Linux Kernel, Board Support and Driver Development training course

Training course

Michael Opdenacker

Root Commit

June 13, 2026

Download Course Materials

Please download the materials for the course:

- Lectures:
  https://rootcommit.com/pub/training/linux-kernel/kernel-lectures.pdf
- Labs: https://rootcommit.com/pub/training/linux-kernel/kernel-labs.pdf

Introduction

License of these training materials

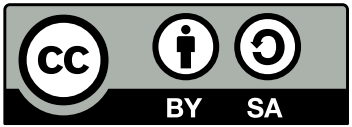
Attribution-ShareAlike 4.0 International

You are free to:

- Share — copy and redistribute the material in any medium or format for any purpose, even commercially.
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.
- The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.



https://creativecommons.org/licenses/by-sa/4.0/

Sources: https://gitlab.com/rootcommit/training-materials/

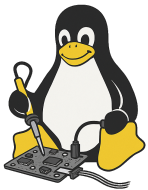
Vincent Branch

You can help making this course better and add your name to the above list by sending suggestions, testing the instructions and reporting typos and bugs.

Root Commit Services

Consulting and engineering work

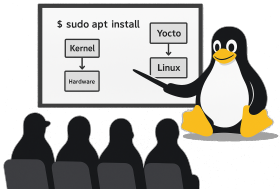
- Yocto Project
  Project reviews, system implementation, new features
- Linux Kernel
  Driver development, board bring-up, debugging
- Embedded Linux
  Boot time, bug fixing, security and other optimizations



Training — https://rootcommit.com/training

- Yocto Project and OpenEmbedded — Free Materials!
- Linux kernel, board support, driver development
- Embedded Linux
- Linux Boot Time Reduction

What's special: focus on practical activities, interactivity and learning techniques.



Michael Opdenacker

root commit

Embedded Linux consultant and trainer

- Former founder of Bootlin
- New founder of Root Commit
- Also worked at STMicroelectronics, Texas Instruments and Linaro
- 22+ years of training experience
- Free Software enthusiast and advocate (member of April.org)

April
promouvoir et défendre le logiciel libre

- Linux kernel trainer since 2004 (pre-git times!)
- First contribution in 2006 (2.6.18)
- 168 contributions as of June 2026
- Made multiple presentations
  at international conferences over the years.
- Currently working on a kernel upstreaming project
  on the RISC-V architecture

https://rootcommit.com/about/michael-opdenacker/

Introduction

Learning Techniques

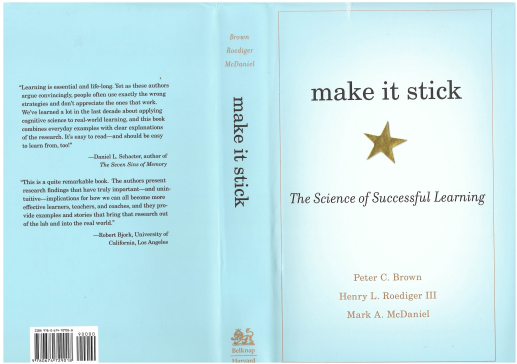
Experience from 20+ years of training

root commit

- Need challenging labs (good)
- But too long series of lectures, people passive for too long
- And tendency to explain the whole theory (as *exhaustively* as possible), before letting people experiment
- And quiz only at the end
- You forget quickly if you stop using

What I'm trying to do

- More practical lab time
- Challenging labs
  - You don't learn from labs that are too easy
  - If you don't know how to do something, it's often because you missed something in the lectures
  - Making mistakes is very positive: that's how you build experience and correct misconceptions
  - And the instructor is here to avoid staying stuck for too long
- Online sessions: people do their labs instead of just watching demos
- More interaction with the audience
- Lectures should never exceed 30 minutes (except if many questions)
- More self-testing (quizzes)

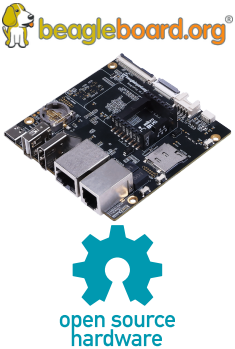


https://rootcommit.com/2024/make-it-stick-book-review/

Supported hardware

BeaglePlay, from BeagleBoard.org

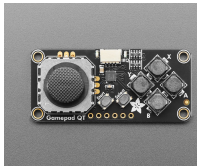
- Texas Instruments AM625x (4xARM Cortex-A53 CPU)
- SoC with 3D acceleration, integrated MCU and many other peripherals.
- 2 GB of RAM
- 16 GB of on-board eMMC storage
- USB host and USB device, microSD, HDMI
- 2.4 and 5 GHz WiFi, Bluetooth and also Ethernet
- 1 MicroBus Header (SPI, I2C, UART, ...), OLDI and CSI connector.

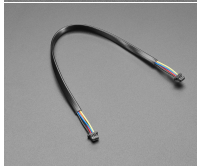


Shopping list: hardware for this course

root commit

- Raspberry Pi Debug Probe or USB Serial Cable - 3.3 V with **female** ends (for serial console) [a]
- Adafruit Mini I2C Gamepad [b]
- Gable with JST-SH female 4-pin connectors (to connect the Gamepad) [c]
- One more Raspberry Pi Debug Probe or USB Serial Cable - 3.3 V with **male** ends (for serial labs, two if possible) [d]





[a] https://www.raspberrypi.com/documentation/microcontrollers/debug-probe.html

[b] https://www.adafruit.com/product/5743

[c] https://www.adafruit.com/product/4401

[d] https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-M/

Lab — Preparing Your Environment

- Download and extract lab data

Introduction

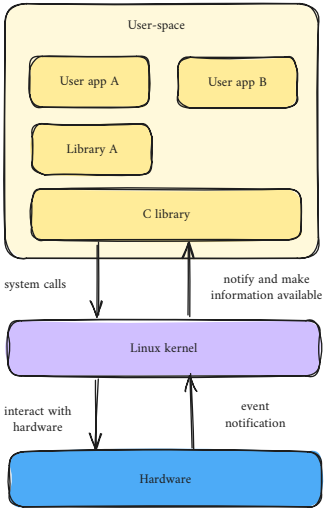
Linux Kernel Introduction

Origin

- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- As of today, about 2,000+ people contribute to each kernel release, individuals or companies big and small.

Linus Torvalds in 2014
Image credits (Wikipedia):
https://bit.ly/2UIa1TD

Linux kernel in the system



Linux kernel main roles

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

System calls

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

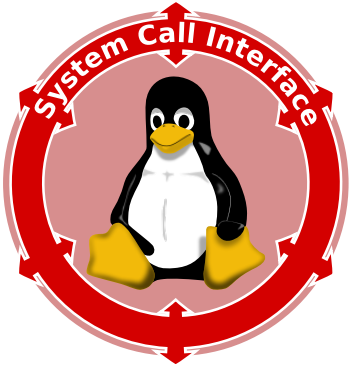


Image credits (Wikipedia): https://bit.ly/2U2rdGB

Pseudo filesystems

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

Lab — BeaglePlay Setup

root commit

- Access the serial console
- Boot to U-Boot
- Set up networking between the PC and board
- tftp networking between the board and PC

Configuring, building, booting and kernel modules

Configuring, building, booting and kernel modules

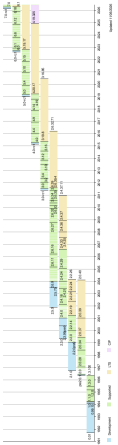
Linux kernel sources

Location of the official kernel sources

- The mainline versions of the Linux kernel, as released by Torvalds
  - These versions follow the development model of the kernel (`master` branch)
  - They may not contain the latest developments from a specific area yet
  - A good pick for products development phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Linux versioning scheme

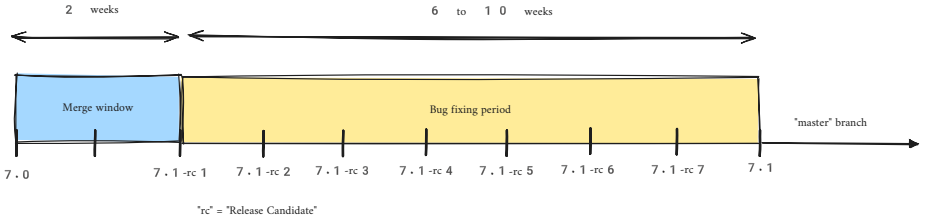
- Until 2003, there was a new "stabilized" release branch of Linux every 2 or 3 years (2.0, 2.2, 2.4). Development branches took 2-3 years to be merged (too slow!).
- Since 2003, there is a new official release of Linux about every 10 weeks:
  - Versions `2.6` (Dec. 2003) to `2.6.39` (May 2011)
  - Versions `3.0` (Jul. 2011) to `3.19` (Feb. 2015)
  - Versions `4.0` (Apr. 2015) to `4.20` (Dec. 2018)
  - Versions `5.0` (Mar. 2019) to `5.19` (July 2022)
  - Versions `6.0` (Oct. 2022) to `6.19` (Feb. 2026)
  - Version `7.0` was released in Apr. 2026.
- Features are added to the kernel in a progressive way. Since 2003, kernel developers have managed to do so without having to introduce a massively incompatible development branch.



Linux kernel version history
https://en.wikipedia.org/wiki/Linux_kernel_version_history

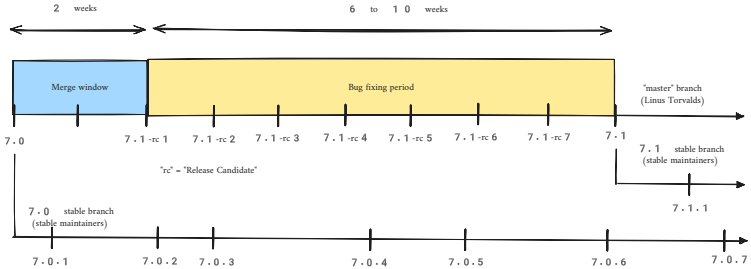
Linux development model

- Each new release starts with a two-week merge window for new features
- Follow about 8 release candidates (one week each)
- Until adoption of a new official release.

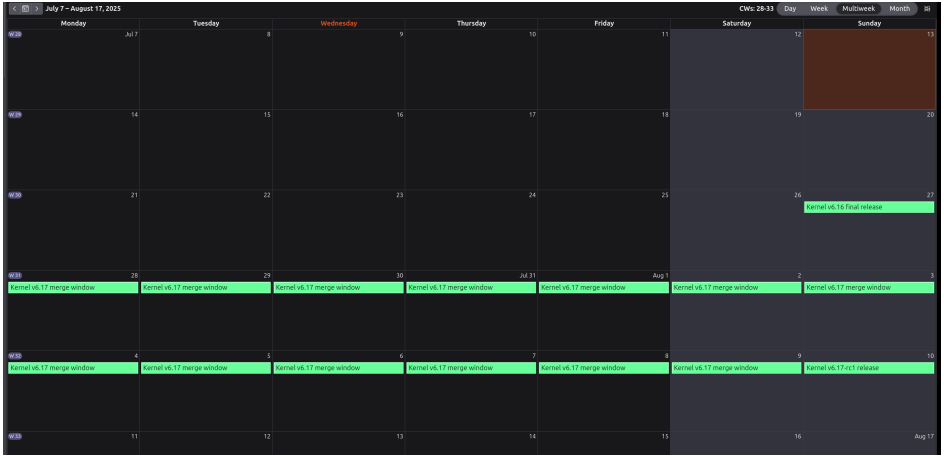


Need to further stabilize the official kernels

- Issue: bug and security fixes only merged into the master branch, need to update to the latest kernel to benefit from them.
- Solution: a stable maintainers team goes through all the patches merged into Torvald's tree and backports the relevant ones into their stable branches for at least a few months.



Fun Fact — Kernel Release Calendar



Subscribe to it as a network calendar: https://www.kernel.org/releases-calendar.ics

Location of the stable kernel sources

- The stable versions of the Linux kernel, as maintained by a maintainers group
  - These versions do not bring new features compared to Linus' tree
  - Only bug fixes and security fixes are pulled there
  - Each version is stabilized during the development period of the next mainline kernel
  - A good pick for products commercialization phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
  - Some versions will be maintained much longer

Need for long term support

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

Longterm release kernels

Version	Maintainer	Released	Projected EOL
6.18	Greg Kroah-Hartman & Sasha Levin	2025-11-30	Dec, 2028
6.12	Greg Kroah-Hartman & Sasha Levin	2024-11-17	Dec, 2028
6.6	Greg Kroah-Hartman & Sasha Levin	2023-10-29	Dec, 2027
6.1	Greg Kroah-Hartman & Sasha Levin	2022-12-11	Dec, 2027
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Dec, 2026
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026

Captured on `https://kernel.org` in June 2026, following the *Releases* link.

Need for even longer term support

- You could also get long term support from a commercial embedded Linux provider.
  - Wind River Linux can be supported for up to 15 years.
  - Ubuntu Core can be supported for up to 10 years.
- *"If you are not using a supported distribution kernel, or a stable / longterm kernel, you have an insecure kernel"* - Greg KH, 2019
  Some vulnerabilities are fixed in stable without ever getting a CVE.
- The *Civil Infrastructure Platform* project is an industry / Linux Foundation effort to support much longer (at least 10 years) selected LTS versions (currently 4.4, 4.19, 5.10, 6.1 and 6.12) on selected architectures. See
  https://wiki.linuxfoundation.org/civilinfrastructureplatform/start.

Location of non-official kernel sources

- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Sometimes they break support for other platforms/devices without caring
  - Useful in early phases only when mainline hasn't caught up yet (many vendors invest in the mainline kernel at the same time)
  - Suitable for PoC, not suitable for products on the long term as usually no updates are provided to these kernels
  - Getting stuck with a deprecated system with broken software that cannot be updated has a real cost in the end

Linux kernel size and structure

root commit

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

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

Lab — Fetching Kernel Sources

- Clone the mainline Linux source tree with git

Configuring, building, booting and kernel modules

Linux kernel source code

Programming language

- Implemented in C like all UNIX systems
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines.
- No C++ used, see http://vger.kernel.org/lkml/#s15-3
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - See https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html
- A majority of supported architectures can be built with the LLVM C compiler (Clang) too: https://clangbuiltlinux.github.io/
- Rust support is currently being introduced: drivers/net/phy/ax88796b_rust.rs is a first driver written in Rust.

No C library

root commit

- The kernel has to be standalone and can't use user space code.
- Architectural reason: user space is implemented on top of kernel services, not the opposite.
- Technical reason: the kernel is on its own during the boot up phase, before it has accessed a root filesystem.
- Hence, kernel code has to supply its own library implementations (string utilities, cryptography, uncompression...)
- So, you can't use standard C library functions in kernel code (`printf()`, `memset()`, `malloc()`,...).
- Fortunately, the kernel provides similar C functions for your convenience, like `printk()`, `memset()`, `kmalloc()`, ...

Portability

- The Linux kernel code is designed to be portable
- All code outside arch/ should be portable
- To this aim, the kernel provides macros and functions to abstract the architecture specific details
  - Endianness
  - I/O memory access
  - Memory barriers to provide ordering guarantees if needed
  - DMA API to flush and invalidate caches if needed
- Never use floating point numbers in kernel code.
  Your code may need to run on a low-end processor without a floating point unit.

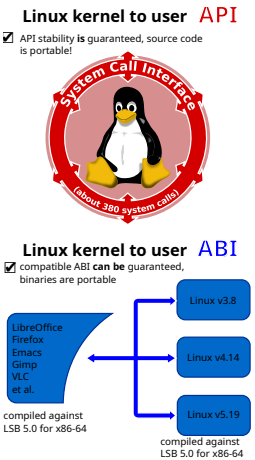
Linux kernel to user API/ABI stability

Linux kernel to userspace API is stable

- Source code for userspace applications will not have to be updated when compiling for a more recent kernel
- System calls, `/proc` and `/sys` content cannot be removed or changed. Only new entries can be added.

Linux kernel to userspace ABI is stable

- Binaries are portable and can be executed on a more recent kernel
- The way memory is accessed, the size of the variables in memory, how structures are organized, the calling convention, etc, are all stable over time.



Modified Image from Wikipedia: https://bit.ly/2U2rdGB

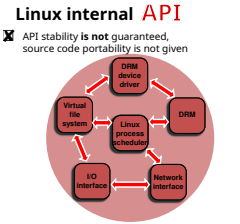
Linux internal API/ABI instability

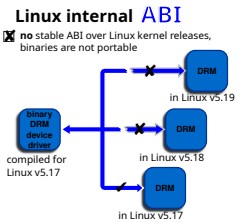
Linux internal API is not stable

- The source code of a driver is not portable across versions
- In-tree drivers are updated by the developer proposing the API change: works great for mainline code
- An out-of-tree driver compiled for a given version may no longer compile or work on a more recent one
- See `process/stable-api-nonsense` for reasons why

Linux internal ABI is not stable

- A binary module compiled for a given kernel version cannot be used with another version
- The module loading utilities will perform this check prior to the insertion





Modified Image from Wikipedia: `https://bit.ly/2U2rdGB`

Kernel memory constraints

root commit

- No memory protection
- The kernel doesn't try to recover from attemps to access illegal memory locations. It just dumps *oops* messages on the system console.
- Fixed size stack (8 or 4 KB).
  Unlike in user space, no mechanism was implemented to make it grow.
  Don't use recursion!
- Swapping is not implemented for kernel memory either
  (except *tmpfs* which lives completely in the page cache and on swap)

Linux kernel licensing constraints

root commit

- The Linux kernel is licensed under the GNU General Public License version 2
  - This license gives you the right to use, study, modify and share the software freely
- However, when the software is redistributed, either modified or unmodified, the GPL requires that you redistribute the software under the same license, with the source code
  - If modifications are made to the Linux kernel (for example to adapt it to your hardware), it is a derivative work of the kernel, and therefore must be released under GPLv2.
- The GPL license has been successfully enforced in courts: https://en.wikipedia.org/wiki/Gpl-violations.org#Notable_victories
- However, you're only required to do so
  - At the time the device starts to be distributed
  - To your customers, not to the entire world

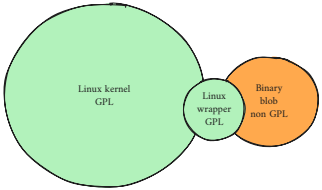
Proprietary code and the kernel

- It is illegal to distribute a binary kernel that includes statically compiled proprietary drivers
- The kernel modules are a gray area: unclear if they are legal or not
  - The general opinion of the kernel community is that proprietary modules are bad: `process/kernel-driver-statement`
  - From a legal point of view, each driver is probably a different case:
    - Are they derived works of the kernel?
    - Are they designed to be used with another operating system?
- Is it really useful to keep drivers secret anyway?

Abusing the kernel licensing constraints

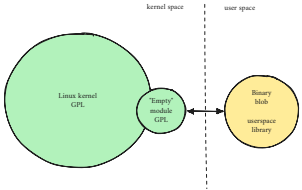
There are some examples of proprietary drivers

- Nvidia uses a wrapper between their drivers and the kernel
- They claim the drivers could be used with a different OS with another wrapper
- Unclear whether it makes it legal or not



The current trend is to hide the logic in the firmware or in userspace. The GPL kernel driver is almost empty and either:

- Blindly writes an incoming flow of bytes in the hardware
- Exposes a huge MMIO region to userspace through `mmap`



Advantages of GPL drivers

- You don't have to write your driver from scratch.
  You can reuse code from similar free software drivers.
- Your drivers can be freely and easily shipped by others
  (for example by Linux distributions or embedded Linux build systems).
- Legal certainty, you are sure that a GPL driver is fine from a legal point of view.

Advantages of mainlining your kernel drivers

root commit

- The community, reviewers and maintainers will review your code before accepting it, offering you the opportunity to enhance it and understand better the internal APIs.
- Once accepted, you will get cost-free bug and security fixes, support for new features, and general improvements.
- Your work will automatically follow the API changes.
- Accessing your code will be much easier for users.
- Your code will remain valid no matter the kernel version.

This will for sure reduce your maintenance and support work

User space device drivers 1/2

root commit

- The kernel provides some mechanisms to access hardware from userspace:
  - USB devices with *libusb*, https://libusb.info/
  - SPI devices with *spidev*, spi/spidev
  - I2C devices with *i2cdev*, i2c/dev-interface
  - GPIOs with *libgpiod*, https://libgpiod.readthedocs.io
  - Memory-mapped devices with *UIO*, including interrupt handling, driver-api/uio-howto
- These solutions can only be used if:
  - There is no need to leverage an existing kernel subsystem such as the networking stack or filesystems.
  - There is no need for the kernel to act as a "multiplexer" for the device: only one application accesses the device.
- Specific classes of devices like printers and scanners do not have any kernel support, they have always been handled in user space for historical reasons.
- Otherwise this is ***not*** how the system should be architectured. Kernel drivers are always preferred.

User space device drivers 2/2

root commit

- Advantages
  - No need for kernel coding skills.
  - Drivers can be written in any language, even Perl!
  - Drivers can be kept proprietary.
  - Driver code can be killed and debugged. Cannot crash the kernel.
  - Can use floating-point computation.
  - Potentially higher performance, especially for memory-mapped devices, thanks to the avoidance of system calls.
- Drawbacks
  - The kernel has no longer access to the device.
  - None of the standard applications will be able to use it.
  - Cannot use any hardware abstraction or software helpers from the kernel
  - Need to adapt applications when changing the hardware.
  - Less straightforward to handle interrupts: increased latency.

Configuring, building, booting and kernel modules

Kernel source management tools

Cscope

root commit

- Tool to browse source code (mainly C, but also C++ or Java)
- Supports huge projects like the Linux kernel. Typically takes less than 1 min. to index the whole Linux sources.
- In Linux kernel sources, two ways of running it:
  - `cscope -Rk`
    All files for all architectures at once
  - `make cscope`
    `cscope -d cscope.out`
    Only files for your current architecture
- Allows searching for a symbol, a definition, functions, strings, files, etc.
- Integration with editors like `vim` and `emacs`.
- http://cscope.sourceforge.net/

Cscope screenshot

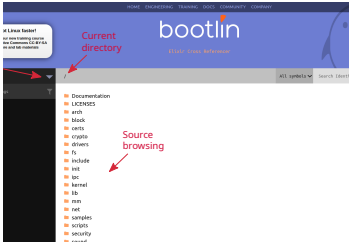
```
C symbol: request_irq

   File                       Function                  Line
0  board-osk.c                osk_mistral_init           519 ret = request_irq(irq,
1  board-palmz71.c            palmz71_gpio_setup         260 if (request_irq(gpio_to_irq(PALMZ71_USBDETECT_GPIO),
2  lcd_dma.c                  omap_init_lcd_dma          436 r = request_irq(INT_DMA_LCD, lcd_dma_irq_handler, 0,
3  serial.c                   omap_serial_set_port_wake  228 ret = request_irq(gpio_to_irq(gpio_nr),
                                                              &omap_serial_wake_interrupt,
4  pm34xx.c                   omap3_pm_init              472 ret = request_irq(omap_prcm_event_to_irq("wkup"),
5  pm34xx.c                   omap3_pm_init              481 ret = request_irq(omap_prcm_event_to_irq("io"),
6  am200epd.c                 am200_setup_irq            295 ret = request_irq(PXA_GPIO_TO_IRQ(RDY_GPIO_PIN),
                                                              am200_handle_irq,
7  am300epd.c                 am300_setup_irq            244 ret = request_irq(PXA_GPIO_TO_IRQ(RDY_GPIO_PIN),
                                                              am300_handle_irq,

* Lines 41-49 of 1688, 1640 more - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

[Tab]: move the cursor between search results and commands

[Ctrl] [D]: exit cscope

Elixir: browsing the Linux kernel sources

- https://github.com/bootlin/elixir
- Generic source indexing tool and code browser for C and C++.
  Inspired by the LXR project (Linux Cross Reference).
- Web server based, very easy and fast to use
- Very easy to find the declaration, implementation or usage of symbols
- Supports huge code projects such as the Linux kernel with a git repository.
  Scales much better than LXR by only indexing new git objects found in each new release.
- Takes a little time and patience to setup
  (configuration, indexing, web server configuration)
- You don't need to set up Elixir by yourself.
  Use the https://elixir.bootlin.com server!



Lab — Exploring Kernel Source Code

- Explore kernel sources manually
- Use automated tools to explore the source code

Configuring, building, booting and kernel modules

Kernel configuration

Kernel configuration

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- Kernel configuration is the process of defining the set of options you want your kernel to be compiled with
- The set of options depends
  - On the target architecture and on your hardware (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.). Such generic options are available in all architectures.

Kernel configuration and build system

root commit

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main Makefile,
  present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the `make` tool, which parses the Makefile
  - through various **targets**, defining which action should be done
    (configuration, compilation, installation, etc.).
  - Run `make help` to see all available targets.
- Example
  - `cd linux/`
  - `make <target>`

Specifying the target architecture

First, specify the architecture for the kernel to build

- Set `ARCH` to the name of a directory under arch/:
  `ARCH=arm` or `ARCH=arm64` or `ARCH=riscv`, etc
- By default, the kernel build system assumes
  that the kernel is configured and built for the host architecture
  (`x86` in our case, native kernel compiling)
- The kernel build system will use this setting to:
  - Use the configuration options for the target architecture.
  - Compile the kernel with source code and headers for the target architecture.

Choosing a compiler

The easiest way to compile the Linux kernel is with Clang from LLVM

- That's because `clang` is a cross-compiler supporting all target architectures at once.
- No more need to install an architecture specific cross-compiler!
- Only one setting necessary:
  `export LLVM=1`
- To compile natively, do not set anything.
- To compile with gcc
  - The Makefile invokes `$(CROSS_COMPILE)gcc` by default
  - Therefore install a specific toolchain package:
    `sudo apt install gcc-aarch64-linux-gnu`
    →The cross-compiler executable is then `aarch64-linux-gnu-gcc`
  - Set `CROSS_COMPILE` to the prefix of the cross-compiler:
    `CROSS_COMPILE=aarch64-linux-gnu-`

Specifying ARCH, LLVM, CROSS_COMPILE...

root commit

There are actually two ways of passing these variables

- Pass them on the `make` command line:
  `make ARCH=arm LLVM=1 ...`
  Drawback: it is easy to forget to pass these variables
  when you run any `make` command,
  causing your build and configuration to be screwed up.
- Define them as environment variables:
  `export ARCH=arm`
  `export LLVM=1`
  Drawback: it only works inside the current shell or terminal.
  You could put these settings in a file that you source every time
  you start working on the project.
  Seee also the https://direnv.net/ project.

Initial configuration

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

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

Create your own default configuration

root commit

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

Built-in or module?

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

Kernel option types

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

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

Kernel option dependencies

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

- `depends on` dependency:

