



## Linux kernel and driver development training

© Copyright 2004-2023, Bootlin.  
Creative Commons BY-SA 3.0 license.  
Latest update: July 28, 2023.

Document updates and training details:  
<https://bootlin.com/training/kernel>

Corrections, suggestions, contributions and translations are welcome!  
Send them to [feedback@bootlin.com](mailto:feedback@bootlin.com)





# Linux kernel and driver development training

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



Icon by Eucalyp, Flaticon

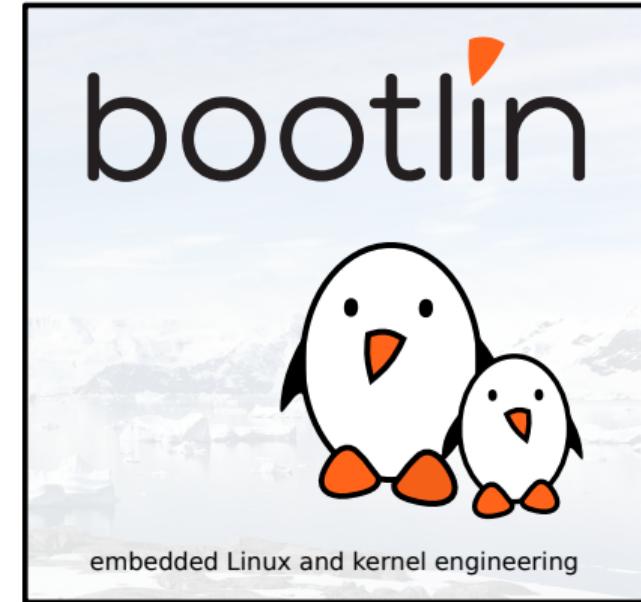


## About Bootlin

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# Bootlin introduction

---

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





# Bootlin engineering services

Bootloader /  
firmware  
development

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

Linux kernel  
porting and  
driver  
development

Linux BSP  
development,  
maintenance  
and upgrade

Embedded Linux  
build systems

Yocto, OpenEmbedded,  
Buildroot, ...

Embedded Linux  
integration

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

Open-source  
upstreaming

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



# Bootlin training courses

## Embedded Linux system development

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

## Linux kernel driver development

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

## Yocto Project system development

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

## Buildroot system development

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

## Understanding the Linux graphics stack

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

## Embedded Linux boot time optimization

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

## Real-Time Linux with PREEMPT\_RT

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

## Linux debugging, tracing, profiling and performance analysis

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



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



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



*Icon by Freepik, Flaticon*

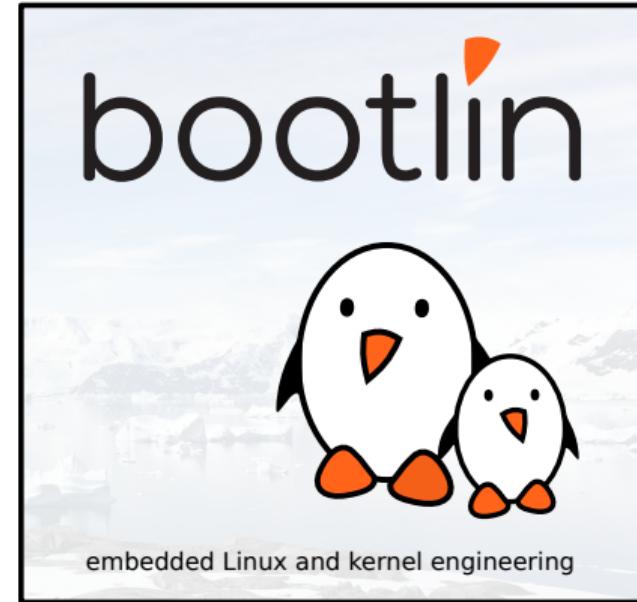


## Generic course information

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# Supported hardware

BeagleBone Black or BeagleBone Black Wireless, from [BeagleBoard.org](https://beagleboard.org)

- ▶ Texas Instruments AM335x (ARM Cortex-A8 CPU)
- ▶ SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- ▶ 512 MB of RAM
- ▶ 4 GB of on-board eMMC storage
- ▶ USB host and USB device, microSD, micro HDMI
- ▶ WiFi and Bluetooth (wireless version), otherwise Ethernet
- ▶ 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- ▶ A huge number of expansion boards, called *capes*. See [https://elinux.org/Beagleboard:BeagleBone\\_Capes](https://elinux.org/Beagleboard:BeagleBone_Capes).





# Shopping list: hardware for this course

- ▶ BeagleBone Black or BeagleBone Black Wireless - Multiple distributors:  
See <https://beagleboard.org/Products/>.
- ▶ USB Serial Cable - 3.3 V - Female ends (for serial console)<sup>1</sup>
- ▶ Nintendo Nunchuk with UEXT connector<sup>2</sup>
- ▶ Breadboard jumper wires - Male ends (to connect the Nunchuk)<sup>3</sup>
- ▶ USB Serial Cable - 3.3 V - Male ends (for serial labs, two if possible):  
Olimex: <https://frama.link/BEGcpgo7>
- ▶ Note that both USB serial cables are the same.  
Only the gender of their connector changes.



<sup>1</sup> <https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/>

<sup>2</sup> <https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/>

<sup>3</sup> <https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/>



## Training quiz and certificate

---

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



# Participate!

---

During the lectures...

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



# Collaborate!

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

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

The screenshot shows a Matrix channel interface with several messages from users Srinath and michael.o. Srinath asks about the CROSS\_COMPILE variable for the Xplained board. Michael.o responds with instructions on setting it correctly. Srinath also mentions asking about labs after the session is over. A message from arnsud.i is visible at the bottom, and a file attachment 'Decrypt Image.png' is shown.

E # embedded-linux-nov2020 Channel for

Srinath

michael.o: What should be CROSS\_COMPILE variable set to in case of the Xplained board? I ran into some issues with my USB hub so doing the u-boot again

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

2 messages deleted.

Srinath

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

michael.o changed their display name to michael.r.

@bootlin-eu@matrix.org

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

arnsud.i

I had the same because I accidentally removed the .overlays from the kernel

arnsud.i: Send an encrypted message...

Decrypt Image.png (109.2 KB)



# Practical lab - Training Setup

---



Prepare your lab environment

- ▶ Download and extract the lab archive

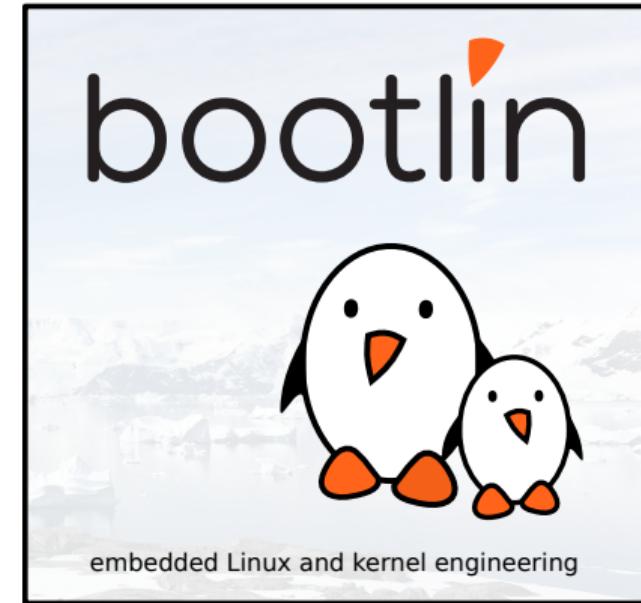


# Linux Kernel Introduction

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





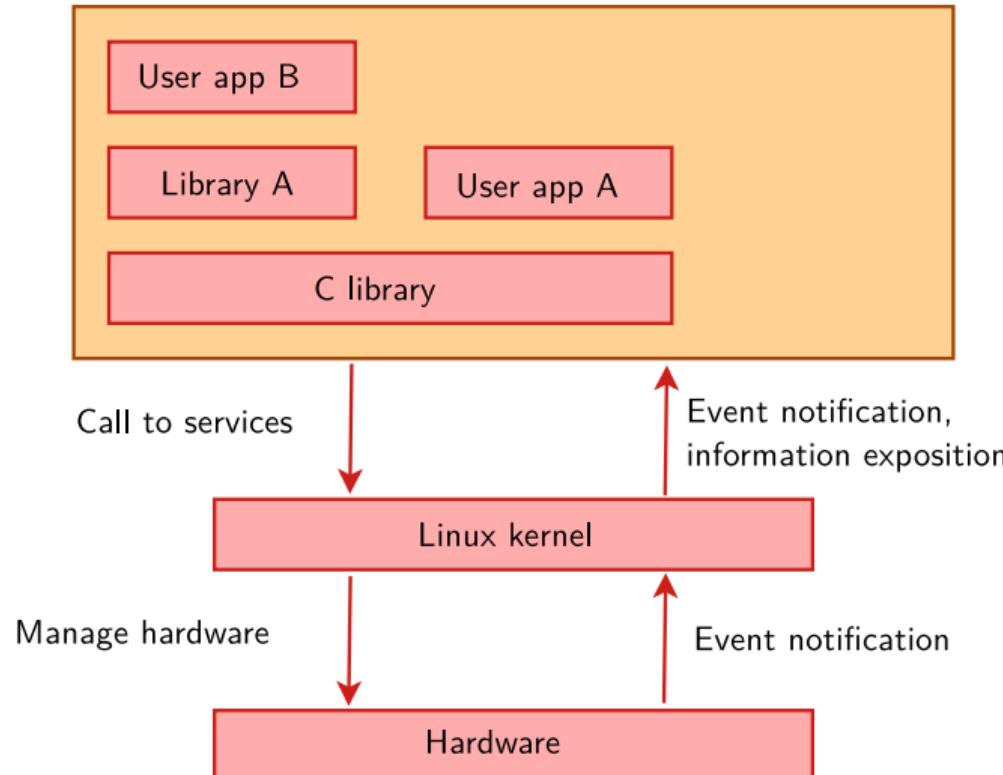
- ▶ The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- ▶ Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- ▶ As of today, about 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





- ▶ **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 system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

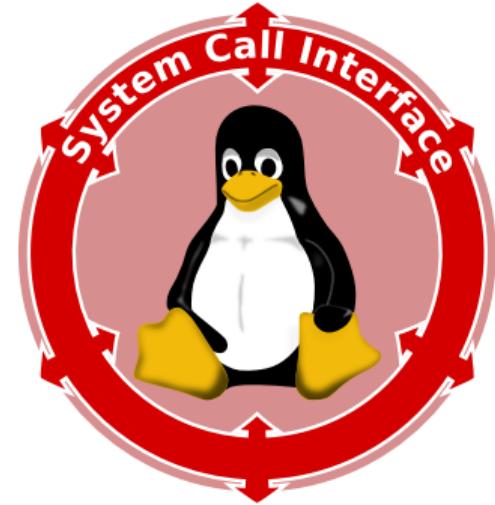


Image credits (Wikipedia):  
<https://bit.ly/2U2rdGB>

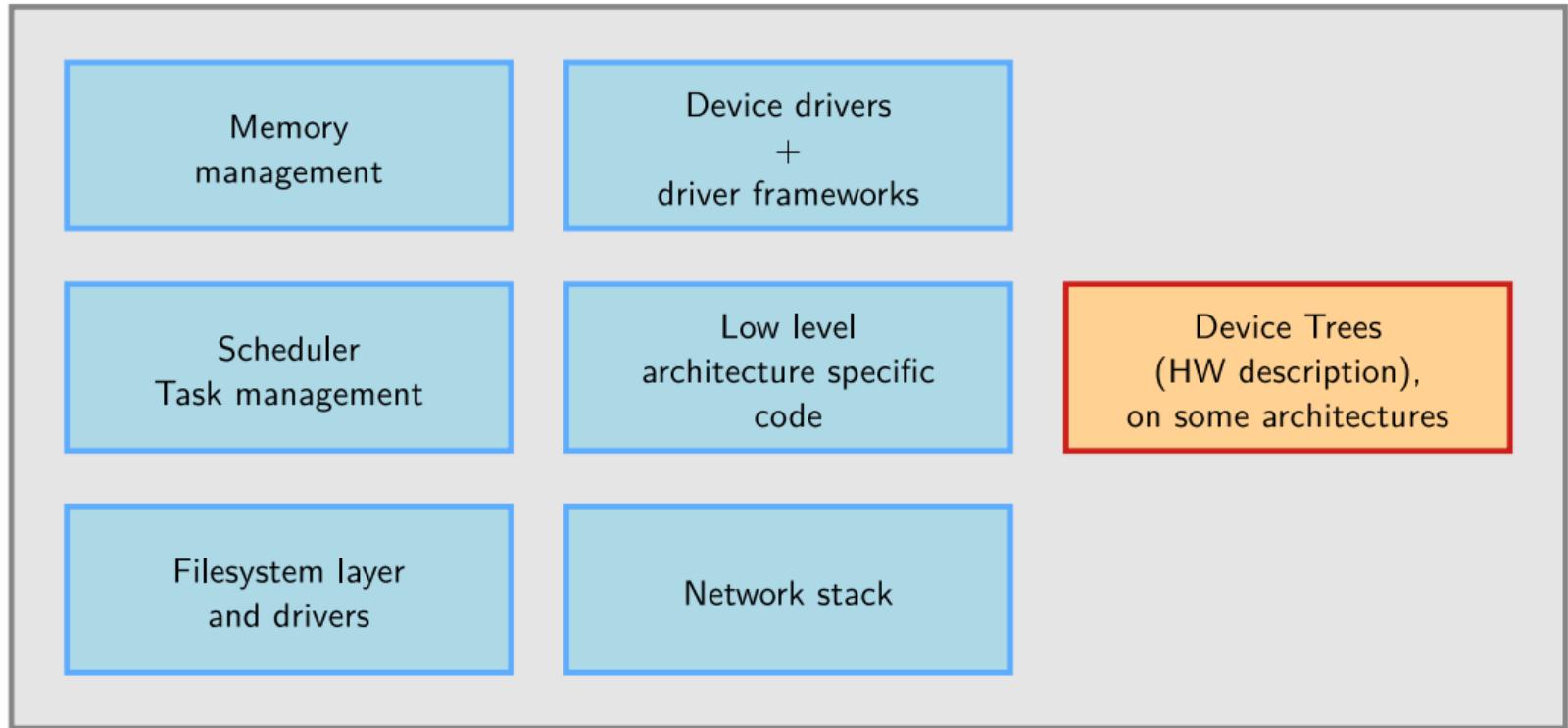


# Pseudo filesystems

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



## Linux Kernel





## 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
  - 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

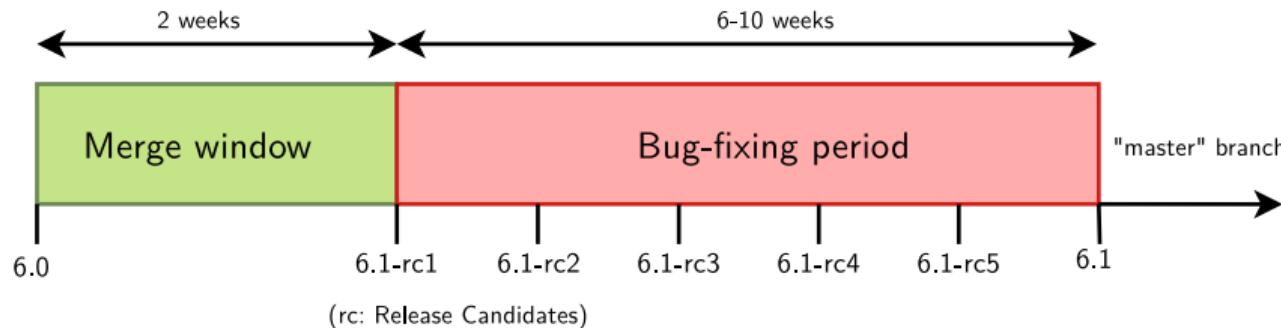
---

- ▶ 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)
  - Version 6.0 was released in Oct. 2022.
- ▶ Features are added to the kernel in a progressive way. Since 2003, kernel developers have managed to do so without having to introduce a massively incompatible development branch.



# Linux development model

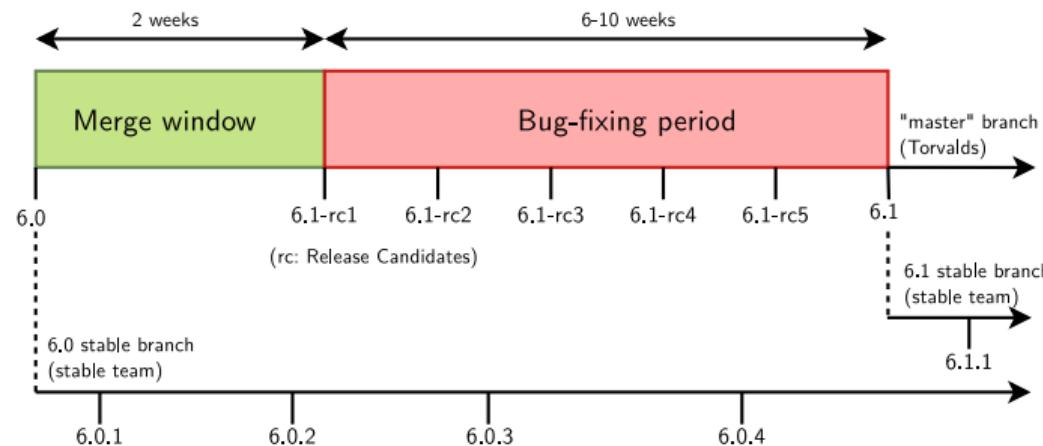
- ▶ Each new release starts with a two-week merge window for new features
- ▶ Follow about 8 release candidates (one week each)
- ▶ Until adoption of a new official release.





# 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.





## 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>
  - Certain 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
6.1	Greg Kroah-Hartman & Sasha Levin	2022-12-11	Dec, 2026
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Oct, 2026
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2025
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2024
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2024

Captured on <https://kernel.org> in Apr. 2023, following the *Releases* link.

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



## Need for 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) on selected architectures. See <https://wiki.linuxfoundation.org/civilinfrastructureplatform/cipkernelmaintenance>.



# Location of non-official kernel sources

---

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



# Linux kernel size and structure

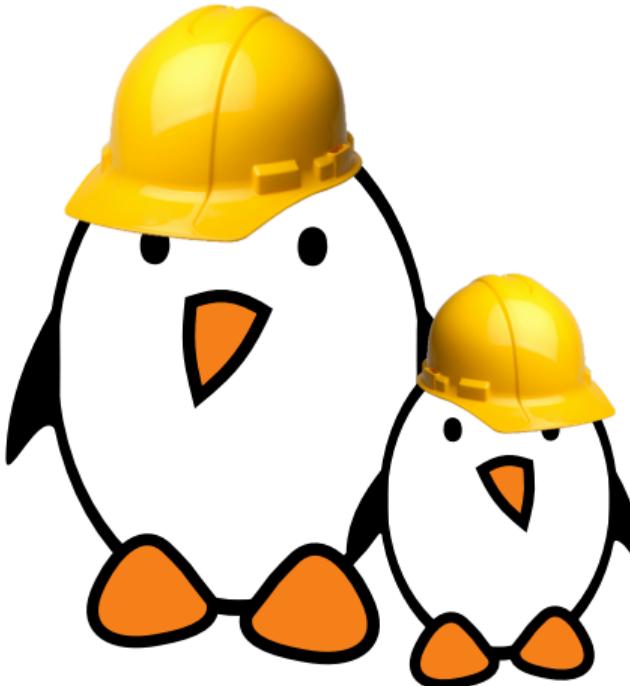
- ▶ Linux v5.18 sources: close to 80k files, 35M lines, 1.3GiB
- ▶ But a compressed Linux kernel just sizes a few megabytes.
- ▶ So, why are these sources so big?

Because they include numerous device drivers, network protocols, architectures, filesystems... The core is pretty small!
- ▶ As of kernel version v5.18 (in percentage of total number of lines):

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



## Practical lab - Downloading kernel source code



- ▶ Clone the mainline Linux source tree with git



## 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-10.2.0/gcc/C-Extensions.html>
- ▶ A subset of the supported architectures can be built with the LLVM C compiler (Clang) too: <https://clangbuiltlinux.github.io/>
- ▶ Rust support is currently being introduced:  
<https://lwn.net/Articles/829858/>, according to Miguel Ojeda (02/2023): "getting closer to a point where the first Rust modules can be upstreamed".



- ▶ 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()`, ...



- ▶ 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.

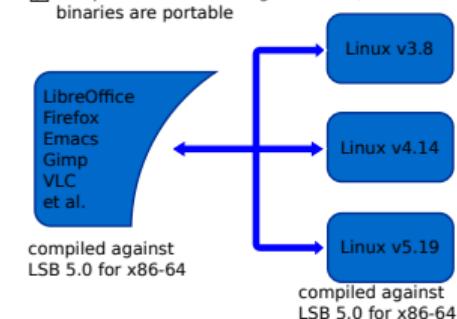
## Linux kernel to user API

API stability **is** guaranteed, source code is portable!



## Linux kernel to user ABI

compatible ABI **can be** guaranteed, binaries are portable



Modified Image from Wikipedia:

<https://bit.ly/2U2rdGB>



# Linux internal API/ABI instability

## Linux internal API is not stable

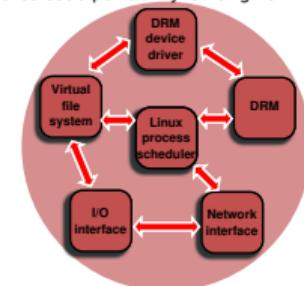
- ▶ The source code of a driver is not portable across versions
  - In-tree drivers are updated by the developer proposing the API change: works great for mainline code
  - An out-of-tree driver compiled for a given version may no longer compile or work on a more recent one
  - See [process/stable-api-nonsense](#)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

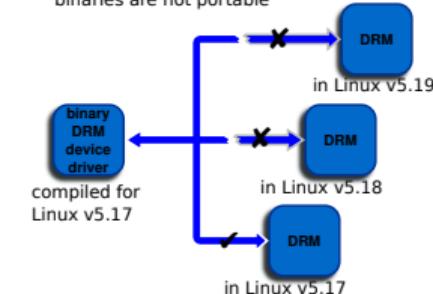
### Linux internal API

☒ API stability is not guaranteed, source code portability is not given



### Linux internal ABI

☒ no stable ABI over Linux kernel releases, binaries are not portable



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



## Kernel memory constraints

---

- ▶ No memory protection
- ▶ The kernel doesn't try to recover from attempts 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

- ▶ 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](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

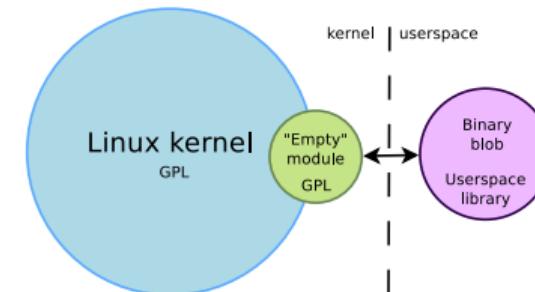
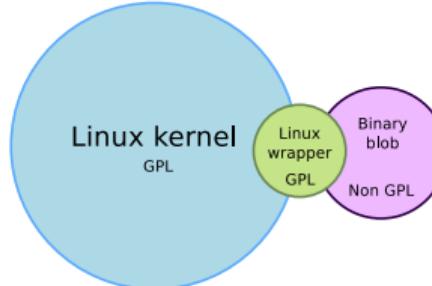
---

- ▶ 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

---

- ▶ 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

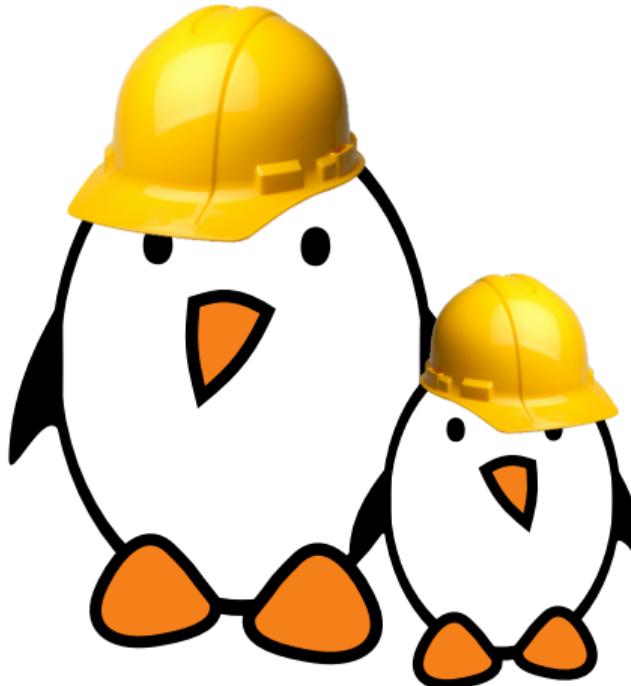
- ▶ The kernel provides certain mechanisms to access hardware directly from userspace:
  - USB devices with *libusb*, <https://libusb.info/>
  - SPI devices with *spidev*, [spi/spidev](#)
  - I2C devices with *i2cdev*, [i2c/dev-interface](#)
  - Memory-mapped devices with *UIO*, including interrupt handling, [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.
- ▶ Certain 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 architected. Kernel drivers should always be preferred!



