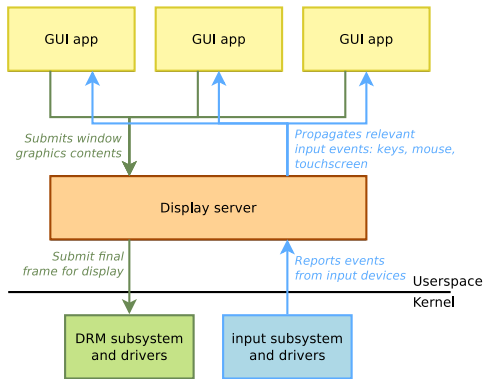
▶ Proprietary

- 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>





- ▶ *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_\(display_server_protocol\)](https://en.wikipedia.org/wiki/Wayland_(display_server_protocol))





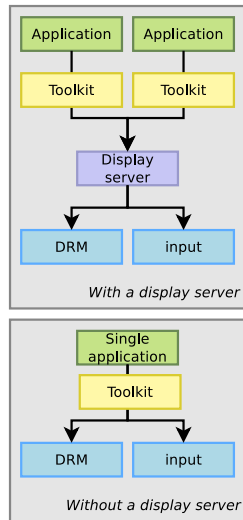
Wayland compositors

- ▶ 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





- ▶ 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

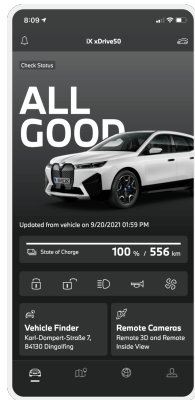
- ▶ Toolkit used as the base for the GNOME desktop environment, the most popular desktop environment for Linux desktop distributions, but loosing traction in embedded projects.
- ▶ Composed of *glib* (core library), *pango* (text handling), *cairo* (vector graphics), *gtk* (widget library)
- ▶ Standard 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/>





- ▶ *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)



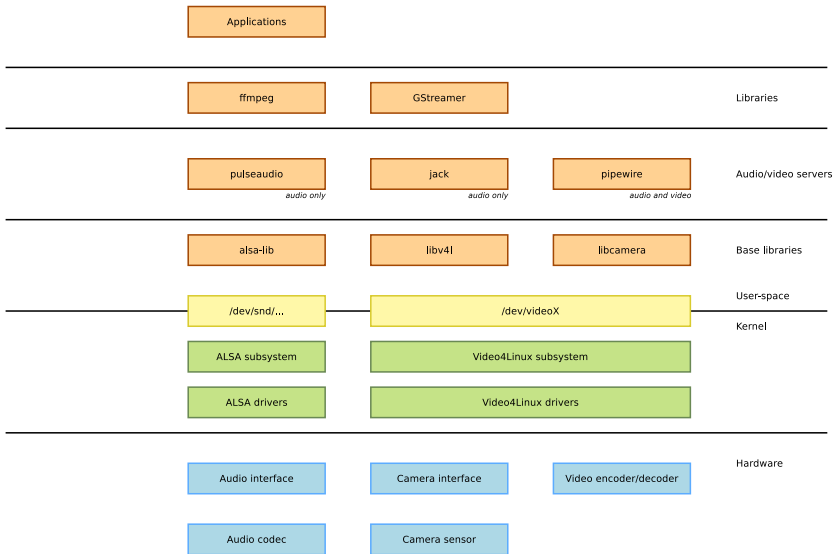


Other graphical toolkits

- ▶ 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



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/>



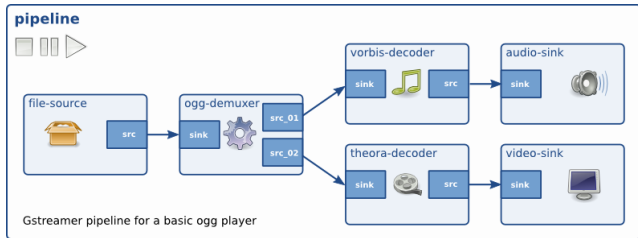
Video stack

- ▶ 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*





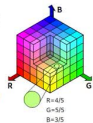
Further details on Linux graphics and multimedia stacks

- ▶ Bootlin's *Understanding the Linux graphics stack* training
- ▶ Bootlin's *Embedded Linux Audio* training
- ▶ Complete courses focused exclusively on those topics
- ▶ Freely available training materials