```
config B
    depends on A
```

  - B is not visible until A is enabled
  - Standard dependency chains

- `select` dependency:

```
config A
    select B
```

  - When A is enabled, B is enabled too (and cannot be disabled manually)
  - Used to declare hardware features or select libraries

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

Kernel configuration details

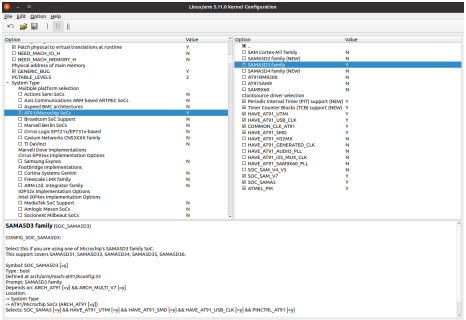
The configuration is stored in the `.config` file at the root of kernel sources

- Simple text file, `CONFIG_PARAM=value`
- Options are grouped by sections and are prefixed with `CONFIG_`
- "No" value is encoded as `# CONFIG_FOO is not set`
- Included by the top-level kernel Makefile
- Typically not edited by hand because of dependencies

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

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

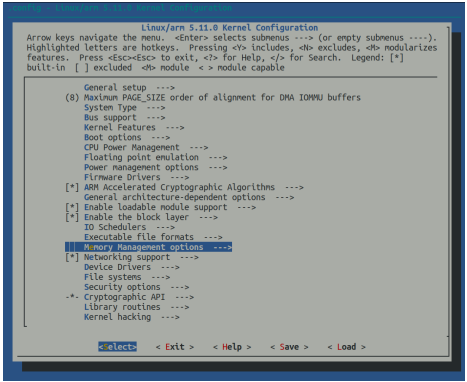
`make xconfig`

- A graphical interface to configure the kernel.
- File browser: easy to load configuration files
- Search interface to look for parameters (`[Ctrl] + [f]`)
- Required Debian/Ubuntu packages:
  `qtbase5-dev` on Ubuntu 22.04 / 24.04



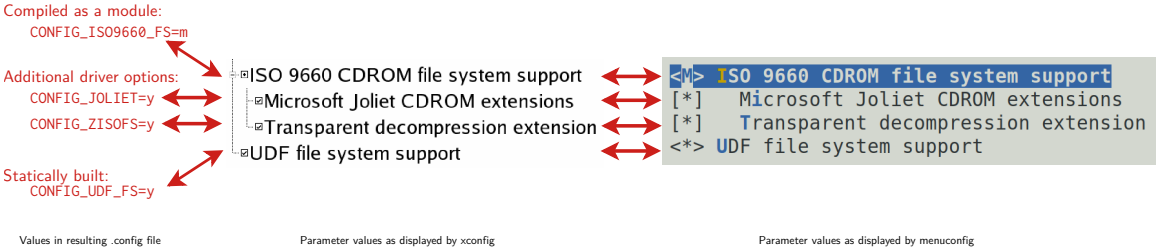
`make menuconfig`

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



Kernel configuration options

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



make oldconfig

`make oldconfig`

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

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

make olddefconfig

`make olddefconfig`

- Like `make oldconfig`, but takes default values without asking
- When you want to upgrade your configuration, and are in a hurry 😁

Undoing configuration changes

A frequent problem:

- After changing several kernel configuration settings, your kernel no longer works.
- If you don't remember all the changes you made,
  you can get back to your previous configuration:
  `$ cp .config.old .config`
- All the configuration tools keep this `.config.old` backup copy.

Configuring, building, booting and kernel modules

Compiling and installing the kernel

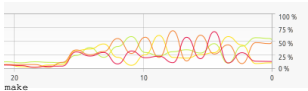
Kernel compilation

`make`

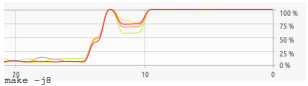
- Only works from the top kernel source directory
- Should not be performed as a privileged user
- Run several **j**obs in parallel.
  Our advice: `$(nproc)`
  to fully load the CPU and I/Os at all times.
  Example: `make -j20`

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

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



Command: make
Total time: 129 s



Command: make -j8
Total time: 67 s

Build the mainline kernel on your PC

Video: https://youtu.be/zeDCODvmo14

Slides: https://rootcommit.com/2025/mainline-linux-on-pc/

Lab — Configuring and compiling the kernel

root commit

- Set up a cross-compiling environment
- Configure the kernel for BeaglePlay
- Start compiling the kernel

Kernel compilation results

- `arch/<arch>/boot/Image`, uncompressed kernel image that can be booted
- `arch/<arch>/boot/*Image*`, compressed kernel images that can also be booted
  - `bzImage` for `x86`, `zImage` for `arm`, `Image.gz` for `arm64` and `riscv`...
- `arch/<arch>/boot/dts/<vendor>/*.dtb`, compiled Device Tree Blobs
- All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.
- `vmlinux`, a raw uncompressed kernel image in the ELF format, useful for debugging purposes but generally not used for booting purposes

Kernel installation: native case

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

Kernel installation: embedded case

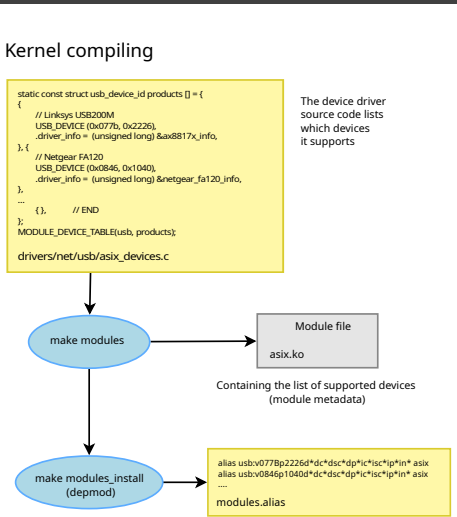
- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle.
- Another reason is that there is no standard way to deploy and use the kernel image.
- Therefore making the kernel image available to the target is usually manual or done through scripts in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

Module installation: native case

root commit

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

Module alias: `modules.alias`



Module installation: embedded case

- In embedded development, you can't directly use `make modules_install` as it would install target modules in `/lib/modules` on the host!
- The `INSTALL_MOD_PATH` variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem (no need to be root):
  `make INSTALL_MOD_PATH=<dir>/ modules_install`

Kernel cleanup targets

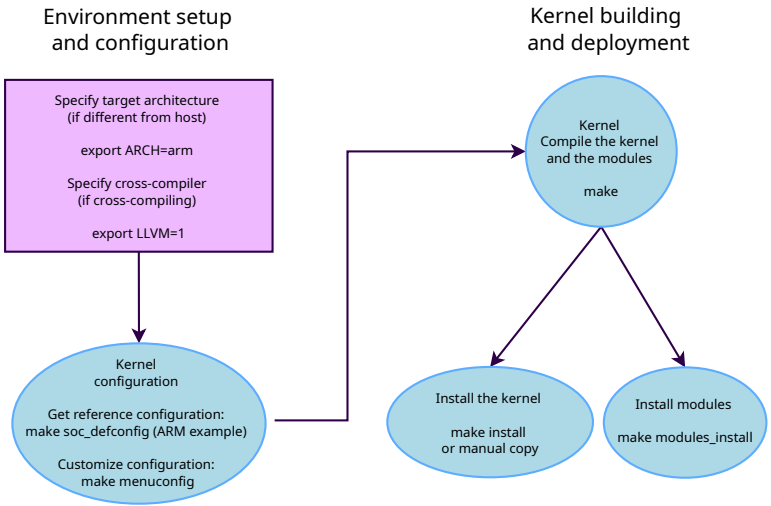
root commit

- From `make help`:

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

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

Kernel building overview



Configuring, building, booting and kernel modules

Booting the kernel

Hardware description

- Many embedded architectures have a lot of non-discoverable hardware (serial, Ethernet, I2C, Nand flash, USB controllers...)
- This hardware needs to be described and passed to the Linux kernel.
- The bootloader/firmware is expected to provide this description when starting the kernel:
  - On x86: using ACPI tables
  - On most embedded devices: using an OpenFirmware Device Tree (DT)
- This way, a kernel supporting different SoCs knows which SoC and device initialization hooks to run on the current board.

Booting with U-Boot

- On ARM32, U-Boot can boot `zImage` (`bootz` command)
- On ARM64 or RISC-V, it boots the `Image` file (`booti` command)
- In addition to the kernel image, U-Boot should also pass a DTB to the kernel.
- The typical boot process is therefore:
  1. Load `zImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `boot[z|i] X - Y`
     The `-` in the middle indicates no *initramfs*

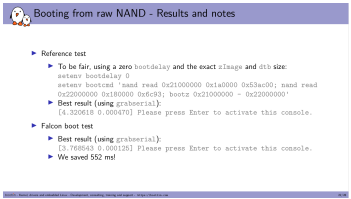
Kernel command line

- In addition to the compile time configuration,
  the kernel behavior can be adjusted with no recompilation
  using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root filesystem (covered later)
  - `console=` for the destination of kernel messages
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 rootwait`
  - Many more exist. The most important ones are documented in
    `admin-guide/kernel-parameters` in kernel documentation.

Passing the kernel command line

- U-Boot carries the Linux kernel command line string in its `bootargs` environment variable
- Right before starting the kernel, it will store the contents of `bootargs` in the `chosen` section of the Device Tree
- The kernel will behave differently depending on its configuration:
  - If `CONFIG_CMDLINE_FROM_BOOTLOADER` is set:
    The kernel will use only the string from the bootloader
  - If `CONFIG_CMDLINE_FORCE` is set:
    The kernel will only use the string received at configuration time in `CONFIG_CMDLINE`
  - If `CONFIG_CMDLINE_EXTEND` is set:
    The kernel will concatenate both strings.
    Note: `arm64` users have to use `CONFIG_BOOT_CONFIG_EMBED` instead.

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



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

Kernel log

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

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

Lab — Kernel booting, NFS root filesystem

- Load kernel and device tree through tftp
- Uncompress kernel in bootloader
- Boot kernel without a filesystem
- Set up NFS server and client
- Boot kernel on NFS root filesystem

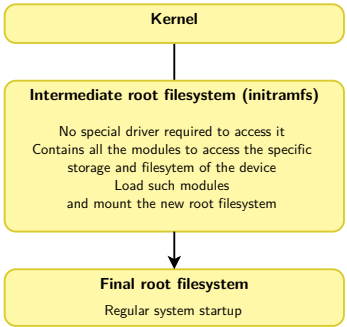
Configuring, building, booting and kernel modules

Using kernel modules

Advantages of modules

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

Using kernel modules to support many different devices and setups



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

Module utilities: extracting information

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

- modinfo <module_name> (for modules in /lib/modules)
  modinfo <module_path>.ko
  Gets information about a module without loading it:
  parameters, license, description and dependencies.

Module utilities: loading

- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.
- `sudo modprobe <top_module_name>`
  Most common usage of `modprobe`: tries to load all the dependencies of the given top module, and then this module. Lots of other options are available.
  `modprobe` automatically looks in `/lib/modules/<version>/`
  for the `.ko` file corresponding to the given module name.
- `lsmod`
  Displays the list of loaded modules
  Compare its output with the contents of `/proc/modules`!

Understanding module loading issues

- When loading a module fails, `insmod` often doesn't give you enough details!
- Details are often available in the kernel log.
- Example:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Module utilities: removals

- `sudo rmmod <module_name>`
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use
  (for example, no more processes opening a device file)
- `sudo modprobe -r <top_module_name>`
  Tries to remove the given top module and all its no longer needed dependencies

Passing parameters to modules

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

Check module parameter values

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

- Check `/sys/module/<name>/parameters`.
- There is one file per parameter, containing the parameter value.
- Also possible to change parameter values
  if these files have write permissions (depends on the module code).
- Example:
  `echo 0 > /sys/module/usb_storage/parameters/delay_use`

Configuring, building, booting and kernel modules

Developing kernel modules

Hello module 1/2

root commit

```c
// SPDX-License-Identifier: GPL-2.0
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Good morrow to this fair assembly.\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

Hello module 2/2

- Code marked as `__init`:
  - Removed after initialization (static kernel or module.)
  - See how init memory is reclaimed when the kernel finishes booting:

```
[    2.689854] VFS: Mounted root (nfs filesystem) on device 0:15.
[    2.698796] devtmpfs: mounted
[    2.704277] Freeing unused kernel memory: 1024K
[    2.710136] Run /sbin/init as init process
```

- Code marked as `__exit`:
  - Discarded when module compiled statically into the kernel, or when module unloading support is not enabled.
- Code of this example module available on

  https://gitlab.com/rootcommit/training-materials/-/raw/main/code/hello/hello.c

Hello module explanations

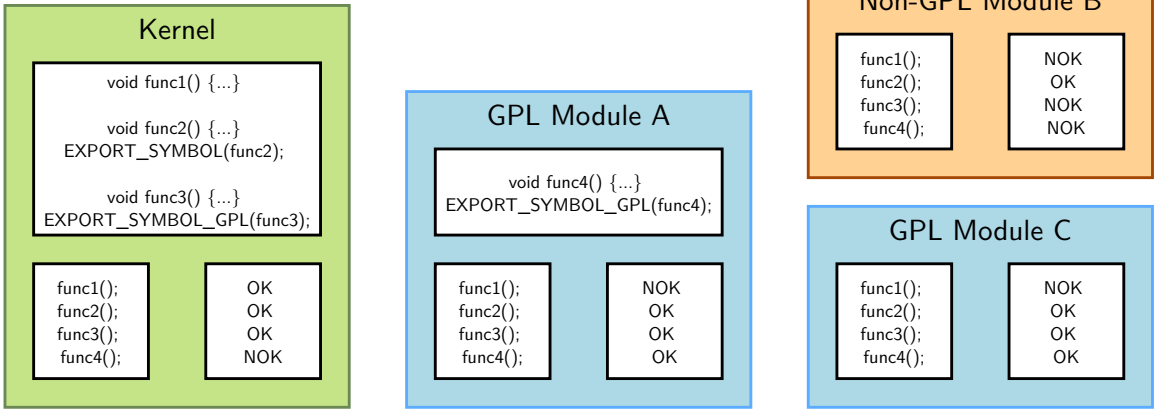
- Headers specific to the Linux kernel: `linux/xxx.h`
  - No access to the usual C library, we're doing kernel programming
- An initialization function
  - Called when the module is loaded, returns an error code (0 on success, negative value on failure)
  - Declared by the `module_init()` macro: the name of the function doesn't matter, even though `<modulename>_init()` is a convention.
- A cleanup function
  - Called when the module is unloaded
  - Declared by the `module_exit()` macro.
- Metadata information declared using `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` and `MODULE_AUTHOR()`

Symbols exported to modules 1/2

root commit

- From a kernel module, only a limited number of kernel functions can be called
- Functions and variables have to be explicitly exported by the kernel to be visible to a kernel module
- Two macros are used in the kernel to export functions and variables:
  - `EXPORT_SYMBOL(symbolname)`
    which exports a function or variable to all modules
  - `EXPORT_SYMBOL_GPL(symbolname)`
    which exports a function or variable only to GPL modules
  - Linux 7.1: contains 20% more symbols with `EXPORT_SYMBOL_GPL()` than with `EXPORT_SYMBOL()`
- A normal driver should not need any non-exported function.

Symbols exported to modules 2/2



Module license

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module
    - Difference between `EXPORT_SYMBOL()` and `EXPORT_SYMBOL_GPL()`
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about ("Tainted" kernel notice in kernel crashes and oopses).
  - See `admin-guide/tainted-kernels` for details about `tainted` flag values.
  - Useful for users to check that their system is 100% free (for the kernel, check `/proc/sys/kernel/tainted`; run `vrms` to check installed packages)
- Values
  - GPL compatible (see include/linux/license.h: `GPL`, `GPL v2`, `GPL and additional rights`, `Dual MIT/GPL`, `Dual BSD/GPL`, `Dual MPL/GPL`)
  - `Proprietary`

Compiling a module

Two solutions

- *Out of tree*, when the code is outside of the kernel source tree, in a different directory
  - Not integrated into the kernel configuration/compilation process
  - Needs to be built separately
  - The driver cannot be built statically, only as a module
- Inside the kernel tree
  - Well integrated into the kernel configuration/compilation process
  - The driver can be built statically or as a module

Compiling an out-of-tree module 1/2

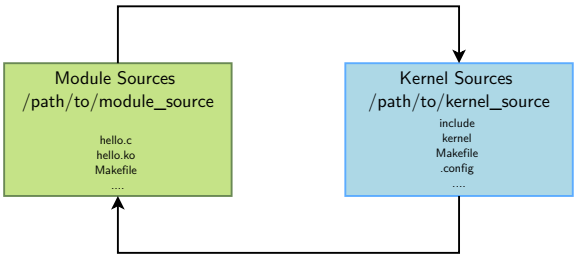
root commit

- The below `Makefile` should be reusable for any single-file out-of-tree Linux module
- The source file is `hello.c`
- Just run `make` to build the `hello.ko` file

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```

- `KDIR`: kernel source or headers directory (see next slides)

Compiling an out-of-tree module 2/2



- The module `Makefile` is interpreted with `KERNELRELEASE` undefined, so it calls the kernel `Makefile`, passing the module directory in the `M` variable
- The kernel `Makefile` knows how to compile a module, and thanks to the `M` variable, knows where the `Makefile` for our module is. This module `Makefile` is then interpreted with `KERNELRELEASE` defined, so the kernel sees the `obj-m` definition.

Modules and kernel version

- To be compiled, a kernel module needs access to *kernel headers*, containing the definitions of functions, types and constants.
- Two solutions
  - Full kernel sources (configured + `make modules_prepare`)
  - Only kernel headers (`linux-headers-*` packages in Debian/Ubuntu distributions, or directory created by `make headers_install`).
- The sources or headers must be configured (`.config` file)
  - Many macros or functions depend on the configuration
- You also need the kernel Makefile, the scripts/ directory, and a few others.
- A kernel module compiled against version X of kernel headers will not load in kernel version Y
  - `modprobe` / `insmod` will say `Invalid module format`

New driver in kernel sources 1/2

root commit

- To add a new driver to the kernel sources:
  - Add your new source file to the appropriate source directory. Example: drivers/usb/serial/navman.c
  - Single file drivers in the common case, even if the file is several thousand lines of code big. Only really big drivers are split in several files or have their own directory.
  - Describe the configuration interface for your new driver by adding the following lines to the `Kconfig` file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

New driver in kernel sources 2/2

root commit

- Add a line in the `Makefile` file based on the `Kconfig` setting:
  `obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o`
- It tells the kernel build system to build `navman.c` when the `USB_SERIAL_NAVMAN` option is enabled. It works both if compiled statically or as a module.
  - Run `make xconfig` and see your new options!
  - Run `make` and your new files are compiled!
  - See `kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

Hello module with parameters 1/2

root commit

```c
// SPDX-License-Identifier: GPL-2.0
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

static char *whom = "world";
module_param(whom, charp, 0644);
MODULE_PARM_DESC(whom, "Recipient of the hello message");

static int howmany = 1;
module_param(howmany, int, 0644);
MODULE_PARM_DESC(howmany, "Number of greetings");
```

Hello module with parameters 2/2

```c
static int __init hello_init(void)
{
    int i;

    for (i = 0; i < howmany; i++)
        pr_alert("(%d) Hello, %s\n", i, whom);
    return 0;
}

static void __exit hello_exit(void)
{
    pr_alert("Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

Thanks to Jonathan Corbet for the examples

Source code available on: https://gitlab.com/rootcommit/training-materials/-/raw/main/code/hello-param/hello_param.c

Declaring a module parameter

```
module_param(
    name, /* name of an already defined variable */
    type, /* standard types (different from C types) are:
           * byte, short, ushort, int, uint, long, ulong
           * charp: a character pointer
           * bool: a bool, values 0/1, y/n, Y/N.
           * invbool: the above, only sense-reversed (N = true). */
    perm  /* for /sys/module/<module_name>/parameters/<param>,
           *  0: no such module parameter value file */
);

/* Example: drivers/block/loop.c */
static int max_loop;
module_param(max_loop, int, 0444);
MODULE_PARM_DESC(max_loop, "Maximum number of loop devices");
```

Lab — Writing modules

root commit

- Create, compile and load your first module
- Add module parameters
- Access kernel internals from your module
- Check code compliance with Linux coding standards

Device model — Bus, drivers and devices

Device model — Bus, drivers and devices

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.

Device model — Bus, drivers and devices

Describing non-discoverable hardware

Describing non-discoverable hardware

root commit

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.

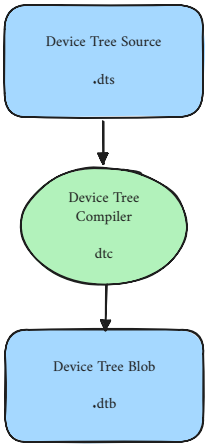
2. Using **ACPI** tables
   - On *x86* systems, but also on a subset of ARM64 platforms
   - Tables provided by the firmware

3. Using a **Device Tree**

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

Device Tree: from source to blob

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



dtc example

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

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

dtc example

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

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

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

dtc example

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

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

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

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

/ {
        welcome = <0xbadcafe>;

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

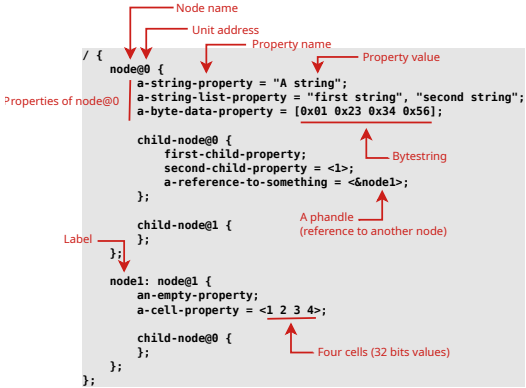
Where are Device Tree Sources located?

root commit

- 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>/`
  - $\approx$ 6,200 Device Tree Source files (`.dts` and `.dtsi`) in Linux as of 7.1.
- Duplicated/synced in various projects
  - U-Boot, Barebox, TF-A

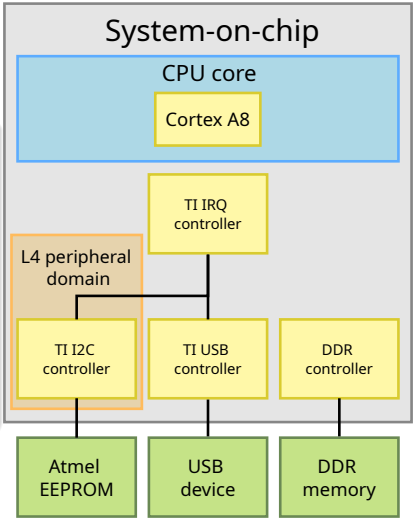
Device Tree base syntax

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



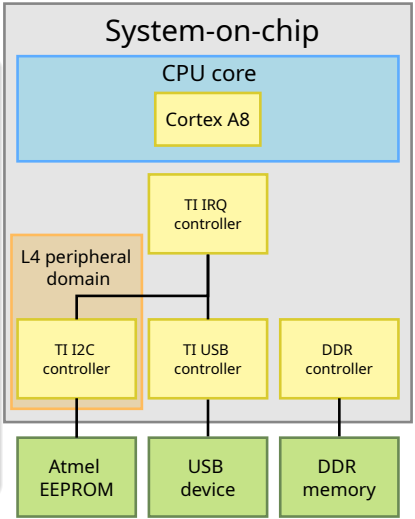
DT overall structure: simplified example

```
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  model = "TI AM335x BeagleBone Black";
  compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";

  cpus { ... };
  memory@80000000 { ... };
  chosen { ... };
  ocp {
    intc: interrupt-controller@48200000 { ... };
    usb0: usb@47401300 { ... };
    l4_per: interconnect@44c00000 {
      i2c0: i2c@40012000 { ... };
    };
  };
};
```



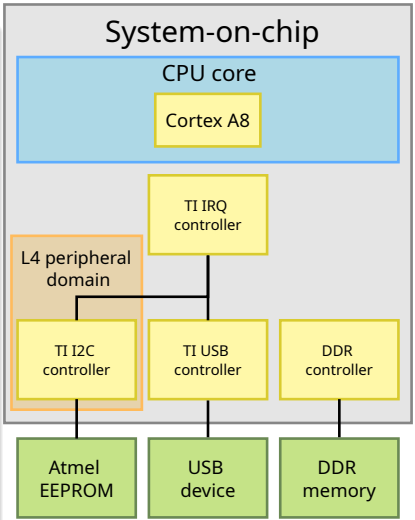
DT overall structure: simplified example

```
/ {
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu0: cpu@0 {
      compatible = "arm,cortex-a8";
      enable-method = "ti,am3352";
      device_type = "cpu";
      reg = <0>;
    };
  };

  memory@0x80000000 {
    device_type = "memory";
    reg = <0x80000000 0x10000000>; /* 256 MB */
  };

  chosen {
    bootargs = "";
    stdout-path = &uart0;
  };