- ▶ 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.



# Practical lab - Kernel Source Code - Exploring



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



# Linux Kernel Usage

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Kernel configuration



# Kernel configuration

---

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



# Kernel configuration and build system

---

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



# Specifying the target architecture

First, specify the architecture for the kernel to build

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



# Choosing a compiler

---

The compiler invoked by the kernel Makefile is \$(CROSS\_COMPILE)gcc

- ▶ Specifying the compiler is already needed at configuration time, as some kernel configuration options depend on the capabilities of the compiler.
- ▶ When compiling natively
  - Leave CROSS\_COMPILE undefined and the kernel will be natively compiled for the host architecture using gcc.
- ▶ When using a cross-compiler
  - Specify the prefix of your cross-compiler executable, for example for arm-linux-gnueabi-gcc:  
CROSS\_COMPILE=arm-linux-gnueabi-

Set LLVM to 1 to compile your kernel with Clang.

See our [LLVM tools for the Linux kernel](#) presentation.



# Specifying ARCH and CROSS\_COMPILE

There are actually two ways of defining ARCH and CROSS\_COMPILE:

- ▶ Pass ARCH and CROSS\_COMPILE on the make command line:

```
make ARCH=arm CROSS_COMPILE=arm-linux- ...
```

Drawback: it is easy to forget to pass these variables when you run any make command, causing your build and configuration to be screwed up.

- ▶ Define ARCH and CROSS\_COMPILE as environment variables:

```
export ARCH=arm
```

```
export CROSS_COMPILE=arm-linux-
```

Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project, see also the <https://direnv.net/> project.



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

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



## Create your own default configuration

---

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



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



# Kernel option types

---

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

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



# Kernel option dependencies

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

- ▶ depends on dependency:

```
config B  
    depends on A
```

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

- ▶ select dependency:

```
config A  
    select B
```

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

```
config SPI_ATH79  
    tristate "Atheros AR71XX/AR724X/AR913X SPI controller driver"  
    depends on ATH79 || COMPILE_TEST  
    select SPI_BITBANG  
    help
```

This enables support for the SPI controller present on the Atheros AR71XX/AR724X/AR913X SoCs.



# Kernel configuration details

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

```
#  
# CD-ROM/DVD Filesystems  
#  
CONFIG_ISO9660_FS=m  
CONFIG_JOLIET=y  
CONFIG_ZISOFS=y  
CONFIG_UDF_FS=y  
# end of CD-ROM/DVD Filesystems  
  
#  
# DOS/FAT/EXFAT/NT Filesystems  
#  
CONFIG_FAT_FS=y  
CONFIG_MSdos_FS=y  
# CONFIG_VFAT_FS is not set  
CONFIG_FAT_DEFAULT_CODEPAGE=437  
# CONFIG_EXFAT_FS is not set
```



## make xconfig

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

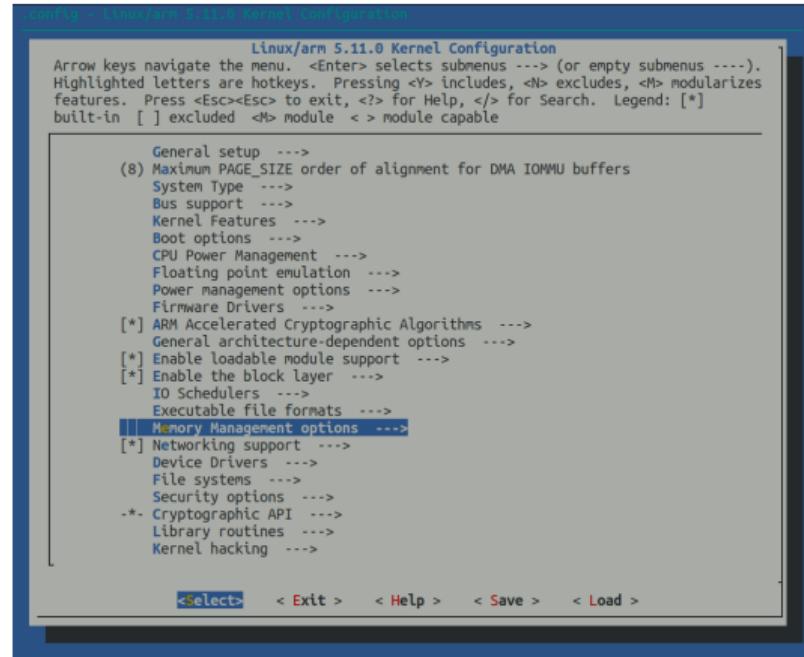




# menuconfig

make menuconfig

- ▶ Useful when no graphics are available.  
Very efficient interface.
- ▶ Same interface found in other tools:  
BusyBox, Buildroot...
- ▶ Convenient number shortcuts to jump  
directly to search results.
- ▶ Required Debian/Ubuntu packages:  
libncurses-dev





# 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

Compiled as a module:

`CONFIG_IS09660_FS=m`

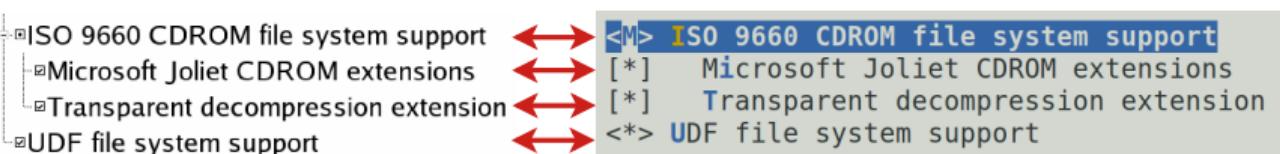
Additional driver options:

`CONFIG_JOLIET=y`

`CONFIG_ZISOFS=y`

Statically built:

`CONFIG_UDF_FS=y`



Values in resulting .config file

Parameter values as displayed by xconfig

Parameter values as displayed by menuconfig



## make oldconfig

---

### make oldconfig

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

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



# Undoing configuration changes

---

A frequent problem:

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



## Compiling and installing the kernel



# Kernel compilation

make

- ▶ Only works from the top kernel source directory
- ▶ Should not be performed as a privileged user
- ▶ Run several jobs in parallel. Our advice: `ncpus * 2` to fully load the CPU and I/Os at all times.

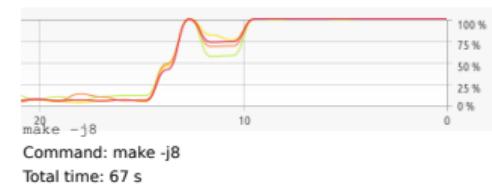
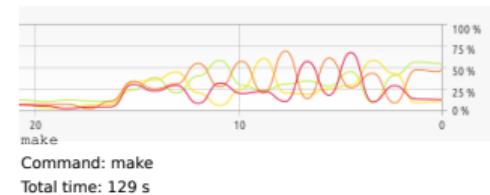
Example: `make -j 8`

- ▶ To recompile faster (7x according to some benchmarks), use the ccache compiler cache:

```
export CROSS_COMPILE="ccache arm-linux-"
```

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

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





# Kernel compilation results

---

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



## Kernel installation: native case

---

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



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



## Module installation: native case

---

- ▶ `sudo make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- ▶ Installs all modules in `/lib/modules/<version>/`
  - `kernel/`  
Module .ko (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias, modules.alias.bin`  
Aliases for module loading utilities , 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 modules that are builtin the kernel.



# Module alias: *modules.alias*

## Kernel compiling

```
static const struct usb_device_id products [] = {  
    {  
        .idVendor = 0x077b,  
        .idProduct = 0x2226,  
        .driver_info = (unsigned long) &ax8817x_info,  
    }, {  
        .idVendor = 0x0846,  
        .idProduct = 0x1040,  
        .driver_info = (unsigned long) &netgear_fa120_info,  
    }, ...  
}; // END  
MODULE_DEVICE_TABLE(usb, products);  
  
drivers/net/usb/asix_devices.c
```

The device driver source code lists which devices it supports

make modules

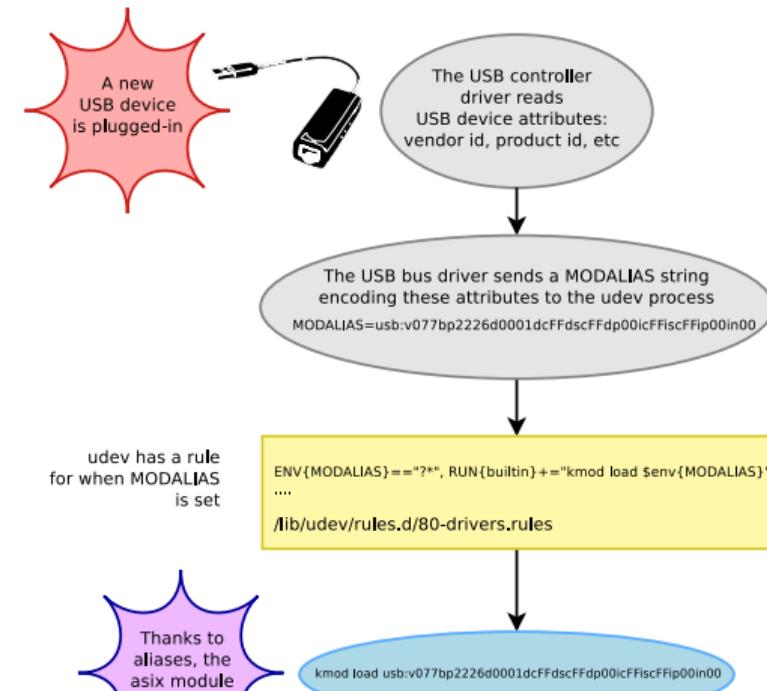
Module file  
asix.ko

Containing the list of supported devices (module metadata)

make modules\_install  
(depmod)

```
alias usb:v077Bp2226d*dc*dsc*d*ic*isc*ip*in* asix  
alias usb:v0846p1040d*dc*dsc*d*ic*isc*ip*in* asix  
...  
modules.alias
```

## System operation





## Module installation: embedded case

---

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

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



# Kernel cleanup targets

- ▶ From make help:

Cleaning targets:

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

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





# Kernel building overview

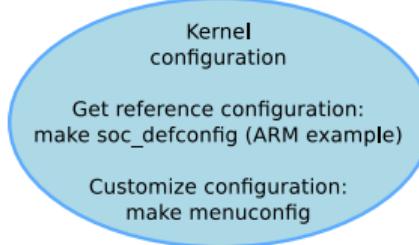
## Environment setup and configuration

Specify target architecture  
(if different from host)

```
export ARCH=arm
```

Specify cross-compiler  
(if cross-compiling)

```
export CROSS_COMPILE=arm-linux-
```



## Kernel building and deployment

Kernel  
compiling  
`make`

Installing modules  
`make modules_install`

Installing the kernel  
`make install`  
or manual copy



## Booting the kernel



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



# Booting with U-Boot

---

- ▶ On ARM32, U-Boot can boot `zImage`, on ARM64 or RISC-V, it boots the `Image` file
- ▶ In addition to the kernel image, U-Boot should also pass a DTB to the kernel.
- ▶ The typical boot process is therefore:
  1. Load `zImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `bootz X - Y`

The `-` in the middle indicates no *initramfs*



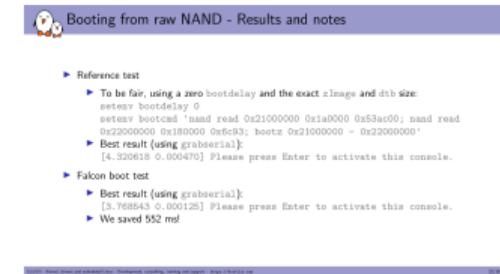
- ▶ In addition to the compile time configuration, the kernel behavior can be adjusted with no recompilation using the **kernel command line**
- ▶ The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root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



# Passing the kernel command line

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

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



Booting from raw NAND - Results and notes

```
▶ Reference test
▶ To be fair, using a zero bootdelay and the exact zImage and dtb size:
setenv bootdelay 0
setenv bootargs zImage read 0x21000000 0x1a0000 0x53ac00; nand read
0x2000000 0x1800000 0xfcfc93; bootz 0x21000000 - 0x22000000
▶ Best result (using grabserial):
[4.320619 0.000470] Please press Enter to activate this console.
▶ Falcon boot test
▶ Best result (using grabserial):
[3.768643 0.000125] Please press Enter to activate this console.
▶ We saved 552 ms!
```

Slides: <https://bootlin.com/pub/conferences/2021/lee/>  
Video: <https://www.youtube.com/watch?v=LFe3x2QMHSo>



- ▶ 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 (**diagnostic message**)
- ▶ 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 to be displayed on the console
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 loglevel=5`
- ▶ It is possible to write to the kernel log from user space:  
`echo "<n>Debug info" > /dev/kmsg`



# Practical lab - Kernel compiling and booting



1st lab: board and bootloader setup:

- ▶ Prepare the board and access its serial port
- ▶ Configure its bootloader to use TFTP

2nd lab: kernel compiling and booting:

- ▶ Set up a cross-compiling environment
- ▶ Cross-compile a kernel for an ARM target platform
- ▶ Boot this kernel from a directory on your workstation, accessed by the board through NFS



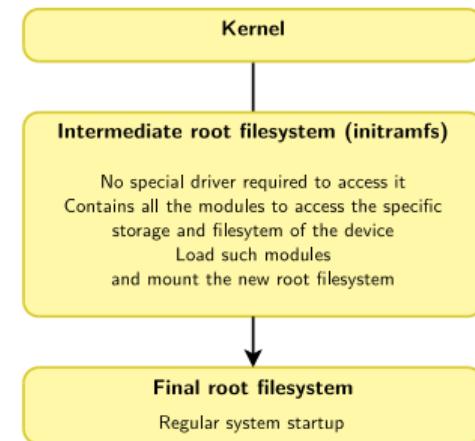
## Using kernel modules



# 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 object 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`



# Developing kernel modules

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# Hello module 1/2

```
// 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://frama.link/Q3CNXnom>



# 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()`

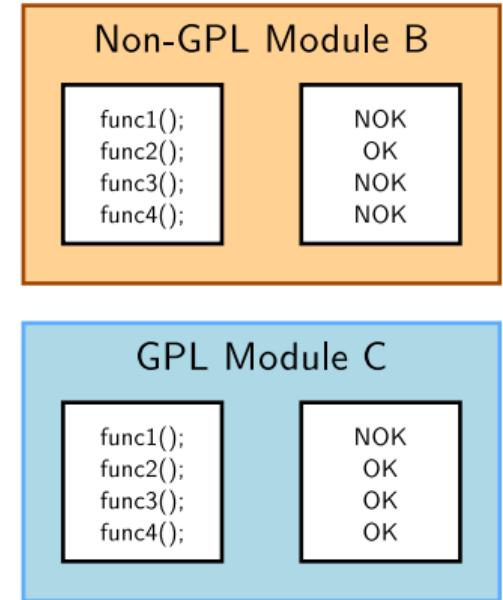
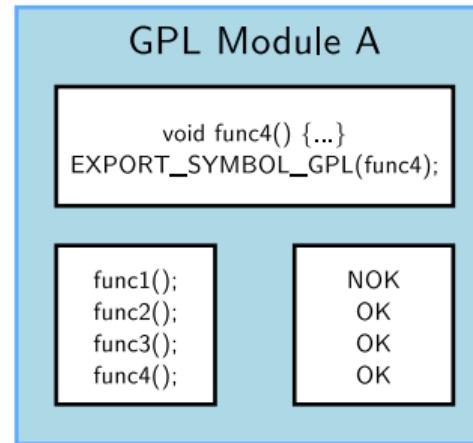
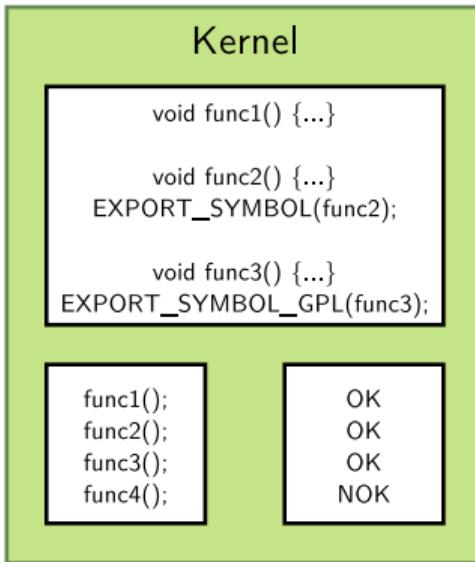


## Symbols exported to modules 1/2

- ▶ 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 5.3: contains the same number of symbols with `EXPORT_SYMBOL()` and symbols with `EXPORT_SYMBOL_GPL()`
- ▶ A normal driver should not need any non-exported function.



# Symbols exported to modules 2/2





- ▶ 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

- ▶ 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

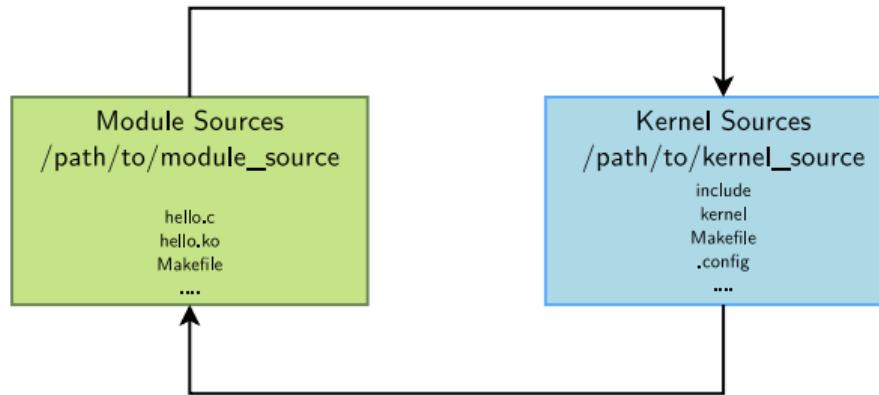
```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

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.



- ▶ 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

## ► 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

- ▶ 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

---

```
// 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

```
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://github.com/bootlin/training-materials/blob/master/code/hello-param/hello\\_param.c](https://github.com/bootlin/training-materials/blob/master/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");
```

Modules parameter arrays are also possible with `module_param_array()`.



## Practical lab - Writing modules

---



- ▶ Create, compile and load your first module
- ▶ Add module parameters
- ▶ Access kernel internals from your module

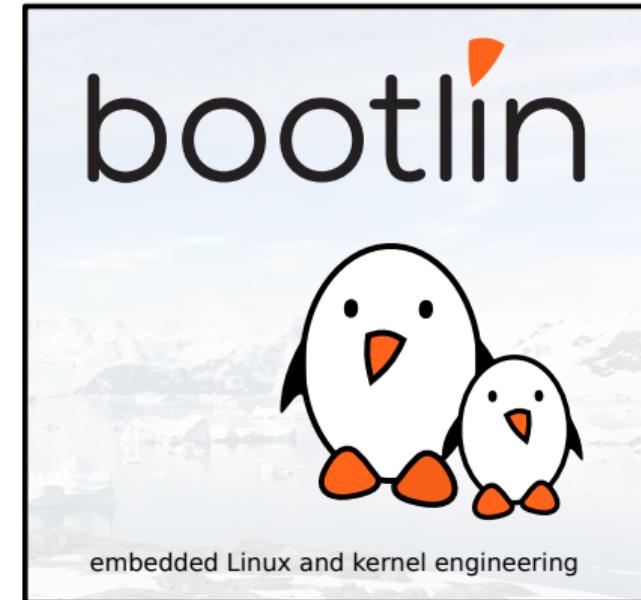


## Describing hardware devices

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## 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.



# Describing non-discoverable hardware



## Non-discoverable buses

---

- ▶ On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- ▶ For example, the devices on I2C buses or SPI buses, or the devices directly part of the system-on-chip.
- ▶ However, we still want all of these devices to be part of the device model.
- ▶ Such devices, instead of being dynamically detected, must be statically described.



# Describing non-discoverable hardware

---

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



# Describing non-discoverable hardware

---

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



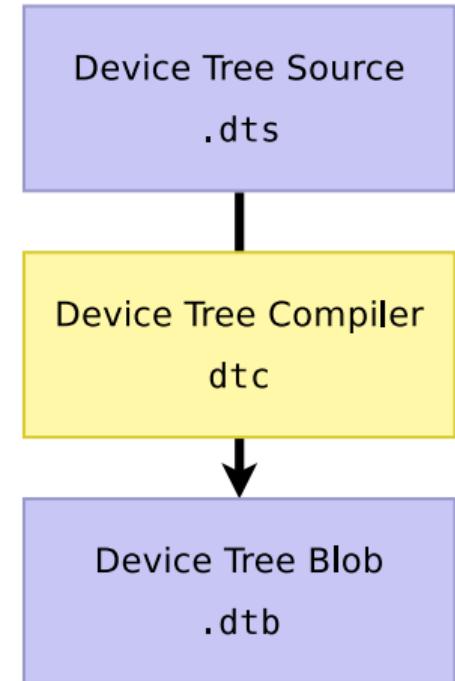
## 3. Using a Device Tree

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



# Device Tree: from source to blob

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





## dtc example

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

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



## dtc example

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

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

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



## dtc example

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

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

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

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

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



# Where are Device Tree Sources located?

---

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



# Device Tree base syntax

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

The diagram illustrates the base syntax of a Device Tree (DT) node. A node is defined by a slash character (/) followed by a node name, a unit address, and a brace block. The brace block contains properties and child nodes. Properties are labeled with their names and values. Child nodes are also labeled with their names and unit addresses. A label is shown with a red arrow pointing to it. Annotations explain various parts:

- Properties of node@0**: Points to the properties block.
- Node name**: Points to the first part of the node definition after the slash.
- Unit address**: Points to the second part of the node definition after the slash.
- Property name**: Points to the name of a property.
- Property value**: Points to the value of a property.
- Bytestring**: Points to a property value represented as a sequence of bytes.
- A phandle (reference to another node)**: Points to a reference to another node.
- Four cells (32 bits values)**: Points to a property value represented as four cells.

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>

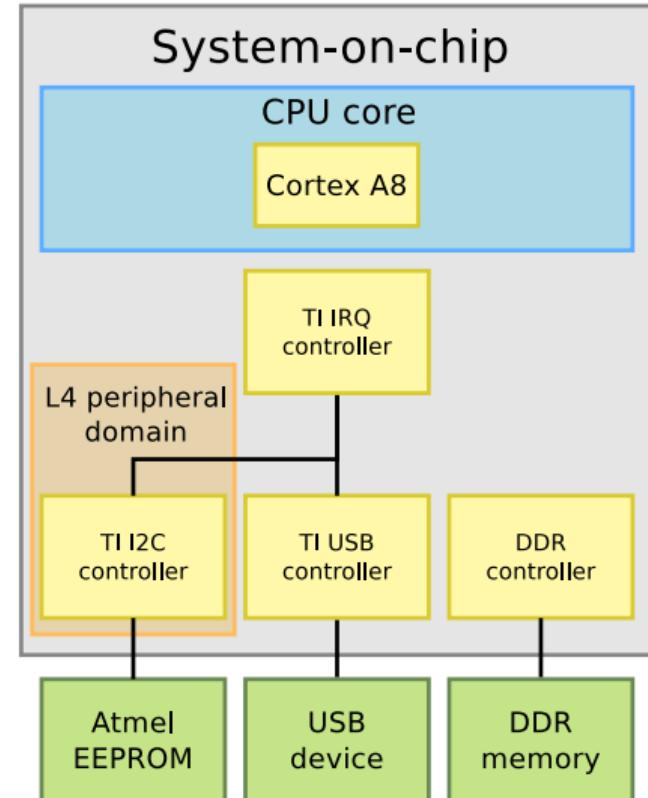
        child-node@0 {
        };
    };
};
```