Color quantization approaches

- ▶ Different approaches exist for **color quantization**:
 - **Uniform** quantization in the color range (most common)
values are attributed to colors with a regular step (resolution)
 - **Irregular** quantization with indexed colors (palettes)
values are attributed to colors as needed
- ▶ Uniform color coordinates are quantized with:
 - A given **resolution**: the smallest possible color difference
 - A given **range**: the span of representable colors
- ▶ A given number of bits are used for quantization: **bit depth**
- ▶ A **trade-off** between range and resolution must be defined
 - Increasing the resolution reduces the range
 - Increasing the range reduces the resolution

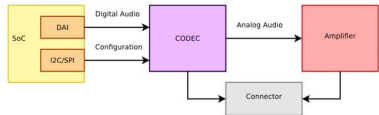


Bootlin - Kernel, drivers and embedded Linux - Development, consulting, training and support - <https://bootlin.com>

23/33



Anatomy



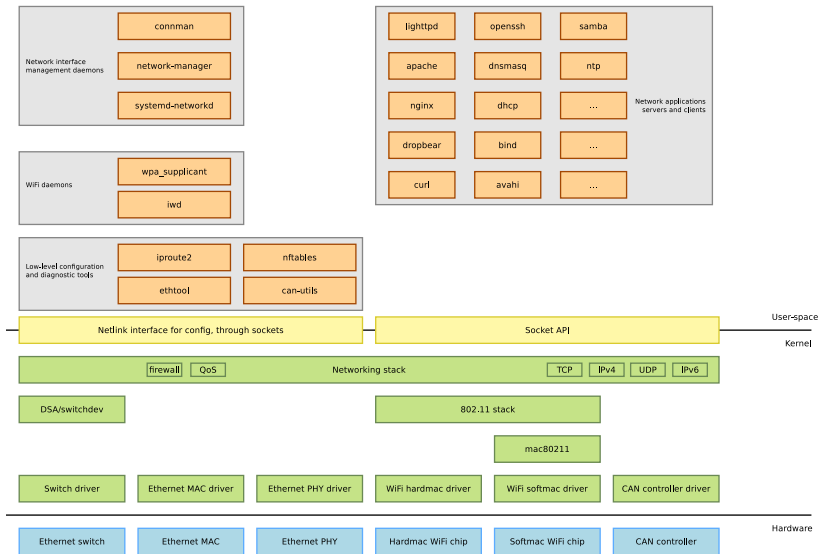
Example of an embedded system sound card

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

23/33



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_\(browser_engine\)](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



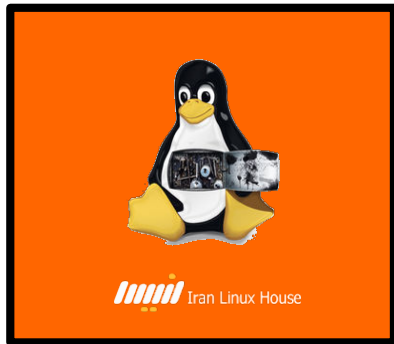
Embedded Linux application development

Based on slides by **Bootlin**

© Copyright 2004-2026, 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, which is often false for embedded systems (binaries are stripped, without `debug_info`) to save storage space.
- ▶ For the same reason, embedding the `gdb` program on embedded targets is rarely desirable (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).