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



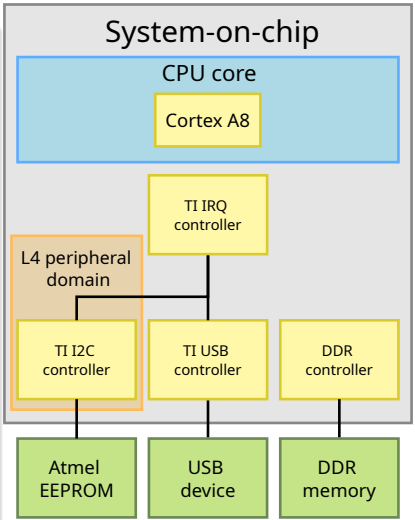
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0x80000000 { ... };
  chosen { ... };
  ocp {

    intc: interrupt-controller@48200000 {
      compatible = "ti,am33xx-intc";
      interrupt-controller;
      #interrupt-cells = <1>;
      reg = <0x48200000 0x1000>;
    };

    usb0: usb@47401300 {
      compatible = "ti,musb-am33xx";
      reg = <0x1400 0x400>, <0x1000 0x200>;
      reg-names = "mc", "control";
      interrupts = <18>;
      dr_mode = "otg";
      dmas = <&cppi41dma  0 0 &cppi41dma  1 0 ...>;
      status = "okay";
    };

    l4_per: interconnect@44c00000 {
      i2c0: i2c@40012000 { ... };
    };
  };
};
```



DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0x80000000 { ... };
  chosen { ... };
  ocp {
    intc: interrupt-controller@48200000 { ... };
    usb0: usb@47401300 { ... };
    l4_per: interconnect@44c00000 {

      i2c0: i2c@40012000 {
        compatible = "ti,omap4-i2c";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x0 0x1000>;
        interrupts = <70>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins>;
        clock-frequency = <400000>;

        baseboard_eeprom: eeprom@50 {
          compatible = "atmel,24c256";
          reg = <0x50>;
        };
      };
    };
  };
};
```



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

root commit

Definition of the AM33xx SoC family

```
&l4_wkup {
    target-module@b000 {
        i2c0: i2c@0 {
            compatible = "ti,omap4-i2c";
            reg = <0x0 0x1000>;
            interrupts = <70>;
            status = "disabled";
        };
    };
};
```

am33xx-l4.dtsi

+

Definition of the Bone Black board

```
#include "am33xx-l4.dtsi"

&i2c0 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c0_pins>;
    status = "okay";

    baseboard_eeprom: eeprom@50 {
        compatible = "atmel,24c256";
        reg = <0x50>;
    };
};
```

am335x-boneblack.dts

Note 1

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

=

Compiled DTB

```
&l4_wkup {
    target-module@b000 {
        i2c0: i2c@0 {
            compatible = "ti,omap4-i2c";
            reg = <0x0 0x1000>;
            interrupts = <70>;
            pinctrl-names = "default";
            pinctrl-0 = <&i2c0_pins>;
            status = "okay";

            baseboard_eeprom: eeprom@50 {
                compatible = "atmel,24c256";
                reg = <0x50>;
            };
        };
    };
};
```

am335x-boneblack.dtb

Note 2

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

Inheritance and labels

Doing:

soc.dtsi

```
/ {
  ocp {
    uart0: serial@0 {
      compatible = "ti,am3352-uart", "ti,omap3-uart";
      reg = <0x0 0x1000>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  ocp {
    serial@0 {
      status = "okay";
    };
  };
};
```

Inheritance and labels

Doing:

soc.dtsi

```
/ {
  ocp {
    uart0: serial@0 {
      compatible = "ti,am3352-uart", "ti,omap3-uart";
      reg = <0x0 0x1000>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  ocp {
    serial@0 {
      status = "okay";
    };
  };
};
```

Is exactly equivalent to:

soc.dtsi

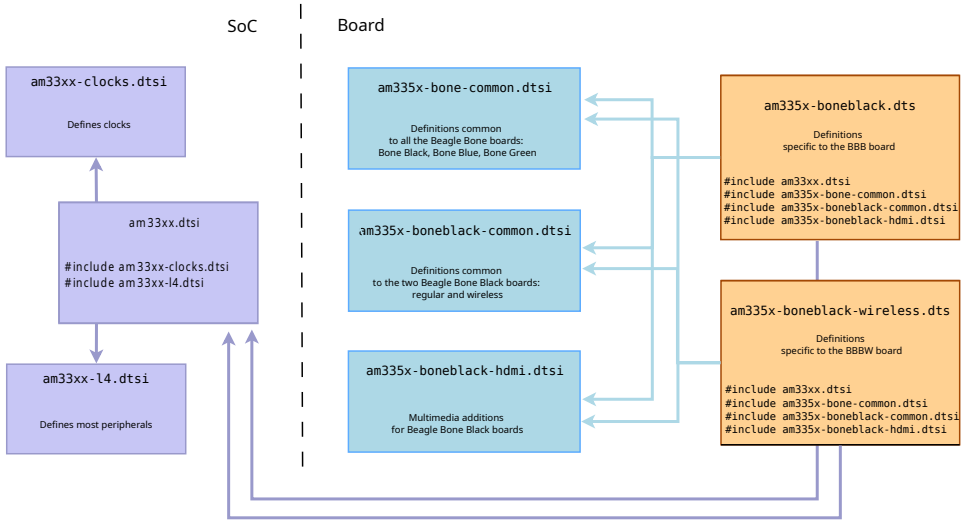
```
/ {
  ocp {
    uart0: serial@0 {
      compatible = "ti,am3352-uart", "ti,omap3-uart";
      reg = <0x0 0x1000>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

&uart0 {
  status = "okay";
};
```

→ this solution is now often preferred

DT inheritance in Bone Black support



Device Tree design principles

root commit

- **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.

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` $\rightarrow$ bus where all sub-nodes are memory-mapped devices

- 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>;
        };
};
```

- 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>;
};
```

`cells` property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
  - Example with a `reg` property using 2 entries of 2 cells:

```
reg = <0x50027000 0x4>, <0x500273f0 0x10>;
```

  - The OS cannot make the difference with:

```
reg = <0x50027000>, <0x4>, <0x500273f0>, <0x10>;
reg = <0x50027000 0x4 0x500273f0>, <0x10>;
reg = <0x50027000>, <0x4 0x500273f0 0x10>;
reg = <0x50027000 0x4 0x500273f0 0x10>;
```

`cells` property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
- Need for other properties to declare the right formatting:
  - `#address-cells`: Indicates the number of cells used to carry the address
  - `#size-cells`: Indicates the number of cells to describe the size of the address range
    0: 1 address, 1: 32 bit address range, 2: 64 bit address range.
- The parent-node declares the children `reg` property formatting
  - Platform devices need memory ranges

```
module@a0000 {
    #address-cells = <1>;
    #size-cells = <1>;

    serial@1000 {
        reg = <0x1000 0x10>, <0x2000 0x10>;
    };
};
```

`cells` property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
- Need for other properties to declare the right formatting:
  - `#address-cells`: Indicates the number of cells used to carry the address
  - `#size-cells`: Indicates the number of cells to describe the size of the address range
    0: 1 address, 1: 32 bit address range, 2: 64 bit address range.
- The parent-node declares the children `reg` property formatting
  - Platform devices need memory ranges
  - SPI devices need chip-selects

```
spi@300000 {
    #address-cells = <1>;
    #size-cells = <0>;

    flash@1 {
        reg = <1>;
    };
};
```

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`

How to validate Device Tree content? 1/2

root commit

- `compatible` properties enforce a specific programming model
- OS expect a specific set of properties in each node
  - The syntax is fixed
  - The content is defined (number of items, their size, their meaning)
  - Some properties are mandatory
- How do I check the validity of a DT snippet?
  - How do I avoid losing half a day on a typo?
  - Looking at drivers to understand the DT structure tends to make it OS-specific

How to validate Device Tree content? 2/2

- **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** → describes how a piece of HW should be described
  - Common bindings are defined in an external repository https://github.com/devicetree-org/dt-schema/tree/main/dtschema/schemas
    - Generic properties: `reg` or `#address-cells`
    - Consumer bindings: `interrupts`, `clocks`, `dmas`, etc
  - Device-specific descriptions are in the Linux kernel sources Documentation/devicetree/bindings/

Devicetree Specification

Release v0.3

devicetree.org

13 February 2020

Device Tree bindings

- Bindings are improved as part of the Linux kernel contribution process
- They are carefully reviewed by DT binding maintainers and can only be merged once approved by them
- Need for automated verifications:
  - Legacy: human readable .txt documents, hardly parsable by tools
  - Current norm: YAML-written specifications, easy to parse by humans and tools at the same time!

Device Tree binding: legacy style

root commit

Documentation/devicetree/bindings/i2c/i2c-omap.txt

```
I2C for OMAP platforms

-Required properties :
- compatible : Must be
       "ti,omap2420-i2c" for OMAP2420 SoCs
       "ti,omap2430-i2c" for OMAP2430 SoCs
       "ti,omap3-i2c" for OMAP3 SoCs
       "ti,omap4-i2c" for OMAP4+ SoCs
       "ti,am654-i2c", "ti,omap4-i2c" for AM654 SoCs
       "ti,j721e-i2c", "ti,omap4-i2c" for J721E SoCs
       "ti,am64-i2c", "ti,omap4-i2c" for AM64 SoCs
- ti,hwmods : Must be "i2c<n>", n being the instance number (1-based)
- #address-cells = <1>;
- #size-cells = <0>;

Recommended properties :
- clock-frequency : Desired I2C bus clock frequency in Hz. Otherwise
  the default 100 kHz frequency will be used.

Optional properties:
- Child nodes conforming to i2c bus binding

Note: Current implementation will fetch base address, irq and dma
from omap hwmod data base during device registration.
Future plan is to migrate hwmod data base contents into device tree
blob so that, all the required data will be used from device tree dts
file.
```

```
Examples :

i2c1: i2c@0 {
    compatible = "ti,omap3-i2c";
    #address-cells = <1>;
    #size-cells = <0>;
    ti,hwmods = "i2c1";
    clock-frequency = <400000>;
};
```

Device Tree binding: YAML style

root commit

Documentation/devicetree/bindings/i2c/ti,omap4-i2c.yaml

```yaml
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
$id: http://devicetree.org/schemas/i2c/ti,omap4-i2c.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: I2C controllers on TI's OMAP and K3 SoCs

maintainers:
  - Vignesh Raghavendra <vigneshr@ti.com>

properties:
  compatible:
    oneOf:
      - enum:
          - ti,omap2420-i2c
          - ti,omap2430-i2c
          - ti,omap3-i2c
          - ti,omap4-i2c
      - items:
          - enum:
              - ti,am4372-i2c
              - ti,am64-i2c
              - ti,am654-i2c
              - ti,j721e-i2c
          - const: ti,omap4-i2c

  reg:
    maxItems: 1

  interrupts:
    maxItems: 1

  clocks:
    maxItems: 1

  clock-names:
    const: fck

  clock-frequency: true

  power-domains: true

  "#address-cells":
    const: 1

  "#size-cells":
    const: 0

  ti,hwmods:
    description:
      Must be "i2c<n>", n being [...]
    $ref: /schemas/types.yaml#/definitions/string
    deprecated: true

required:
  - compatible
  - reg
  - interrupts

additionalProperties: false

if:
  properties:
    compatible:
      enum:
        - ti,omap2420-i2c
        - ti,omap2430-i2c
        - ti,omap3-i2c
        - ti,omap4-i2c
then:
  properties:
    ti,hwmods:
      items:
        - pattern: "^i2c([1-9])$"
else:
  properties:
    ti,hwmods: false

examples:
  - |
    #include <dt-bindings/interrupt-controller/irq.h>
    #include <dt-bindings/interrupt-controller/arm-gic.h>

    main_i2c0: i2c@2000000 {
        compatible = "ti,j721e-i2c", "ti,omap4-i2c";
        reg = <0x2000000 0x100>;
        interrupts = <GIC_SPI 200 IRQ_TYPE_LEVEL_HIGH>;
    };
```

Validating Device Trees

root commit

- `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, particularly useful if you write examples!
  - `make dtbs_check`
    validate DTs currently enabled against YAML bindings
- The combination of DTS and bindings growing, it may sometimes be relevant to only check against a subset of matching schema by adding the `DT_SCHEMA_FILES` specifier on the `make` command line:
  - eg. `make DT_SCHEMA_FILES=Documentation/devicetree/bindings/trivial-devices.yaml dtbs_check`
  - Can be used with both `dt_binding_check` and `dtbs_check`

References

- Kernel documentation
  - driver-api/driver-model/
  - devicetree/
  - filesystems/sysfs
- https://devicetree.org
- The kernel source code
  - Full of Device Tree examples
  - And compatible drivers!

Lab — Describing hardware devices - LEDs

- Browse and update Device Trees.
- Drive GPIOs LEDs from `/sys`
- Modify the Device Tree to make the changes permanent

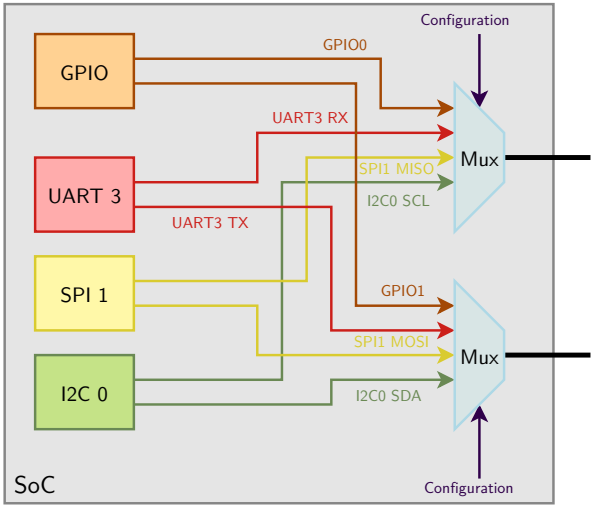
Device model — Bus, drivers and devices

Pin Multiplexing

What is pin muxing?

- Modern SoCs (System on Chip) include more and more hardware blocks, many of which need to interface with the outside world using *pins*.
- However, the physical size of the chips remains small, and therefore the number of available pins is limited.
- For this reason, not all of the internal hardware block features can be exposed on the pins simultaneously.
- The pins are **multiplexed**: they expose either the functionality of hardware block A **or** the functionality of hardware block B.
- This *multiplexing* is usually software configurable.

Pin muxing diagram

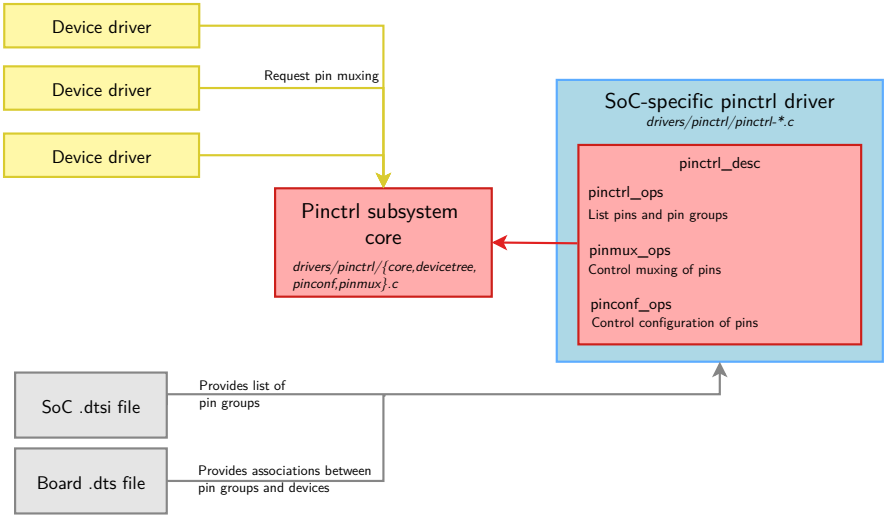


Pin muxing in the Linux kernel

root commit

- Linux has a dedicated `pinctrl` subsystem
- This subsystem, located in drivers/pinctrl/ provides a generic subsystem to handle pin muxing. It offers:
  - A pin muxing driver interface,
    to implement the system-on-chip specific drivers that configure the muxing.
  - A pin muxing consumer interface, for device drivers.
- Most *pinctrl* drivers provide a Device Tree binding,
  and the pin muxing must be described in the Device Tree.
  - The exact Device Tree binding depends on each driver.
    Each binding is defined in `devicetree/bindings/pinctrl`.

pinctrl subsystem diagram



Device Tree properties for consumer devices

The devices that require specific pins to be muxed will use the `pinctrl-<x>` and `pinctrl-names` Device Tree properties.

- The `pinctrl-0`, `pinctrl-1`, `pinctrl-<x>` properties link to a pin configuration for a given state of the device.
- The `pinctrl-names` property associates a name to each state. The name `default` is special, and is automatically selected by a device driver, without having to make an explicit *pinctrl* function call.
- See `devicetree/bindings/pinctrl/pinctrl-bindings.txt` for details.

Device Tree properties for consumer devices - Examples

```
i2c0: i2c@11000 {
        ...
        pinctrl-0 = <&pmx_twsi0>;
        pinctrl-names = "default";
        ...
};
```

Most common case
(arch/arm/boot/dts/marvell/kirkwood.dtsi)

```
i2c0: i2c@f8014000 {
        ...
        pinctrl-names = "default", "gpio";
        pinctrl-0 = <&pinctrl_i2c0>;
        pinctrl-1 = <&pinctrl_i2c0_gpio>;
        ...
};
```

Case with multiple pin states
(arch/arm/boot/dts/microchip/sama5d4.dtsi)

Defining pinctrl configurations

- The different *pinctrl configurations* must be defined as child nodes of the main *pinctrl device* (which controls the muxing of pins).
- The configurations may be defined at:
  - the SoC level (`.dtsi` file)
    for pin configurations that are often shared between multiple boards
  - at the board level (`.dts` file)
    for configurations that are board specific.
- The `pinctrl-<x>` property of the consumer device points to the pin configuration it needs through a DT *phandle*.
- The description of the configurations is specific to each *pinctrl driver*. See `devicetree/bindings/pinctrl` for the pinctrl bindings.

Example on OMAP/AM33xx

- On OMAP/AM33xx, the `pinctrl-single` driver is used. It is common between multiple SoCs and simply allows to configure pins by writing a value to a register.
  - In each pin configuration, a `pinctrl-single,pins` value gives a list of *(register, value)* pairs needed to configure the pins.
- To know the correct values, one must use the SoC and board datasheets.

```
/* Excerpt from am335x-bone-common.dts */

&am33xx_pinmux {
   ...
   i2c2_pins: pinmux_i2c2_pins {
      pinctrl-single,pins = <
         AM33XX_PADCONF(AM335X_PIN_UART1_CTSN, PIN_INPUT_PULLUP, MUX_MODE3)
         /* uart1_ctsn.i2c2_sda */
         AM33XX_PADCONF(AM335X_PIN_UART1_RTSN, PIN_INPUT_PULLUP, MUX_MODE3)
         /* uart1_rtsn.i2c2_scl */
      >;
   };
};

&i2c2 {
   pinctrl-names = "default";
   pinctrl-0 = <&i2c2_pins>;

   status = "okay";
   clock-frequency = <400000>;
   ...

   pressure@76 {
      compatible = "bosch,bmp280";
      reg = <0x76>;
   };
};
```

Example on the Allwinner A20 SoC

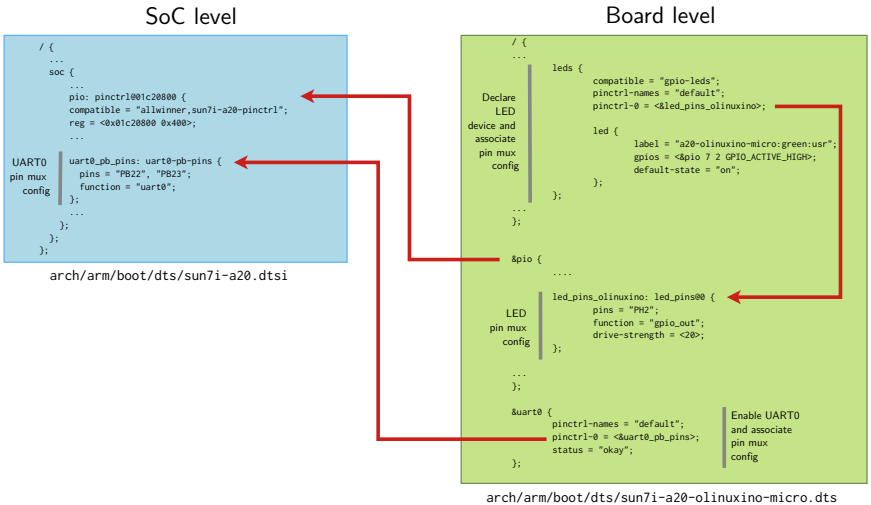
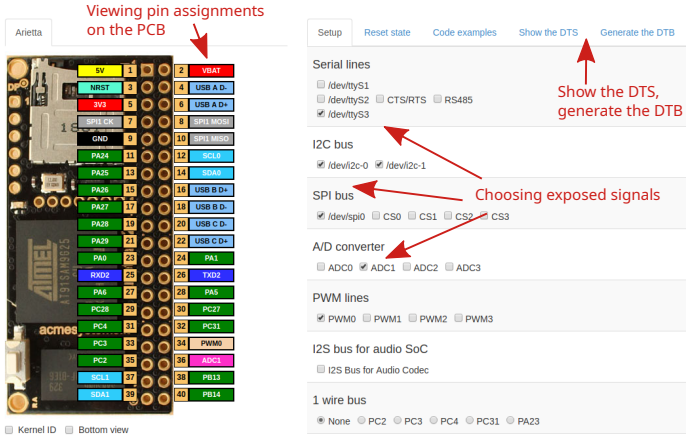


Illustration: live pin muxing configuration



Try ACME Systems' on-line pin-out generator: http://linux.tanzilli.com/

Lab — I2C bus and pin muxing

- Explore I2C buses on the system
- Understand and enable pin muxing for I2C buses
- Detect I2C devices
- Add description of the I2C gamepad to the device tree

Device model — Bus, drivers and devices

Linux device and driver model

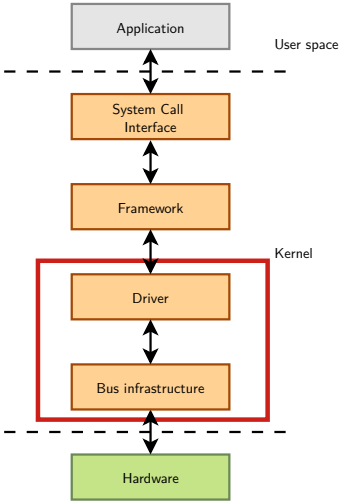
The need for a device model?

- The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to **maximize the reusability** of code between platforms.
- For example, we want the same *USB device driver* to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these platforms are different.
- This requires a clean organization of the code, with the *device drivers* separated from the *controller drivers*, the hardware description separated from the drivers themselves, etc.
- This is what the Linux kernel **Device Model** allows, in addition to other advantages covered in this section.

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features in a generic way.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *bus infrastructure*, while *kernel frameworks* are covered later in this course.



Device model data structures

- The *device model* is organized around three main data structures:
  - The `struct bus_type` structure, which represents one type of bus (USB, PCI, I2C, etc.)
  - The `struct device_driver` structure, which represents one driver capable of handling specific devices on a given bus.
  - The `struct device` structure, which represents one device connected to a bus
- The kernel uses inheritance to create more specialized versions of `struct device_driver` and `struct device` for each bus subsystem.

Bus drivers

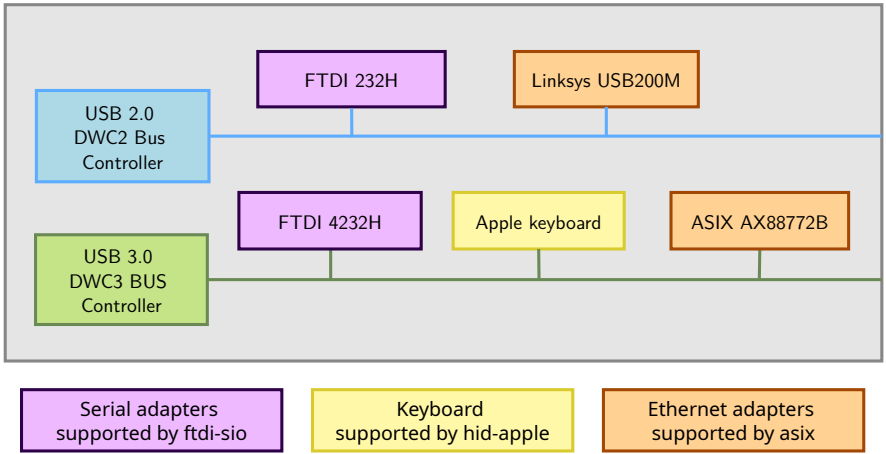
- The first component of the device model is the bus driver
  - One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (`struct bus_type`)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices (if possible), and providing a communication mechanism with the devices
  - Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
  - Matching the device drivers against the devices detected by the adapter drivers.
  - Provides an API to implement both adapter drivers and device drivers
  - Defining driver and device specific structures, eg. `struct usb_driver` and `struct usb_interface`

- The bus, device, drivers, etc. structures are internal to the kernel
- The `sysfs` virtual filesystem offers a mechanism to export such information to user space
- Used for example by `udev` to provide automatic module loading, firmware loading, mounting of external media, etc.
- `sysfs` is usually mounted in `/sys`
  - `/sys/bus/` contains the list of buses
  - `/sys/devices/` contains the list of devices
  - `/sys/class` enumerates devices by the framework they are registered to (`net`, `input`, `block`...), whatever bus they are connected to. Very useful!