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





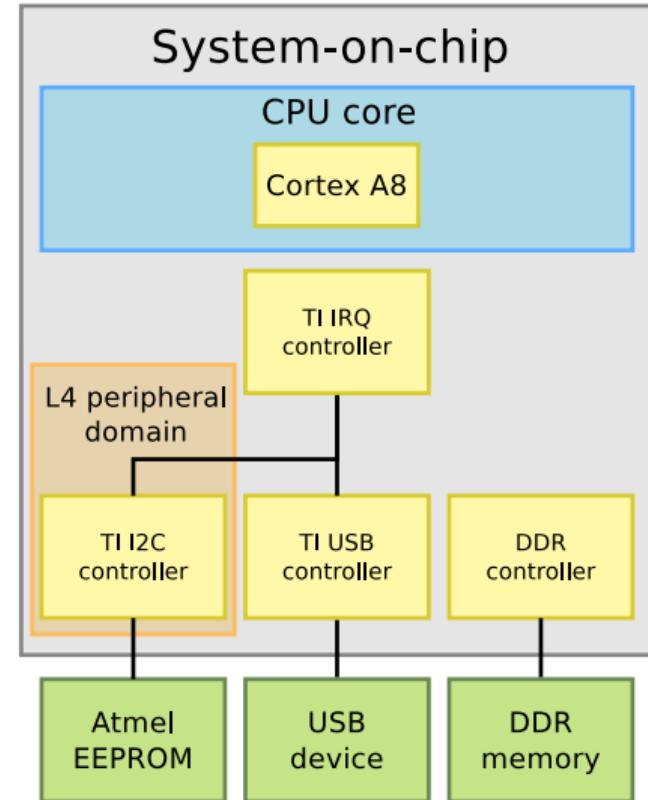
# DT overall structure: simplified example

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





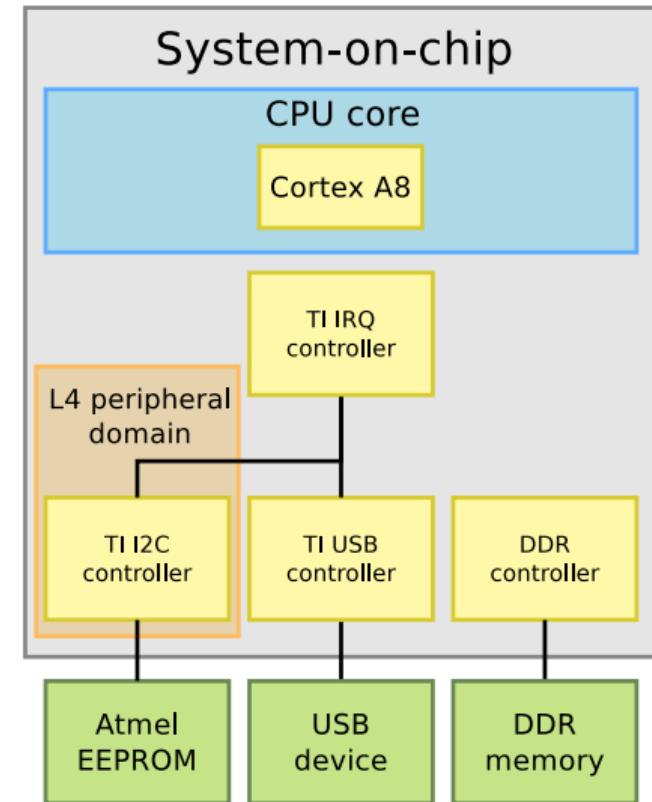
# DT overall structure: simplified example

```
/ {
    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";
            dmams = <&cppi41dma 0 0 &cppi41dma 1 0 ...>;
            status = "okay";
        };

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





# DT overall structure: simplified example

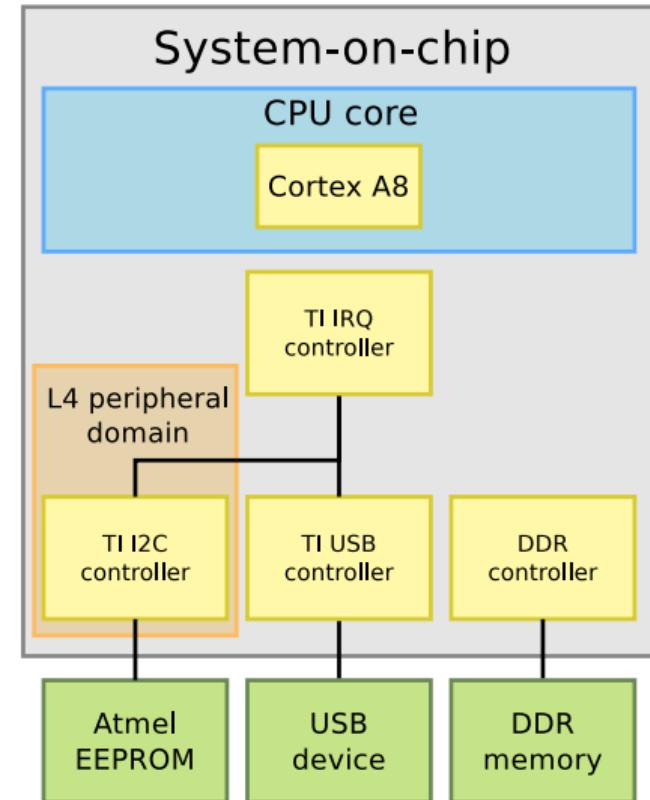
```
/ {
    cpus { ... };
    memory@0x80000000 { ... };
    chosen { ... };

    ocp {
        compatible = "simple-pm-bus";
        clocks = <&l3_clkctrl AM3_L3_L3_MAIN_CLKCTRL 0>;
        clock-names = "fck";
        #address-cells = <1>;
        #size-cells = <1>

        intc: interrupt-controller@48200000 { ... };
        usb0: usb@47401300 { ... };

        l4_per: interconnect@44c00000 {
            compatible = "ti,am33xx-l4-wkup", "simple-pm-bus";
            reg = <0x44c00000 0x800>, <0x44c00800 0x800>,
                  <0x44c01000 0x400>, <0x44c01400 0x400>;
            reg-names = "ap", "la", "ia0", "ia1";
            #address-cells = <1>;
            #size-cells = <1>

            i2c0: i2c@40012000 { ... };
        };
    };
}
```



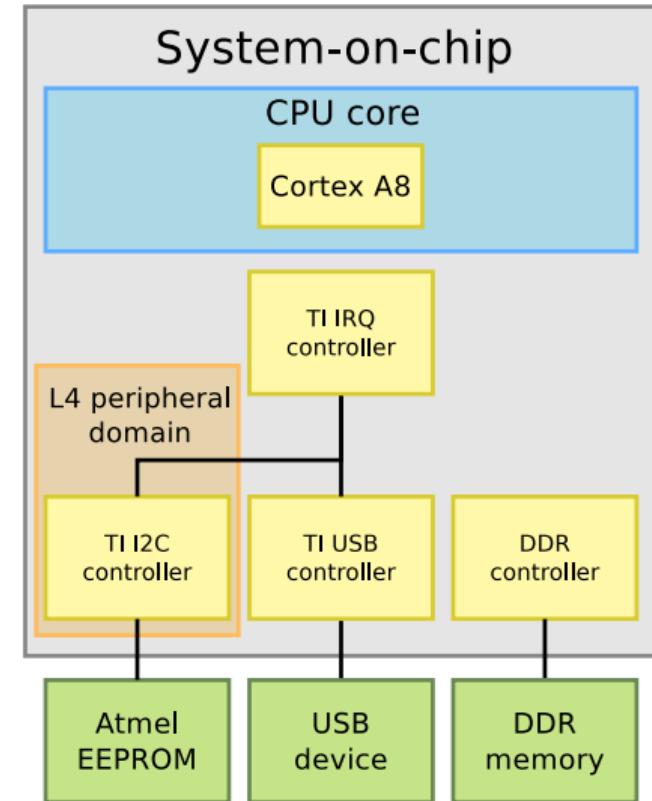


# DT overall structure: simplified example

```
/ {
    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 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 **overlaid** the tree of the including file over the tree of the included file, according to the order of the #include directives.
- ▶ Allows an including file to **override** values specified by an included file.
- ▶ Uses the C pre-processor #include directive



# Device Tree inheritance example

Definition of the AM33xx SoC family

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



am33xx-14.dtsi

Definition of the Bone Black board

```
#include "am33xx-14.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`

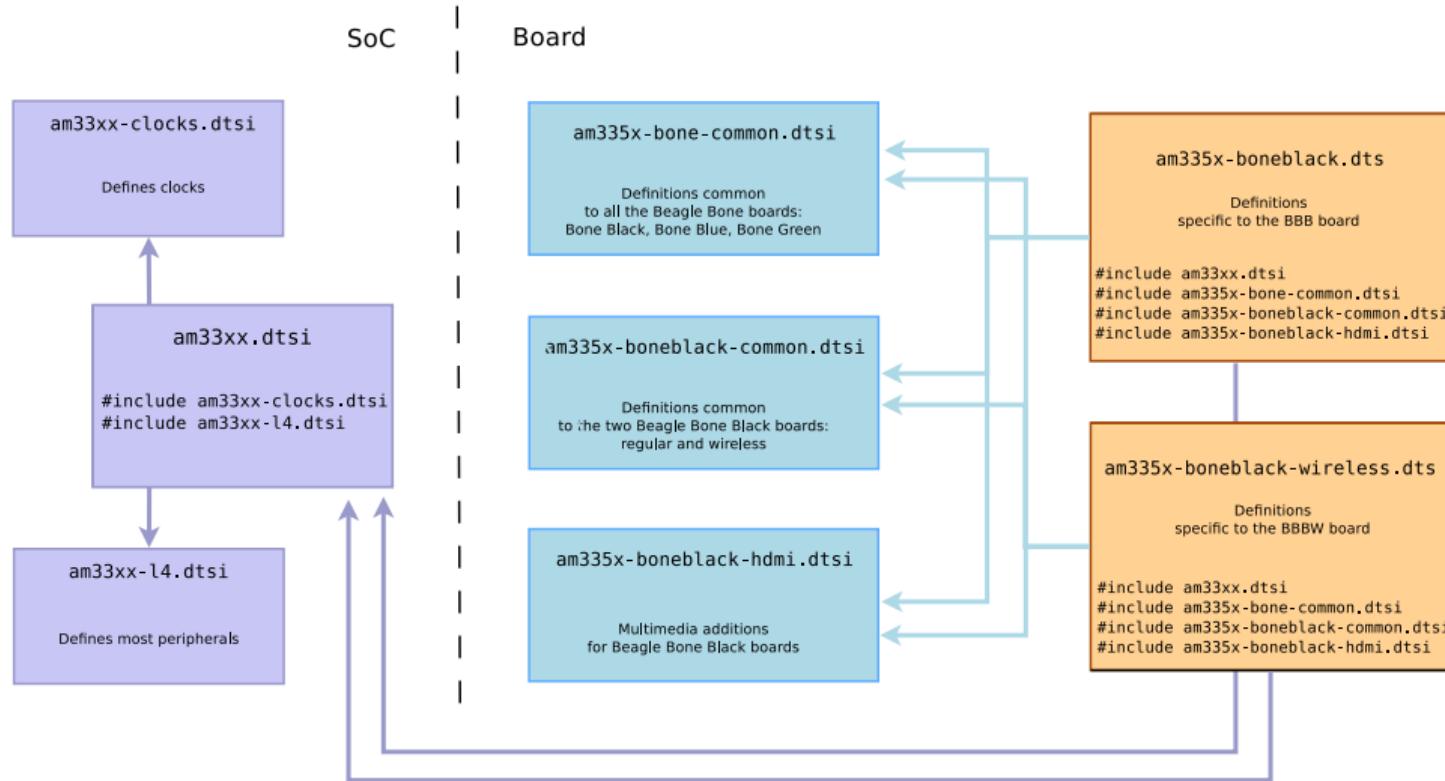
```
#include "soc.dtsi"

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

→ this solution is now often preferred



# DT inheritance in Bone Black support





# Device Tree design principles

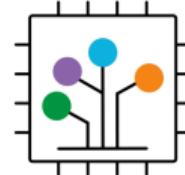
---

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



# Device Tree specifications

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



Devicetree Specification  
Release v0.3

[devicetree.org](http://devicetree.org)

13 February 2020



# Device Tree binding: legacy style

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

## I2C for OMAP platforms

-Required properties :

- compatible : Must be
  - "ti,omap240-i2c" for OMAP240 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

Documentation/devicetree/bindings/i2c/ti,omap4-i2c.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 Tree in Linux

---

- ▶ dtc only does syntactic validation
- ▶ YAML bindings allow to do semantic validation
- ▶ Linux kernel make rules:
  - make dt\_binding\_check  
verify that YAML bindings are valid
  - make dtbs\_check  
validate DTs currently enabled against YAML bindings
  - make DT\_SCHEMA\_FILES=Documentation/devicetree/bindings/trivial-devices.yaml dtbs\_check  
validate DTs against a specific YAML binding



# The properties

---

Device tree properties can be:

- ▶ Fully generic
  - Their meaning is usually described in one place: the core DT schema available at <https://github.com/devicetree-org/dt-schema>.
  - compatible, reg, #address-cells, etc
- ▶ Subsystem specific and cover generic consumer bindings
  - Their meaning is either described in the dt-schema Github or under [Documentation/device-tree/bindings](#).
  - clocks, interrupts, regulators, etc
- ▶ Subsystem specific
  - All devices of a certain class may use them, often starting with the class name
  - spi-cpha, i2c-scl-internal-delay-ns, nand-ecc-engine, mac-address, etc
- ▶ Vendor/device specific
  - To describe uncommon or very specific properties
  - Always described in the device's binding file and prefixed with <vendor>,
  - ti, hwmods, xlnx, num-channels, nxp, tx-output-mode, etc
- ▶ Some of them are deprecated, watch out the bindings!



# The compatible property

---

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



# compatible property and Linux kernel drivers

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

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



# Matching with drivers in Linux: platform driver

drivers/i2c/busses/i2c-omap.c

```
static const struct of_device_id omap_i2c_of_match[] = {
    {
        .compatible = "ti,omap4-i2c",
        .data = &omap4_pdata,
    },
    {
        .compatible = "ti,omap3-i2c",
        .data = &omap3_pdata,
    },
    [...]
};

MODULE_DEVICE_TABLE(of, omap_i2c_of_match);

[...]

static struct platform_driver omap_i2c_driver = {
    .probe      = omap_i2c_probe,
    .remove     = omap_i2c_remove,
    .driver     = {
        .name  = "omap_i2c",
        .pm     = &omap_i2c_pm_ops,
        .of_match_table = of_match_ptr(omap_i2c_of_match),
    },
};
```



# Matching with drivers in Linux: I2C driver

sound/soc/codecs/cs42151.c

```
const struct of_device_id cs42151_of_match[] = {
    { .compatible = "cirrus,cs42151", },
    { }
};

MODULE_DEVICE_TABLE(of, cs42151_of_match);
```

sound/soc/codecs/cs42151-i2c.c

```
static struct i2c_driver cs42151_i2c_driver = {
    .driver = {
        .name = "cs42151",
        .of_match_table = cs42151_of_match,
        .pm = &cs42151_pm_ops,
    },
    .probe = cs42151_i2c_probe,
    .remove = cs42151_i2c_remove,
    .id_table = cs42151_i2c_id,
};
```



- ▶ 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>;  
    };  
    cs42151: cs42151@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>;  
};
```



## Property cells

- ▶ 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>;
```



# Property cells

- ▶ 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 used to carry the size of the 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>;  
    };  
};
```



# Property cells

- ▶ 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 used to carry the size of the 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



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

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

```
intc: interrupt-controller@a0021000 {  
    compatible = "arm,cortex-a7-gic";  
    #interrupt-cells = <3>;  
    interrupt-controller;  
    reg = <0xa0021000 0x1000>, <0xa0022000 0x2000>;  
};  
  
rcc: rcc@50000000 {  
    compatible = "st,stm32mp1-rcc", "syscon";  
    reg = <0x50000000 0x1000>;  
    #clock-cells = <1>;  
    #reset-cells = <1>;  
};  
  
dmamux1: dma-router@48002000 {  
    compatible = "st,stm32h7-dmamux";  
    reg = <0x48002000 0x1c>;  
    #dma-cells = <3>;  
    clocks = <&rcc DMAMUX>;  
    resets = <&rcc DMAMUX_R>;  
};  
  
spi3: spi@4000c000 {  
    interrupts = <GIC_SPI 51 IRQ_TYPE_LEVEL_HIGH>;  
    clocks = <&rcc SPI3_K>;  
    resets = <&rcc SPI3_R>;  
    dmas = <&dmamux1 61 0x400 0x05>, <&dmamux1 62 0x400 0x05>;  
};
```



# Generic suffixes

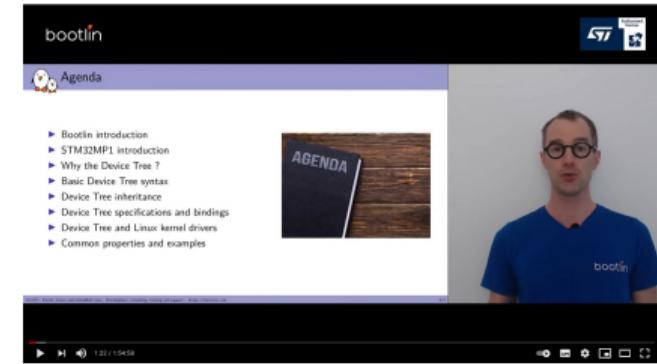
- ▶ xxx-gpios
  - When drivers need access to GPIOs
  - May be subsystem-specific or vendor-specific
  - Examples: enable-gpios, ts, txrx-gpios
- ▶ xxx-names
  - Sometimes naming items is relevant
  - Allows drivers to perform lookups by name rather than ID
  - The order of definition of each item still matters
  - Examples: gpio-names, clock-names, reset-names

```
uart0@4000c000 {  
    dmas = <&edma 26 0>, <&edma 27 0>;  
    dma-names = "tx", "rx";  
    ...  
};
```



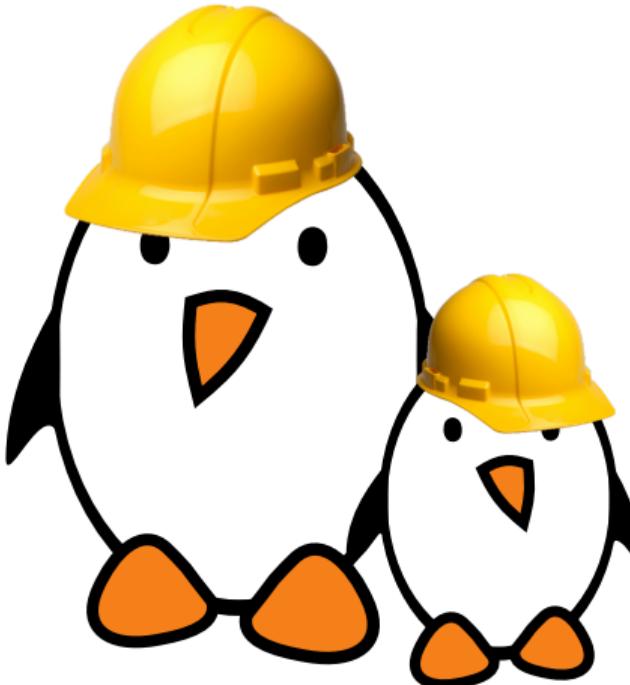
# References

- ▶ Device Tree 101 webinar, Thomas Petazzoni (2021):  
Slides: <https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/>  
Video: <https://youtu.be/a9CZ1Uk30YQ>
- ▶ Kernel documentation
  - [driver-api/driver-model/](#)
  - [devicetree/](#)
  - [filesystems/sysfs](#)
- ▶ <https://devicetree.org>
- ▶ The kernel source code
  - Full of examples of other drivers!





# Practical lab - Describing hardware devices



- ▶ Browse and update Device Trees.
- ▶ Use GPIO LEDs.
- ▶ Modify the Device Tree to enable an I2C controller and describe an I2C device.

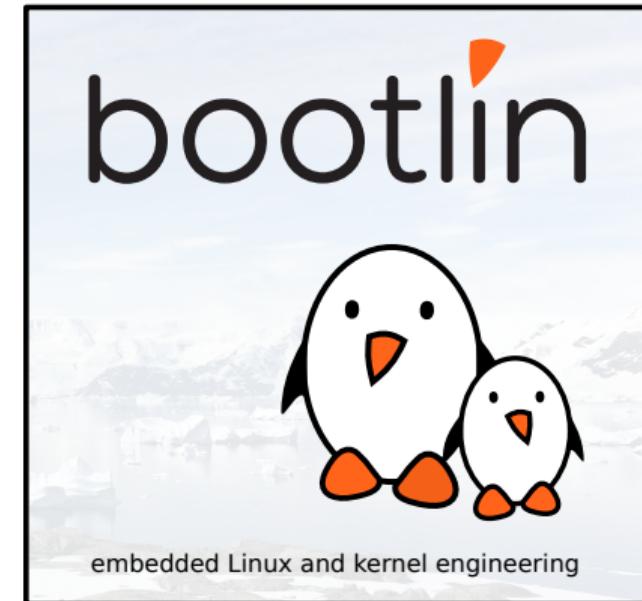


# Introduction to pin muxing

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





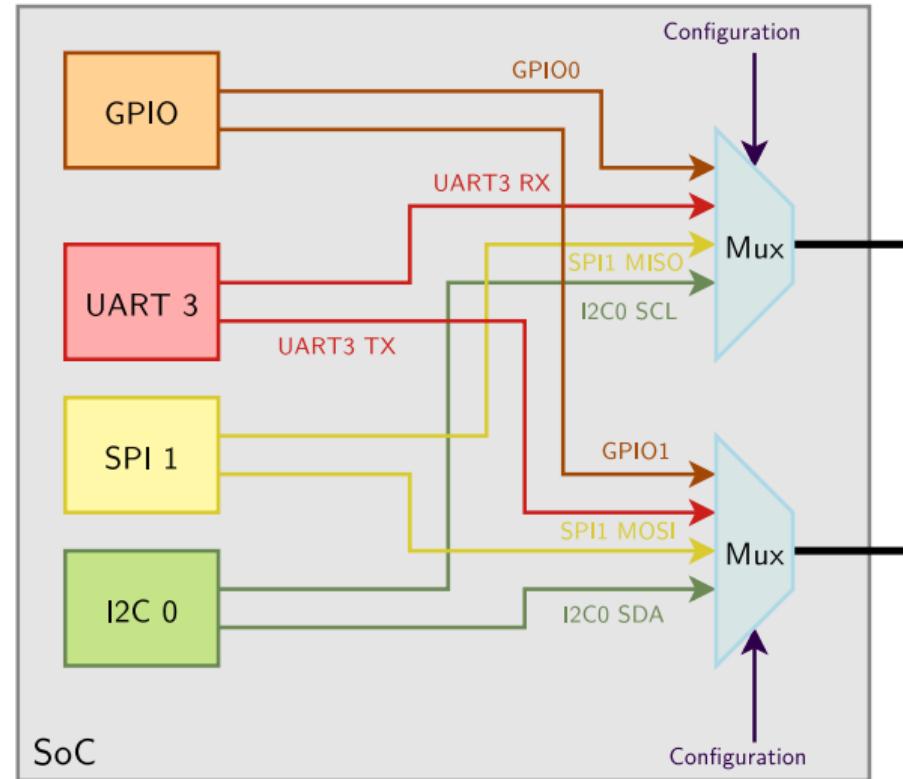
# What is pin muxing?

---

- ▶ 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





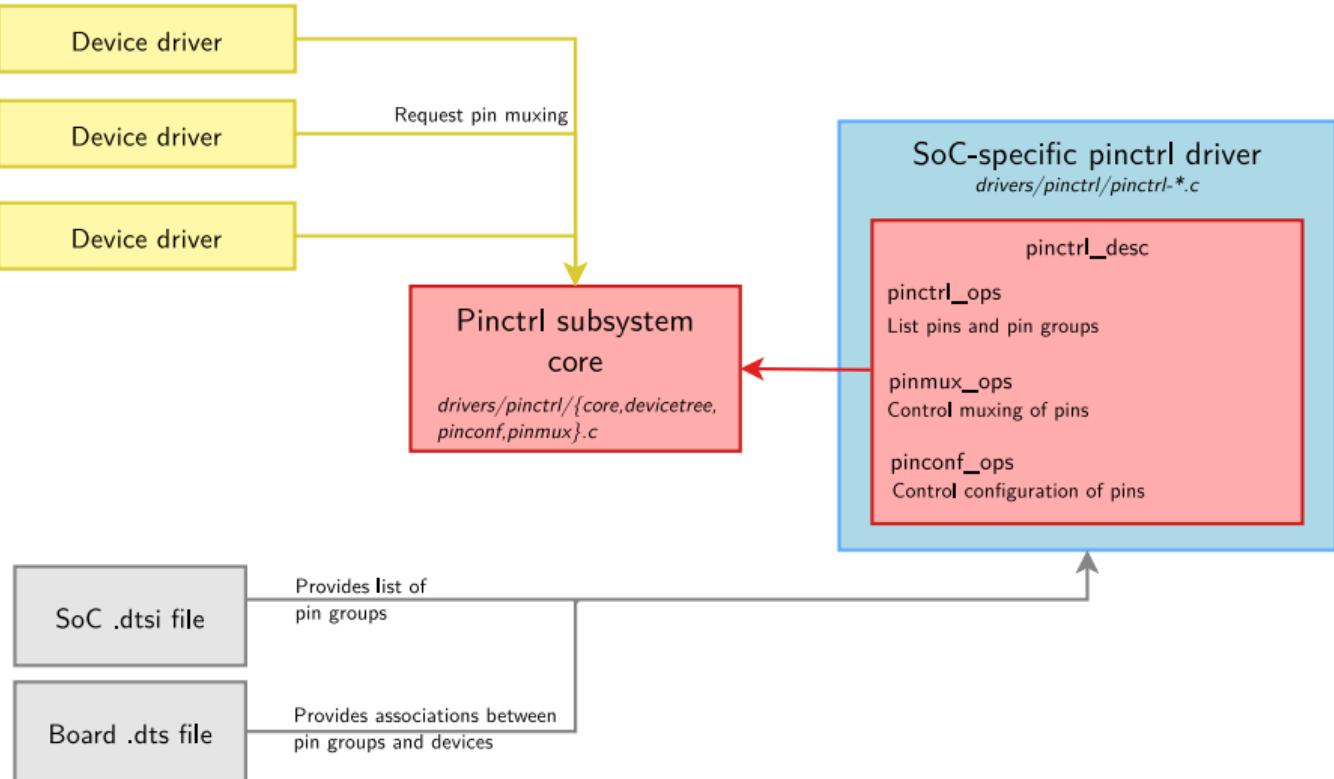
## Pin muxing in the Linux kernel

---

- ▶ Since Linux 3.2, a `pinctrl` subsystem has been added.
- ▶ 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 [Documentation/devicetree/bindings/pinctrl](#).



# pinctrl subsystem diagram





## Device Tree properties for consumer devices

---

The devices that require certain 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 <Documentation/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/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/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 [Documentation/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-boneblue.dts */

&am33xx_pinmux {
    ...
    i2c2_pins: pinmux_i2c2_pins {
        pinctrl-single,pins = <
            AM33XX_IOPAD(0x978, PIN_INPUT_PULLUP | MUX_MODE3)
            /* (D18) uart1_ctsn.I2C2_SDA */
            AM33XX_IOPAD(0x97c, PIN_INPUT_PULLUP | MUX_MODE3)
            /* (D17) 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

SoC level

```
/ {  
    ...  
    soc {  
        ...  
        pio: pinctrl001c20800 {  
            compatible = "allwinner,sun7i-a20-pinctrl";  
            reg = <0x01c20800 0x400>;  
            ...  
  