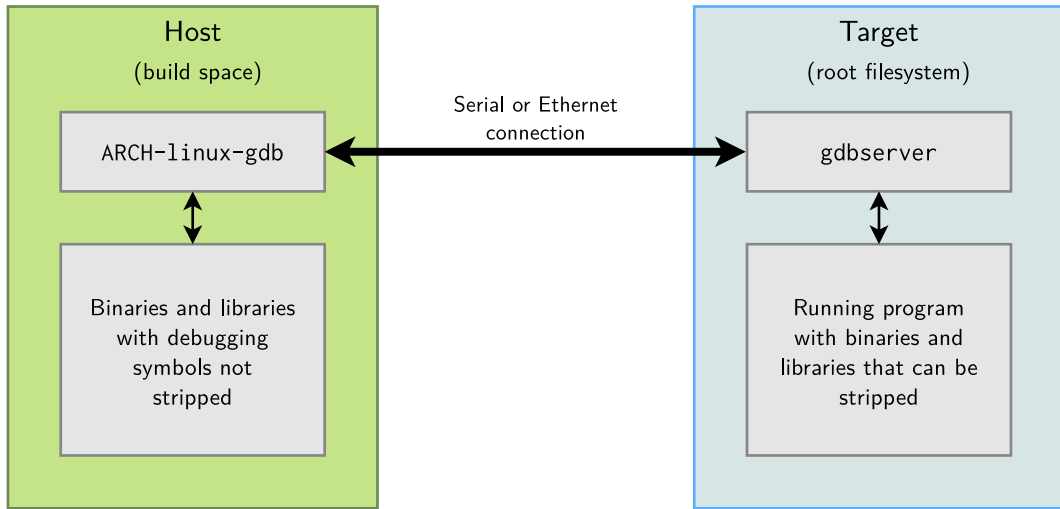
ARCH-linux-gdb

`gdbserver`





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

- ▶ It is sometime not possible to have a debugger attached when a crash occurs
- ▶ Fortunately, Linux can generate a `core` file (a snapshot of the whole process 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 and path for the coredump file can be modified using `/proc/sys/kernel/core_pattern` (see [man 5 core](#))
 - Example: `echo /tmp/mycore > /proc/sys/kernel/core_pattern`
 - Depending on the system configuration, the `core_pattern` file may be rewritten automatically by some software to handle core files or even disable core generation (eg: `systemd`)
- ▶ 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`



- ▶ 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=q2zwmwrgLJGs>
 - Slides: elinux.org/images/8/81/Eoss2023_ogness_minicoredumper.pdf



Tracing and profiling



strace

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>` (follow child processes too)
`strace -p <pid>` (tracing an existing process)
`strace -c <command>` (time statistics per system call)
`strace -e <expr> <command>` (use expression for advanced filtering)

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

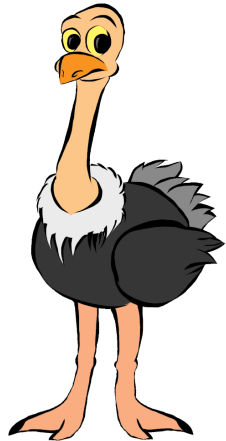


Image credits: <https://strace.io/>



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\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) = 0xf7f290d28000  
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		writew
2.69	0.038865	10	3712		read
2.14	0.030927	3	10807		getpid
0.28	0.003977	1	3341	34	futex
0.21	0.002991	3	1030	269	openat
0.20	0.002889	2	1619	975	stat
0.18	0.002534	4	568		mmap
0.13	0.001851	5	356		mprotect
0.10	0.001512	2	784		close
0.08	0.001171	3	461	315	access
0.07	0.001036	2	538		fstat

```
...
```



A tool to trace **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	strcmp
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