Device model — Bus, drivers and devices

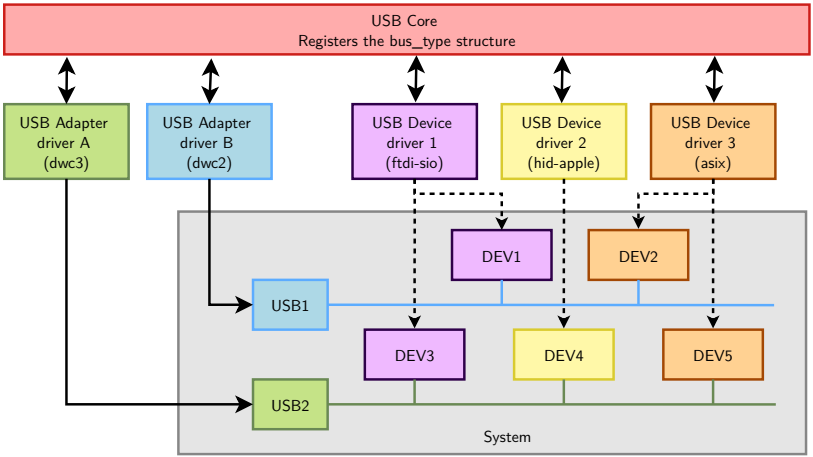
Example of the USB bus

Example: USB bus 1/3



Hardware view of the bus

Example: USB bus 2/3



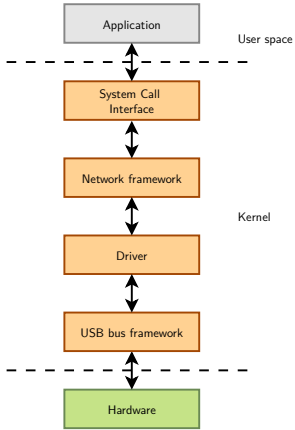
Device model view of the bus

Example: USB bus 3/3

- Core infrastructure (bus driver)
  - drivers/usb/core/
  - `struct bus_type` is defined in drivers/usb/core/driver.c and registered in drivers/usb/core/usb.c
- Adapter drivers
  - drivers/usb/host/
  - For EHCI, UHCI, OHCI, XHCI, and their implementations on various systems (Microchip, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Device drivers
  - Everywhere in the kernel tree, classified by their type (Example: drivers/net/usb/)

Example of device driver

- To illustrate how drivers are implemented to work with the device model, we will study the source code of a driver for a USB network card
  - It is USB device, so it has to be a USB device driver
  - It exposes a network device, so it has to be a network driver
  - Most drivers rely on a bus infrastructure (here, USB) and register themselves in a framework (here, network)
- We will only look at the device driver side, and not the adapter driver side
- The driver we will look at is drivers/net/usb/rtl8150.c



Device identifiers

- Defines the set of devices that this driver can manage,
  so that the USB core knows for which devices this driver should be used
- The `MODULE_DEVICE_TABLE()` macro allows `depmod` (run by `make modules_install`) to extract the relationship between device identifiers and drivers, so that drivers can be loaded automatically by `udev`.
  See `/lib/modules/$(uname -r)/modules.alias,usbmap`

```
static struct usb_device_id rtl8150_table[] = {
    { USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
    { USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
    { USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR) },
    { USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },
    [...]
    {}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instantiation of usb_driver

root commit

- `struct usb_driver` is a structure defined by the USB core. Each USB device driver must instantiate it, and register itself to the USB core using this structure
- This structure inherits from `struct device_driver`, which is defined by the device model.

```
static struct usb_driver rtl8150_driver = {
    .name = "rtl8150",
    .probe = rtl8150_probe,
    .disconnect = rtl8150_disconnect,
    .id_table = rtl8150_table,
    .suspend = rtl8150_suspend,
    .resume = rtl8150_resume
};
```

Driver registration and unregistration

root commit

- When the driver is loaded / unloaded, it must register / unregister itself to / from the USB core
- Done using `usb_register()` and `usb_deregister()`, provided by the USB core.

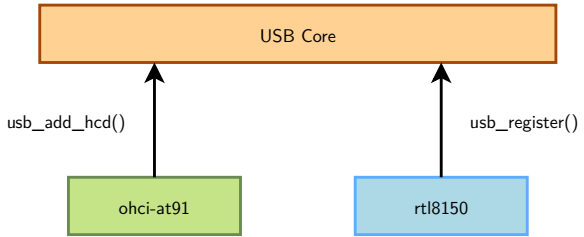
```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

- All this code is actually replaced by a call to the `module_usb_driver()` macro:

```
module_usb_driver(rtl8150_driver);
```

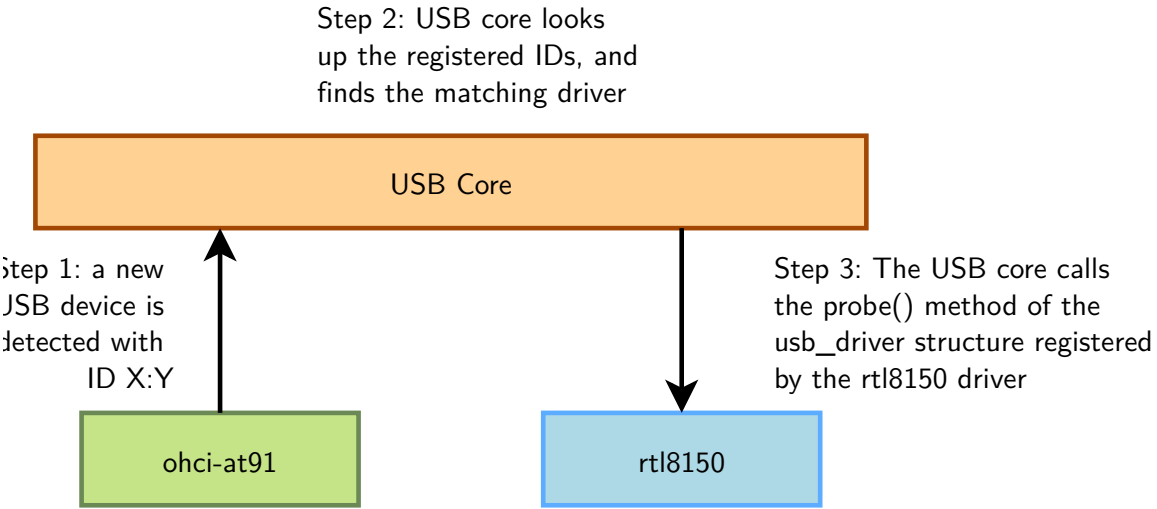
At Initialization

- The USB adapter driver that corresponds to the USB controller of the system registers itself to the USB core
- The `rtl8150` USB device driver registers itself to the USB core



- The USB core now knows the association between the vendor/product IDs of `rtl8150` and the `struct usb_driver` structure of this driver

When a device is detected



Probe method

- Invoked **for each device** bound to a driver
- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`struct pci_dev`, `struct usb_interface`, etc.)
- This function is responsible for
  - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information.
  - Registering the device to the proper kernel framework, for example the network infrastructure.

Example: probe() and disconnect() methods

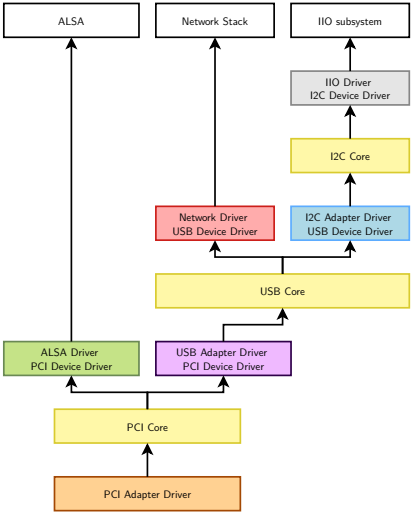
root commit

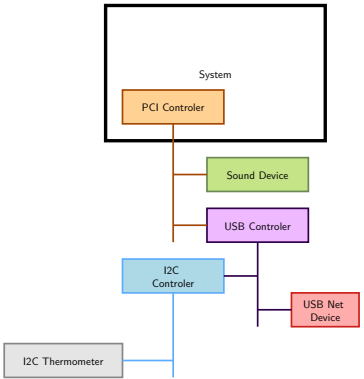
```c
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    [...]
    dev = netdev_priv(netdev);
    tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
    spin_lock_init(&dev->rx_pool_lock);
    [...]
    netdev->netdev_ops = &rtl8150_netdev_ops;
    alloc_all_urbs(dev);
    [...]
    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);
    register_netdev(netdev);

    return 0;
}
```

```c
static void rtl8150_disconnect(struct usb_interface *intf)
{
        rtl8150_t *dev = usb_get_intfdata(intf);

        usb_set_intfdata(intf, NULL);
        if (dev) {
                set_bit(RTL8150_UNPLUG, &dev->flags);
                tasklet_kill(&dev->tl);
                unregister_netdev(dev->netdev);
                unlink_all_urbs(dev);
                free_all_urbs(dev);
                free_skb_pool(dev);
                if (dev->rx_skb)
                        dev_kfree_skb(dev->rx_skb);
                kfree(dev->intr_buff);
                free_netdev(dev->netdev);
        }
}
```

Source: drivers/net/usb/rtl8150.c

The model is recursive





Device model — Bus, drivers and devices

Platform drivers

Platform devices

- Amongst the non-discoverable devices, a huge family are the devices that are directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- In the Linux kernel, a special bus, called the **platform bus** has been created to handle such devices.
- It supports **platform drivers** that handle **platform devices**.
- It works like any other bus (USB, PCI), except that devices are enumerated statically instead of being discovered dynamically.

Implementation of a platform driver (1)

The driver implements a `struct platform_driver` structure (example taken from drivers/tty/serial/imx.c, simplified)

```
static struct platform_driver serial_imx_driver = {
        .probe          = serial_imx_probe,
        .remove         = serial_imx_remove,
        .id_table       = imx_uart_devtype,
        .driver         = {
                .name   = "imx-uart",
                .of_match_table = imx_uart_dt_ids,
                .pm     = &imx_serial_port_pm_ops,
        },
};
```

Implementation of a platform driver (2)

root commit

... and registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void) {
    return platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}

module_init(imx_serial_init);
module_exit(imx_serial_cleanup);
```

Most drivers actually use the `module_platform_driver()` macro when they do nothing special in `init()` and `exit()` functions:

```
module_platform_driver(serial_imx_driver);
```

As platform devices cannot be detected dynamically, they are defined statically

- Legacy way: by direct instantiation of `struct platform_device` structures, as done on a few old ARM platforms. The device was part of a list, and the list of devices was added to the system during board initialization.
- Current way: by parsing an "external" description, like a *device tree* on most embedded platforms today, from which `struct platform_device` structures are created.

Using additional hardware resources

root commit

- Regular DT descriptions contain many information, including phandles (pointers) towards additional hardware blocks or hardware details which cannot be discovered.
  - Some of them are available through a generic array of resources, like addresses for the I/O registers and IRQ lines:
    - Such information can be represented using `struct resource`, and an array of `struct resource` is associated to each `struct platform_device`.
  - Common information/dependencies are parsed by the relevant subsystems, like clocks, GPIOs, or DMA channels:
    - Each subsystem is responsible of instantiating its components, and offering an API to retrieve these objects and use them from device drivers.
  - Specific information might be directly be retrieved by device drivers, through (expensive) direct DT lookups (old drivers use `struct platform_data`).
- All these methods allow the same driver to be used with multiple devices functioning similarly, but with different addresses, IRQs, etc.

Using resources

- The platform driver has access to the resources:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

- As well as the various common dependencies through individual APIs:
  - `clk_get()`
  - `gpio_request()`
  - `dma_request_channel()`

Driver data

- In addition to the per-device resources and information, drivers may require driver-specific information to behave slighlty differently when different flavors of an IP block are driven by the same driver.
- A `const void *data` pointer can be used to store per-compatible specificities:

```
static const struct of_device_id marvell_nfc_of_ids[] = {
        {
                .compatible = "marvell,armada-8k-nand-controller",
                .data = &marvell_armada_8k_nfc_caps,
        },
};
```

- Which can be retrieved in the probe with:

```
        /* Get NAND controller capabilities */
        if (pdev->id_entry) /* legacy way */
                nfc->caps = (void *)pdev->id_entry->driver_data;
        else /* current way */
                nfc->caps = of_device_get_match_data(&pdev->dev);
```

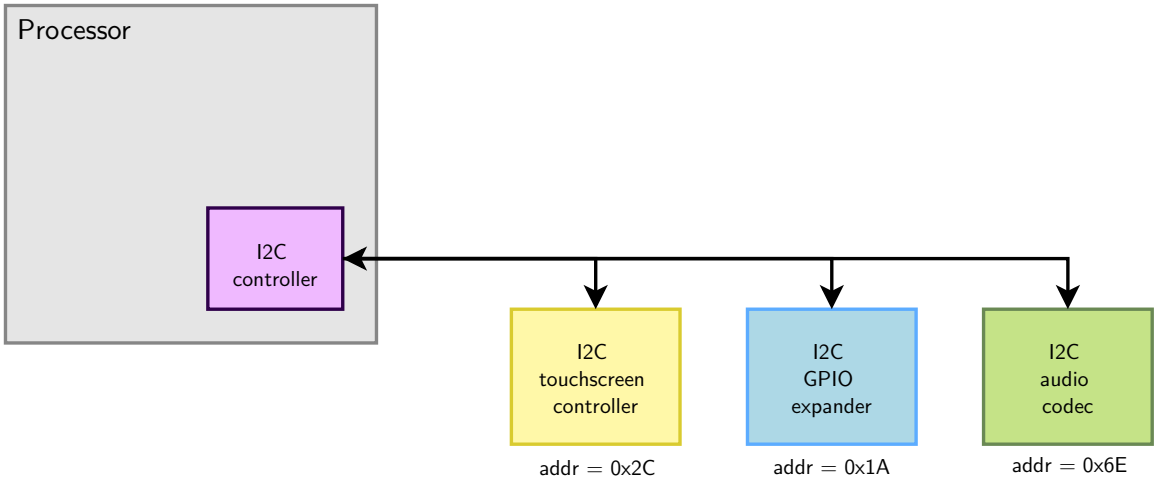
Device model — Bus, drivers and devices

The I2C subsystem

What is I2C?

- A very commonly used low-speed bus to connect on-board and external devices to the processor.
- Uses only two wires: SDA for the data, SCL for the clock.
- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- Each slave device is identified by an I2C address (you can't have 2 devices with the same address on the same bus). Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.

An I2C bus example



The I2C bus driver

- Like all bus subsystems, the I2C bus driver is responsible for:
  - Providing an API to implement I2C controller drivers
  - Providing an API to implement I2C device drivers, in kernel space
  - Providing an API to implement I2C device drivers, in user space
- The core of the I2C bus driver is located in drivers/i2c/.
- The I2C controller drivers are located in drivers/i2c/busses/.
- The I2C device drivers are located throughout drivers/, depending on the framework used to expose the devices (e.g. drivers/input/ for input devices).

Registering an I2C device driver

root commit

- Like all bus subsystems, the I2C subsystem defines a `struct i2c_driver` that inherits from `struct device_driver`, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the `->probe()` and `->remove()` functions.
  - It also contains a legacy `id_table`, used for non-DT based probing of I2C devices.
- The `i2c_add_driver()` and `i2c_del_driver()` functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its `init()`/`exit()` functions, it is advised to use the `module_i2c_driver()` macro instead.

Registering an I2C device driver: example

```
static const struct i2c_device_id adxl345_i2c_id[] = {
        { "adxl345", ADXL345 },
        { "adxl375", ADXL375 },
        { }
};

MODULE_DEVICE_TABLE(i2c, adxl345_i2c_id);

static const struct of_device_id adxl345_of_match[] = {
        { .compatible = "adi,adxl345" },
        { .compatible = "adi,adxl375" },
        { },
};

MODULE_DEVICE_TABLE(of, adxl345_of_match);

static struct i2c_driver adxl345_i2c_driver = {
        .driver = {
                .name   = "adxl345_i2c",
                .of_match_table = adxl345_of_match,
        },
        .probe          = adxl345_i2c_probe,
        .remove         = adxl345_i2c_remove,
        .id_table       = adxl345_i2c_id,
};

module_i2c_driver(adxl345_i2c_driver);
```

From drivers/iio/accel/adxl345_i2c.c

Registering an I2C device, in the DT

root commit

- In the Device Tree, the I2C controller device is typically defined in the `.dtsi` file that describes the processor.
  - Normally defined with `status = "disabled"`.
- At the board/platform level:
  - the I2C controller device is enabled (`status = "okay"`)
  - the I2C bus frequency is defined, using the `clock-frequency` property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the `reg` property gives the I2C slave address on the bus.
- See the binding for the corresponding driver for a specification of the expected DT properties. Example: `devicetree/bindings/i2c/i2c-omap.txt`

Registering an I2C device, DT example (1/2)

root commit

Definition of the I2C controller

```
i2c0: i2c@01c2ac00 {
        compatible = "allwinner,sun7i-a20-i2c",
                     "allwinner,sun4i-a10-i2c";
        reg = <0x01c2ac00 0x400>;
        interrupts = <GIC_SPI 7 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&apb1_gates 0>;
        status = "disabled";
        #address-cells = <1>;
        #size-cells = <0>;
};
```

From arch/arm/boot/dts/allwinner/sun7i-a20.dtsi

Registering an I2C device, DT example (2/2)

root commit

Definition of the I2C device

```
&i2c0 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins_a>;
        status = "okay";

        axp209: pmic@34 {
                compatible = "x-powers,axp209";
                reg = <0x34>;
                interrupt-parent = <&nmi_intc>;
                interrupts = <0 IRQ_TYPE_LEVEL_LOW>;

                interrupt-controller;
                #interrupt-cells = <1>;
        };
};
```

From arch/arm/boot/dts/allwinner/sun7i-a20-olinuxino-micro.dts

- The ->probe() function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - An `struct i2c_client` pointer, which represents the I2C device itself. This structure inherits from `struct device`.
- The ->remove() function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:
  - The same `struct i2c_client` pointer as received by probe()

remove() may be left undefined if everything is automatically unregistered (explained later).

Probe example (drivers/iio/pressure/mpl3115.c)

root commit

With added comments to explain each section!

```
static int mpl3115_probe(struct i2c_client *client)
{
        const struct i2c_device_id *id = i2c_client_get_device_id(client);
        struct mpl3115_data *data;
        struct iio_dev *indio_dev;
        int ret;

        /* Device check */
        ret = i2c_smbus_read_byte_data(client, MPL3115_WHO_AM_I);
        if (ret < 0)
                return ret;
        if (ret != MPL3115_DEVICE_ID)
                return -ENODEV;

        /* Allocate and initialize framework structure */
        indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
        if (!indio_dev)
                return -ENOMEM;

        /* Access per device structure */
        data = iio_priv(indio_dev);

        /* Store reference to physical device structure */
        data->client = client;

        /* Initialize per device lock */
        mutex_init(&data->lock);

        /* Store reference to framework structure */
        i2c_set_clientdata(client, indio_dev);

        /* Initialize framework structure */
        indio_dev->info = &mpl3115_info;
        indio_dev->name = id->name;
        indio_dev->modes = INDIO_DIRECT_MODE;
        indio_dev->channels = mpl3115_channels;
        indio_dev->num_channels = ARRAY_SIZE(mpl3115_channels);

        /* Device initialization */
        /* software reset, I2C transfer is aborted (fails) */
        i2c_smbus_write_byte_data(client, MPL3115_CTRL_REG1,
                MPL3115_CTRL_RESET);
        msleep(50);

        data->ctrl_reg1 = MPL3115_CTRL_OS_258MS;
        ret = i2c_smbus_write_byte_data(client, MPL3115_CTRL_REG1,
                data->ctrl_reg1);
        if (ret < 0)
                return ret;

        /* Further framework device preparation work */
        ret = iio_triggered_buffer_setup(indio_dev, NULL,
                mpl3115_trigger_handler, NULL);
        if (ret < 0)
                return ret;

        /* Everything is ready, register framework device */
        ret = iio_device_register(indio_dev);
        if (ret < 0)
                goto buffer_cleanup;
        return 0;

        /* Failure, cleanup before returning */
buffer_cleanup:
        iio_triggered_buffer_cleanup(indio_dev);
        return ret;
}
```

Remove example (drivers/iio/pressure/mpl3115.c)

root commit

```
static void mpl3115_remove(struct i2c_client *client)
{
        /* Get reference to per device structure */
        struct iio_dev *indio_dev = i2c_get_clientdata(client);

        /* Unregister framework structure */
        iio_device_unregister(indio_dev);

        /* Clean up resources */
        iio_triggered_buffer_cleanup(indio_dev);

        /* Standby device */
        mpl3115_standby(iio_priv(indio_dev));
}
```

Lab — Basic I2C driver

- Create a basic I2C driver for the gamepad device
- Make sure that the driver is bound to our device

Communicating with the I2C device: raw API

The most **basic API** to communicate with the I2C device provides functions to either send or receive data:

- Send a `buf` to the I2C device with:

```
int i2c_master_send(const struct i2c_client *client, const char *buf, int count);
```

- Receive a `count` bytes from the I2C device and save them in `buf` with:

```
int i2c_master_recv(const struct i2c_client *client, char *buf, int count);
```

Both functions return a negative error number in case of failure, otherwise the number of transmitted bytes.

Communicating with the I2C device: message transfer

The message transfer API allows to describe **transfers** that consists of several **messages**, with each message being a transaction in one direction:

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
```

- The `struct i2c_adapter` pointer can be found by using `client->adapter`
- The `struct i2c_msg` structure defines the length, location, and direction of the message.

I2C: message transfer example

root commit

```
static int st1232_ts_read_data(struct st1232_ts_data *ts)
{
        ...
        struct i2c_client *client = ts->client;
        struct i2c_msg msg[2];
        int error;
        ...
        u8 start_reg = ts->chip_info->start_reg;
        u8 *buf = ts->read_buf;

        /* read touchscreen data */
        msg[0].addr = client->addr;
        msg[0].flags = 0;
        msg[0].len = 1;
        msg[0].buf = &start_reg;

        msg[1].addr = ts->client->addr;
        msg[1].flags = I2C_M_RD;
        msg[1].len = ts->read_buf_len;
        msg[1].buf = buf;

        error = i2c_transfer(client->adapter, msg, 2);
        ...
}
```

From drivers/input/touchscreen/st1232.c

References

- https://en.wikipedia.org/wiki/I2C, general presentation of the I2C protocol
- i2c/, details about Linux support for I2C
  - i2c/writing-clients
    How to write I2C kernel device drivers
  - i2c/dev-interface
    How to write I2C user-space device drivers
  - i2c/instantiating-devices
    How to instantiate devices
- See also Luca Ceresoli's introduction to I2C (slides, video).

Lab — I2C communication

Expand the `probe()` function by:

- Using raw I2C commands to initialize the device
- Using raw I2C commands to access the states of the joystick and its buttons.

Kernel frameworks, memory and I/O

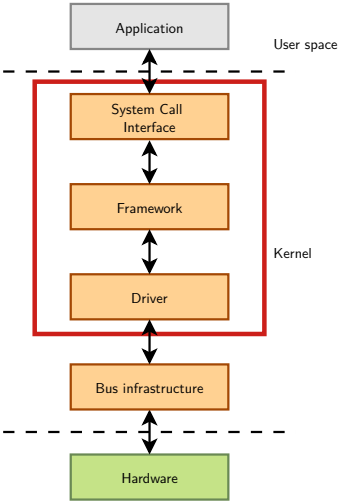
Kernel frameworks, memory and I/O

Introduction

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features to user space applications.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *kernel frameworks*, while the *bus infrastructure* was covered earlier in this course.



Kernel frameworks, memory and I/O

User space vision of devices

Types of devices

Under Linux, there are essentially four types of devices:

- **Network devices**. They are represented as network interfaces, visible in user space using `ip a`
- **Block devices**. They are used to provide user space applications access to raw storage devices (hard disks, USB keys). They are visible to the applications as *device files* in `/dev`.
- **Character devices**. They are used to provide user space applications access to all other types of devices (input, sound, graphics, serial, etc.). They are also visible to the applications as *device files* in `/dev`.
- **Sysfs devices**. They don't have any of the above user space interfaces, only a representation in sysfs. "Internal" device drivers fall under this (e.g. pinctrl), but also some user-space accessible devices. E.g. gpio (deprecated), IIO (Industrial I/O).

→ Most devices are *character devices*, so we will study these in more details.

Major and minor numbers

root commit

- Within the kernel, all block and character devices are identified using a *major* and a *minor* number.
- The *major number* typically indicates the family of the device.
- The *minor number* allows drivers to distinguish the various devices they manage.
- Some major numbers are statically allocated, and identical across all Linux systems.
- Since approximately 2016, new frameworks use dynamically allocated major numbers when possible.
- Minor numbers are almost always (partially) dynamically allocated by the framework itself. Allocation happens in order of enumeration of the devices.
- Pre-defined values, ranges and rules can be found in `admin-guide/devices`.

Devices: everything is a file

root commit

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

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda /dev/sda1 /dev/sda2 /dev/sdc1 /dev/zero
brw-rw---- 1 root disk    8,  0 2011-05-27 08:56 /dev/sda
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
brw-rw---- 1 root disk    8, 32 2011-05-27 08:56 /dev/sdc
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a serial port

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

Creating device files

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

A character driver in the kernel

root commit

- From the point of view of an application, a *character device* is essentially a **file**.
- Character device drivers therefore implement **operations** that let applications think the device is a file.
- In order to achieve this, a character driver implements the operations it wants from the `struct file_operations` structure: `read`, `write`, `ioctl`, etc.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

Implementation details are detailed in a later section.