        UART0 pin mux config  
        uart0_pb_pins: uart0_pb-pins {  
            pins = "PB22", "PB23";  
            function = "uart0";  
        };  
        ...  
    };  
};
```

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

Board level

```
/ {  
    ...  
    Declare LED device and associate pin mux config  
    pinctrl-0 = <&led_pins_olinuxino>;  
    ...  
};  
  
led {  
    label = "a20-olinuxino-micro:green:usr";  
    gpios = <pio 7 2 GPIO_ACTIVE_HIGH>;  
    default-state = "on";  
};  
...  
};
```

```
8pio {  
    ...  
    LED pin mux config  
    led_pins_olinuxino: led_pins@0 {  
        pins = "PH2";  
        function = "gpio_out";  
        drive-strength = <20>;  
    };  
    ...  
};  
  
&uart0 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&uart0_pb_pins>;  
    status = "okay";  
};
```

Enable UART0 and associate pin mux config

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



# Illustration: live pin muxing configuration

Viewing pin assignments on the PCB

The diagram shows the physical pin layout for the Arietta board, with pins numbered 1 through 40. Various signals are color-coded and labeled:

- Power: 5V (yellow), GND (black), VBAT (red)
- Reset: NRST (cyan)
- USB: USB A D-, USB A D+, USB B D-, USB B D+, USB C D-, USB C D+
- SPI: SPI1 CK (orange), SPI1 MOSI (blue), SPI1 MISO (green)
- I2C: SCL0 (blue), SDA0 (blue), SCL1 (blue), SDA1 (blue)
- SPI: PA24 (green), PA25 (green), PA26 (green), PA27 (green), PA28 (green), PA29 (green), PA0 (green), RXD2 (blue), PA6 (green), PC28 (green), PC4 (green), PC3 (green), PC2 (green), SCL1 (blue), SDA1 (blue)
- A/D converter: ADC0 (grey), ADC1 (blue), ADC2 (grey), ADC3 (grey)
- PWM: PWM0 (grey), PWM1 (grey), PWM2 (grey), PWM3 (grey)
- I2S: PC2 (grey), PC3 (grey), PC4 (grey), PC31 (grey), PC27 (green), PC31 (grey), PC26 (orange), PC28 (green)
- 1-wire: PB13 (green), PB14 (green), PA23 (grey)

Setup    Reset state    Code examples    Show the DTS    Generate the DTB

Serial lines

- /dev/ptyS1
- /dev/ptyS2
- CTS/RTS
- RS485
- /dev/ptyS3

I2C bus

- /dev/i2c-0
- /dev/i2c-1

SPI bus

- /dev/spi0
- CS0
- CS1
- CS2
- CS3

A/D converter

- ADC0
- ADC1
- ADC2
- ADC3

PWM lines

- PWM0
- PWM1
- PWM2
- PWM3

I2S bus for audio SoC

- I2S Bus for Audio Codec

1 wire bus

- None
- PC2
- PC3
- PC4
- PC31
- PA23

Show the DTS, generate the DTB

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



- ▶ Configure the pinmuxing for the I2C bus used to communicate with the Nunchuk
- ▶ Validate that the I2C communication works with user space tools.



# Linux device and driver model

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Introduction



## 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.

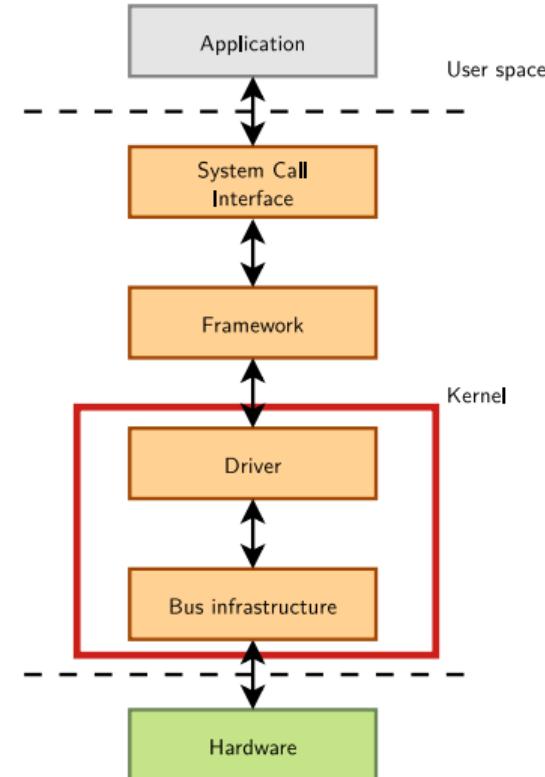


# Kernel and Device Drivers

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 training.





- ▶ 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 certain devices on a certain 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.
- ▶ In order to explore the device model, we will
  - First look at a popular bus that offers dynamic enumeration, the *USB bus*
  - Continue by studying how buses that do not offer dynamic enumeration are handled.



- ▶ 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, mainly `struct usb_driver` and `struct usb_interface`



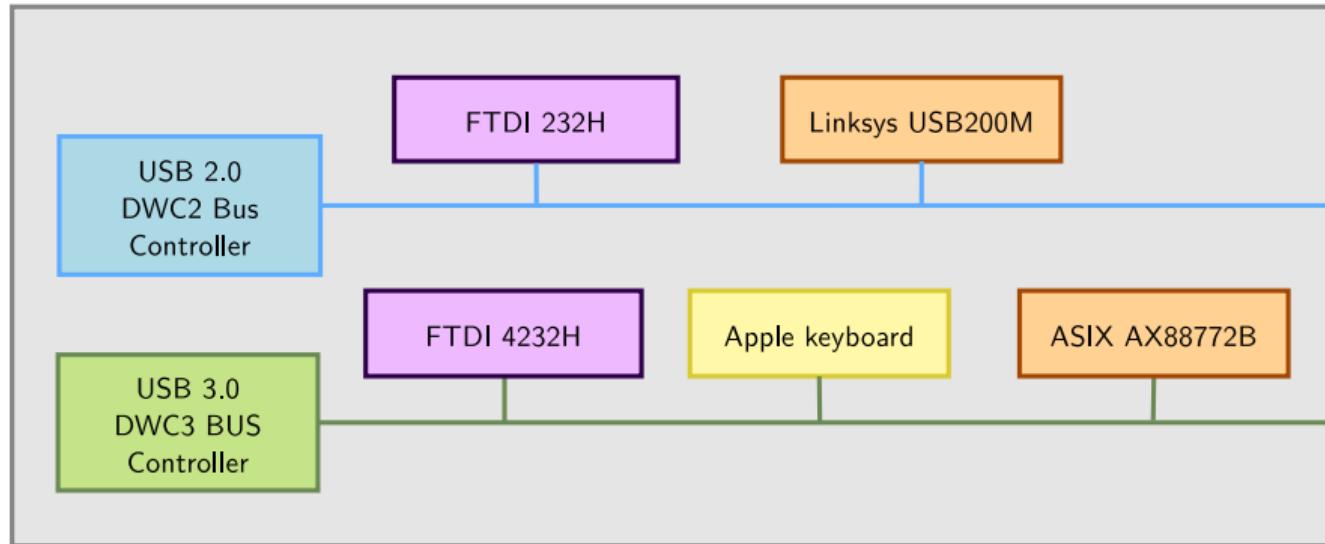
- ▶ The bus, device, drivers, etc. structures are internal to the kernel
- ▶ The sysfs virtual filesystem offers a mechanism to export such information to user space
- ▶ Used for example by udev to provide automatic module loading, firmware loading, mounting of external media, etc.
- ▶ sysfs is usually 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!



## Example of the USB bus



# Example: USB Bus 1/3



Serial adapters  
supported by ftdi-sio

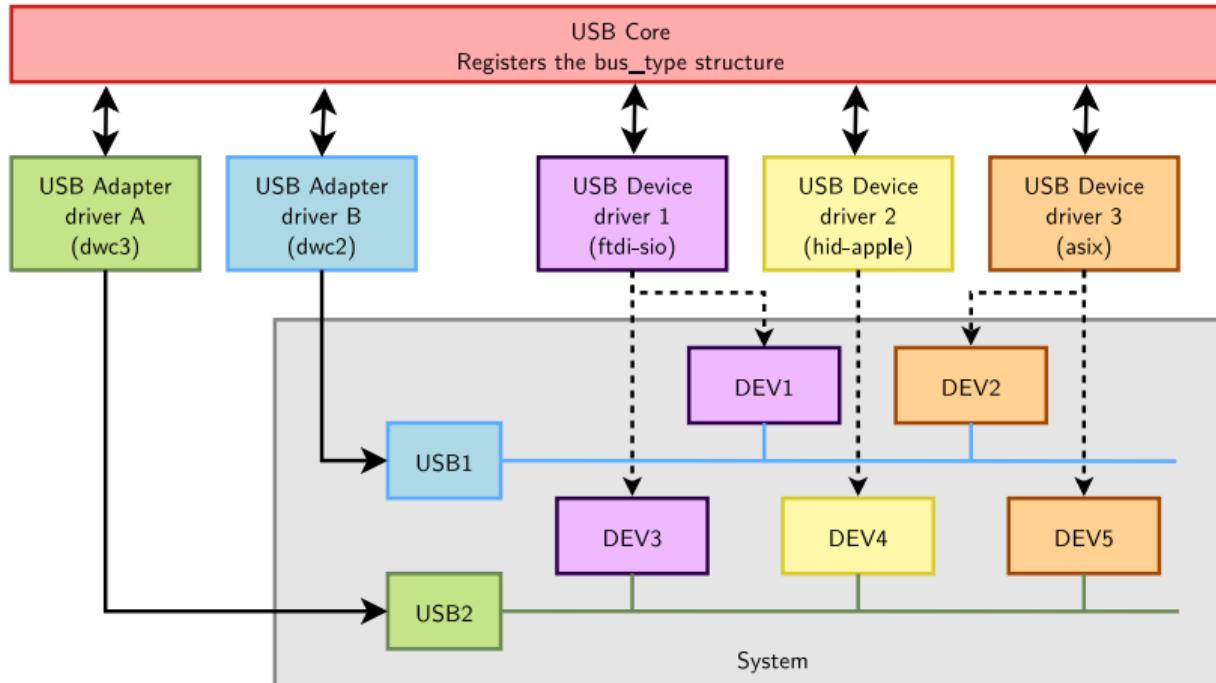
Keyboard  
supported by hid-apple

Ethernet adapters  
supported by asix

Hardware view of the bus



# Example: USB Bus 2/3



Device model view of the bus



## Example: USB Bus 3/3

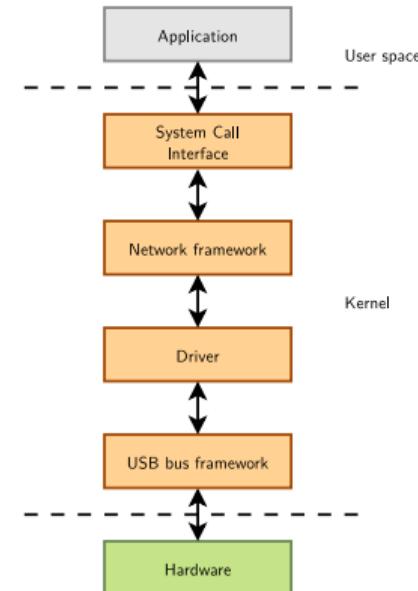
---

- ▶ 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

- ▶ `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

- ▶ 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);
```

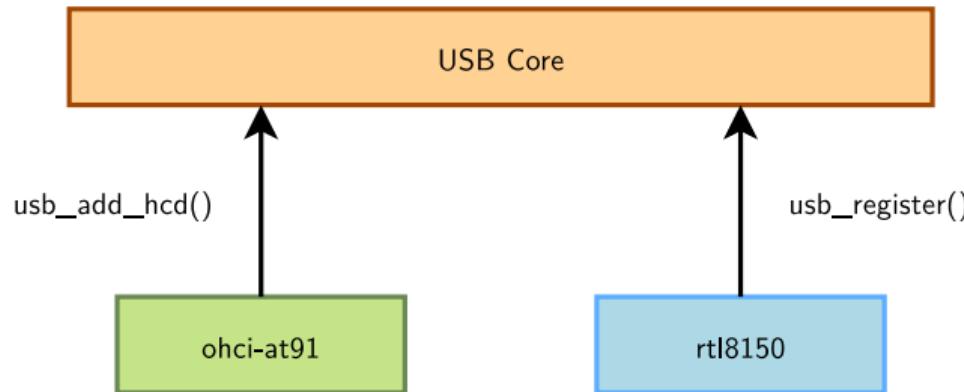
- ▶ All this code is actually replaced by a call to the `module_usb_driver()` macro:

```
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

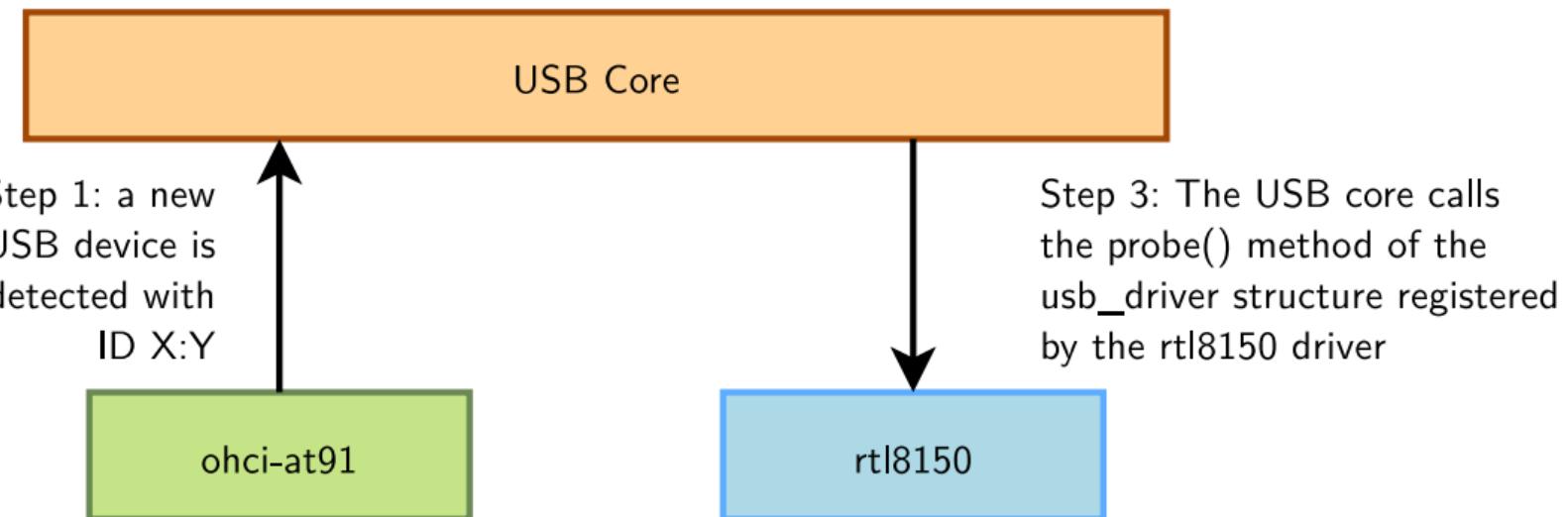


- ▶ The USB core now knows the association between the vendor/product IDs of `rtl8150` and the `struct usb_driver` structure of this driver



## When a device is detected

Step 2: USB core looks up the registered IDs, and finds the matching driver





- ▶ 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

```
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;
}
```

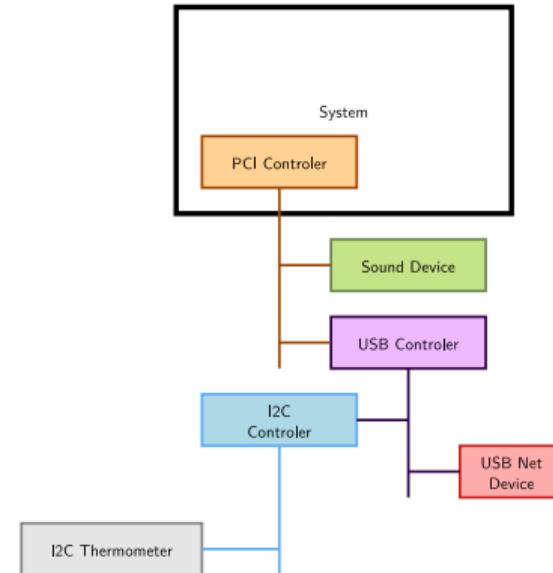
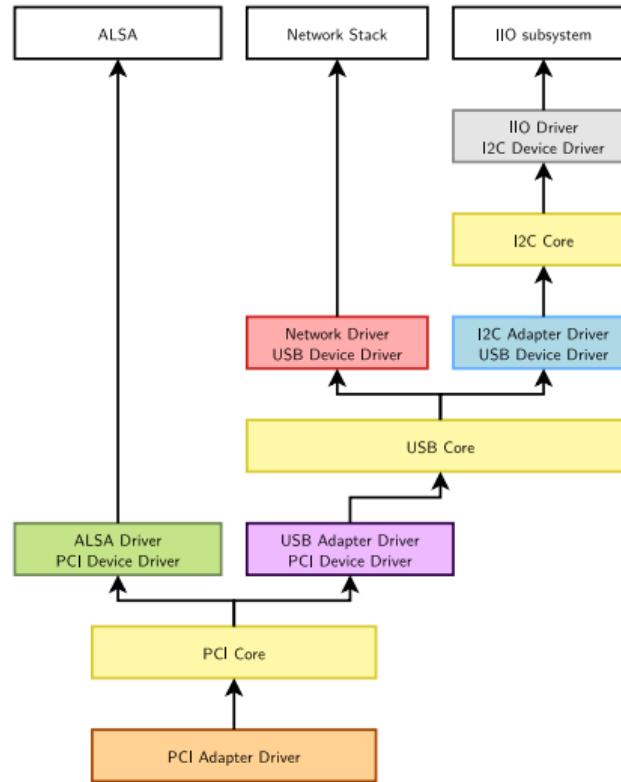
```
static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

    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](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/usb/rtl8150.c)



# The Model is Recursive





## Platform drivers



- ▶ 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)

... 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

- ▶ 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 slightly 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);
```



# Introduction to the I2C subsystem

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





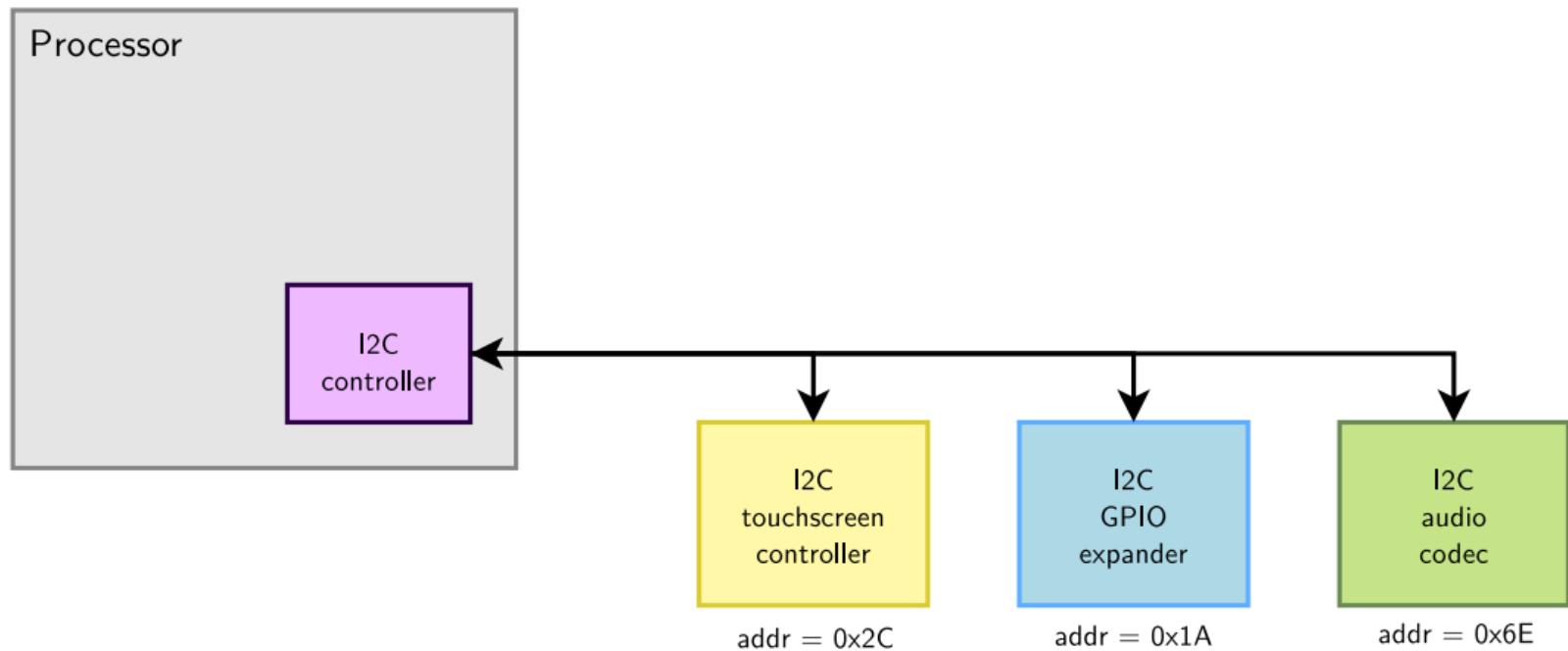
## What is I2C?

---

- ▶ 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

- ▶ 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 an `id_table`, used for non-DT based probing of I2C devices.
  - A `->probe_new()` function can replace `->probe()` when no `id_table` is provided.
- ▶ 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: non-DT

---

- ▶ On non-DT platforms, the `struct i2c_board_info` structure allows to describe how an I2C device is connected to a board.
- ▶ Such structures are normally defined with the `I2C_BOARD_INFO()` helper macro.
  - Takes as argument the device name and the slave address of the device on the bus.
- ▶ An array of such structures is registered on a per-bus basis using `i2c_register_board_info()`, when the platform is initialized.



# Registering an I2C device, non-DT example

```
static struct i2c_board_info __initdata em7210_i2c_devices[] = {
{
    I2C_BOARD_INFO("rs5c372a", 0x32),
},
};

...

static void __init em7210_init_machine(void)
{
    register_iop32x_gpio();
    platform_device_register(&em7210_serial_device);
    platform_device_register(&iop3xx_i2c0_device);
    platform_device_register(&iop3xx_i2c1_device);
    platform_device_register(&em7210_flash_device);
    platform_device_register(&iop3xx_dma_0_channel);
    platform_device_register(&iop3xx_dma_1_channel);

    i2c_register_board_info(0, em7210_i2c_devices,
        ARRAY_SIZE(em7210_i2c_devices));
}
```

From arch/arm/mach-iop32x/em7210.c



# Registering an I2C device, in the DT

- ▶ 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: <Documentation/devicetree/bindings/i2c/i2c-omap.txt>



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

## 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/sun7i-a20.dtsi](https://github.com/Bootlin/arch/arm/boot/dts/sun7i-a20.dtsi)

#address-cells: number of 32-bit values needed to encode the address fields

#size-cells: number of 32-bit values needed to encode the size fields

See details in [https://elinux.org/Device\\_Tree\\_Usage](https://elinux.org/Device_Tree_Usage)



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

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/sun7i-a20-olinuxino-micro.dts



## probe\_new() / probe() and remove()

---

- ▶ The `->probe_new()` function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - A `struct i2c_client` pointer, which represents the I2C device itself. This structure inherits from `struct device`.
- ▶ Alternatively, the `->probe()` function receives as arguments:
  - A similar `struct i2c_client` pointer.
  - A `struct i2c_device_id` pointer, which points to the I2C device ID entry that matched the device that is being probed.
- ▶ 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 that was passed as argument to `->probe_new()` or `->probe()`



## Probe example

```
static int da311_probe(struct i2c_client *client,
                      const struct i2c_device_id *id)
{
    struct iio_dev *indio_dev;           // framework structure
    da311_data *data;                  // per device structure
    ...
    // Allocate framework structure with per device struct inside
    indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
    data = iio_priv(indio_dev);
    data->client = client;
    i2c_set_clientdata(client, indio_dev);
    // Prepare device and initialize indio_dev
    ...
    // Register device to framework
    ret = iio_device_register(indio_dev);
    ...
    return ret;
}
```

From drivers/iio/accel/da311.c



## Remove example

```
static int da311_remove(struct i2c_client *client)
{
    struct iio_dev *indio_dev = i2c_get_clientdata(client);
    // Unregister device from framework
    iio_device_unregister(indio_dev);
    return da311_enable(client, false);
}
```

From `drivers/iio/accel/da311.c`



## 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:

- ▶ `int i2c_master_send(const struct i2c_client *client, const char *buf, int count);`  
Sends the contents of buf to the client.
- ▶ `int i2c_master_recv(const struct i2c_client *client, char *buf, int count);`  
Receives count bytes from the client, and store them into buf.

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

```
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](#)



## SMBus calls

---

- ▶ SMBus is a subset of the I2C protocol.
- ▶ It defines a standard set of transactions, such as reading/writing from a register-like interface.
- ▶ Linux provides SMBus functions that should preferably be used instead of the raw API with devices supporting SMBus.
- ▶ Such a driver will be usable with both SMBus and I2C adapters
  - SMBus adapters cannot send raw I2C commands
  - I2C adapters will receive an SMBus-like command crafted by the core
- ▶ Example: the `i2c_smbus_read_byte_data()` function allows to read one byte of data from a device “register”.
  - It does the following operations:  
S Addr Wr [A] Comm [A] Sr Addr Rd [A] [Data] NA P
  - Which means it first writes a one byte data command (*Comm*, which is the “register” address), and then reads back one byte of data ([*Data*]).
- ▶ See `i2c/smbus-protocol` for details.