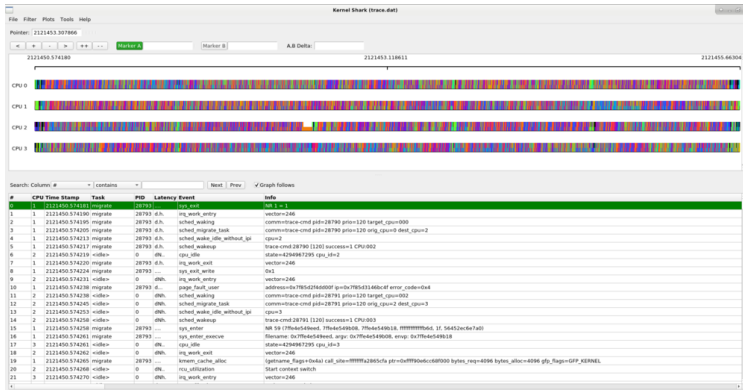
ftrace

- ▶ 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/>



kernelshark

- ▶ 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





- ▶ 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/>



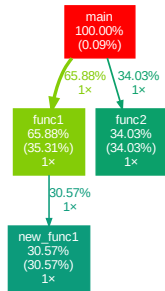
gprof example

```
$ ./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 L1, 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*

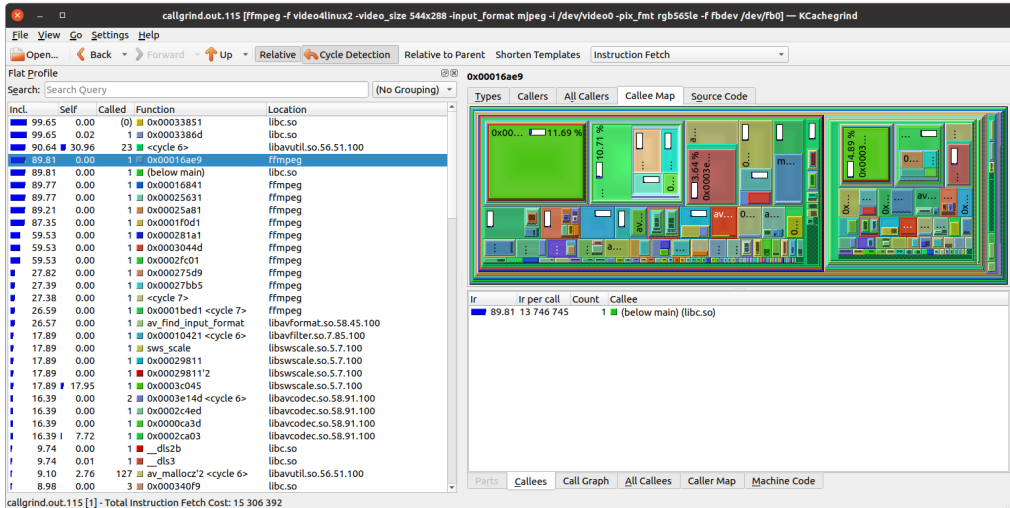
```
$ valgrind --leak-check=yes <program>
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==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

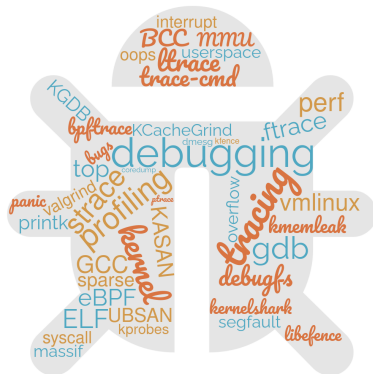


<https://github.com/KDE/kcachegrind>



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/>.





Practical lab - Application development and 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.



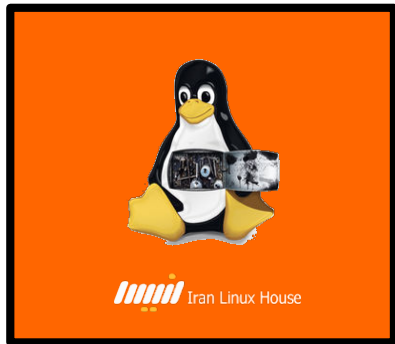
Useful resources

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





- ▶ **Mastering Embedded Linux Programming, 4th Edition**¹

By Frank Vasquez, Chris Simmonds, Packt Publishing, May 2025

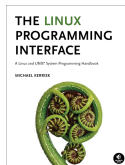
An up-to-date resource covering most aspects of embedded Linux development.



- ▶ **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.amazon.com/dp/1803232595>

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)

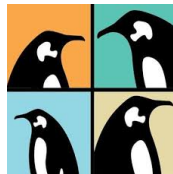
▶ Embedded Linux Conference:

- <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

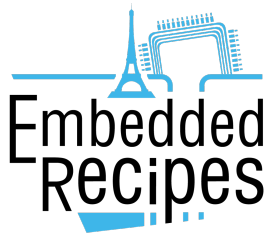
- <https://lpc.events/>
- About the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc.
- Not really a conventional conference with formal presentations, but rather a place where contributors on each topic meet, share their progress and make plans for work 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.





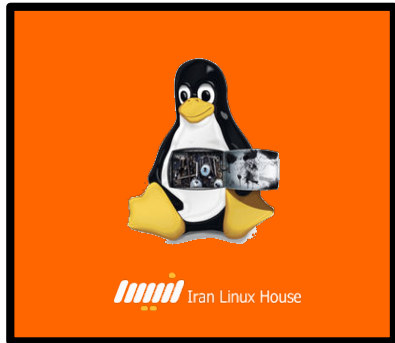
Last slides

Based on slides by **Bootlin**

© Copyright 2004-2026, 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 2004-2026, 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/>



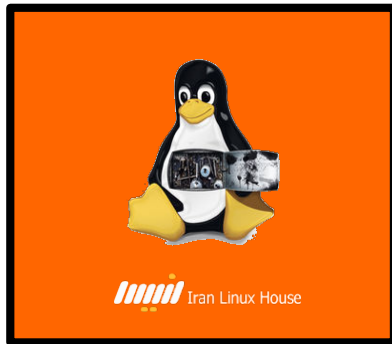
Extra slides

Based on slides by **Bootlin**

© Copyright 2004-2026, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Linux connectivity stack