Kernel frameworks, memory and I/O

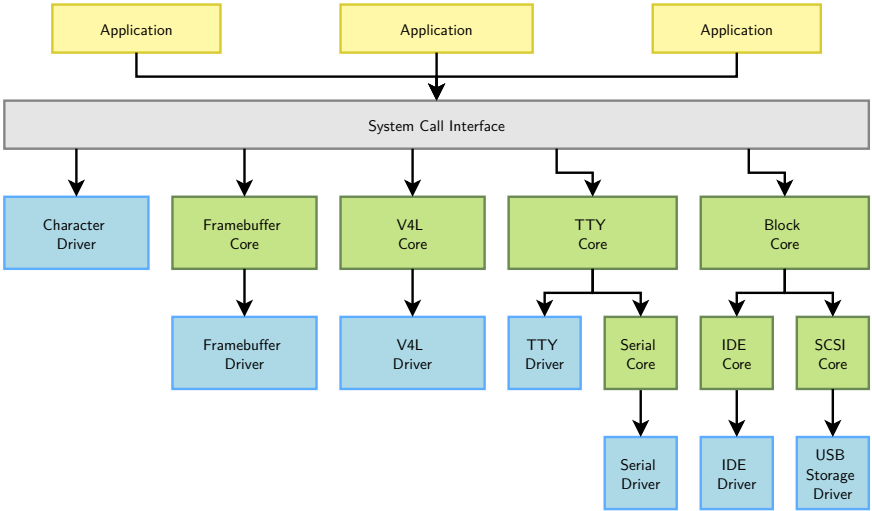
The concept of kernel frameworks

Beyond character drivers: kernel frameworks

root commit

- Many device drivers are not implemented directly as character drivers
- They are implemented under a *framework*, specific to a given device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From user space, they are still seen as character devices by the applications
  - The framework allows to provide a coherent user space interface (`ioctl`, etc.) for every type of device, regardless of the driver

Example: Some Kernel Frameworks



Example: Framebuffer Framework

root commit

- Kernel option `CONFIG_FB`
  - `menuconfig FB`
    - `tristate "Support for frame buffer devices"`
- Implemented in C files in drivers/video/fbdev/core/
- Defines the user/kernel API
  - include/uapi/linux/fb.h (constants and structures)
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
  - `struct fb_ops`
  - include/linux/fb.h

Framebuffer driver operations

root commit

Here are the operations a framebuffer driver can or must implement, and define them in a `struct fb_ops` structure (excerpt from drivers/video/fbdev/skeletonfb.c)

```
static struct fb_ops xxxfb_ops = {
    .owner = THIS_MODULE,
    .fb_open = xxxfb_open,
    .fb_read = xxxfb_read,
    .fb_write = xxxfb_write,
    .fb_release = xxxfb_release,
    .fb_check_var = xxxfb_check_var,
    .fb_set_par = xxxfb_set_par,
    .fb_setcolreg = xxxfb_setcolreg,
    .fb_blank = xxxfb_blank,
    .fb_pan_display = xxxfb_pan_display,
    .fb_fillrect = xxxfb_fillrect,          /* Needed !!! */
    .fb_copyarea = xxxfb_copyarea,          /* Needed !!! */
    .fb_imageblit = xxxfb_imageblit,        /* Needed !!! */
    .fb_cursor = xxxfb_cursor,              /* Optional !!! */
    .fb_rotate = xxxfb_rotate,
    .fb_sync = xxxfb_sync,
    .fb_ioctl = xxxfb_ioctl,
    .fb_mmap = xxxfb_mmap,
};
```

Framebuffer driver code

- In the `probe()` function, registration of the framebuffer device and operations

```
static int xxxfb_probe (struct pci_dev *dev, const struct pci_device_id *ent)
{
    struct fb_info *info;
    [...]
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    [...]
    info->fbops = &xxxfb_ops;
    [...]
    if (register_framebuffer(info) < 0)
        return -EINVAL;
    [...]
}
```

- `register_framebuffer()` will create a new character device in *devtmpfs* that can be used by user space applications with the generic framebuffer API.

Kernel frameworks, memory and I/O

Device-managed allocations

Device managed allocations

- The `probe()` function is typically responsible for allocating a significant number of resources: memory, mapping I/O registers, registering interrupt handlers, etc.
- These resource allocations have to be properly freed:
  - In the `probe()` function, in case of failure
  - In the `remove()` function
- This required a lot of failure handling code that was rarely tested
- To solve this problem, *device managed* allocations have been introduced.
- The idea is to associate resource allocation with the `struct device`, and automatically release those resources
  - When the device disappears
  - When the device is unbound from the driver
- Functions prefixed by `devm_`
- See `driver-api/driver-model/devres` for details

Memory allocation example

root commit

- Normally done with `kmalloc(size_t, gfp_t)`, released with `kfree(void *)`
- Device managed with `devm_kmalloc(struct device *, size_t, gfp_t)`

Without devm functions

```
int foo_probe(struct platform_device *pdev)
{
        struct foo_t *foo = kmalloc(sizeof(struct foo_t),
                                    GFP_KERNEL);
        /* Register to framework, store
         * reference to framework structure in foo */
        ...
        if (failure) {
                kfree(foo);
                return -EBUSY;
        }
        platform_set_drvdata(pdev, foo);
        return 0;
}

void foo_remove(struct platform_device *pdev)
{
        struct foo_t *foo = platform_get_drvdata(pdev);
        /* Retrieve framework structure from foo
           and unregister it */
        ...
        kfree(foo);
}
```

With devm functions

```
int foo_probe(struct platform_device *pdev)
{
        struct foo_t *foo = devm_kmalloc(&pdev->dev,
                                         sizeof(struct foo_t),
                                         GFP_KERNEL);
        /* Register to framework, store
         * reference to framework structure in foo */
        ...
        if (failure)
                return -EBUSY;
        platform_set_drvdata(pdev, foo);
        return 0;
}

void foo_remove(struct platform_device *pdev)
{
        struct foo_t *foo = platform_get_drvdata(pdev);
        /* Retrieve framework structure from foo
           and unregister it */
        ...
        /* foo automatically freed */
}
```

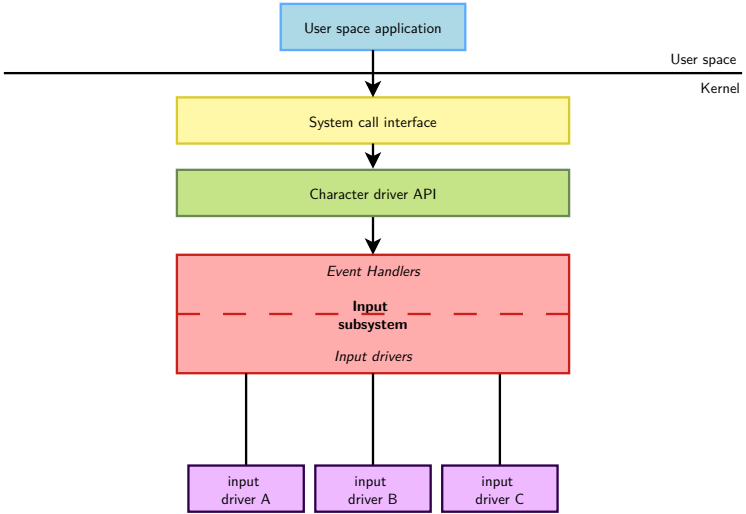
Kernel frameworks, memory and I/O

The input subsystem

What is the input subsystem?

- The input subsystem takes care of all the input events coming from the human user.
- Initially written to support the USB *HID* (Human Interface Device) devices, it quickly grew up to handle all kinds of inputs (using USB or not): keyboards, mice, joysticks, touchscreens, etc.
- The input subsystem is split in two parts:
  - **Device drivers**: they talk to the hardware (for example via USB), and provide events (keystrokes, mouse movements, touchscreen coordinates) to the input core
  - **Event handlers**: they get events from drivers and pass them where needed via various interfaces (most of the time through `evdev`)
- In user space it is usually used by the graphic stack such as *X.Org*, *Wayland* or *Android's InputManager*.

Input subsystem diagram



Input subsystem overview

- Kernel option `CONFIG_INPUT`
  - `menuconfig INPUT`
    - `tristate "Generic input layer (needed for keyboard, mouse, ...)"`
- Implemented in drivers/input/
  - `input.c`, `input-polldev.c`, `evdev.c`...
- Defines the user/kernel API
  - include/uapi/linux/input.h
- Defines the set of operations an input driver must implement and helper functions for the drivers
  - `struct input_dev` for the device driver part
  - `struct input_handler` for the event handler part
  - include/linux/input.h

Input subsystem API 1/3

root commit

An *input device* is described by a very long `struct input_dev` structure, an excerpt is:

```
struct input_dev {
    const char *name;
    [...]
    struct input_id id;
    [...]
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    [...]
    int (*getkeycode)(struct input_dev *dev,
                      struct input_keymap_entry *ke);
    [...]
    int (*open)(struct input_dev *dev);
    [...]
    int (*event)(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value);
    [...]
};
```

Before being used, this structure must be allocated and initialized, typically with: `struct input_dev *devm_input_allocate_device(struct device *dev);`

Input subsystem API 2/3

- Depending on the type of events that will be generated, the input bit fields `evbit` and `keybit` must be configured: For example, for a button we only generate `EV_KEY` type events, and from these only `BTN_0` events code:

```
set_bit(EV_KEY, myinput_dev.evbit);
set_bit(BTN_0, myinput_dev.keybit);
```

- Once the *input device* is allocated and filled, the function to register it is: `int input_register_device(struct input_dev *);`

Input subsystem API 3/3

The states of input devices are sent by the driver to the event handler using

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)
```

- The event handler generates an event **only when a state changes**
- The event types are documented in `input/event-codes`
- The input subsystem provides simpler wrappers to report specific states, such as:
  - `input_report_key()`
  - `input_report_abs()`

After submitting potentially multiple states, the *input* core must be notified by calling:

```
void input_sync(struct input_dev *dev)
```

Example from drivers/hid/usbhid/usbmouse.c

root commit

```
static void usb_mouse_irq(struct urb *urb)
{
        struct usb_mouse *mouse = urb->context;
        signed char *data = mouse->data;
        struct input_dev *dev = mouse->dev;
        ...

        input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
        input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
        input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
        input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
        input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);

        input_report_rel(dev, REL_X,     data[1]);
        input_report_rel(dev, REL_Y,     data[2]);
        input_report_rel(dev, REL_WHEEL, data[3]);

        input_sync(dev);
        ...
}
```

Polling input devices

- The input subsystem provides an API to support simple input devices that *do not raise interrupts* but have to be *periodically scanned or polled* to detect changes in their state.
- Setting up polling is done using `input_setup_polling()`:

```
int input_setup_polling(struct input_dev *dev, void (*poll_fn)(struct input_dev *dev));
```

- `poll_fn` is the function that will be called periodically.
- The polling interval can be set using `input_set_poll_interval()` or `input_set_min_poll_interval()` and `input_set_max_poll_interval()`

- The main user space interface to *input devices* is the **event interface**
- Each *input device* is represented as a `/dev/input/event<X>` character device
- A user space application can use blocking and non-blocking reads, but also `select()` (to get notified of events) after opening this device.
- Each read will return `struct input_event` structures of the following format:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

- A very useful application for *input device* testing is `evtest`, from `https://cgit.freedesktop.org/evtest/`

Polled input drivers: summary of steps

- Make sure your `probe()` and `remove()` routines are called.
- In the `probe()` routine, initialize your device.
  Also add temporary code to make sure you can access device data.
- Allocate and initialize a `struct input_dev` structure:

```
struct input_dev *input;
input = devm_input_allocate_device(&client->dev);
```

- Implement a polling function reporting the state of your device:

```
static void gamepad_poll(struct input_dev *input)
```

- Fill `input` structure fields
- Attach the polling function to the input structure:

```
ret = input_setup_polling(input, gamepad_poll);
```

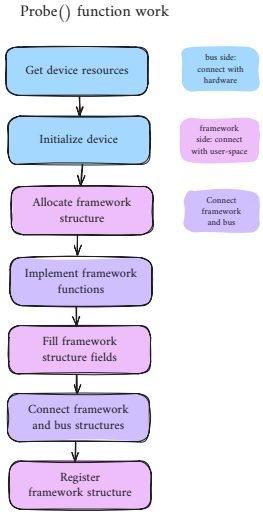
- When everything is ready, register the input structure:

```
ret = input_register_device(input);
```

- Empty `remove()`: the input structure is automatically unregistered and freed.
- Load and test your driver:

```
sudo evtest
```

Good example in kernel sources: drivers/input/mouse/gpio_mouse.c



Lab — Expose the gamepad to user space

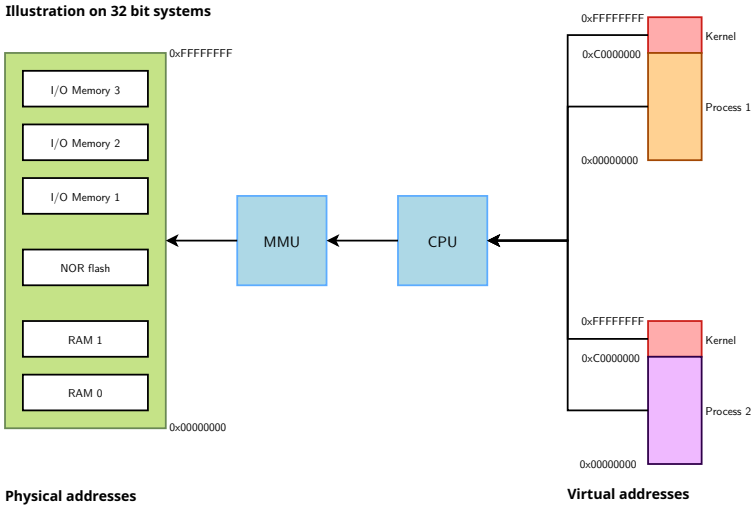
root commit

- Extend the gamepad driver to expose device data to user space applications, as an *input* device.
- Test the operation of the gamepad using `evtest`
- Play games with the gamepad!

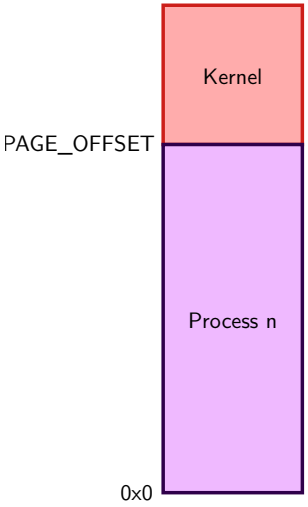
Kernel frameworks, memory and I/O

Memory Management

Physical and virtual memory

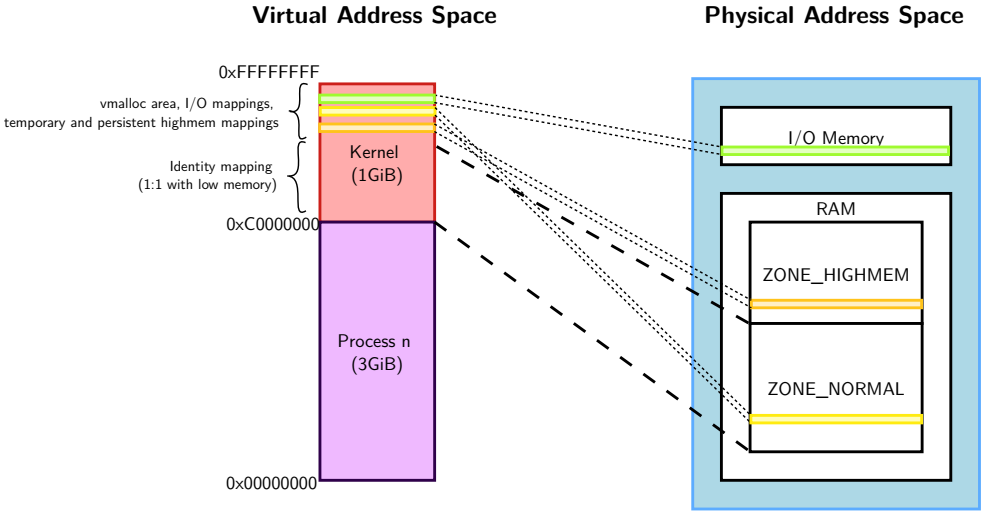


Virtual memory organization



- The top quarter reserved for kernel-space
  - Contains kernel code and core data structures
  - Allocations for loading modules
  - All kernel physical mappings
  - Identical in all address spaces
- The lower part is a per user process exclusive mapping
  - Process code and data (program, stack, ...)
  - Memory-mapped files
  - Each process has its own address space!
- The exact virtual mapping in-use is displayed in the kernel log early at boot time

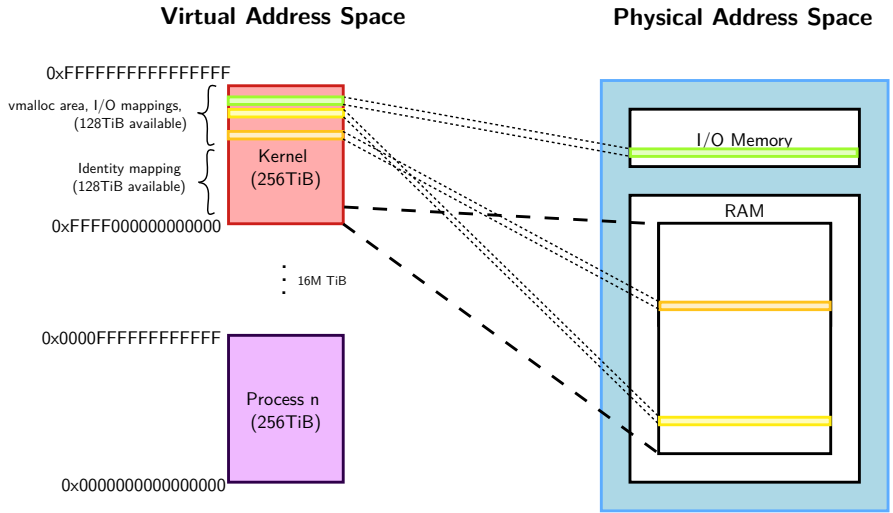
Physical/virtual memory mapping on 32-bit systems



32-bit systems limitations

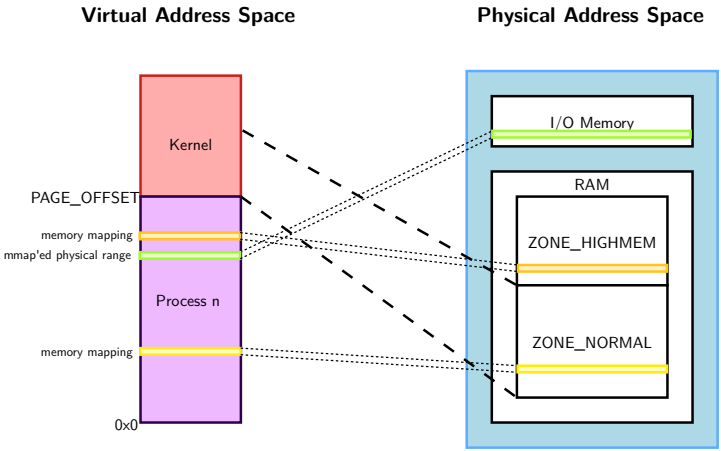
- Only less than 1GB memory addressable directly through kernel virtual addresses
- If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by user space
- To allow the kernel to access more physical memory:
  - Change the 3GB/1GB memory split to 2GB/2GB or 1GB/3GB (`CONFIG_VMSPLIT_2G` or `CONFIG_VMSPLIT_1G`) $\Rightarrow$ reduce total user memory available for each process
  - Activate *highmem* support if available for your architecture:
    - Allows kernel to map parts of its non-directly accessible memory
    - Mapping must be requested explicitly
    - Limited addresses ranges reserved for this usage
- See Arnd Bergmann's *4GB by 4GB split* presentation (video and slides) at Linaro Connect virtual 2020.

Physical/virtual memory mapping on 64-bit systems



User space virtual address space

- When a process starts, the executable code is loaded in RAM and mapped into the process virtual address space.
- During execution, additional mappings can be created:
  - Memory allocations
  - Memory mapped files
  - `mmap`'ed areas
  - ...

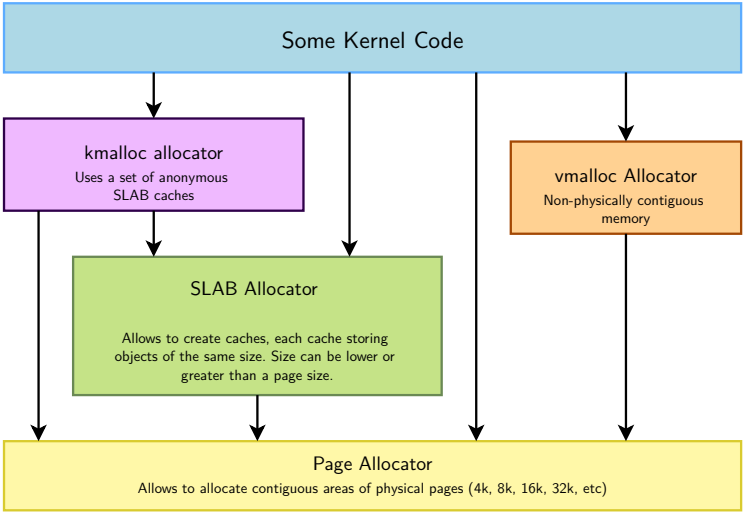


Userspace memory allocations

root commit

- Userspace mappings can target the full memory
- When allocated, memory may not be physically allocated:
  - Kernel uses demand fault paging to allocate the physical page (the physical page is allocated when access to the virtual address generates a page fault)
  - ... or may have been swapped out, which also induces a page fault
    - See the `mlock`/`mlockall` system calls for workarounds
- User space memory allocation is allowed to over-commit memory (more than available physical memory) $\Rightarrow$ can lead to out of memory situations.
  - Can be prevented with the use of `/proc/sys/vm/overcommit_*`
- OOM killer kicks in and selects a process to kill to retrieve some memory. That's better than letting the system freeze.

Kernel memory allocators



Page allocator

- Appropriate for medium-size allocations
- A page is usually 4K, but can be made greater in some architectures (sh, mips: 4, 8, 16 or 64 KB, but not configurable in x86 or arm).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is contiguous in the kernel virtual address space, but also maps to physically contiguous pages. It is allocated in the identity-mapped part of the kernel memory space.
  - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.
  - The *Contiguous Memory Allocator* (`CMA`) can be used to reserve a given amount of memory at boot (see `https://lwn.net/Articles/486301/`).

Page allocator API

- `unsigned long get_zeroed_page(gfp_t gfp_mask)`
  - Returns the virtual address of a free page, initialized to zero
  - `gfp_mask`: see the next pages for details.
- `unsigned long __get_free_page(gfp_t gfp_mask)`
  - Same, but doesn't initialize the contents
- `unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)`
  - Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being `log2(number_of_pages)`.Can be computed from the size with the `get_order()` function.
- `void free_page(unsigned long addr)`
  - Frees one page.
- `void free_pages(unsigned long addr, unsigned int order)`
  - Frees multiple pages. Need to use the same order as in allocation.

Page allocator flags

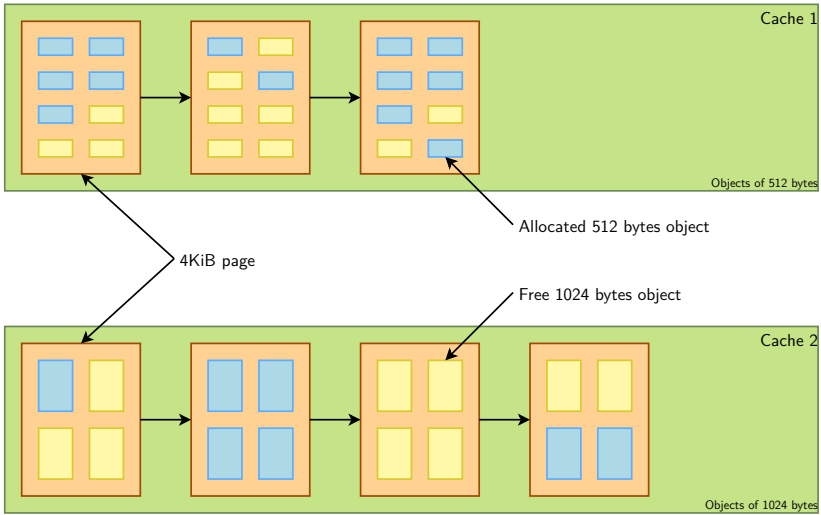
The most common ones are:

- `GFP_KERNEL`
  - Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.
- `GFP_ATOMIC`
  - RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.
- Others are defined in include/linux/gfp_types.h.
  See also the documentation in `core-api/memory-allocation`

SLAB allocator 1/2

- The SLAB allocator allows to create *caches*, which contain a set of objects of the same size. In English, *slab* means *tile*.
- The object size can be smaller or greater than the page size
- The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects. It uses the page allocator to allocate and free pages.
- SLAB caches are used for data structures that are present in many instances in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.
  - See `/proc/slabinfo`
- They are rarely used for individual drivers.
- See include/linux/slab.h for the API

SLAB allocator 2/2



kmalloc allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches, named `kmalloc-XXX` in `/proc/slabinfo`
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be physically contiguous
- The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, etc.) with the same semantics.
- Maximum sizes, on `x86` and `arm` (see `https://j.mp/YIGq6W`):
  - Per allocation: 4 MB
  - Total allocations: 128 MB
- Should be used as the primary allocator unless there is a strong reason to use another one.

kmalloc API 1/2

- `#include <linux/slab.h>`
- `void *kmalloc(size_t size, gfp_t flags);`
  - Allocate `size` bytes, and return a pointer to the area (virtual address)
  - `size`: number of bytes to allocate
  - `flags`: same flags as the page allocator
- `void kfree(const void *objp);`
  - Free an allocated area
- Example: (drivers/infiniband/core/cache.c)