# List of SMBus functions

## ► Read/write one byte

- `s32 i2c_smbus_read_byte(const struct i2c_client *client);`
- `s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value);`

## ► Write a command byte, and read or write one byte

- `s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command);`
- `s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value);`

## ► Write a command byte, and read or write one word

- `s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command);`
- `s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value);`

## ► Write a command byte, and read or write a block of data (max 32 bytes)

- `s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 command, u8 *values);`
- `s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);`

## ► Write a command byte, and read or write a block of data (no limit)

- `s32 i2c_smbus_read_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, u8 *values);`
- `s32 i2c_smbus_write_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);`



# I2C functionality

---

- ▶ Not all I2C controllers support all functionalities.
- ▶ The I2C controller drivers therefore tell the I2C core which functionalities they support.
- ▶ An I2C device driver must check that the functionalities they need are provided by the I2C controller in use on the system.
- ▶ The `i2c_check_functionality()` function allows to make such a check.
- ▶ Examples of functionalities: `I2C_FUNC_I2C` to be able to use the raw I2C functions, `I2C_FUNC_SMBUS_BYTE_DATA` to be able to use SMBus commands to write a command and read/write one byte of data.
- ▶ See `include/uapi/linux/i2c.h` for the full list of existing functionalities.



## 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
  - [i2c/smbus-protocol](#)  
Details on the SMBus functions
  - [i2c/functionality](#)  
How the functionality mechanism works
- ▶ See also Luca Ceresoli's introduction to I2C ([slides](#), [video](#)).



## Practical lab - Communicate with the Nunchuk



- ▶ Explore the content of `/dev` and `/sys` and the devices available on the embedded hardware platform.
- ▶ Implement a driver that registers as an I2C driver.
- ▶ Communicate with the Nunchuk and extract data from it.

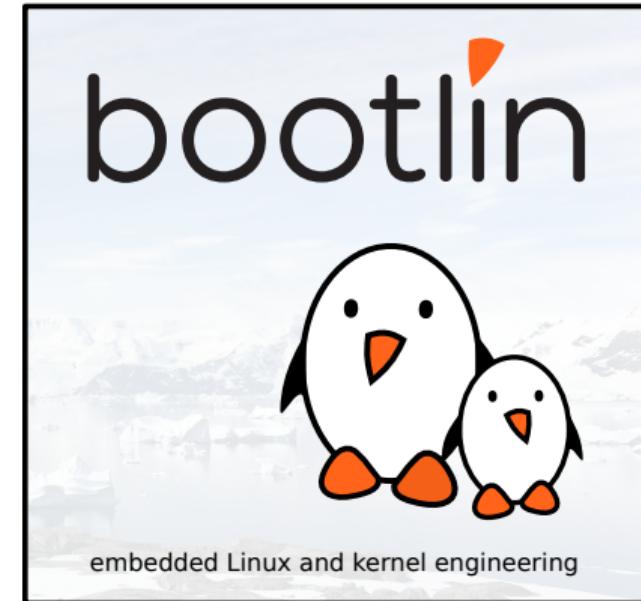


# Kernel frameworks for device drivers

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



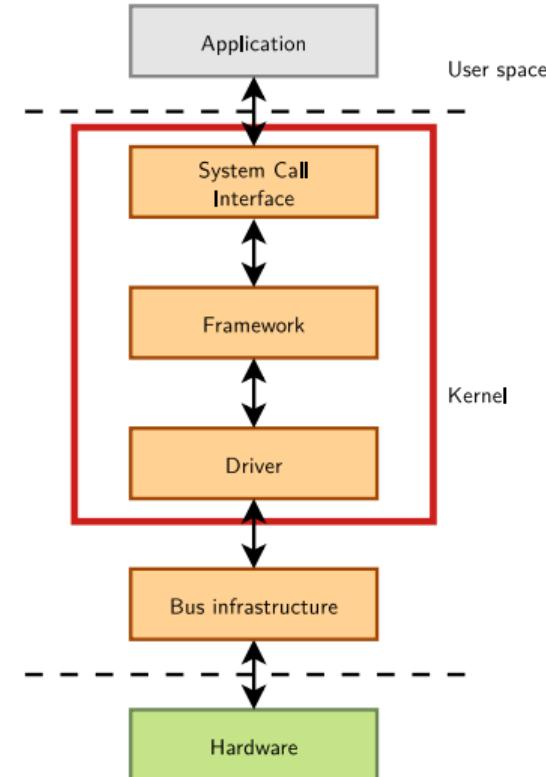


# Kernel and Device Drivers

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 training.





## User space vision of devices



## Types of devices

---

Under Linux, there are essentially three 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`.
- Most devices are *character devices*, so we will study these in more details.



## Major and minor numbers

---

- ▶ 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.
- ▶ Most major and minor numbers are statically allocated, and identical across all Linux systems.
- ▶ They are defined in [admin-guide/devices](#).



## Devices: everything is a file

---

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



# Device files examples

## Example of device files in a Linux system

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

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

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



## Creating device files

---

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



## Character drivers



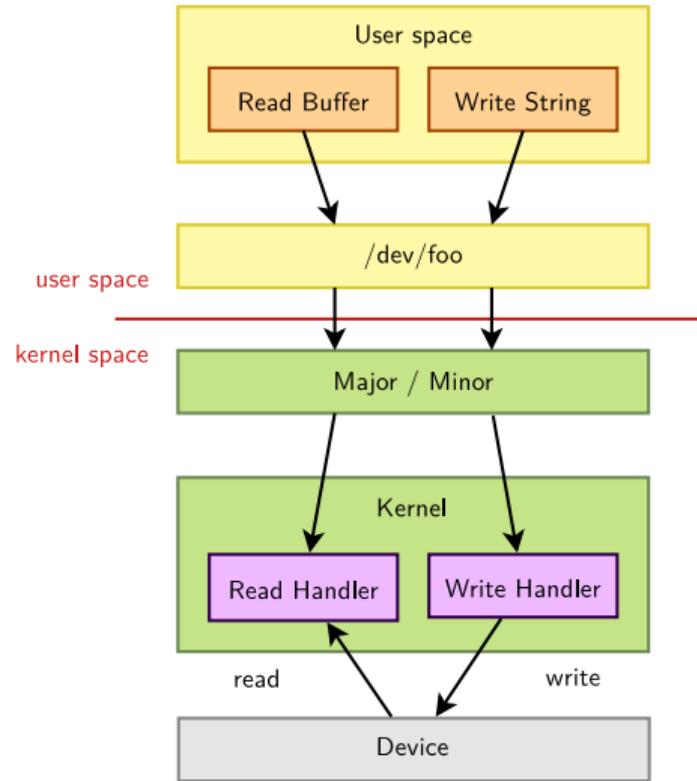
## A character driver in the kernel

---

- ▶ From the point of view of an application, a *character device* is essentially a **file**.
- ▶ The driver of a character device must therefore implement **operations** that let applications think the device is a file: open, close, read, write, etc.
- ▶ In order to achieve this, a character driver must implement the operations described in the `struct file_operations` structure and register them.
- ▶ The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.



# 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 more operations exist. All of them are 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()

- ▶ `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

- ▶ 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

## ► 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`.

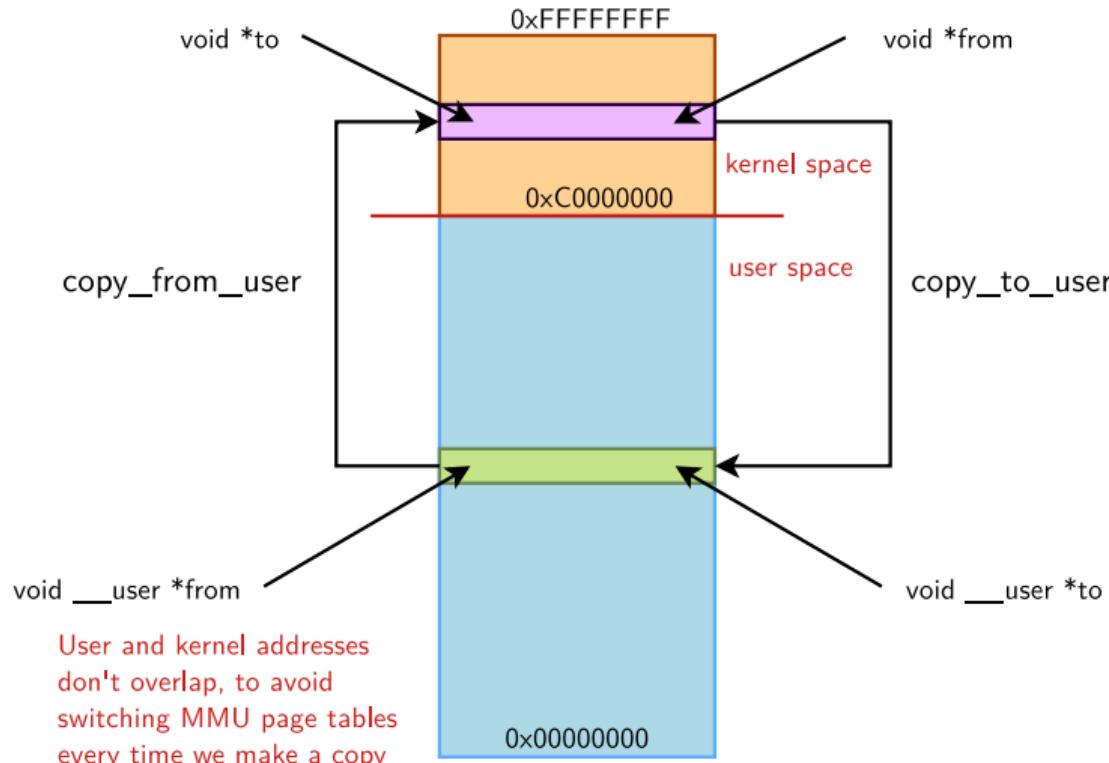
## ► A buffer

- `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





- ▶ 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

```
#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

```
#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;
}
```



## The concept of kernel frameworks



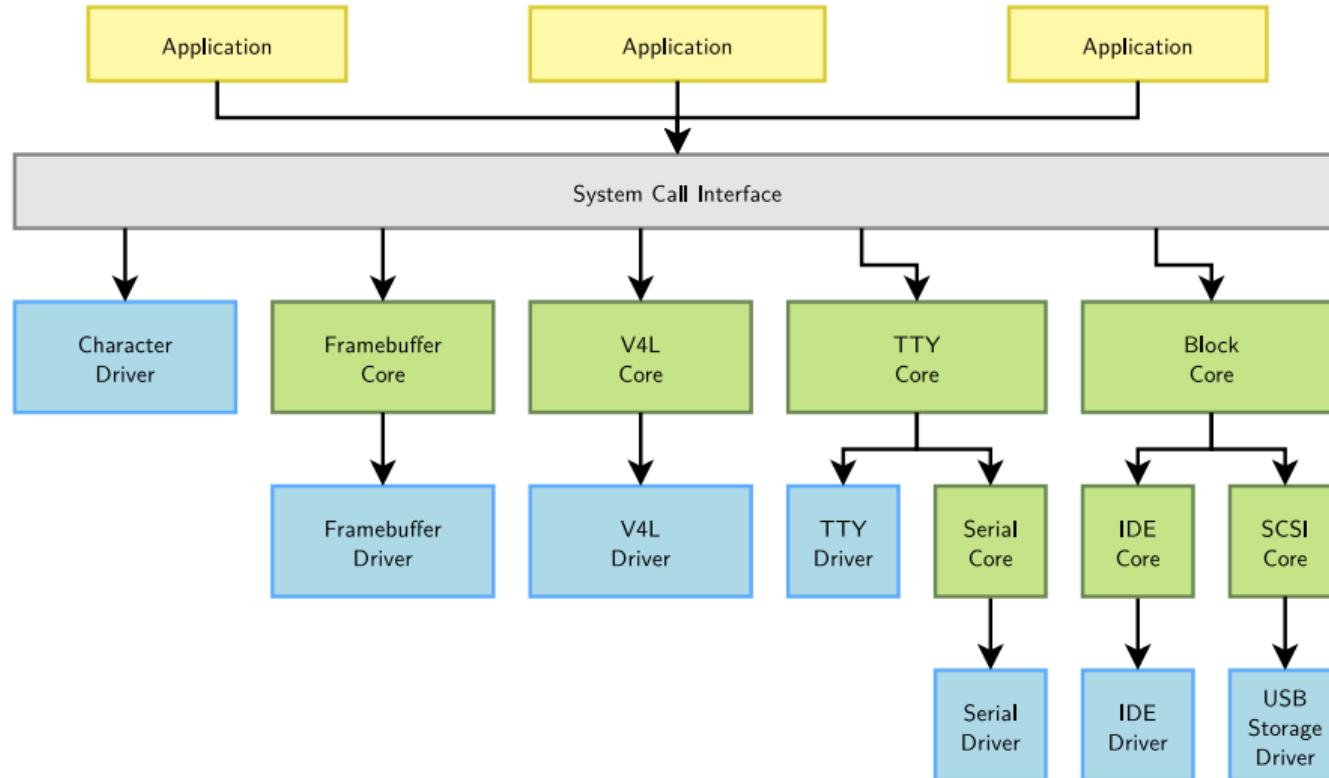
## Beyond character drivers: kernel frameworks

---

- ▶ 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

---

- ▶ 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

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.



## 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



# Device managed allocations: memory allocation example

- ▶ 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 */
}
```



## Driver data structures and links



- ▶ Each *framework* defines a structure that a device driver must register to be recognized as a device in this framework
  - `struct uart_port` for serial ports, `struct net_device` for network devices, `struct fb_info` for framebuffers, etc.
- ▶ In addition to this structure, the driver usually needs to store additional information about each device
- ▶ This is typically done
  - By subclassing the appropriate framework structure
  - By storing a reference to the appropriate framework structure
  - Or by including your information in the framework structure



## Driver-specific Data Structure Examples 1/2

- ▶ i.MX serial driver: `struct imx_port` is a subclass of `struct uart_port`

```
struct imx_port {  
    struct uart_port port;  
    struct timer_list timer;  
    unsigned int old_status;  
    int txirq, rxirq, rtsirq;  
    unsigned int have_rtscts:1;  
    [...]  
};
```

- ▶ ds1305 RTC driver: `struct ds1305` has a reference to `struct rtc_device`

```
struct ds1305 {  
    struct spi_device      *spi;  
    struct rtc_device     *rtc;  
    [...]  
};
```



## Driver-specific Data Structure Examples 2/2

- ▶ rtl8150 network driver: `struct rtl8150` has a reference to `struct net_device` and is allocated within that framework structure.

```
struct rtl8150 {  
    unsigned long flags;  
    struct usb_device *udev;  
    struct tasklet_struct tl;  
    struct net_device *netdev;  
    [...]  
};
```



## Links between structures 1/4

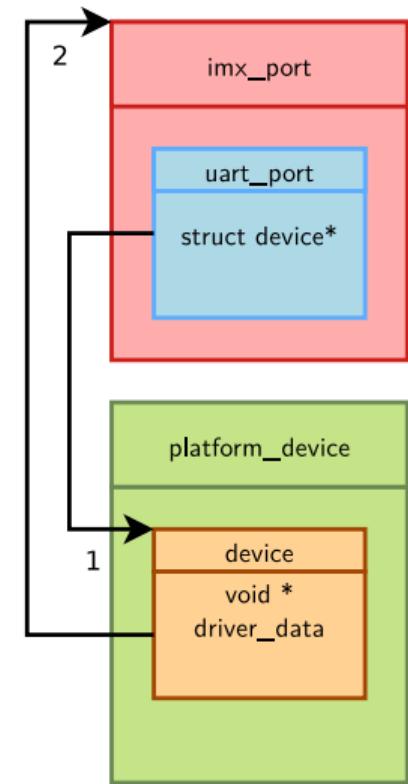
- ▶ The framework structure typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
  - It's the relationship between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- ▶ The device structure also contains a `void *` pointer that the driver can freely use.
  - It's often used to link back the device to the higher-level structure from the framework.
  - It allows, for example, from the `struct platform_device` structure, to find the structure describing the logical device



# Links between structures 2/4

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport; /* per device structure */
    [...]
    sport = devm_kzalloc(&pdev->dev, sizeof(*sport), GFP_KERNEL);
    [...]
    /* setup the link between uart_port and the struct
     * device inside the platform_device */
    sport->port.dev = &pdev->dev;                                // Arrow 1
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, sport);                            // Arrow 2
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrieve the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    [...]
    uart_remove_one_port(&imx_reg, &sport->port);
    [...]
}
```

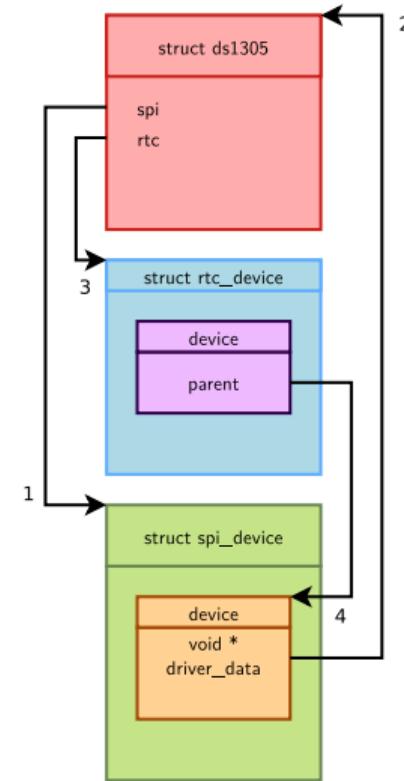




# Links between structures 3/4

```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305             *ds1305;
    [...]
    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
        return -ENOMEM;
    ds1305->spi = spi;           // Arrow 1
    spi_set_drvdata(spi, ds1305); // Arrow 2
    [...]
    ds1305->rtc = devm_rtc_allocate_device(&spi->dev);
    // Arrows 3 and 4
    [...]
}

static int ds1305_remove(struct spi_device *spi)
{
    struct ds1305 *ds1305 = spi_get_drvdata(spi);
    [...]
}
```





# Links between structures 4/4

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

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    [...]

    dev->udev = udev;      // Arrow 1
    dev->netdev = netdev;  // Arrow 2

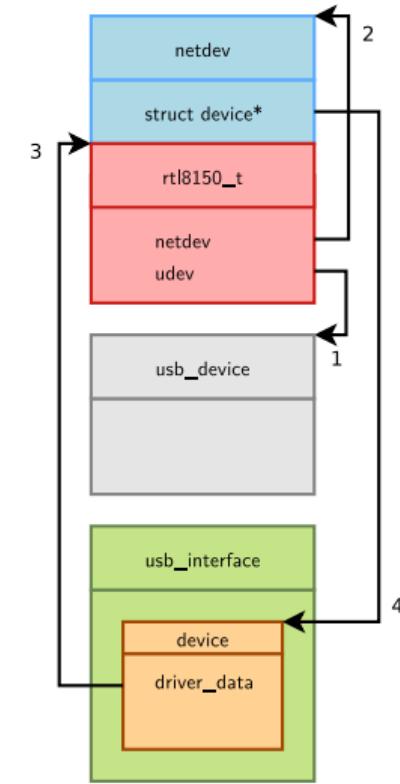
    [...]

    usb_set_intfdata(intf, dev);      // Arrow 3
    SET_NETDEV_DEV(netdev, &intf->dev); // Arrow 4

    [...]
}

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

    [...]
}
```





## The input subsystem

# The input subsystem

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





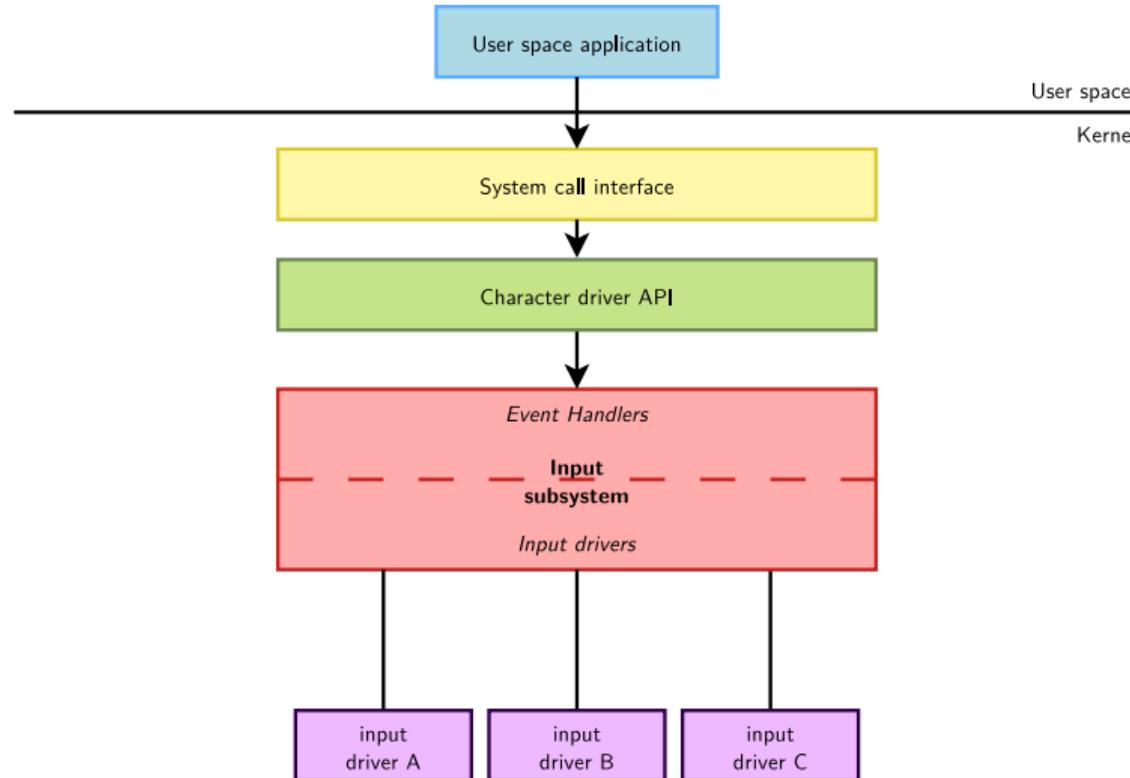
# What is the input subsystem?