```
struct ib_port_attr *tprops;
tprops = kmalloc(sizeof *tprops, GFP_KERNEL);
...
kfree(tprops);
```

kmalloc API 2/2

- `void *kzalloc(size_t size, gfp_t flags);`
  - Allocates a zero-initialized buffer
- `void *kcalloc(size_t n, size_t size, gfp_t flags);`
  - Allocates memory for an array of `n` elements of `size` size, and zeroes its contents.
- `void *krealloc(const void *p, size_t new_size, gfp_t flags);`
  - Changes the size of the buffer pointed by `p` to `new_size`, by reallocating a new buffer and copying the data, unless `new_size` fits within the alignment of the existing buffer.

devm_kmalloc functions

Allocations with automatic freeing when the corresponding device or module is unprobed.

- `void *devm_kmalloc(struct device *dev, size_t size, gfp_t gfp);`
- `void *devm_kzalloc(struct device *dev, size_t size, gfp_t gfp);`
- `void *devm_kcalloc(struct device *dev, size_t n, size_t size, gfp_t flags);`
- `void *devm_kfree(struct device *dev, void *p);`
  Useful to immediately free an allocated buffer

For use in `probe()` functions, in which you have access to a `struct device` structure.

vmalloc allocator

- The `vmalloc()` allocator can be used to obtain memory zones that are contiguous in the virtual addressing space, but not made out of physically contiguous pages.
- The requested memory size is rounded up to the next page (not efficient for small allocations).
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible (almost as big as total available memory, see `https://j.mp/YIGq6W` again), since physical memory fragmentation is not an issue.
- Not suitable for DMA purposes.
- API in include/linux/vmalloc.h
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

Kernel memory debugging

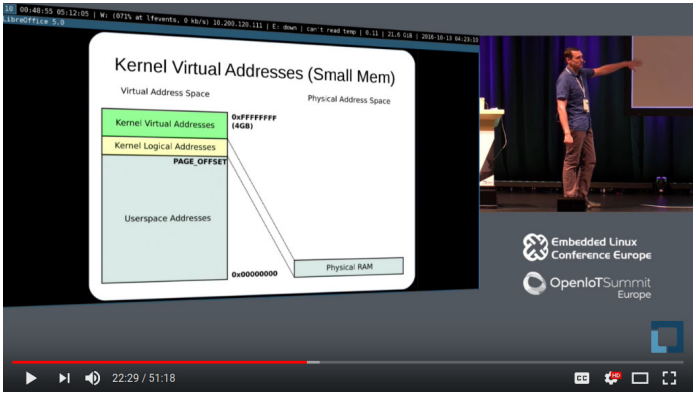
root commit

- `KASAN` (*Kernel Address Sanitizer*)
  - Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.
  - Available on most architectures
  - See `dev-tools/kasan` for details.
- `KFENCE` (*Kernel Electric Fence*)
  - A low overhead alternative to KASAN, trading performance for precision. Meant to be used in production systems.
  - Available on most architectures.
  - See `dev-tools/kfence` for details.
- `Kmemleak`
  - Dynamic checker for memory leaks
  - This feature is available for all architectures.
  - See `dev-tools/kmemleak` for details.

KASAN and Kmemleak have a significant overhead. Only use them in development!

Kernel memory management: resources

Virtual memory and Linux, Alan Ott and Matt Porter, 2016
Great and much more complete presentation about this topic
https://bit.ly/2Af1G2i (video: https://bit.ly/2Bwwv0C)

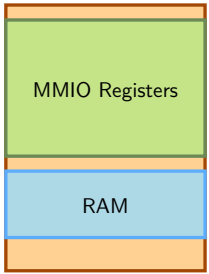


Kernel frameworks, memory and I/O

I/O Memory

Memory-Mapped I/O

- Same address bus to address memory and I/O device registers
- Access to the I/O device registers using regular instructions
- Most widely used I/O method across the different architectures supported by Linux



Physical Memory address space, accessed with normal load/store instructions

Requesting I/O memory

- Tells the kernel which driver is using which I/O registers
- ```
  struct resource *request_mem_region(unsigned long start,
                                      unsigned long len, char *name);
  ```
- ```
  void release_mem_region(unsigned long start, unsigned long len);
  ```
- Allows to prevent other drivers from requesting the same I/O registers, but is purely voluntary.

/proc/iomem — BeagleBone Black, Linux 5.11

root commit

```
40300000-4030ffff : 40300000.sram sram@0
44e00c00-44e00cff : 44e00c00.prm prm@c00
44e00d00-44e00dff : 44e00d00.prm prm@d00
44e00e00-44e00eff : 44e00e00.prm prm@e00
44e00f00-44e00fff : 44e00f00.prm prm@f00
44e01000-44e010ff : 44e01000.prm prm@1000
44e01100-44e011ff : 44e01100.prm prm@1100
44e01200-44e012ff : 44e01200.prm prm@1200
44e07000-44e07fff : 44e07000.gpio gpio@0
44e09000-44e0901f : serial
44e0b000-44e0bfff : 44e0b000.i2c i2c@0
44e10800-44e10a37 : pinctrl-single
44e10f90-44e10fcf : 44e10f90.dma-router dma-router@f90
48024000-48024fff : 48024000.serial serial@0
48042000-480423ff : 48042000.timer timer@0
48044000-480443ff : 48044000.timer timer@0
48046000-480463ff : 48046000.timer timer@0
48048000-480483ff : 48048000.timer timer@0
4804a000-4804a3ff : 4804a000.timer timer@0
4804c000-4804cfff : 4804c000.gpio gpio@0
48060000-48060fff : 48060000.mmc mmc@0
4819c000-4819cfff : 4819c000.i2c i2c@0
481a8000-481a8fff : 481a8000.serial serial@0
481ac000-481acfff : 481ac000.gpio gpio@0
481ae000-481aefff : 481ae000.gpio gpio@0
481d8000-481d8fff : 481d8000.mmc mmc@0
49000000-4900ffff : 49000000.dma edma3_cc
4a100000-4a1007ff : 4a100000.ethernet ethernet@0
4a101200-4a1012ff : 4a100000.ethernet ethernet@0
80000000-9fdfffff : System RAM
80008000-80cfffff : Kernel code
80e00000-80f3d807 : Kernel data
```

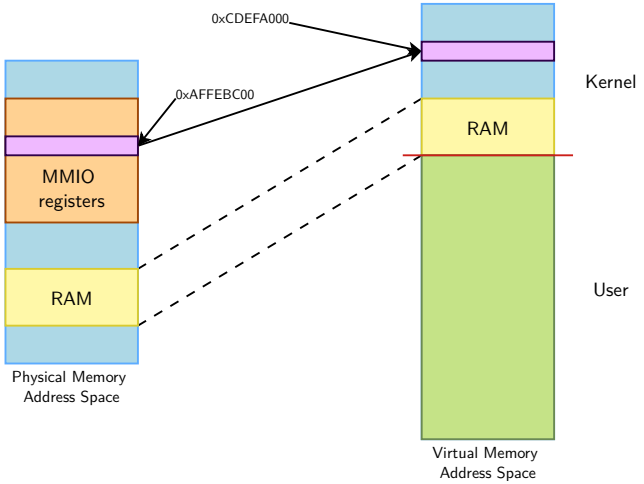
Mapping I/O memory in virtual memory

root commit

- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.
- The `ioremap` function satisfies this need:

```
#include <linux/io.h>

void __iomem *ioremap(phys_addr_t phys_addr, unsigned long size);
void iounmap(void __iomem *addr);
```

- Caution: check that `ioremap()` doesn't return a `NULL` address!




```
ioremap(0xAFFEBC00, 4096) = 0xCDEFA000
```

Managed API

Using `request_mem_region()` and `ioremap()` in device drivers is now deprecated. You should use the below "managed" functions instead, which simplify driver coding and error handling:

- `devm_ioremap()`, `devm_iounmap()`
- `devm_ioremap_resource()`
  - Takes care of both the request and remapping operations!
- `devm_platform_ioremap_resource()`
  - Takes care of `platform_get_resource()`, `request_mem_region()` and `ioremap()`
  - Caution: unlike the other `devm_` functions, its first argument is of type `struct platform_device`, not a pointer to `struct device`:

```
base = devm_platform_ioremap_resource(pdev, 0);
if (IS_ERR(base))
        return PTR_ERR(base);
```

Accessing MMIO devices: using accessor functions

root commit

- Directly reading from or writing to addresses returned by `ioremap()` (*pointer dereferencing*) may not work on some architectures.
- A family of architecture-independent accessor functions are available covering most needs.
- A few architecture-specific accessor functions also exists.

MMIO access functions

- `read[b/w/l/q]` and `write[b/w/l/q]` for access to little-endian devices, includes memory barriers
- `ioread[8/16/32/64]` and `iowrite[8/16/32/64]` are very similar to read/write but also work with port I/O (not covered in the course), includes memory barriers
- `ioread[8/16/32/64]be` and `iowrite[8/16/32/64]be` for access to big-endian devices, includes memory barriers
- `__raw_read[b/w/l/q]` and `__raw_write[b/w/l/q]` for raw access: no endianness conversion, no memory barriers
- `read[b/w/l/q]_relaxed` and `write[b/q/l/w]_relaxed` for access to little-endian devices, without memory barriers
- All functions work on a `void __iomem *`

MMIO access functions summary

Name	Device endianness	Memory barriers
`read/write`	little	yes
`ioread/iowrite`	little	yes
`ioreadbe/iowritebe`	big	yes
`__raw_read/__raw_write`	native	no
`read_relaxed/write_relaxed`	little	no

More details at `driver-api/device-io`

Ordering

- Reads/writes to MMIO-mapped registers of given device are done in program order
- However reads/writes to RAM can be re-ordered between themselves, and between MMIO-mapped read/writes
- Some of the accessor functions include memory barriers to help with this:
  - Write operation starts with a write memory barrier which prior writes cannot cross
  - Read operation ends with a read memory barrier which guarantees the ordering with regard to the subsequent reads
- Sometimes compiler/CPU reordering is not an issue, in this case the code may be optimized by dropping the memory barriers, using the raw or relaxed helpers
- See Documentation/memory-barriers.txt

/dev/mem

- Used to provide user space applications with direct access to physical addresses.
- Usage: open `/dev/mem` and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- Easy to use from a shell with the devmem2 program
- For security reasons, on `x86`, `arm`, `arm64`, `riscv`, `powerpc`, `parisc`, `s390`:
  - `CONFIG_STRICT_DEVMEM` restricts `/dev/mem` to non-RAM addresses (from v5.12)
  - `CONFIG_IO_STRICT_DEVMEM` goes beyond and only allows to access *idle* I/O ranges (not appearing in `/proc/iomem`).

Lab — I/O memory and ports

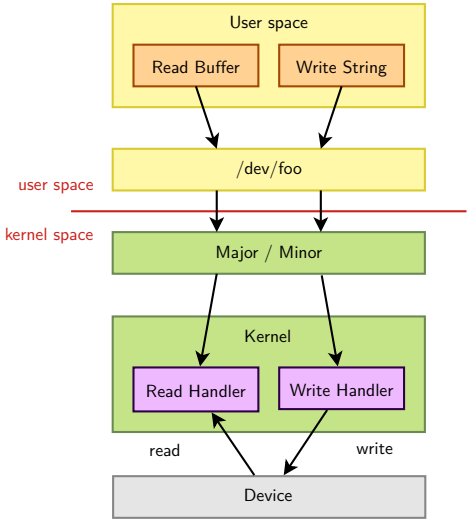
root commit

- Add UART devices to the board device tree
- Access I/O registers to control the device and send first characters to it.

Kernel frameworks, memory and I/O

Character drivers — Implementation details

From user space to the kernel: character devices



File operations

Here are the most important operations for a character driver, from the definition of `struct file_operations`:

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

Many operations exist, they are all optional.

open() and release()

- `int foo_open(struct inode *i, struct file *f)`
  - Called when user space opens the device file.
  - **Only implement this function when you do something special with the device at `open()` time.**
  - `struct inode` is a structure that uniquely represents a file in the filesystem (be it a regular file, a directory, a symbolic link, a character or block device)
  - `struct file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains information like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the `file` structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user space closes the file.
  - **Only implement this function when you do something special with the device at `close()` time.**

read() and write()

root commit

- `ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `read()` system call on the device.
  - Must read data from the device, write at most `sz` bytes to the user space buffer `buf`, and update the current position in the file `off`. `f` is a pointer to the same file structure that was passed in the `open()` operation
  - Must return the number of bytes read.
    0 is usually interpreted by userspace as the end of the file.
  - On UNIX, `read()` operations typically block when there isn't enough data to read from the device
- `ssize_t foo_write(struct file *f, const char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `write()` system call on the device
  - The opposite of `read`, must read at most `sz` bytes from `buf`, write it to the device, update `off` and return the number of bytes written.

Exchanging data with user space 1/3

root commit

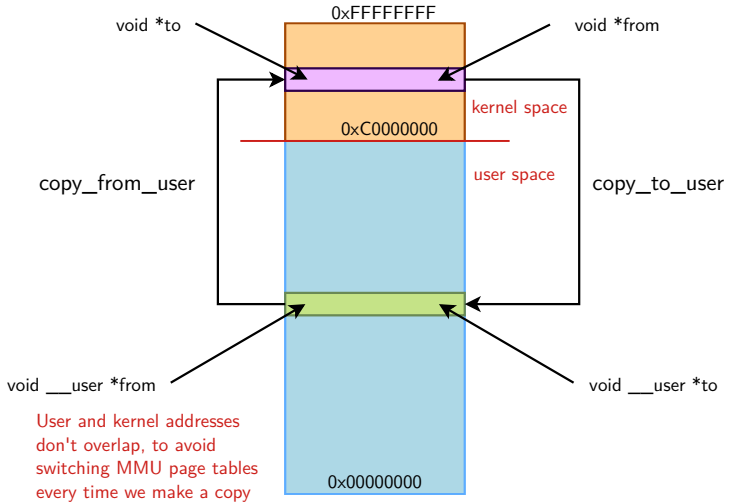
- Kernel code isn't allowed to directly access user space memory, using `memcpy()` or direct pointer dereferencing
  - User pointer dereferencing is disabled by default to make it harder to exploit vulnerabilities.
  - If the address passed by the application was invalid, the kernel could segfault.
  - **Never** trust user space. A malicious application could pass a kernel address which you could overwrite with device data (`read` case), or which you could dump to the device (`write` case).
  - Doing so does not work on some architectures anyway.
- To keep the kernel code portable, secure, and have proper error handling, your driver must use special kernel functions to exchange data with user space.

Exchanging data with user space 2/3

root commit

- A single value
  - `get_user(v, p);`
    - The kernel variable `v` gets the value pointed by the user space pointer `p`
  - `put_user(v, p);`
    - The value pointed by the user space pointer `p` is set to the contents of the kernel variable `v`.
- A buffer
  - ```
    unsigned long copy_to_user(void __user *to, const void *from,
                               unsigned long n);
    ```
  - ```
    unsigned long copy_from_user(void *to, const void __user *from,
                                 unsigned long n);
    ```
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

Exchanging data with user space 3/3



Zero copy access to user memory

root commit

- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).
- *Zero copy* options are possible:
  - `mmap()` system call to allow user space to directly access memory mapped I/O space. See our `mmap()` chapter.
  - `get_user_pages()` and related functions to get a mapping to user pages without having to copy them.

unlocked_ioctl()

- `long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`
  - Associated to the `ioctl()` system call.
  - Called unlocked because it didn't hold the Big Kernel Lock (gone now).
  - Allows to extend the driver capabilities beyond the limited read/write API.
  - For example: changing the speed of a serial port, setting video output format, querying a device serial number... Used extensively in the V4L2 (video) and ALSA (sound) driver frameworks.
  - `cmd` is a number identifying the operation to perform.
    See `driver-api/ioctl` for the recommended way of choosing `cmd` numbers.
  - `arg` is the optional argument passed as third argument of the `ioctl()` system call. Can be an integer, an address, etc.
  - The semantic of `cmd` and `arg` is driver-specific.

ioctl() example: kernel side

```c
#include <linux/phantom.h>

static long phantom_ioctl(struct file *file, unsigned int cmd,
    unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    default:
        return -ENOTTY;
    }

    return 0;
}
```

Selected excerpt from drivers/misc/phantom.c

Ioctl() Example: Application Side

root commit

```c
#include <linux/phantom.h>

int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, &reg);
    assert(ret == 0);

    return 0;
}
```

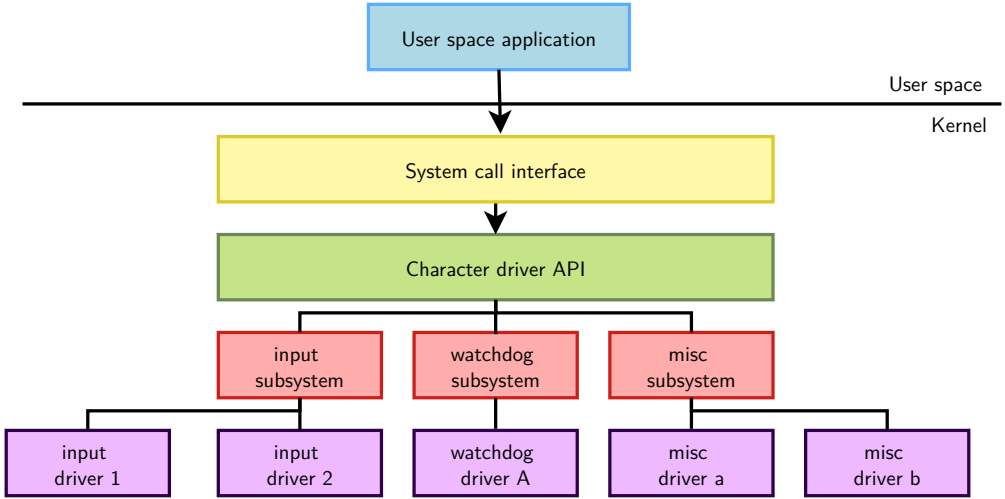
Kernel frameworks, memory and I/O

The misc subsystem

Why a *misc* subsystem?

- The kernel offers a large number of **frameworks** covering a wide range of device types: input, network, video, audio, etc.
  - These frameworks allow to factorize common functionality between drivers and offer a consistent API to user space applications.
- However, there are some devices that **really do not fit in any of the existing frameworks**.
  - Highly customized devices implemented in a FPGA, or other weird devices for which implementing a complete framework is not useful.
- The drivers for such devices could be implemented directly as raw *character drivers* (with `cdev_init()` and `cdev_add()`).
- But there is a subsystem that makes this work a little bit easier: the **misc subsystem**.
  - It is really only a **thin layer** above the *character driver* API.
  - Another advantage is that devices are integrated in the Device Model (device files appearing in *devtmpfs*, which you don't have with raw character devices).

Misc subsystem diagram



Misc subsystem API (1/2)

- The misc subsystem API mainly provides two functions, to register and unregister **a single** *misc device*:
  - `int misc_register(struct miscdevice * misc);`
  - `void misc_deregister(struct miscdevice *misc);`
- A *misc device* is described by a `struct miscdevice` structure:

```
struct miscdevice  {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        umode_t mode;
};
```

Misc subsystem API (2/2)

The main fields to be filled in `struct miscdevice` are:

- `minor`, the minor number for the device, or `MISC_DYNAMIC_MINOR` to get a minor number automatically assigned.
- `name`, name of the device, which will be used to create the device node if *devtmpfs* is used.
- `fops`, pointer to the same `struct file_operations` structure that is used for raw character drivers, describing which functions implement the *read*, *write*, *ioctl*, etc. operations.
- `parent`, pointer to the `struct device` of the underlying "physical" device (platform device, I2C device, etc.)

User space API for misc devices

- *misc devices* are regular character devices
- The operations they support in user space depends on the operations the kernel driver implements:
  - The `open()` and `close()` system calls to open/close the device.
  - The `read()` and `write()` system calls to read/write to/from the device.
  - The `ioctl()` system call to call some driver-specific operations.

Lab — Output-only serial port driver

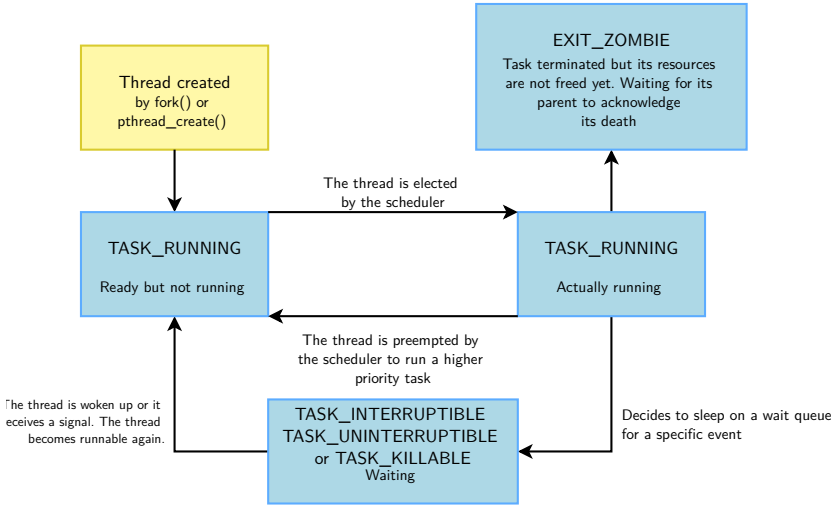
- Extend the driver started in the previous lab by registering it into the *misc* subsystem.
- Implement serial output functionality through the *misc* subsystem.
- Test serial output using user space applications.

Interrupts, deferring work, locking

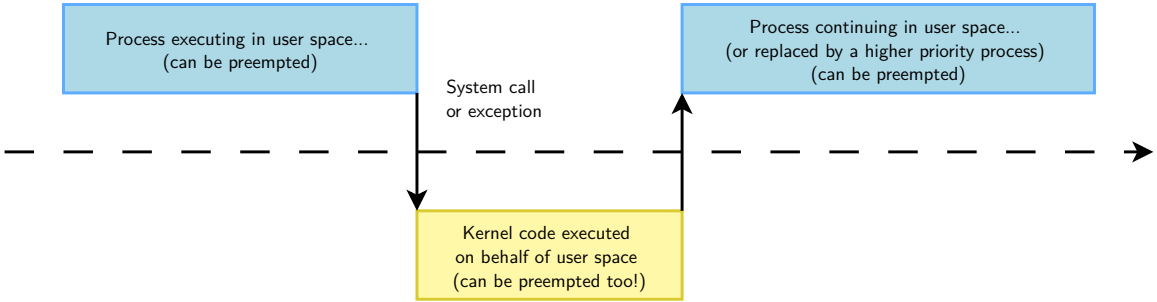
Interrupts, deferring work, locking

Processes and scheduling

A thread life



Execution of system calls

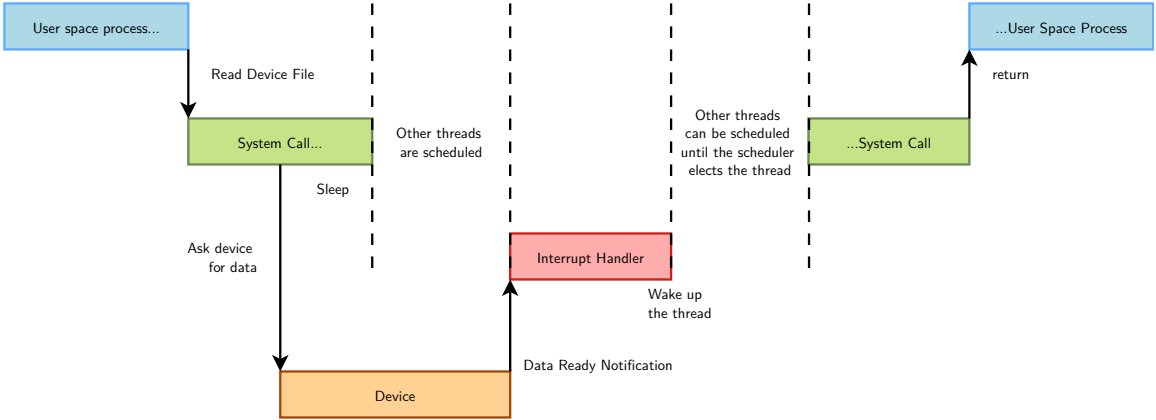


The execution of system calls takes place in the context of the thread requesting them.

Interrupts, deferring work, locking

Sleeping

Sleeping



Sleeping is needed when a process (user space or kernel space) is waiting for data.

How to sleep with a wait queue 1/3

- Must declare a wait queue, to store the list of threads waiting for an event
- Dynamic queue declaration:
  - Typically one queue per device managed by the driver
  - It's convenient to embed the wait queue inside a per-device data structure.
  - Example from drivers/net/ethernet/marvell/mvmdio.c:

```
struct orion_mdio_dev {
        ...
        wait_queue_head_t smi_busy_wait;
};
struct orion_mdio_dev *dev;
...
init_waitqueue_head(&dev->smi_busy_wait);
```

How to sleep with a waitqueue 2/3

root commit

Several ways to make a kernel process sleep

- `void wait_event(queue, condition);`
  - Sleeps until the task is woken up **and** the given C expression is true.
    Caution: can't be interrupted (can't kill the user space process!)
- `int wait_event_killable(queue, condition);`
  - Can be interrupted, but only by a *fatal* signal (`SIGKILL`).
    Returns `-ERESTARTSYS` if interrupted.
- `int wait_event_interruptible(queue, condition);`
  - The most common variant
  - Can be interrupted by any signal.
    Returns `-ERESTARTSYS` if interrupted.

How to sleep with a waitqueue 3/3

root commit

- `int wait_event_timeout(queue, condition, timeout);`
  - Also stops sleeping when the task is woken up **or** the timeout expired (a timer is used).
  - Returns 0 if the timeout elapsed, non-zero if the condition was met.
- `int wait_event_interruptible_timeout(queue, condition, timeout);`
  - Same as above, interruptible.
  - Returns 0 if the timeout elapsed,
    -ERESTARTSYS if interrupted,
    positive value if the condition was met.

How to sleep with a waitqueue - Example

```
sig = wait_event_interruptible(ibmvtpm->wq,
                               !ibmvtpm->tpm_processing_cmd);
if (sig)
        return -EINTR;
```

From drivers/char/tpm/tpm_ibmvtpm.c

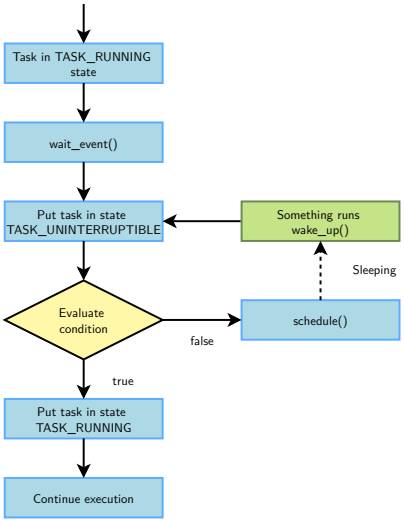
Waking up!

Typically done by interrupt handlers when data sleeping processes are waiting for become available.

- `wake_up(&queue);`
  - Wakes up all threads in the wait queue
- `wake_up_interruptible(&queue);`
  - Wakes up all threads waiting in an interruptible sleep on the given queue

Note: exclusive variants also exist (only one thread woken up)

Sleeping and waking up - Implementation



The scheduler doesn't keep evaluating the sleeping condition!

- `wait_event(queue, cond);`
  The process is put in the `TASK_UNINTERRUPTIBLE` state.
- `wake_up(&queue);`
  All processes waiting in `queue` are woken up, so they get scheduled later and have the opportunity to evaluate the condition again and go back to sleep if it is not met.

See include/linux/wait.h for implementation details.

Waiting when there is no interrupt

root commit

- When there is no interrupt mechanism tied to a particular hardware state, it is tempting to implement a custom busy-wait loop.
  - Spoiler alert: this is highly discouraged!
- For long lasting pauses, rely on helpers which leverage the system clock
  - `wait_event()` helpers are (also) very useful outside of interrupt situations
  - Release the CPU with `schedule()`
- For shorter pauses, use helpers which implement software loops
  - `msleep()`/`msleep_interruptible()` put the process in sleep for a given amount of milliseconds
  - `udelay()`/`udelay_range()` waste CPU cycles in order to save a couple of context switches for a sub-millisecond period
  - `cpu_relax()` does nothing, but may be used as a way to not being optimized out by the compiler when busy looping for very short periods

Interrupts, deferring work, locking

Interrupt Management

Registering an interrupt handler 1/2

The *managed* API is recommended:

```
int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                     unsigned long irq_flags, const char *devname, void *dev_id);
```

- `device` for automatic freeing at device or module release time.
- `irq` is the requested IRQ channel. For platform devices, use `platform_get_irq()` to retrieve the interrupt number.
- `handler` is a pointer to the IRQ handler function
- `irq_flags` are option masks (see next slide)
- `devname` is the registered name (for `/proc/interrupts`). For platform drivers, good idea to use `pdev->name` which allows to distinguish devices managed by the same driver (example: `44e0b000.i2c`).
- `dev_id` is an opaque pointer. It can typically be used to pass a pointer to a per-device data structure. It cannot be `NULL` as it is used as an identifier for freeing interrupts on a shared line.

Registering an interrupt handler 2/2

Here are the most frequent `irq_flags` bit values in drivers (can be combined):

- `IRQF_SHARED`: interrupt channel can be shared by several devices.
  - When an interrupt is received, all the interrupt handlers registered on the same interrupt line are called.
  - This requires a hardware status register telling whether an IRQ was raised or not.
- `IRQF_ONESHOT`: for use by threaded interrupts (see next slides). Keeping the interrupt line disabled until the thread function has run.

Interrupt handler constraints

- No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space.
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Interrupt handlers are run with all interrupts disabled on the local CPU (see https://lwn.net/Articles/380931). Therefore, they have to complete their job quickly enough, to avoiding blocking interrupts for too long.

/proc/interrupts on Raspberry Pi 2 (ARM, Linux 4.19)

```
           CPU0       CPU1       CPU2       CPU3
 17:    1005317          0          0          0  ARMCTRL-level   1 Edge      3f00b880.mailbox
 18:         36          0          0          0  ARMCTRL-level   2 Edge      VCHIQ doorbell
 40:          0          0          0          0  ARMCTRL-level  48 Edge      bcm2708_fb DMA
 42:     427715          0          0          0  ARMCTRL-level  50 Edge      DMA IRQ
 56:  478426356          0          0          0  ARMCTRL-level  64 Edge      dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 80:     411468          0          0          0  ARMCTRL-level  88 Edge      mmc0
 81:        502          0          0          0  ARMCTRL-level  89 Edge      uart-pl011
161:          0          0          0          0  bcm2836-timer   0 Edge      arch_timer
162:   10963772    6378711   16583353    6406625  bcm2836-timer   1 Edge      arch_timer
165:          0          0          0          0  bcm2836-pmu     9 Edge      arm-pmu
FIQ:                                             usb_fiq
IPI0:         0          0          0          0  CPU wakeup interrupts
IPI1:         0          0          0          0  Timer broadcast interrupts
IPI2:   2625198    4404191    7634127    3993714  Rescheduling interrupts
IPI3:      3140      56405      49483      59648  Function call interrupts
IPI4:         0          0          0          0  CPU stop interrupts
IPI5:   2167923     477097    5350168     412699  IRQ work interrupts
IPI6:         0          0          0          0  completion interrupts
Err:          0
```

Note: interrupt numbers shown on the left-most column are virtual numbers when the Device Tree is used. The physical interrupt numbers can be found in `/sys/kernel/debug/irq/irqs/<nr>` files when `CONFIG_GENERIC_IRQ_DEBUGFS=y`.

Interrupt handler prototype

- `irqreturn_t foo_interrupt(int irq, void *dev_id)`
  - `irq`, the IRQ number
  - `dev_id`, the per-device pointer that was passed to `devm_request_irq()`
- Return value
  - `IRQ_HANDLED`: recognized and handled interrupt
  - `IRQ_NONE`: used by the kernel to detect spurious interrupts, and disable the interrupt line if none of the interrupt handlers has handled the interrupt.
  - `IRQ_WAKE_THREAD`: handler requests to wake the handler thread (see next slides)

Typical interrupt handler's job

- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any process waiting for such data, typically on a per-device wait queue: `wake_up_interruptible(&device_queue);`

Threaded interrupts

The kernel also supports threaded interrupts:

- The interrupt handler is executed inside a thread.
- Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs time to communicate with them.
- Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux

```
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                              irq_handler_t handler, irq_handler_t thread_fn,
                              unsigned long flags, const char *name,
                              void *dev);
```

- `handler`, "hard IRQ" handler
- `thread_fn`, executed in a thread

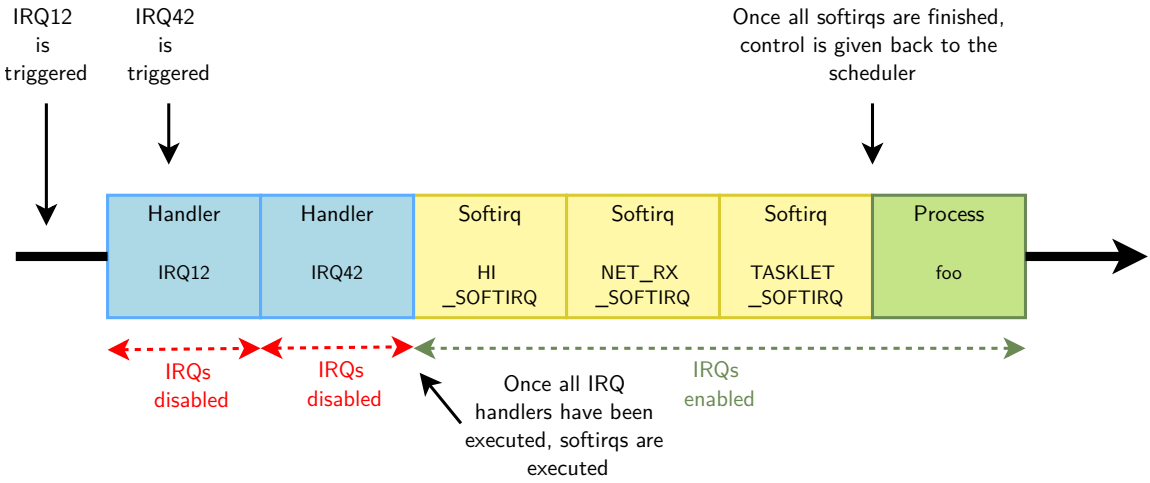
Top half and bottom half processing

root commit

Splitting the execution of interrupt handlers in 2 parts

- Top half
  - This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. It takes the data out of the device and if substantial post-processing is needed, schedule a bottom half to handle it.
- Bottom half
  - Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as: softirqs, tasklets, threaded handlers or workqueues.
  - And yet, the abbreviation "bh" often means "softirqs"...

Top half and bottom half diagram



Softirqs

- Softirqs are a form of bottom half processing
- The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by kernel subsystems (network, etc.)
- The list of softirqs is defined in include/linux/interrupt.h: `HI_SOFTIRQ`, `TIMER_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ`, `BLOCK_SOFTIRQ`, `IRQ_POLL_SOFTIRQ`, `TASKLET_SOFTIRQ`, `SCHED_SOFTIRQ`, `HRTIMER_SOFTIRQ`, `RCU_SOFTIRQ`
- `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` are used to execute tasklets

Example usage of softirqs - NAPI

NAPI = *New API*

- Interface in the Linux kernel used for interrupt mitigation in network drivers
- Principle: when the network traffic exceeds a given threshhold ("budget"), disable network interrupts and consume incoming packets through a polling function, instead of processing each new packet with an interrupt.
- This reduces overhead due to interrupts and yields better network throughput.
- The polling function is run by `napi_schedule()`, which uses `NET_RX_SOFTIRQ`.
- See https://en.wikipedia.org/wiki/New_API for details
- See also our commented network driver on https://bootlin.com/pub/drivers/r6040-network-driver-with-comments.c

Tasklets

- Tasklets are executed within the `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
- Tasklets are typically created with the `tasklet_init()` function, when your driver manages multiple devices, otherwise statically with `DECLARE_TASKLET()`. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
- The interrupt handler can schedule tasklet execution with:
  - `tasklet_schedule()` to get it executed in `TASKLET_SOFTIRQ`
  - `tasklet_hi_schedule()` to get it executed in `HI_SOFTIRQ` (highest priority)

Note: new kernel code should not introduce any new tasklet, because tasklets are now deprecated (since 6.9) and being slowly replaced by the new BH workqueue (Bottom Half workqueue)

Workqueues

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts. It can typically be used for background jobs.
- Functions registered to run in workqueues are called works:
  - They can be created with the macro `INIT_WORK()`
  - When scheduled, they become threads (called workers) running in process context, which means:
    - All interrupts are enabled
    - Sleeping is allowed
  - Works can be queued on:
    - The default workqueue, with `schedule_work()`
    - A workqueue allocated by the subsystem or the drivers, with `alloc_workqueue()`
- The complete API is in include/linux/workqueue.h
- Example (drivers/crypto/atmel-i2c.c):

```
INIT_WORK(&work_data->work, atmel_i2c_work_handler);
schedule_work(&work_data->work);
```

Interrupt management summary

- Device driver
  - In the `probe()` function, for each device, use `devm_request_irq()` to register an interrupt handler for the device's interrupt channel.
- Interrupt handler
  - Called when an interrupt is raised.
  - Acknowledge the interrupt
  - If needed, schedule a per-device tasklet taking care of handling data.
  - Wake up processes waiting for the data on a per-device queue
- Device driver
  - In the `remove()` function, for each device, the interrupt handler is automatically unregistered.

Lab — Interrupts

- Adding read capability to the character driver developed earlier.
- Register an interrupt handler for each device.
- Waiting for data to be available in the read file operation.
- Waking up the code when data are available from the devices.

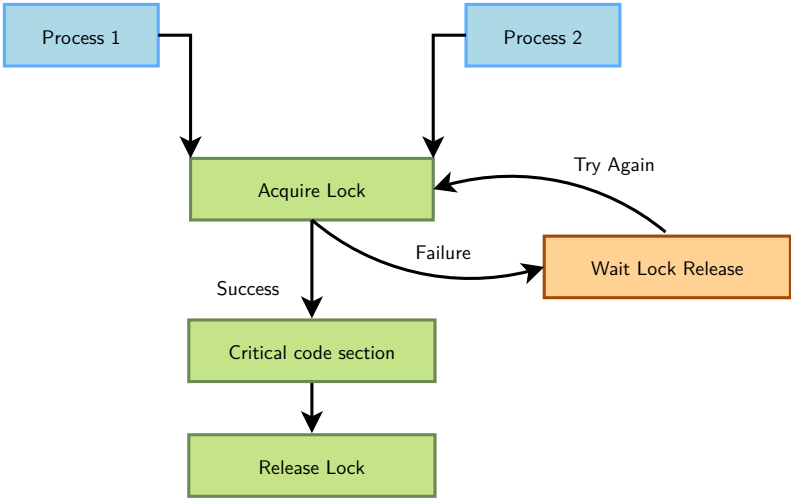
Concurrent Access to Resources: Locking

Sources of concurrency issues

root commit

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- Concurrency arises because of
  - *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources (memory addresses, hardware registers...)
  - *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one thread to another. They may be using shared resources.
  - *Multiprocessing*, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.
- The solution is to keep as much local state as possible and for the shared resources that can't be made local (such as hardware ones), use locking.

Concurrency protection with locks



Linux mutexes

mutex = **mut***ual* **ex***clusion*

- The kernel's main locking primitive. It's a *binary lock*. Note that *counting locks* (*semaphores*) are also available, but used 30x less frequently.
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Mutex definition:
  - `#include <linux/mutex.h>`
- Initializing a mutex statically (unusual case):
  - `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically (the usual case, on a per-device basis):
  - `void mutex_init(struct mutex *lock);`

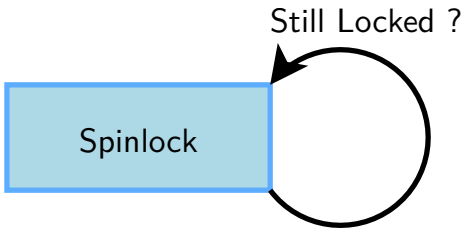
Locking and unlocking mutexes 1/2

root commit

- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: cannot be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible(struct mutex *lock);`
  - Same, but can be interrupted by any signal.
- `void mutex_unlock(struct mutex *lock);`
  - Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- The critical section protected by a spinlock is not allowed to sleep.

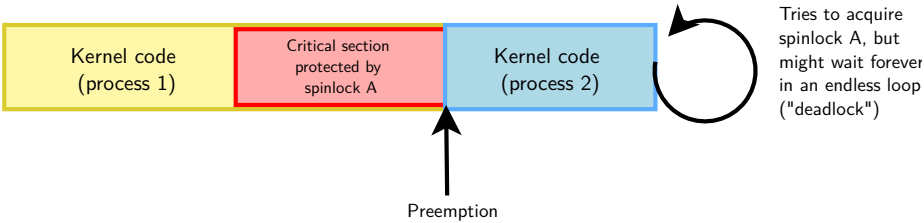


The spinlock API

- Spinlocks can be initialized:
  - Statically (unusual)
    - `DEFINE_SPINLOCK(my_lock);`
  - Dynamically (the usual case, on a per-device basis)
    - `void spin_lock_init(spinlock_t *lock);`
- They can be acquired and released with:
  - `void spin_lock(spinlock_t *lock);`
    - Used for locking in process context (critical sections in which you do not want to sleep) as well as atomic sections.
  - `void spin_unlock(spinlock_t *lock);`

Using spinlocks 1/2

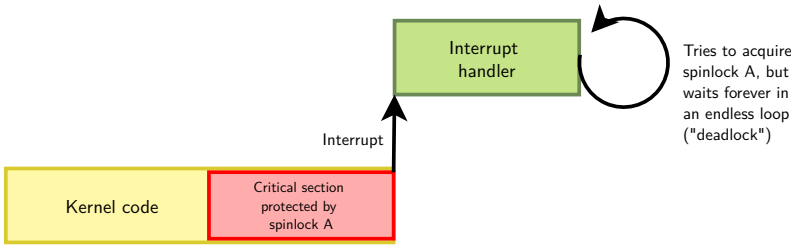
- Manipulating spinlocks implies some care:



- So, kernel preemption on the local CPU is disabled. We need to avoid deadlocks (and unbounded latencies) because of preemption from processes that want to get the same lock.
- Disabling kernel preemption also disables migration to avoid the same kind of issue as pictured above from happening.

Using spinlocks 2/2

- We also need to avoid deadlocks because of interrupts that could want to get the same lock:



- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
  - Disables/restores IRQs on the local CPU.
  - Typically used when the lock can be accessed in both process and interrupt context.

Using spinlocks 3/3

- `void spin_lock_bh(spinlock_t *lock);`
- `void spin_unlock_bh(spinlock_t *lock);`
  - Disables software interrupts, but not hardware ones.
  - Useful to protect shared data accessed in process context and in a soft interrupt (*bottom half*).
  - No need to disable hardware interrupts in this case.
- Note that reader/writer spinlocks also exist, allowing for multiple simultaneous readers.

Spinlock example

root commit

- From drivers/tty/serial/uartlite.c
- Spinlock structure embedded into `struct uart_port`

```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

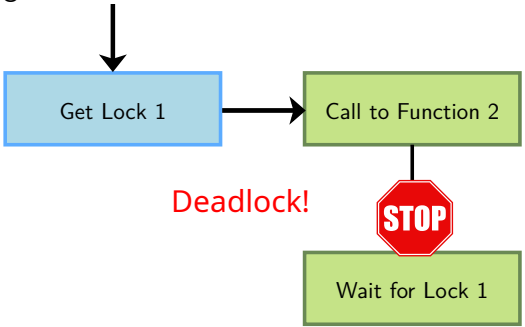
- Spinlock taken/released with protection against interrupts

```
static unsigned int ulite_tx_empty(struct uart_port *port) {
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

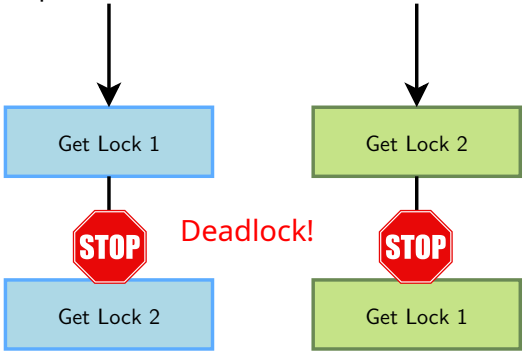
More deadlock situations

They can lock up your system. Make sure they never happen!

Rule 1: don't call a function that can try to get access to the same lock



Rule 2: if you need multiple locks, always acquire them in the same order!



Debugging locking

- Lock debugging: prove locking correctness
  - `CONFIG_PROVE_LOCKING`
  - Adds instrumentation to kernel locking code
  - Detect violations of locking rules during system life, such as:
    - Locks acquired in different order (keeps track of locking sequences and compares them).
    - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
  - Not suitable for production systems but acceptable overhead in development.
  - See `locking/lockdep-design` for details
- `CONFIG_DEBUG_ATOMIC_SLEEP` allows to detect code that incorrectly sleeps in atomic section (while holding lock typically).
  - Warning displayed in `dmesg` in case of such violation.

Concurrency issues

- Kernel Concurrency SANitizer framework
- CONFIG_KCSAN, introduced in Linux 5.8.
- Dynamic race detector relying on compile time instrumentation.
- Can find concurrency issues (mainly data races) in your system.
- See dev-tools/kcsan and https://lwn.net/Articles/816850/ for details.

Alternatives to locking

As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.

- By using lock-free algorithms like *Read Copy Update* (RCU).
  - RCU API available in the kernel
  - See https://en.wikipedia.org/wiki/Read-copy-update for a coverage of how it works.
- When relevant, use atomic operations.

Atomic variables 1/2

root commit

```
#include <linux/atomic.h>
```

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Ideal for RMW (Read-Modify-Write) operations
- Main atomic operations on `atomic_t` (signed integer, at least 24 bits):
  - Set or read the counter:
    - `void atomic_set(atomic_t *v, int i);`
    - `int atomic_read(atomic_t *v);`
  - Operations without return value:
    - `void atomic_inc(atomic_t *v);`
    - `void atomic_dec(atomic_t *v);`
    - `void atomic_add(int i, atomic_t *v);`
    - `void atomic_sub(int i, atomic_t *v);`

Atomic variables 2/2

- Similar functions testing the result:
  - `int atomic_inc_and_test(...);`
  - `int atomic_dec_and_test(...);`
  - `int atomic_sub_and_test(...);`
- Functions returning the new value:
  - `int atomic_inc_return(...);`
  - `int atomic_dec_return(...);`
  - `int atomic_add_return(...);`
  - `int atomic_sub_return(...);`

Atomic bit operations

root commit

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long *` type.
- Apply to a `void *` type on a few others.
- Ideal for bitmaps
- Set, clear, toggle a given bit:
  - `void set_bit(int nr, unsigned long *addr);`
  - `void clear_bit(int nr, unsigned long *addr);`
  - `void change_bit(int nr, unsigned long *addr);`
- Test bit value:
  - `int test_bit(int nr, unsigned long *addr);`
- Test and modify (return the previous value):
  - `int test_and_set_bit(...);`
  - `int test_and_clear_bit(...);`
  - `int test_and_change_bit(...);`

Kernel locking: summary and references

- Use mutexes in code that is allowed to sleep
- Use spinlocks in code that is not allowed to sleep (interrupts) or for which sleeping would be too costly (critical sections)
- Use atomic operations to protect integers or addresses

See `kernel-hacking/locking` in kernel documentation for many details about kernel locking mechanisms.

Further reading: see the classical *dining philosophers problem* for a nice illustration of synchronization and concurrency issues.

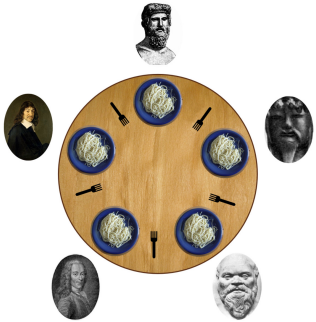


Image source: https://en.wikipedia.org/wiki/Dining_philosophers_problem)

Lab — Locking

- Identify issues caused by concurrent access to shared resources
- Understand why they are happening
- Use atomic variables and operations, as well as mutexes, to fix such issues.

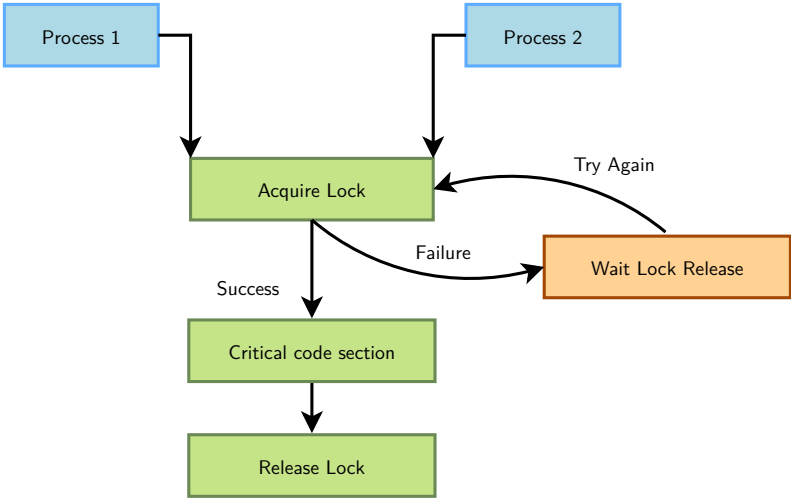
Concurrent Access to Resources: Locking

Sources of concurrency issues

root commit

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- Concurrency arises because of
  - *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources (memory addresses, hardware registers...)
  - *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one thread to another. They may be using shared resources.
  - *Multiprocessing*, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.
- The solution is to keep as much local state as possible and for the shared resources that can't be made local (such as hardware ones), use locking.

Concurrency protection with locks



Linux mutexes

mutex = **mut***ual* **ex***clusion*

- The kernel's main locking primitive. It's a *binary lock*. Note that *counting locks* (*semaphores*) are also available, but used 30x less frequently.
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Mutex definition:
  - `#include <linux/mutex.h>`
- Initializing a mutex statically (unusual case):
  - `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically (the usual case, on a per-device basis):
  - `void mutex_init(struct mutex *lock);`

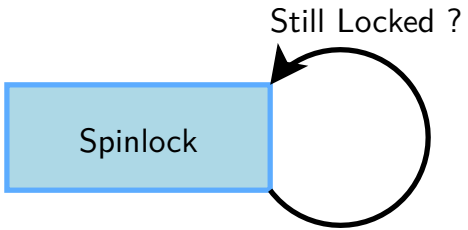
Locking and unlocking mutexes 1/2

root commit

- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: cannot be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible(struct mutex *lock);`
  - Same, but can be interrupted by any signal.
- `void mutex_unlock(struct mutex *lock);`
  - Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- The critical section protected by a spinlock is not allowed to sleep.

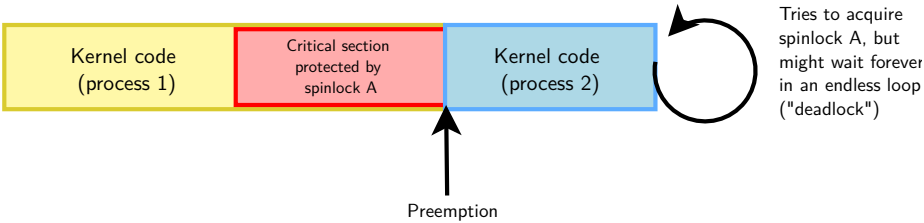


The spinlock API

- Spinlocks can be initialized:
  - Statically (unusual)
    - `DEFINE_SPINLOCK(my_lock);`
  - Dynamically (the usual case, on a per-device basis)
    - `void spin_lock_init(spinlock_t *lock);`
- They can be acquired and released with:
  - `void spin_lock(spinlock_t *lock);`
    - Used for locking in process context (critical sections in which you do not want to sleep) as well as atomic sections.
  - `void spin_unlock(spinlock_t *lock);`

Using spinlocks 1/2

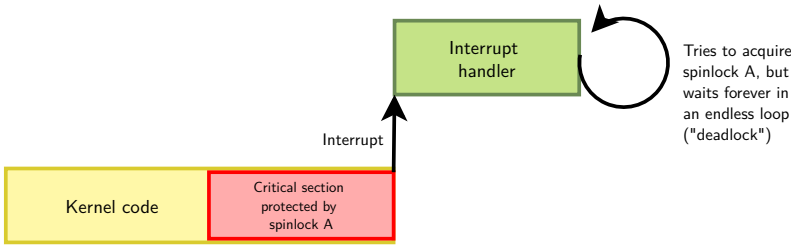
- Manipulating spinlocks implies some care:



- So, kernel preemption on the local CPU is disabled. We need to avoid deadlocks (and unbounded latencies) because of preemption from processes that want to get the same lock.
- Disabling kernel preemption also disables migration to avoid the same kind of issue as pictured above from happening.

Using spinlocks 2/2

- We also need to avoid deadlocks because of interrupts that could want to get the same lock:



- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
  - Disables/restores IRQs on the local CPU.
  - Typically used when the lock can be accessed in both process and interrupt context.

Using spinlocks 3/3

- `void spin_lock_bh(spinlock_t *lock);`
- `void spin_unlock_bh(spinlock_t *lock);`
  - Disables software interrupts, but not hardware ones.
  - Useful to protect shared data accessed in process context and in a soft interrupt (*bottom half*).
  - No need to disable hardware interrupts in this case.
- Note that reader/writer spinlocks also exist, allowing for multiple simultaneous readers.

Spinlock example

- From drivers/tty/serial/uartlite.c
- Spinlock structure embedded into `struct uart_port`