---

- ▶ 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

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);`



- ▶ 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);
```

- ▶ `set_bit()` is an atomic operation allowing to set a particular bit to 1 (explained later).
- ▶ Once the *input device* is allocated and filled, the function to register it is:  
`int input_register_device(struct input_dev *);`



The events are sent by the driver to the event handler using `input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);`

- ▶ The event types are documented in [input/event-codes](#)
- ▶ An event is composed by one or several input data changes (packet of input data changes) such as the button state, the relative or absolute position along an axis, etc..
- ▶ After submitting potentially multiple events, the *input* core must be notified by calling: `void input_sync(struct input_dev *dev);`
- ▶ The input subsystem provides other wrappers such as [input\\_report\\_key\(\)](#), [input\\_report\\_abs\(\)](#), ...



## Example from drivers/hid/usbhid/usbmouse.c

```
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/>



## Practical lab - Expose the Nunchuk to user space



- ▶ Extend the Nunchuk driver to expose the Nunchuk features to user space applications, as an *input* device.
- ▶ Test the operation of the Nunchuk using `evtest`

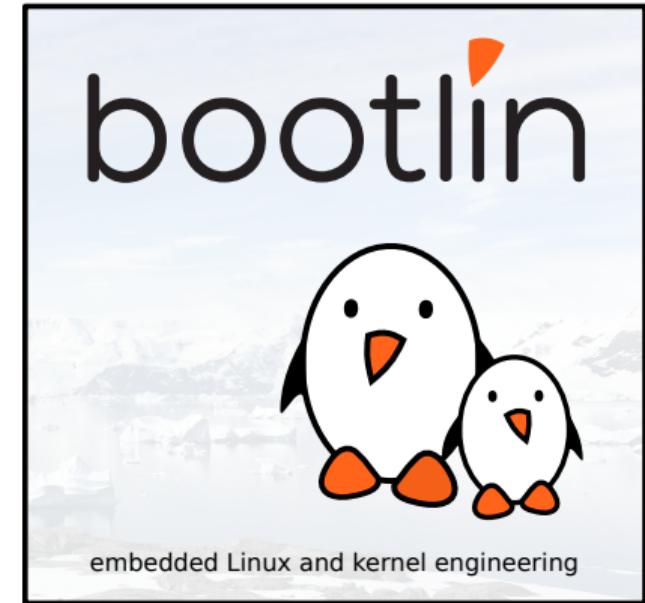


# Memory Management

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

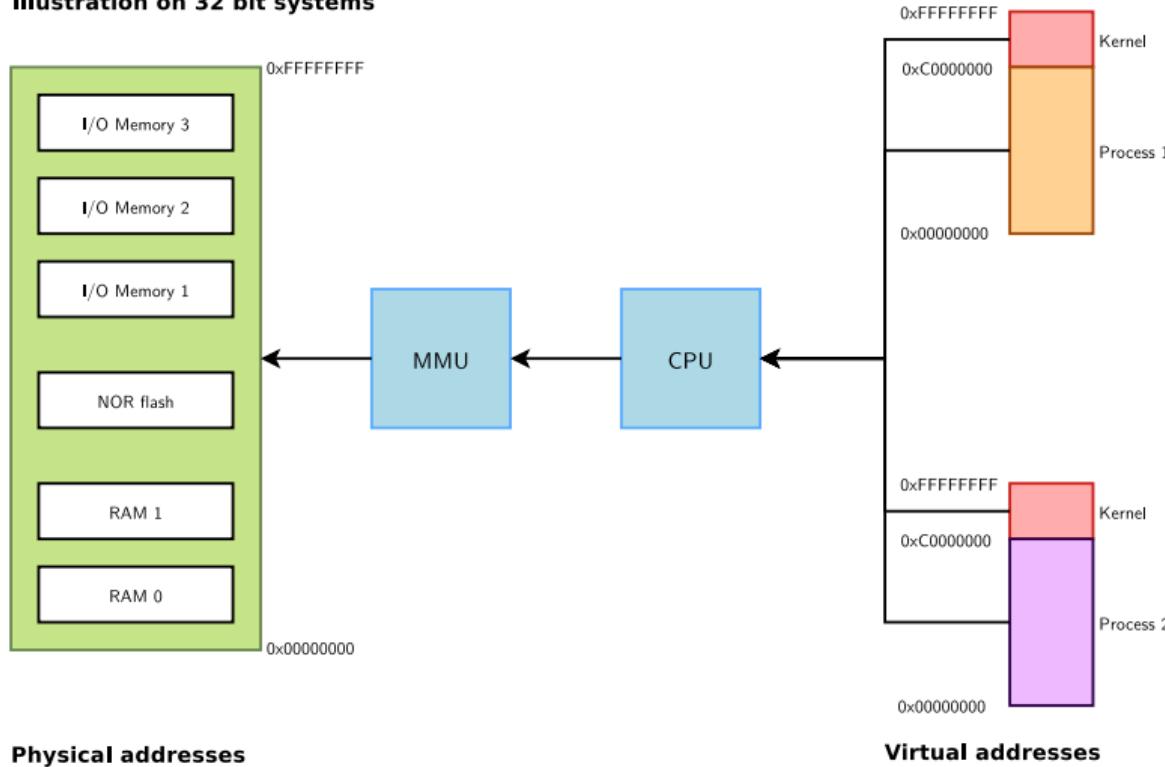
Corrections, suggestions, contributions and translations are welcome!





# Physical and virtual memory

Illustration on 32 bit systems

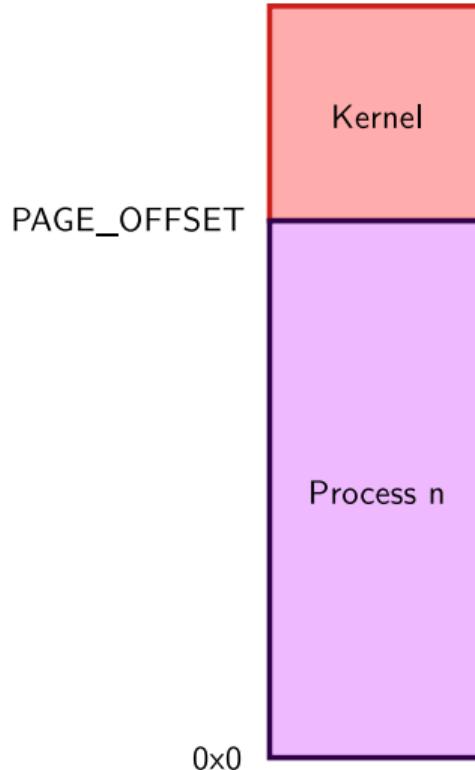


**Physical addresses**

**Virtual addresses**



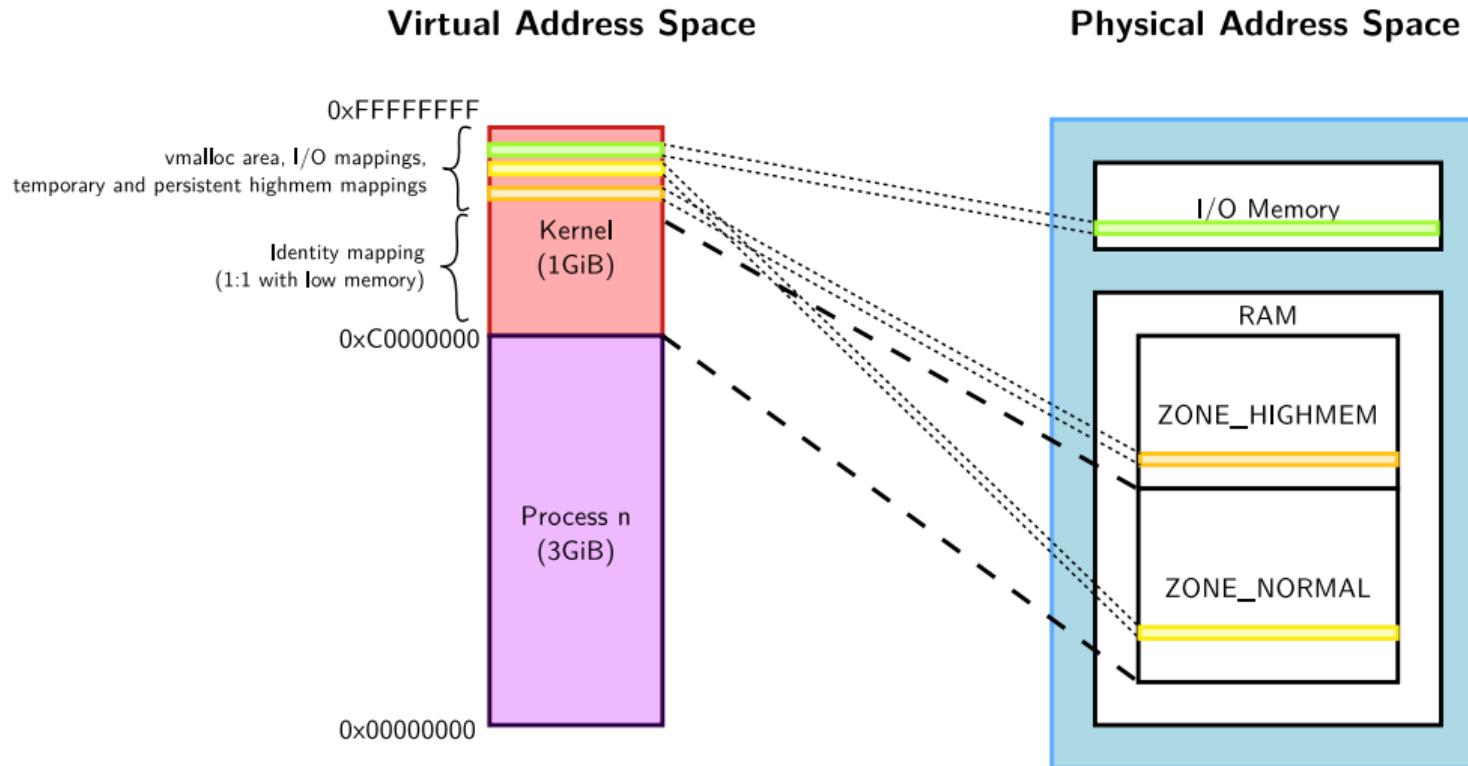
# Virtual memory organization



- ▶ 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



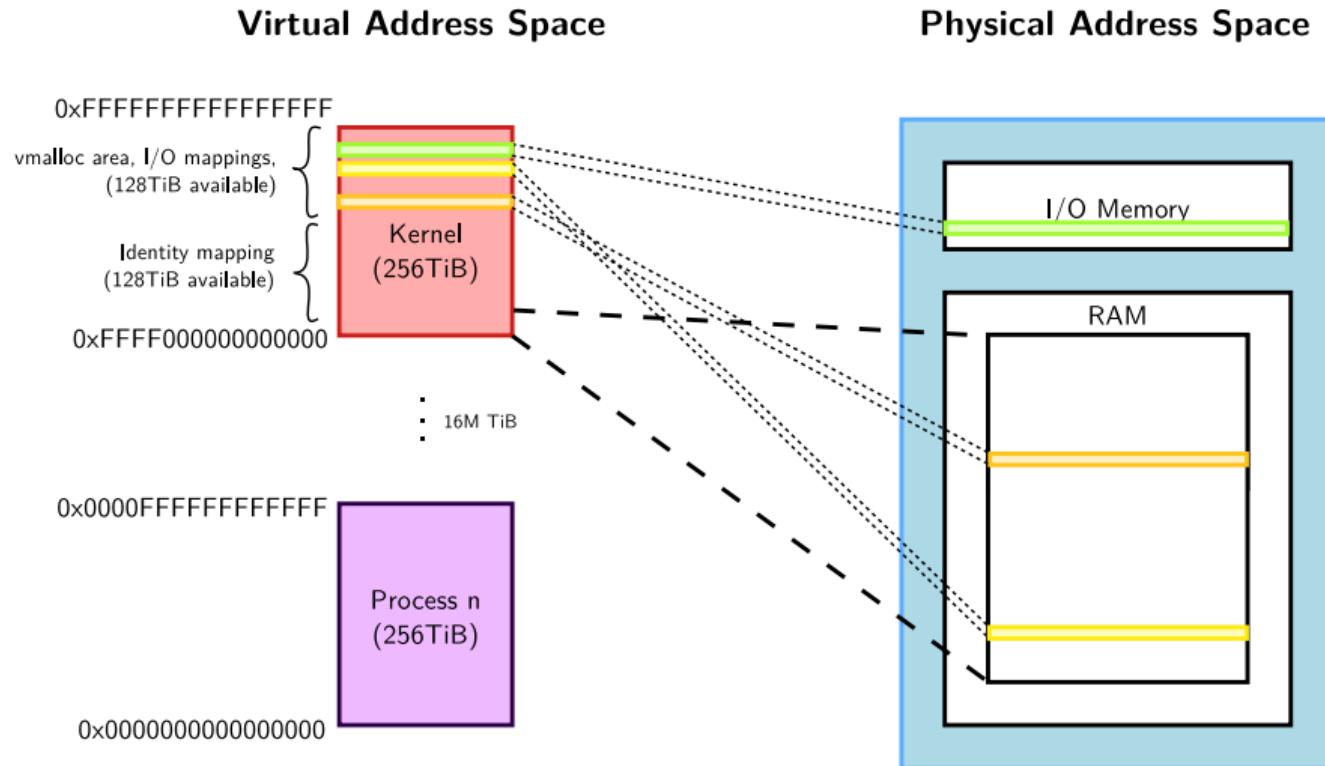


## 32-bit systems limitations

- ▶ 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`) ⇒ 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: <https://frama.link/fD1HvuVP>



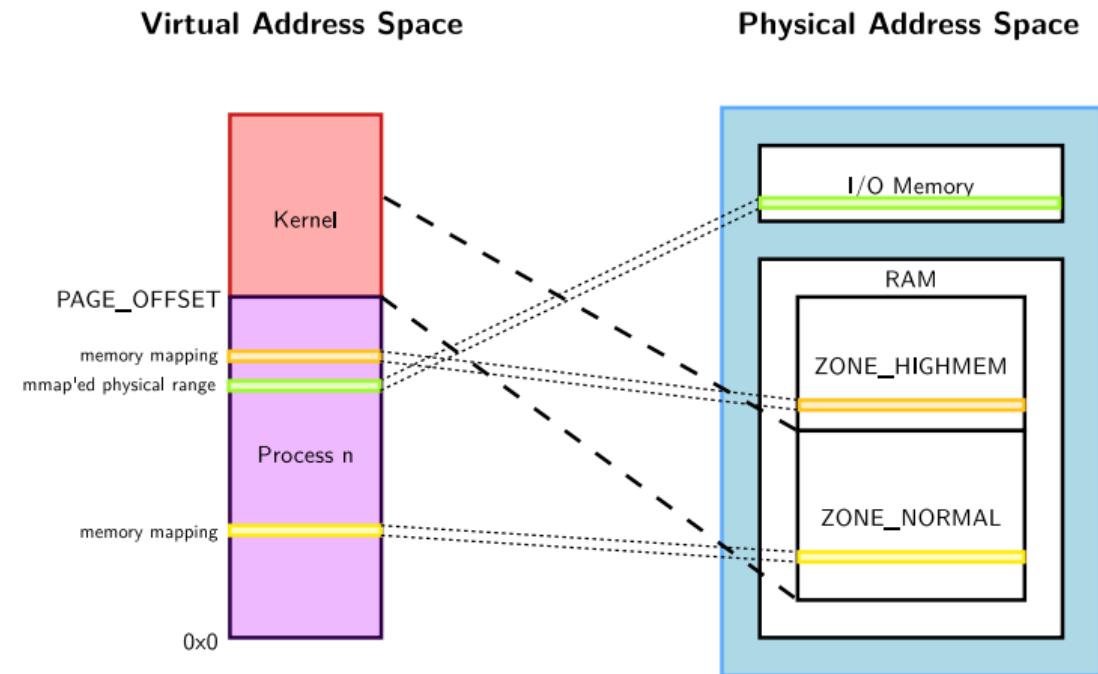
# Physical/virtual memory mapping on 64-bit systems (4kiB-pages)





# 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
  - ...





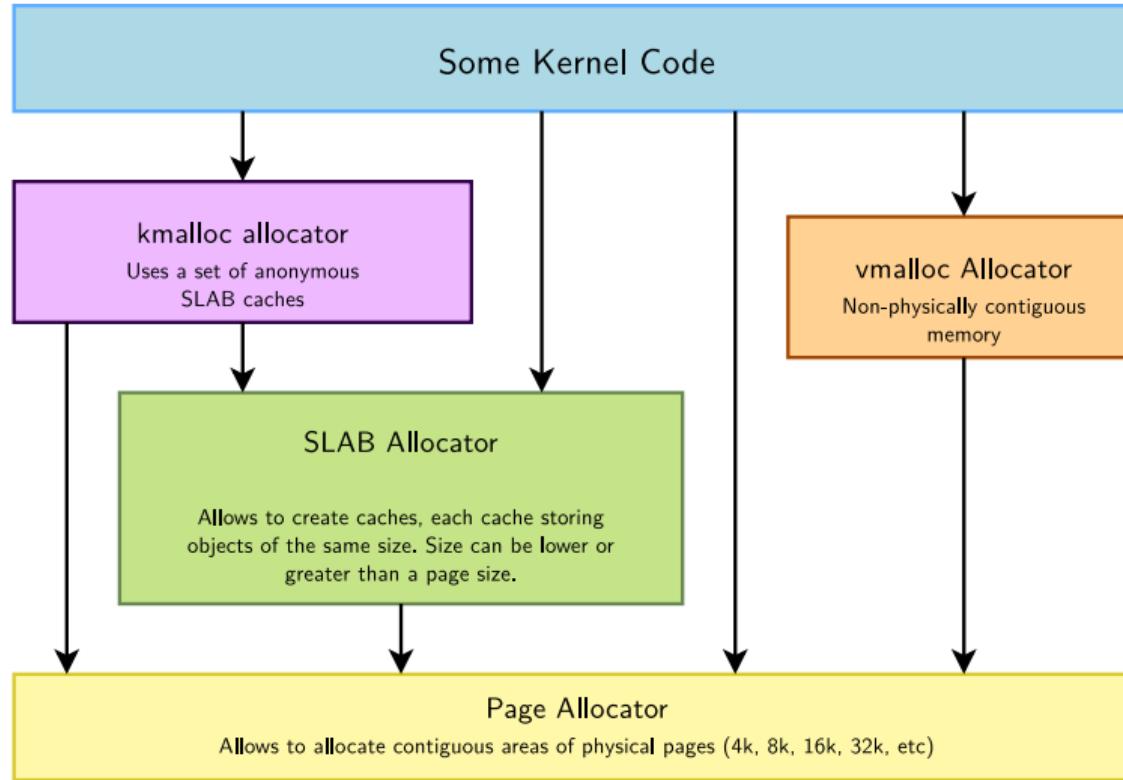
# Userspace memory allocations

---

- ▶ 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) ⇒ 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: get free pages

- ▶ `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  $\log_2(\text{number\_of\_pages})$ . 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.h](#).  
See also the documentation in [core-api/memory-allocation](#)



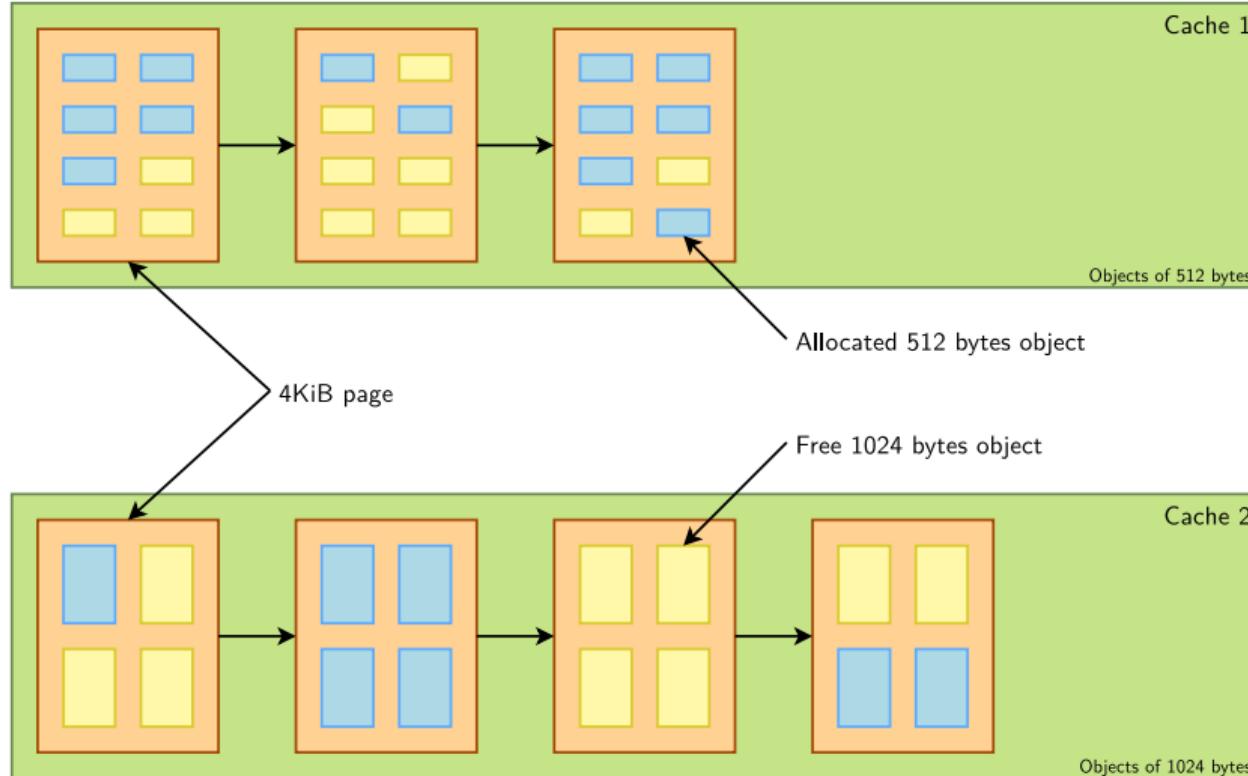
# SLAB allocator 1/2

---

- ▶ 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





# Different SLAB allocators

There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.

- ▶ SLAB: legacy, well proven allocator.  
Linux 5.10 on arm (32 bit): used in 39 defconfig files
- ▶ SLOB: much simpler. More space efficient but doesn't scale well.  
Can save space in small systems (depends on `CONFIG_EXPERT`).  
Linux 5.10 on arm (32 bit): used in 7 defconfig files  
Results on BeagleBone Black: -5 KB compressed kernel size, +1.43 s boot time!
- ▶ SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation. Now the default allocator.  
Linux 5.10 on arm (32 bit): used in 9 defconfig files  
Results on BeagleBone Black: +4 KB compressed kernel, + 2ms total boot time.