```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

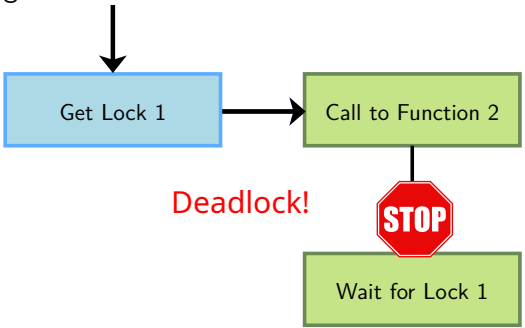
- Spinlock taken/released with protection against interrupts

```
static unsigned int ulite_tx_empty(struct uart_port *port) {
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

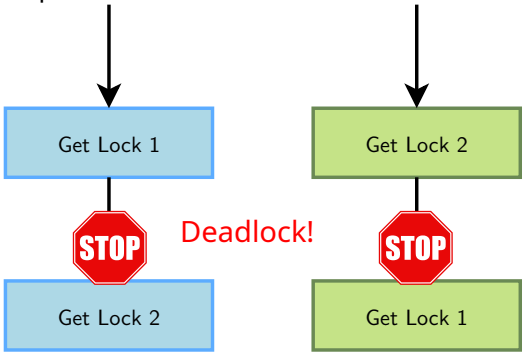
More deadlock situations

They can lock up your system. Make sure they never happen!

Rule 1: don't call a function that can try to get access to the same lock



Rule 2: if you need multiple locks, always acquire them in the same order!



Debugging locking

- Lock debugging: prove locking correctness
  - CONFIG_PROVE_LOCKING
  - Adds instrumentation to kernel locking code
  - Detect violations of locking rules during system life, such as:
    - Locks acquired in different order (keeps track of locking sequences and compares them).
    - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
  - Not suitable for production systems but acceptable overhead in development.
  - See locking/lockdep-design for details
- CONFIG_DEBUG_ATOMIC_SLEEP allows to detect code that incorrectly sleeps in atomic section (while holding lock typically).
  - Warning displayed in dmesg in case of such violation.

Concurrency issues

- Kernel Concurrency SANitizer framework
- CONFIG_KCSAN, introduced in Linux 5.8.
- Dynamic race detector relying on compile time instrumentation.
- Can find concurrency issues (mainly data races) in your system.
- See dev-tools/kcsan and https://lwn.net/Articles/816850/ for details.

Alternatives to locking

As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.

- By using lock-free algorithms like *Read Copy Update* (RCU).
  - RCU API available in the kernel
  - See https://en.wikipedia.org/wiki/Read-copy-update for a coverage of how it works.
- When relevant, use atomic operations.

Atomic variables 1/2

root commit

```
#include <linux/atomic.h>
```

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Ideal for RMW (Read-Modify-Write) operations
- Main atomic operations on `atomic_t` (signed integer, at least 24 bits):
  - Set or read the counter:
    - `void atomic_set(atomic_t *v, int i);`
    - `int atomic_read(atomic_t *v);`
  - Operations without return value:
    - `void atomic_inc(atomic_t *v);`
    - `void atomic_dec(atomic_t *v);`
    - `void atomic_add(int i, atomic_t *v);`
    - `void atomic_sub(int i, atomic_t *v);`

Atomic variables 2/2

root commit

- Similar functions testing the result:
  - `int atomic_inc_and_test(...);`
  - `int atomic_dec_and_test(...);`
  - `int atomic_sub_and_test(...);`
- Functions returning the new value:
  - `int atomic_inc_return(...);`
  - `int atomic_dec_return(...);`
  - `int atomic_add_return(...);`
  - `int atomic_sub_return(...);`

Atomic bit operations

root commit

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long *` type.
- Apply to a `void *` type on a few others.
- Ideal for bitmaps
- Set, clear, toggle a given bit:
  - `void set_bit(int nr, unsigned long *addr);`
  - `void clear_bit(int nr, unsigned long *addr);`
  - `void change_bit(int nr, unsigned long *addr);`
- Test bit value:
  - `int test_bit(int nr, unsigned long *addr);`
- Test and modify (return the previous value):
  - `int test_and_set_bit(...);`
  - `int test_and_clear_bit(...);`
  - `int test_and_change_bit(...);`

Kernel locking: summary and references

- Use mutexes in code that is allowed to sleep
- Use spinlocks in code that is not allowed to sleep (interrupts) or for which sleeping would be too costly (critical sections)
- Use atomic operations to protect integers or addresses

See `kernel-hacking/locking` in kernel documentation for many details about kernel locking mechanisms.

Further reading: see the classical *dining philosophers problem* for a nice illustration of synchronization and concurrency issues.

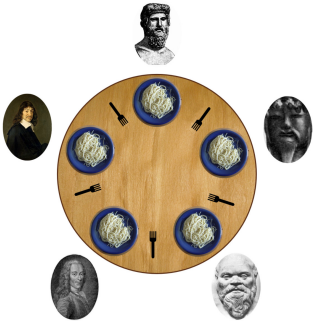


Image source: https://en.wikipedia.org/wiki/Dining_philosophers_problem)

Lab — Locking

- Identify issues caused by concurrent access to shared resources
- Understand why they are happening
- Use atomic variables and operations, as well as mutexes, to fix such issues.

Board support, debugging, testing and contribution

Kernel debugging

Debugging using messages (1/3)

root commit

Three APIs are available

- The old `printk()`, no longer recommended for new debugging messages
- The `pr_*()` family of functions: `pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_err()`, `pr_warn()`, `pr_notice()`, `pr_info()`, `pr_cont()`
  and the special `pr_debug()` (see next pages)
  - Defined in include/linux/printk.h
  - They take a classic format string with arguments
  - Example:

    ```
    pr_info("Booting CPU %d\n", cpu);
    ```

  - Here's what you get in the kernel log:

    ```
    [  202.350064] Booting CPU 1
    ```

- `print_hex_dump_debug()`: useful to dump a buffer with `hexdump` like display

Debugging using messages (2/3)

root commit

- The `dev_*()` family of functions: `dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_err()`, `dev_warn()`, `dev_notice()`, `dev_info()` and the special `dev_dbg()` (see next page)
  - They take a pointer to `struct device` as first argument, and then a format string with arguments
  - Defined in include/linux/dev_printk.h
  - To be used in drivers integrated with the Linux device model
  - Example:

    ```
    dev_info(&pdev->dev, "in probe\n");
    ```

  - Here's what you get in the kernel log:

    ```
    [   25.878382] serial 48024000.serial: in probe
    [   25.884873] serial 481a8000.serial: in probe
    ```

- `*_ratelimited()` version exists which limits the amount of printing if called too often, based on `/proc/sys/kernel/printk_ratelimit_burst` values

Debugging using messages (3/3)

root commit

- The kernel defines many more format specifiers than the standard `printf()` existing ones.
  - `%p`: Display the hashed value of pointer by default.
  - `%px`: Always display the address of a pointer (use carefully on non-sensitive addresses).
  - `%pK`: Display hashed pointer value, zeros or the pointer address depending on `kptr_restrict` sysctl value.
  - `%pOF`: Device-tree node format specifier.
  - `%pr`: Resource structure format specifier.
  - `%pa`: Physical address display (work on all architectures 32/64 bits)
  - `%pe`: Error pointer (displays the string corresponding to the error number)
- `/proc/sys/kernel/kptr_restrict` should be set to `1` in order to display pointers which use `%pK`
- See `core-api/printk-formats` for an exhaustive list of supported format specifiers

pr_debug() and dev_dbg()

root commit

- When the driver is compiled with `DEBUG` defined, all these messages are compiled and printed at the debug level. `DEBUG` can be defined by `#define DEBUG` at the beginning of the driver, or using `ccflags-$(CONFIG_DRIVER) += -DDEBUG` in the `Makefile`
- When the kernel is compiled with `CONFIG_DYNAMIC_DEBUG`, then these messages can dynamically be enabled on a per-file, per-module or per-message basis, by writing commands to `/proc/dynamic_debug/control`. Note that messages are not enabled by default.
  - Details in `admin-guide/dynamic-debug-howto`
  - Very powerful feature to only get the debug messages you're interested in.
- When neither `DEBUG` nor `CONFIG_DYNAMIC_DEBUG` are used, these messages are not compiled in.

Configuring the priority

- Each message is associated to a priority, ranging from 0 for emergency to 7 for debug, as specified in include/linux/kern_levels.h.
- All the messages, regardless of their priority, are stored in the kernel log ring buffer
  - Typically accessed using the `dmesg` command
- Some of the messages may appear on the console, depending on their priority and the configuration of
  - The `loglevel` kernel parameter, which defines the priority number below which messages are displayed on the console. Details in `admin-guide/kernel-parameters`.
    Examples: `loglevel=0`: no message, `loglevel=8`: all messages
  - The value of `/proc/sys/kernel/printk`, which allows to change at runtime the priority above which messages are displayed on the console. Details in `admin-guide/sysctl/kernel`

Debug logs troubleshooting

root commit

- When using dynamic debug, make sure that your debug call is enabled: it must be visible in `control` file in debugfs **and** be actived (=p)
- Is your log output only in kernel log buffer?
  - You can see it thanks to `dmesg`
  - You can lower the `loglevel` to output it to the console directly
  - You can also set `ignore_loglevel` in the kernel command line to force all kernel logs to console
- If you are working on an out-of-tree module, you may prefer to define `DEBUG` in your module source or Makefile instead of using dynamic debug
- If configuration is done through kernel command line, is it properly interpreted?
  - Starting from 5.14, kernel will let you know about faulty command line: `Unknown kernel command line parameters laglevel, will be passed to user space.`
  - You may need to take care of special characters escaping (e.g: quotes)
- Be aware that a few subsystems bring their own logging infrastructure, with specific configuration/controls, eg: `drm.debug=0x1ff`

DebugFS

A virtual filesystem to export debugging information to user space.

- Kernel configuration: `CONFIG_DEBUG_FS`
  - `Kernel hacking -> Debug Filesystem`
- The debugging interface disappears when Debugfs is configured out.
- You can mount it as follows:
  - `sudo mount -t debugfs none /sys/kernel/debug`
- First described on https://lwn.net/Articles/115405/
- API documented in the Linux Kernel Filesystem API: `filesystems/debugfs`

DebugFS API

root commit

- Create a sub-directory for your driver:
  - `struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);`
- Expose an integer as a file in DebugFS. Example:
  - `struct dentry *debugfs_create_u8 (const char *name, mode_t mode, struct dentry *parent, u8 *value);`
    - u8, u16, u32, u64 for decimal representation
    - x8, x16, x32, x64 for hexadecimal representation
- Expose a binary blob as a file in DebugFS:
  - `struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob);`
- Also possible to support writable DebugFS files or customize the output using the more generic `debugfs_create_file()` function.

Deprecated debugging mechanisms

Some additional debugging mechanisms, whose usage is now considered deprecated

- Adding special `ioctl()` commands for debugging purposes. DebugFS is preferred.
- Adding special entries in the `proc` filesystem. DebugFS is preferred.
- Adding special entries in the `sysfs` filesystem. DebugFS is preferred.
- Using `printk()`. The `pr_*()` and `dev_*()` functions are preferred.

Using Magic SysRq

Functionnality provided by serial drivers

- Allows to run multiple debug / rescue commands even when the kernel seems to be in deep trouble
  - On PC: press `[Alt]` + `[Prnt Scrn]` + `<character>` simultaneously (`[SysRq]` = `[Alt]` + `[Prnt Scrn]`)
  - On embedded: in the console, send a break character (Picocom: press `[Ctrl]` + a followed by `[Ctrl]` +), then press `<character>`
- Example commands:
  - `h`: show available commands
  - `s`: sync all mounted filesystems
  - `b`: reboot the system
  - `n`: makes RT processes nice-able.
  - `w`: shows the kernel stack of all sleeping processes
  - `t`: shows the kernel stack of all running processes
  - You can even register your own!
- Detailed in `admin-guide/sysrq`

kgdb - A kernel debugger

root commit

- `CONFIG_KGDB` in *Kernel hacking*.
- The execution of the kernel is fully controlled by `gdb` from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Feature supported for the most popular CPU architectures
- `CONFIG_GDB_SCRIPTS` allows to build GDB python scripts that are provided by the kernel.
  - See `process/debugging/gdb-kernel-debugging` for more information

Using kgdb (1/2)

- Details available in the kernel documentation: `process/debugging/kgdb`
- You must include a kgdb I/O driver. One of them is `kgdb` over serial console (`kgdboc`: `kgdb` over console, enabled by `CONFIG_KGDB_SERIAL_CONSOLE`)
- Configure `kgdboc` at boot time by passing to the kernel:
  - `kgdboc=<tty-device>,<bauds>`.
  - For example: `kgdboc=ttyS0,115200`
- Or at runtime using sysfs:
  - `echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc`
  - If the console does not have polling support, this command will yield an error.

Using kgdb (2/2)

- Then also pass `kgdbwait` to the kernel: it makes `kgdb` wait for a debugger connection.
- Boot your kernel, and when the console is initialized, interrupt the kernel with a break character and then g in the serial console (see our *Magic SysRq* explanations).
- On your workstation, start gdb as follows:
  - `arm-linux-gdb ./vmlinux`
  - `(gdb) set remotebaud 115200`
  - `(gdb) target remote /dev/ttyS0`
- Once connected, you can debug a kernel the way you would debug an application program.
- On GDB side, the first threads represent the CPU context (ShadowCPU<x>), then all the other threads represents a task.

Early traces

- If something breaks before the `tty` layer, serial driver and serial console are properly registered, you might just have nothing else after "`Starting kernel...`"
- On ARM, if your platform implements it, you can activate (`CONFIG_DEBUG_LL` and `CONFIG_EARLYPRINTK`), and add `earlyprintk` to the kernel command line
  - Assembly routines to just push a character and wait for it to be sent
  - Extremely basic, but is part of the uncompressed section, so available even if the kernel does not uncompress correctly!
- On other platforms, hoping that your serial driver implements `OF_EARLYCON_DECLARE()`, you can enable `CONFIG_SERIAL_EARLYCON`
  - The kernel will try to hook an appropriate `earlycon` UART driver using the `stdout-path` of the device-tree.

More kernel debugging tips

- Make sure `CONFIG_KALLSYMS_ALL` is enabled
  - To get oops messages with symbol names instead of raw addresses
  - Turned on by default
- Make sure `CONFIG_DEBUG_INFO` is also enabled
  - This way, the kernel is compiled with `$(CROSSCOMPILE)gcc -g`, which keeps the source code inside the binaries.

Lab — Kernel debugging

root commit

- Use the dynamic debug feature.
- Add debugfs entries
- Load a broken driver and see it crash
- Analyze the error information dumped by the kernel.
- Disassemble the code and locate the exact C instruction which caused the failure.

Kernel debugging

Porting the Linux kernel to an ARM board

Porting the Linux kernel

root commit

- The Linux kernel supports a lot of different CPU architectures
- Each of them is maintained by a different group of contributors
  - See the MAINTAINERS file for details
- The organization of the source code and the methods to port the Linux kernel to a new board are therefore very architecture-dependent
  - For example, some architectures use the Device Tree, some do not.
- This presentation is mainly focused on the ARM (32-bit) architecture

Architecture, CPU and Machine

- In the source tree, each architecture has its own directory
  - arch/arm/ for the ARM 32-bit architecture
  - arch/arm64/ for the ARM 64-bit architecture
- The arch/arm/ directory contains generic ARM code
  - boot/, common/, configs/, kernel/, lib/, mm/, nwfpe/, vfp/, tools/ and several others.
- And many directories for different SoC families
  - mach-* directories: mach-pxa/ for PXA SoCs, mach-imx/ for Freescale iMX SoCs, etc. They essentially contain:
    - a small SoC description file
    - power management code
    - SMP code
- Some SoC families share some code, in directories named plat-*
- Device Tree source files are in arch/arm/boot/dts/<vendor>//

Before the Device Tree and ARM cleanup

root commit

- Until 2011, the ARM architecture wasn't using the Device Tree, and a large portion of SoC support was located in `arch/arm/mach-<soc>`.
- Each board supported by the kernel was associated to an unique *machine ID*.
- The entire list of *machine ID* can be downloaded at `https://www.arm.linux.org.uk/developer/machines/download.php` and one could freely register an additional one.
- The Linux kernel was defining a *machine structure* for each board, which associates the *machine ID* with a set of information and callbacks.
- The bootloader had to pass the *machine ID* to the kernel in a specific ARM register.

This way, the kernel knew what board it was booting on, and which init callbacks it had to execute.

The Device Tree and the ARM cleanup

- As the ARM architecture gained significantly in popularity, some major refactoring was needed.
- First, the Device Tree was introduced on ARM: instead of using C code to describe SoCs and boards, a specialized language is used.
- Second, many driver infrastructures were created to replace custom code in `arch/arm/mach-<soc>`:
  - The common clock framework in drivers/clk/
  - The pinctrl subsystem in drivers/pinctrl/
  - The irqchip subsystem in drivers/irqchip/
  - The clocksource subsystem in drivers/clocksource/
- The amount of code in `mach-<soc>` has now significantly reduced.

Adding support for a new ARM board

Provided the SoC used on your board is supported by the Linux kernel:

1. Create a *Device Tree* file in arch/arm/boot/dts/<vendor>/, generally named `<soc-name>-<board-name>.dts`, and make it include the relevant SoC `.dtsi` file.
   - Your Device Tree will describe all the SoC peripherals that are enabled, the pin muxing, as well as all the devices on the board.
2. Modify arch/arm/boot/dts/<vendor>/Makefile to make sure your Device Tree gets built as a *DTB* during the kernel build.
3. Tweak an existing configuration that matches your SoC and save it as `<board-name>_defconfig` in arch/arm/configs/
4. If needed, develop the missing device drivers for the devices that are on your board outside the SoC.

Understanding SoC support

- Let's consider another ARM platform here for the kernel side: the Marvell Armada 370/XP.
- For this platform, the core of SoC support is located in arch/arm/mach-mvebu/
- The `board-v7.c` file (see code on the next slide) contains the "*entry point*" of the SoC definition, the `DT_MACHINE_START` .. `MACHINE_END` definition:
  - Defines the list of platform compatible strings that will match this platform, in this case `marvell,armada-370-xp`. This allows the kernel to know which `DT_MACHINE` structure to use depending on the DTB that is passed at boot time.
  - Defines various callbacks for the platform initialization, the most important one being the `.init_machine` callback, running initialization code for the associated SoC.

arch/arm/mach-mvebu/board-v7.c (Linux 5.3)

root commit

```
static void __init mvebu_dt_init(void)
{
        if (of_machine_is_compatible("marvell,armadaxp"))
                i2c_quirk();
}

static void __init armada_370_xp_dt_fixup(void)
{
#ifdef CONFIG_SMP
        smp_set_ops(smp_ops(armada_xp_smp_ops));
#endif
}

static const char * const armada_370_xp_dt_compat[] __initconst = {
        "marvell,armada-370-xp",
        NULL,
};

DT_MACHINE_START(ARMADA_370_XP_DT, "Marvell Armada 370/XP (Device Tree)")
        .l2c_aux_val    = 0,
        .l2c_aux_mask   = ~0,
        .init_machine   = mvebu_dt_init,
        .init_irq       = mvebu_init_irq,
        .restart        = mvebu_restart,
        .reserve        = mvebu_memblock_reserve,
        .dt_compat      = armada_370_xp_dt_compat,
        .dt_fixup       = armada_370_xp_dt_fixup,
MACHINE_END
```

Components of minimal SoC support

root commit

Minimal SoC support consists of

- An SoC *entry point* file, arch/arm/mach-mvebu/board-v7.c
- At least one SoC `.dtsi` DT and one board `.dts` DT, in arch/arm/boot/dts/<vendor>/
- An interrupt controller driver, drivers/irqchip/irq-armada-370-xp.c
- A timer driver, drivers/clocksource/timer-armada-370-xp.c
- An earlyprintk implementation to get early messages from the console, arch/arm/Kconfig.debug and arch/arm/include/debug/
- A serial port driver in drivers/tty/serial/. For Armada 370/XP, the 8250 driver drivers/tty/serial/8250/ is used.

This allows to boot a minimal system up to user space, using a root filesystem in *initramfs*.

Extending minimal SoC support

Once minimal SoC support is in place, the following core components should be added:

- Support for the clocks. Usually requires some clock drivers, as well as DT representations of the clocks. See drivers/clk/mvebu/ for Armada 370/XP clock drivers.
- Support for pin muxing, through the *pinctrl* subsystem. See drivers/pinctrl/mvebu/ for the Armada 370/XP drivers.
- Support for GPIOs, through the *GPIO* subsystem. See drivers/gpio/gpio-mvebu.c for the Armada 370/XP GPIO driver.
- Support for SMP, through `struct smp_operations`. See arch/arm/mach-mvebu/platsmp.c.

Adding controller drivers

Once the core pieces of SoC support have been implemented, the remaining part is to add drivers for the different hardware blocks:

- Ethernet controller driver, in drivers/net/ethernet/marvell/mvneta.c
- SATA controller driver, in drivers/ata/sata_mv.c
- I2C controller driver, in drivers/i2c/busses/i2c-mv64xxx.c
- SPI controller driver, in drivers/spi/spi-orion.c
- PCIe controller driver, in drivers/pci/controller/pci-mvebu.c
- USB controller driver, in drivers/usb/host/ehci-orion.c
- etc.

Kernel debugging

Kernel Resources

Linux Weekly News

- https://lwn.net/
- The weekly digest off all Linux and free software information sources
- In depth technical discussions about the kernel
- Coverage of the features accepted in each merge window
- Subscribe to finance the editors ($9 / month)
- Articles available for non subscribers after 1 week.

Useful Online Resources

- Kernel documentation
  - https://kernel.org/doc/
- Linux kernel mailing list etiquette
  - https://subspace.kernel.org/etiquette.html
  - Read this before asking a question to the mailing list
- Linux kernel mailing lists archives
  - https://lore.kernel.org/
  - Easy browsing and referencing of all e-mail threads
  - Easy access to an mbox in order to answer to e-mails you were not Cc'ed to
- Kernel Newbies
  - https://kernelnewbies.org/
  - Articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.
  - Glossary: https://kernelnewbies.org/KernelGlossary
  - In depth coverage of the new features in each kernel release: https://kernelnewbies.org/LinuxChanges
- The https://elinux.org wiki

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.

Embedded Linux Conference

International Conferences (2)

- Kernel Recipes: https://kernel-recipes.org/
  - Well attended conference in Europe (Paris), only one track at a time, with a format that really allows for discussions.
- FOSDEM: https://fosdem.org/
  - Huge conference in Brussels, free of cost, with kernel and embedded "devrooms".
- Most conferences are available on-line too.
  They are much more affordable and sometimes free.

After the course

Continue to learn:

- Run your labs again on your own hardware. The Gamepad lab should be rather straightforward, but the serial lab will be quite different if you use a different processor.
- Learn by reading the kernel code by yourself, ask questions and propose improvements.
- Implement and share drivers for your own hardware, of course!

Hobbyists can make their first contributions by:

- Helping with tasks keeping the kernel code clean and up-to-date: https://kernelnewbies.org/KernelJanitors/Todo
- Proposing fixes for issues reported by the *Coccinelle* tool: `make coccicheck`
- Participating to improving drivers in drivers/staging/
- Investigating and do the triage of issues reported by Coverity Scan: https://scan.coverity.com/projects/linux

Contribute your changes

root commit

Recommended resources

- See `process/submitting-patches` for guidelines and `https://kernelnewbies.org/UpstreamMerge` for very helpful advice to have your changes merged upstream (by Rik van Riel).
- Watch the *Write and Submit your first Linux kernel Patch* talk by Greg. K.H: `https://www.youtube.com/watch?v=LLBrBBImJt4`

Thank you!

Here are all courses from Root Commit if you are interested

- Embedded Linux Training
  https://rootcommit.com/training/embedded-linux/
- Linux Kernel, Board Support and Driver Development Training
  https://rootcommit.com/training/linux-kernel/
- Yocto Project and OpenEmbedded Training
  https://rootcommit.com/training/yocto/
- Linux Boot Time Reduction Training
  https://rootcommit.com/training/boot-time/