## Choose SLAB allocator

SLAB

SLUB (Unqueued Allocator)  
 SLOB (Simple Allocator)



## 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);
```



- ▶ `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

- ▶ 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.
  - Only available on x86, arm64 and powerpc (Linux 5.13 status)
  - 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 Virtual Addresses (Small Mem)

Virtual Address Space      Physical Address Space

Kernel Virtual Addresses (4GB)

Kernel Logical Addresses

PAGE\_OFFSET

Userspace Addresses

Physical RAM

0x00000000

Embedded Linux Conference Europe

OpenIoTSummit Europe

▶ ▶ 🔍 22:29 / 51:18 CC 🔍 ☰



# I/O Memory

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

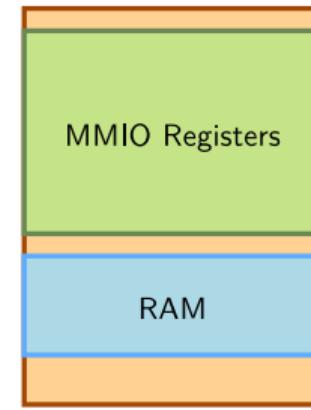
Corrections, suggestions, contributions and translations are welcome!





# 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 example - ARM 32 bit (BeagleBone Black, Linux 5.11)

```
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
4804c000-4804cffff : 4804c000 gpio gpio@0
48060000-48060fff : 48060000 mmc mmc@0
4819c000-4819cffff : 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

- ▶ 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:

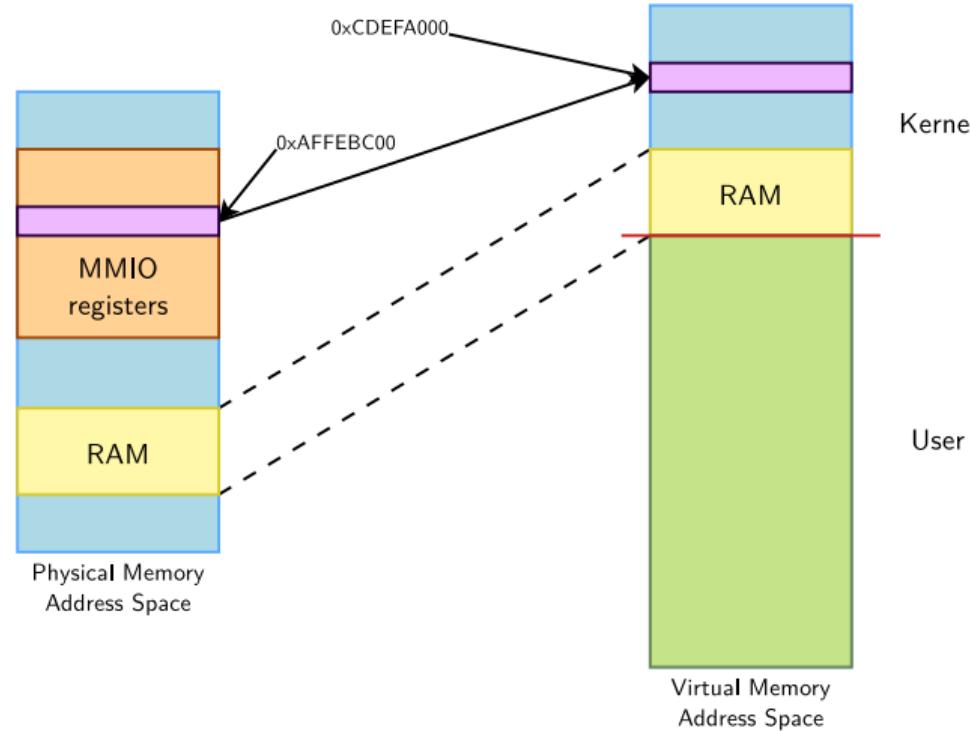
```
#include <asm/io.h>
```

```
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()



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



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 pdev`, not a pointer to `struct device`:
  - Example: `drivers/char/hw_random/st-rng.c`:

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



## Accessing MMIO devices: using accessor functions

---

- ▶ 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 <https://docs.kernel.org/driver-api/device-io.html>



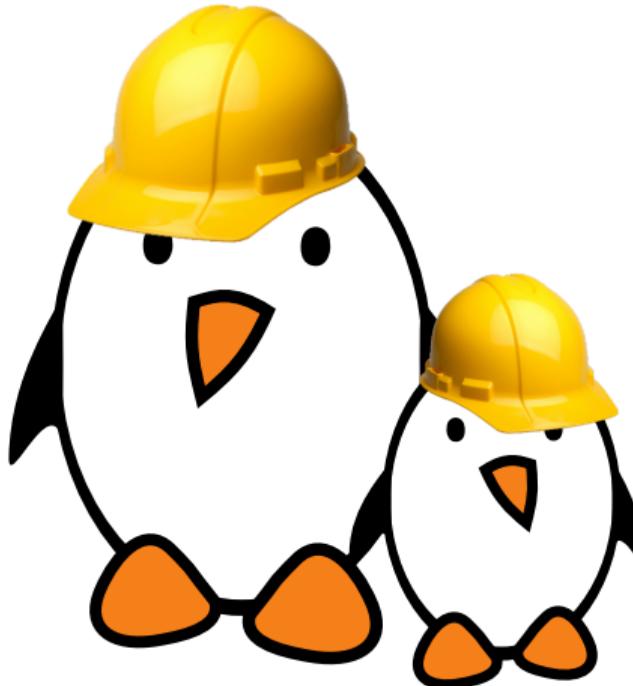
- ▶ Reads/writes to MMIO-mapped registers of given device are done in program order
- ▶ However reads/writes to RAM can be re-ordered between themselves, and between MMIO-mapped read/writes
- ▶ Some of the accessor functions include memory barriers to help with this:
  - Write operation starts with a 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](#)



- ▶ 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.
- ▶ On x86, arm, arm64, riscv, powerpc, parisc, s390: `CONFIG_STRICT_DEVMEM` option to restrict `/dev/mem` to non-RAM addresses, for security reasons (Linux 5.12 status). `CONFIG_IO_STRICT_DEVMEM` goes beyond and only allows to access *idle* I/O ranges (not appearing in `/proc/iomem`).



## Practical lab - I/O memory and ports



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



## The misc subsystem

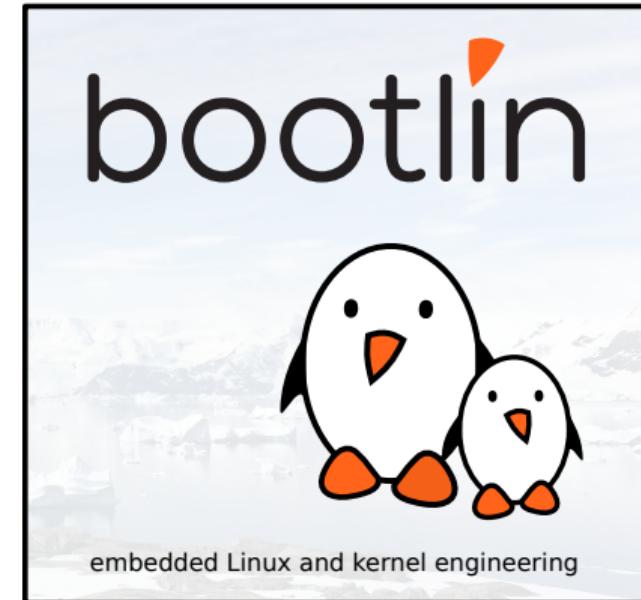
---

# The misc subsystem

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





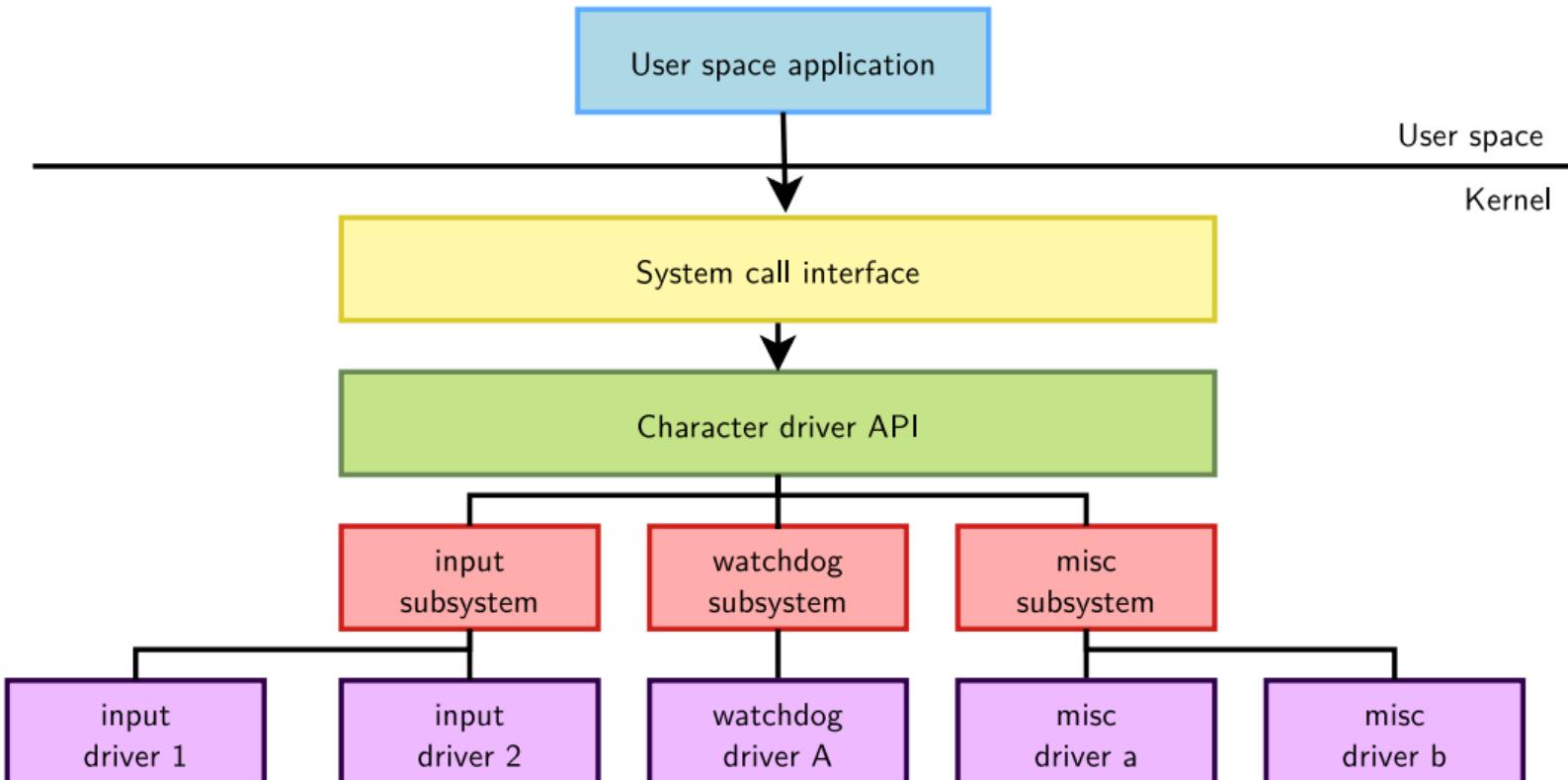
## Why a *misc* subsystem?

---

- ▶ 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:

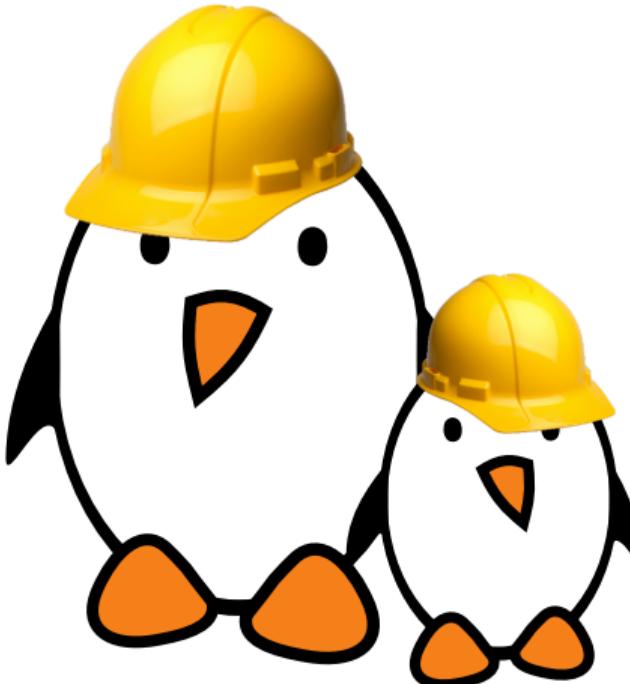
- ▶ `minor`, the minor number for the device, or `MISC_DYNAMIC_MINOR` to get a minor number automatically assigned.
- ▶ `name`, name of the device, which will be used to create the device node if `devtmpfs` is used.
- ▶ `fops`, pointer to the same `struct file_operations` structure that is used for raw 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.)



- ▶ *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.



## Practical 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.

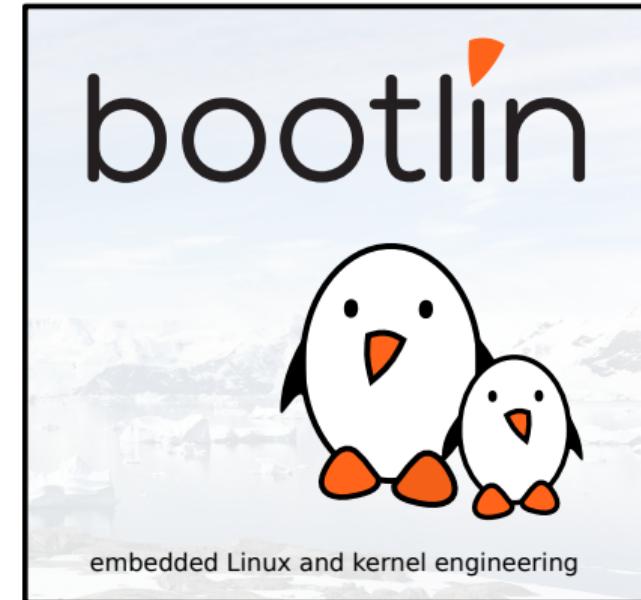


# Processes, scheduling and interrupts

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Processes and scheduling

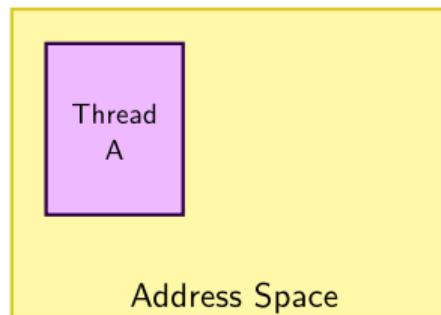


- ▶ Confusion about the terms *process*, *thread* and *task*
- ▶ In UNIX, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - A single thread, which is the only entity known by the scheduler.
- ▶ Additional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

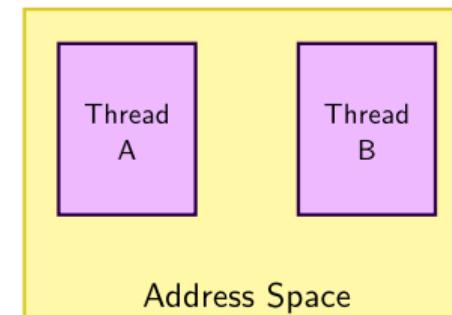


## Process, thread: kernel point of view

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



Process after `fork()`

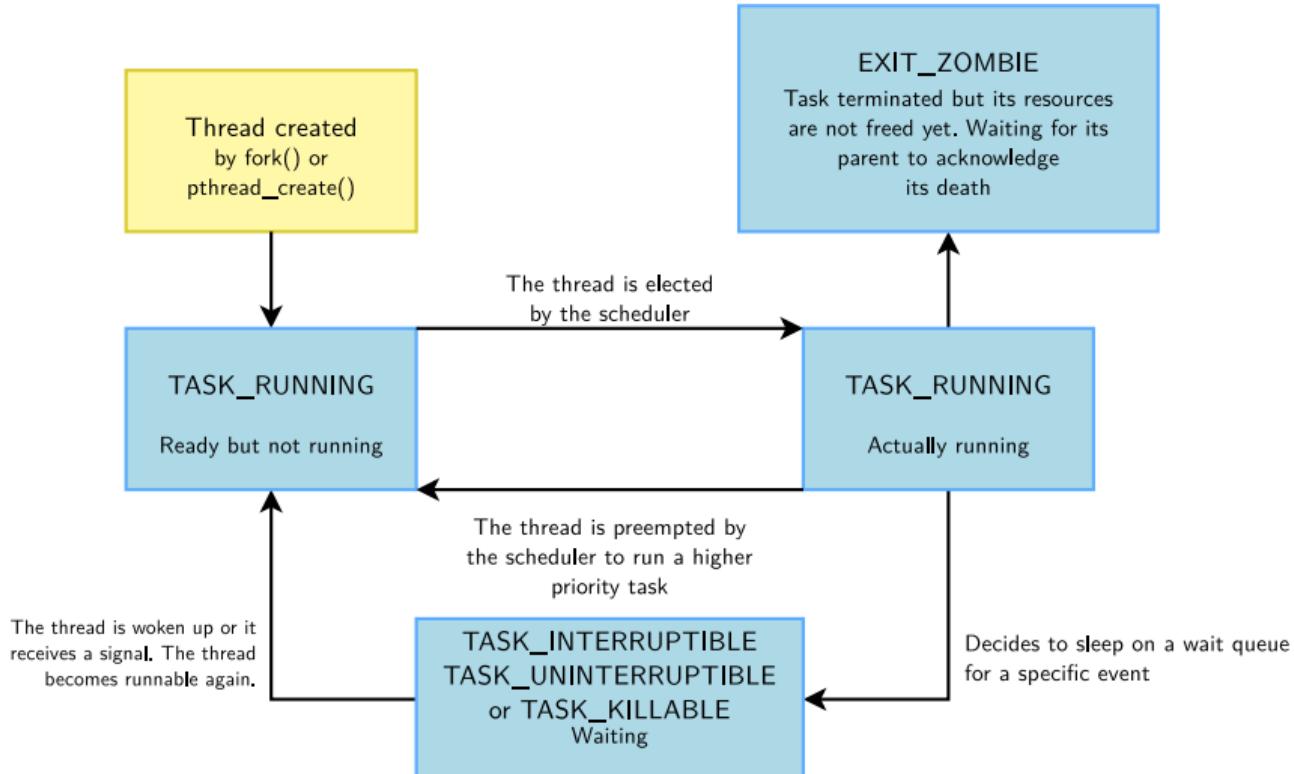


Same process after  
`pthread_create()`

- ▶ When speaking about *process* and *thread*, these concepts need to be clarified:
  - *Mode* is the level of privilege allowing to perform some operations:
    - *Kernel Mode*: in this level CPU can perform any operation allowed by its architecture; any instruction, any I/O operation, any area of memory accessed.
    - *User Mode*: in this level, certain instructions are not permitted (especially those that could alter the global state of the machine), some memory areas cannot be accessed.
  - Linux splits its *address space* in *kernel space* and *user space*
    - *Kernel space* is reserved for code running in *Kernel Mode*.
    - *User space* is the place where applications execute (accessible from *Kernel Mode*).
  - *Context* represents the current state of an execution flow.
    - The *process context* can be seen as the content of the registers associated to this process: execution register, stack register...
    - The *interrupt context* replaces the *process context* when the interrupt handler is executed.

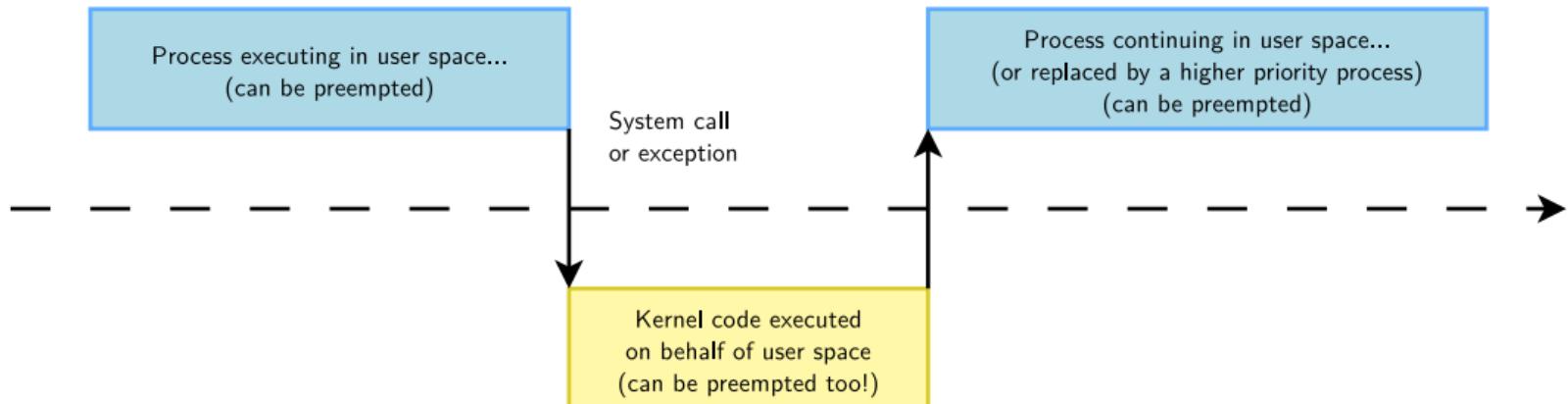


# A thread life





# Execution of system calls



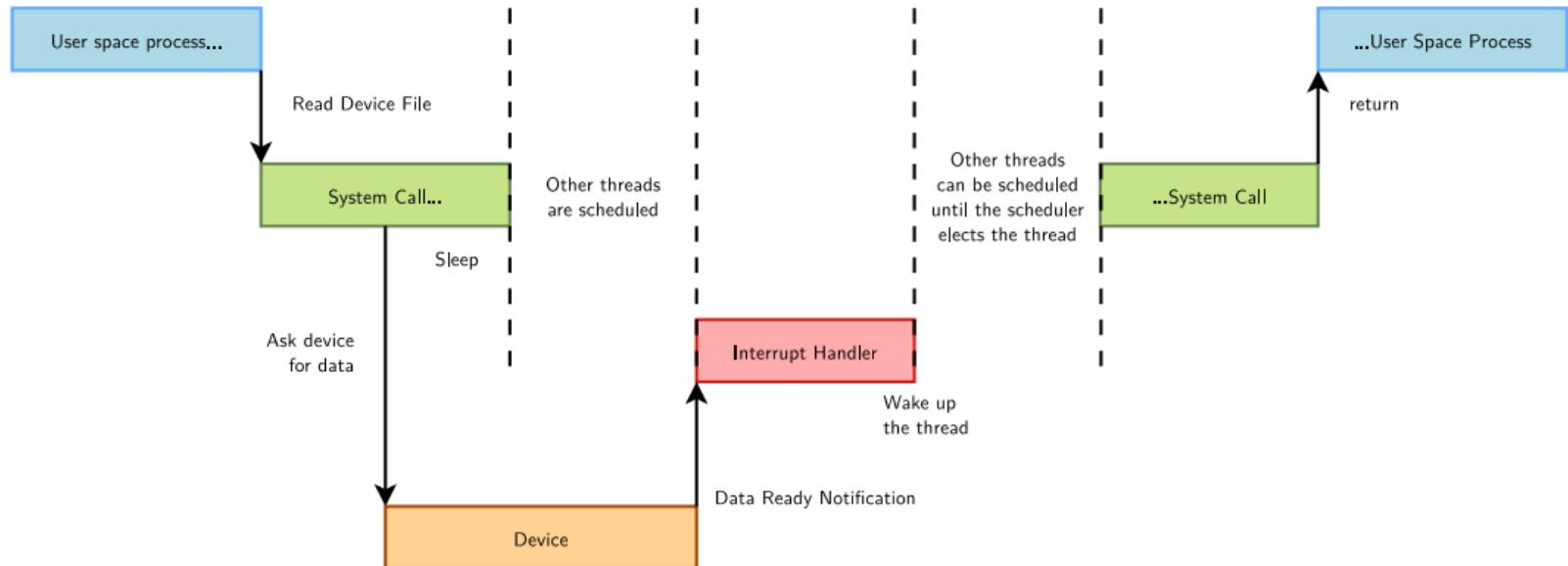
The execution of system calls takes place in the context of the thread requesting them.



## 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, which will be used 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);
```

- ▶ Static queue declaration:
  - Using a global variable when a global resource is sufficient
  - DECLARE\_WAIT\_QUEUE\_HEAD(module\_queue);



## How to sleep with a waitqueue 2/3

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

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



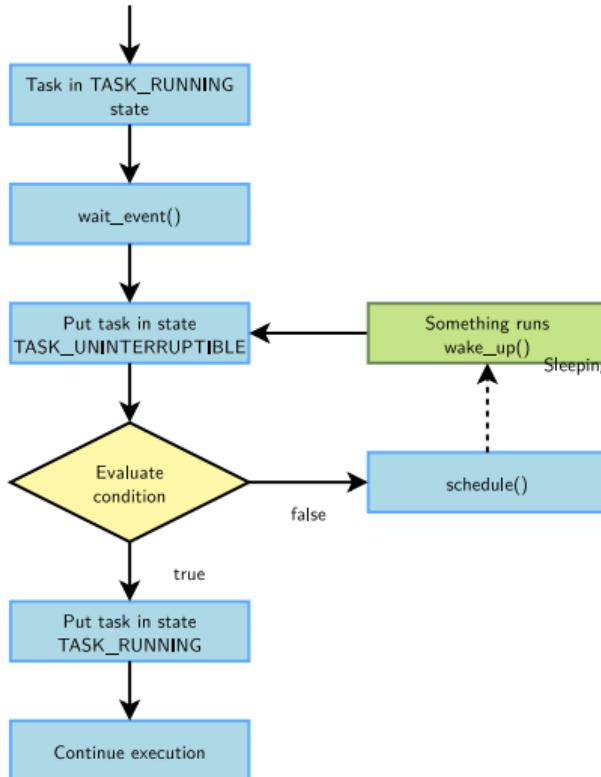
## Exclusive vs. non-exclusive

---

- ▶ `wait_event_interruptible()` puts a task in a non-exclusive wait.
  - All non-exclusive tasks are woken up by `wake_up()` / `wake_up_interruptible()`
- ▶ `wait_event_interruptible_exclusive()` puts a task in an exclusive wait.
  - `wake_up()` / `wake_up_interruptible()` wakes up all non-exclusive tasks and only one exclusive task
  - `wake_up_all()` / `wake_up_interruptible_all()` wakes up all non-exclusive and all exclusive tasks
- ▶ Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to “consume” the event.
- ▶ Non-exclusive sleeps are useful when the event can “benefit” to multiple tasks.



# 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.



## How to sleep with completions 1/2

- ▶ Use `wait_for_completion()` when no particular condition must be enforced at the time of the wake-up
  - Leverages the power of wait queues
  - Simplifies its use
  - Highly efficient using low level scheduler facilities
- ▶ Preparation of the completion structure:
  - Static declaration and initialization:

```
DECLARE_COMPLETION(setup_done);
```
  - Dynamic declaration:

```
init_completion(&object->setup_done);
```
  - The completion object should get a meaningful name (eg. not just “done”).
- ▶ Ready to be used by signal consumers and providers as soon as the completion object is initialized
- ▶ See `include/linux/completion.h` for the full API
- ▶ Internal documentation at `scheduler/completion.rst`



## How to sleep with completions 2/2

- ▶ Enter a wait state with

```
void wait_for_completion(struct completion *done)
```

- All `wait_event()` flavors are also supported, such as:  
`wait_for_completion_timeout()`,  
`wait_for_completion_interruptible{,_timeout}()`,  
`wait_for_completion_killable{,_timeout}()`, etc

- ▶ Wake up consumers with

```
void complete(struct completion *done)
```

- Several calls to `complete()` are valid, they will wake up the same number of threads waiting on this object (acts as a FIFO).
- A single `complete_all()` call would wake up all present and future threads waiting on this completion object

- ▶ Reset the counter with

```
void reinit_completion(struct completion *done)
```

- Resets the number of “done” completions still pending
- Mind not to call `init_completion()` twice, which could confuse the enqueued tasks



# Waiting when there is no interrupt

- ▶ 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 interruption 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



# Waiting when hardware is involved

- ▶ When hardware is involved in the waiting process
  - but there is no interrupt available
  - or because a context switch would be too expensive
- ▶ Specific polling I/O accessors may be used:
  - Exhaustive list in [include/iopoll.h](#)

```
int read[bwlq]_poll[_timeout[_atomic]](addr, val, cond,
                                         delay_us, timeout_us)
```

    - addr: I/O memory location
    - val: Content of the register pointed with
    - cond: Boolean condition based on val
    - delay\_us: Polling delay between reads
    - timeout\_us: Timeout delay after which the operation fails and returns -ETIMEDOUT
  - \_atomic variant uses [udelay\(\)](#) instead of [usleep\(\)](#).



## 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.



## Releasing an interrupt handler

---

```
void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);
```

- ▶ Explicitly release an interrupt handler. Done automatically in normal situations.

Defined in [include/linux/interrupt.h](#)



## 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

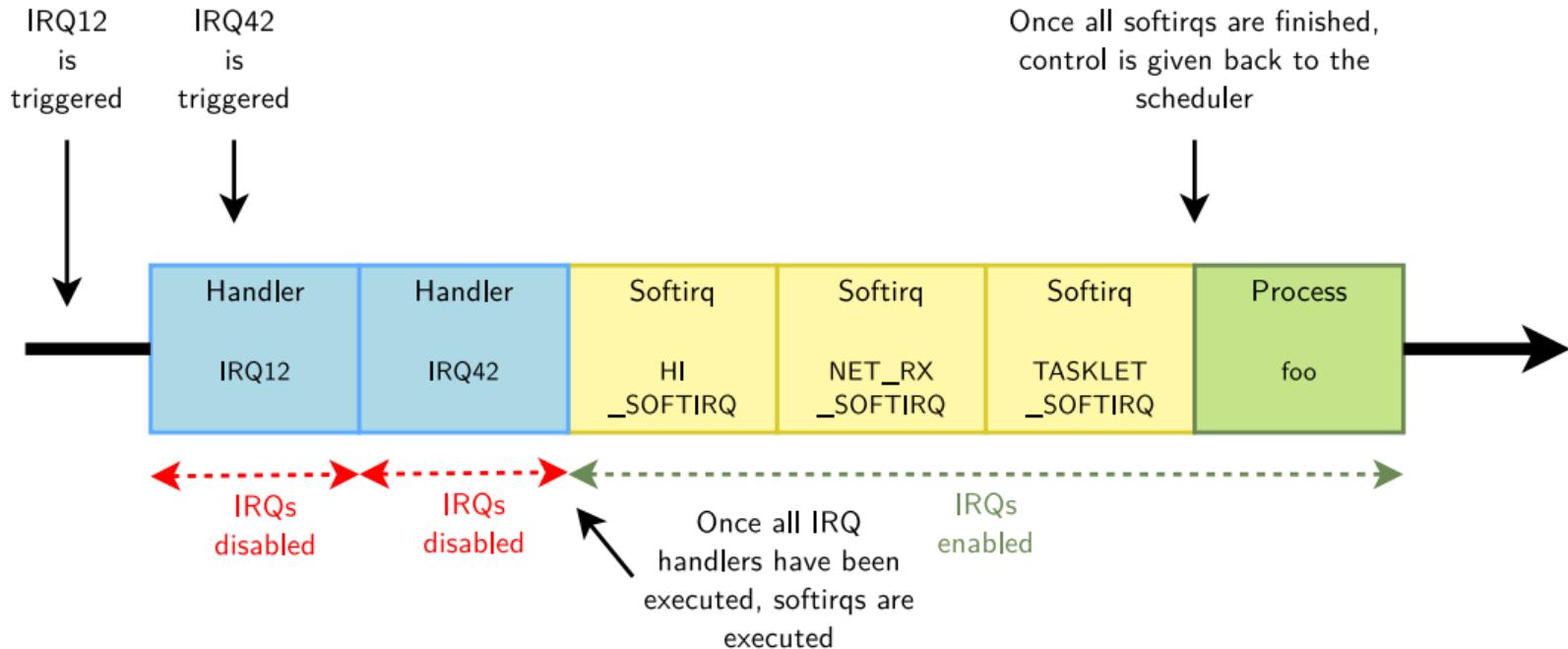
---

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 or workqueues.



# Top half and bottom half diagram





- ▶ 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 threshold ("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](https://en.wikipedia.org/wiki/New_API) for details
- ▶ See also our commented network driver on <https://frama.link/qCaWu1-U>



- ▶ 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)



## Tasklet example: drivers/crypto/atmel-sha.c 1/2

```
/* The tasklet function */
static void atmel_sha_done_task(unsigned long data)
{
    struct atmel_sha_dev *dd = (struct atmel_sha_dev *)data;
    [...]
}

/* Probe function: registering the tasklet */
static int atmel_sha_probe(struct platform_device *pdev)
{
    struct atmel_sha_dev *sha_dd; /* Per device structure */
    [...]
    platform_set_drvdata(pdev, sha_dd);
    [...]
    tasklet_init(&sha_dd->done_task, atmel_sha_done_task,
                (unsigned long)sha_dd);
    [...]
    err = devm_request_irq(&pdev->dev, sha_dd->irq, atmel_sha_irq,
                          IRQF_SHARED, "atmel-sha", sha_dd);
    [...]
}
```



## Tasklet example: drivers/crypto/atmel-sha.c 2/2

```
/* Remove function: removing the tasklet */
static int atmel_sha_remove(struct platform_device *pdev)
{
    static struct atmel_sha_dev *sha_dd;
    sha_dd = platform_get_drvdata(pdev);
    [...]
    tasklet_kill(&sha_dd->done_task);
    [...]
}

/* Interrupt handler: triggering execution of the tasklet */
static irqreturn_t atmel_sha_irq(int irq, void *dev_id)
{
    struct atmel_sha_dev *sha_dd = dev_id;
    [...]
    tasklet_schedule(&sha_dd->done_task);
    [...]
}
```



# 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 work which can be scheduled.
- ▶ Workqueues may be created by subsystems or drivers with `alloc_workqueue()`. The default queue can also be used.
- ▶ Functions registered to run in workqueues, called workers, are executed in thread context which means:
  - All interrupts are enabled
  - Sleeping is allowed
- ▶ A worker is usually allocated in a per-device structure, initialized and registered with `INIT_WORK()` and typically triggered with `queue_work()` when using a dedicated queue or `schedule_work()` when using the default queue
- ▶ The complete API is in `include/linux/workqueue.h`
- ▶ Example (`drivers/crypto/atmel-i2c`):

```
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

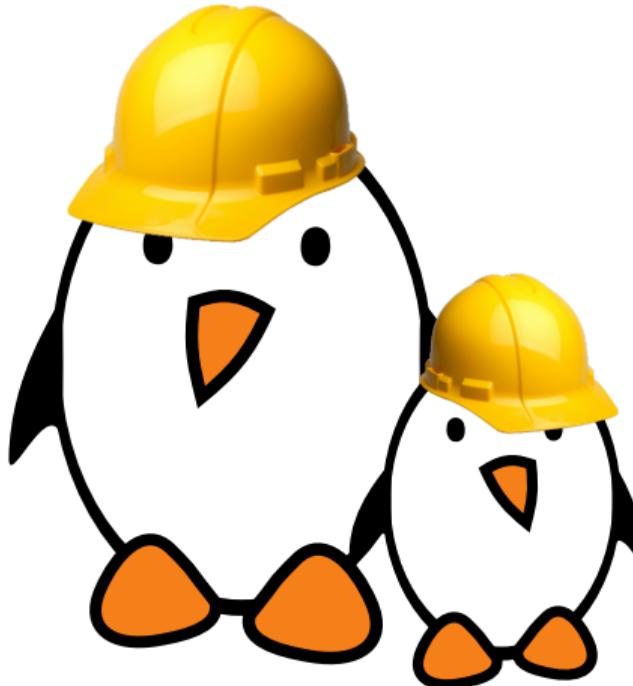
## ► Device driver

- In the `remove()` function, for each device, the interrupt handler is automatically unregistered.



## Practical lab - Interrupts

---

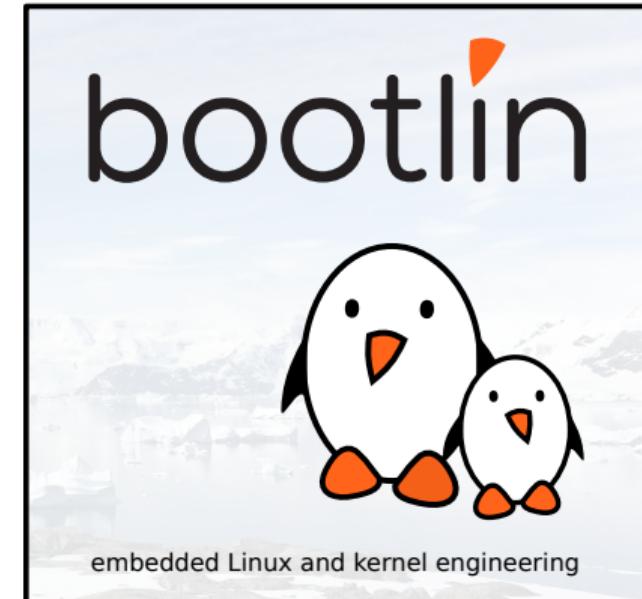


- ▶ Adding read capability to the character driver developed earlier.
- ▶ Register an interrupt handler for each device.
- ▶ Waiting for data to be available in the read file operation.
- ▶ Waking up the code when data are available from the devices.



# Concurrent Access to Resources: Locking

© Copyright 2004-2023, Bootlin.  
Creative Commons BY-SA 3.0 license.  
Corrections, suggestions, contributions and translations are welcome!





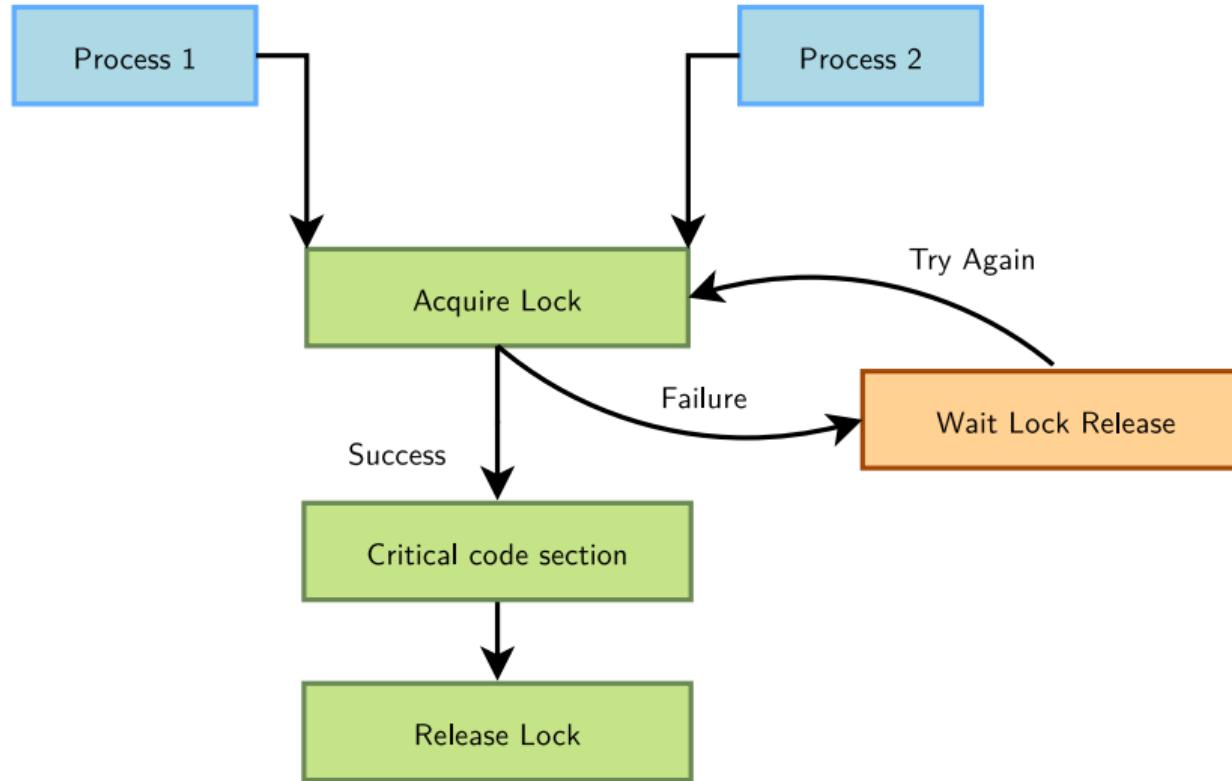
# Sources of concurrency issues

---

- ▶ 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





*mutex = mutual exclusion*

- ▶ 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

- ▶ `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: can't 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.



## Locking and unlocking mutexes 2/2

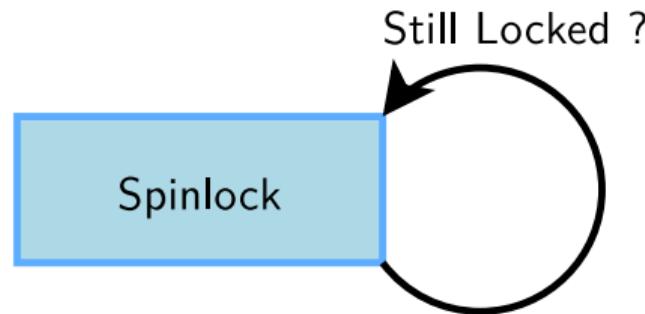
---

- ▶ `int mutex_trylock(struct mutex *lock);`
  - Never waits. Returns a non zero value if the mutex is not available.
- ▶ `int mutex_is_locked(struct mutex *lock);`
  - Just tells whether the mutex is locked or not.
- ▶ `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.





# Initializing spinlocks

---

- ▶ Statically (unusual)
  - `DEFINE_SPINLOCK(my_lock);`
- ▶ Dynamically (the usual case, on a per-device basis)
  - `void spin_lock_init(spinlock_t *lock);`

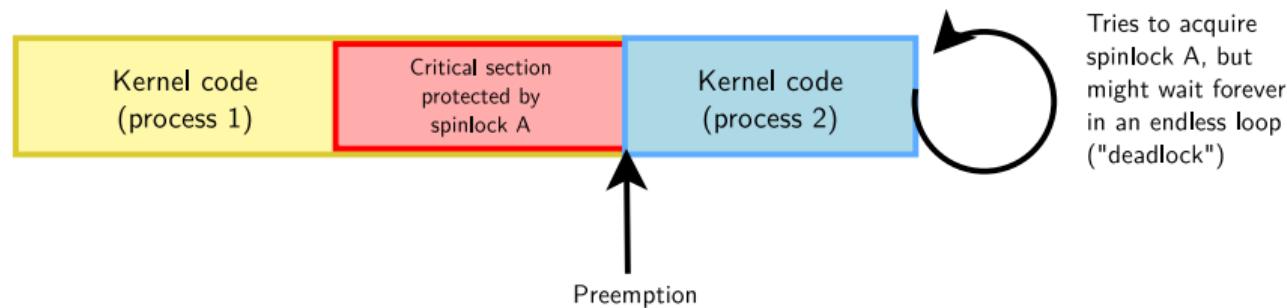


# Using spinlocks 1/3

Several variants, depending on where the spinlock is called:

- ▶ `void spin_lock(spinlock_t *lock);`
- ▶ `void spin_unlock(spinlock_t *lock);`

- Used for locking in process context (critical sections in which you do not want to sleep) as well as atomic sections.
- 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:

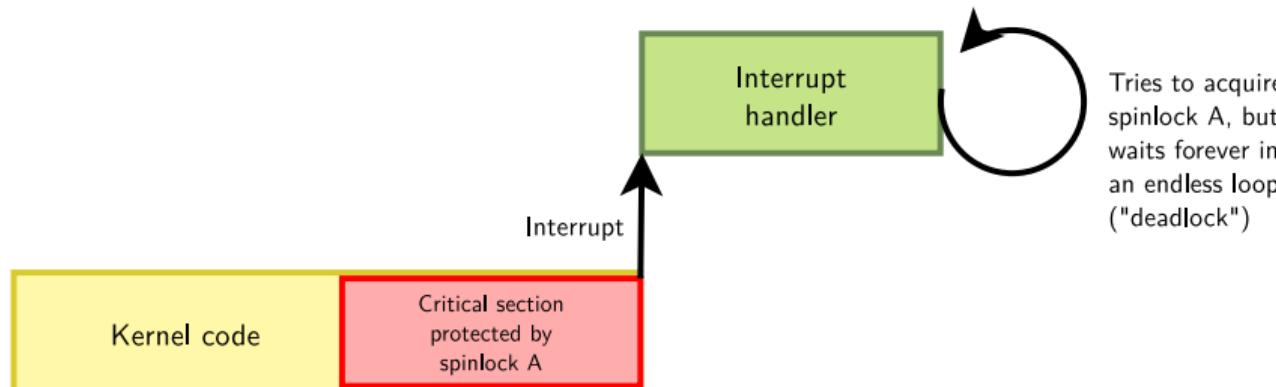


- Disabling kernel preemption also disables migration to avoid the same kind of issue as pictured above from happening.



## Using spinlocks 2/3

- ▶ `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.
  - We need to avoid deadlocks because of interrupts that want to get the same lock.





## 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

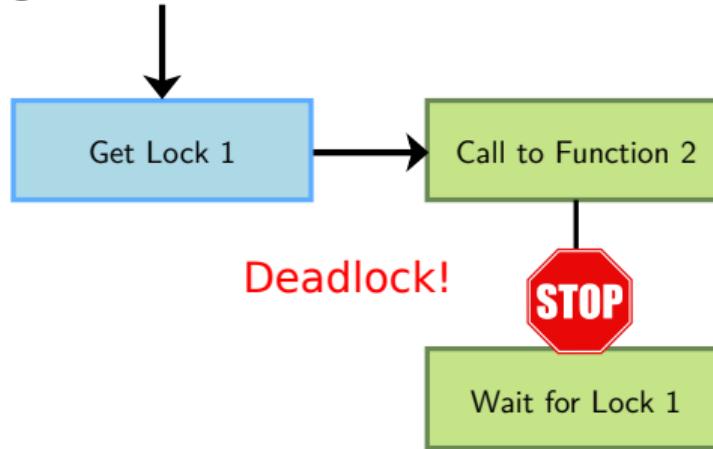
```
static unsigned int ultite_tx_empty(struct uart_port *port) {  
    unsigned long flags;  
  
    spin_lock_irqsave(&port->lock, flags);  
    /* Do something */  
    spin_unlock_irqrestore(&port->lock, flags);  
}
```



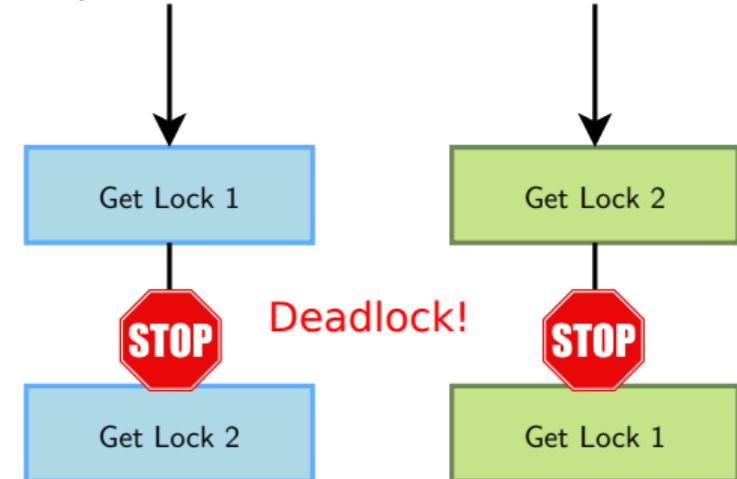
## More deadlock situations

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!





- ▶ 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 the dmesg in case of such violation.



- ▶ 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.



- ▶ Conditions where RCU is useful:
  - Frequent reads but infrequent writes
  - Focus on getting consistent data rather than getting the latest data
- ▶ Kind of enforces ownership by enforcing space/time synchronization
- ▶ RCU API ([Documentation/RCU/whatisRCU.rst](#)):
  - `rcu_read_lock()` and `rcu_read_unlock()`: reclaim/release read access
  - `synchronize_rcu()`, `call_rcu()` or `kfree_rcu()`: wait for pre-existing readers
  - `rcu_assign_pointer()`: update RCU-protected pointer
  - `rcu_dereference()`: load RCU-protected pointer
- ▶ RCU mentorship session by Paul E. McKenney: <https://youtu.be/K-4TI5gFsig>



# RCU example: ensuring consistent accesses (1/2)

## Unsafe read/write

```
struct myconf { int a, b; } *shared_conf; /* initialized */

unsafe_get(int *cur_a, int *cur_b)
{
    *cur_a = shared_conf->a;
    /* What if *current_conf gets updated now? The assignment is inconsistent! */
    *cur_b = shared_conf->b;
};

unsafe_set(int new_a, int new_b)
{
    shared_conf->a = new_a;
    shared_conf->b = new_b;
};
```



# RCU example: ensuring consistent accesses (2/2)

## Safe read/write with RCU

```
struct myconf { int a, b; } *shared_conf; /* initialized */

safe_get(int *cur_a, int *cur_b)
{
    struct myconf *temp;

    rcu_read_lock();
    temp = rcu_dereference(shared_conf);
    *cur_a = temp->a;
    /* If *shared_conf is updated, temp->a and temp->b will remain consistent! */
    *cur_b = temp->b;
    rcu_read_unlock();
};

safe_set(int new_a, int new_b)
{
    struct myconf *newconf = kmalloc(...);
    struct myconf *oldconf;

    oldconf = rcu_dereference(current_conf);
    newconf->a = new_a;
    newconf->b = new_b;
    rcu_assign_pointer(current_conf, newconf);
    /* Readers might still have a reference over the old struct here... */
    synchronize_rcu();
    /* ...but not here! No more readers of the old struct, kfree() is safe! */
    kfree(oldconf);
};
```



# Atomic variables 1/2

```
#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);`



► 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

---

- ▶ 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.

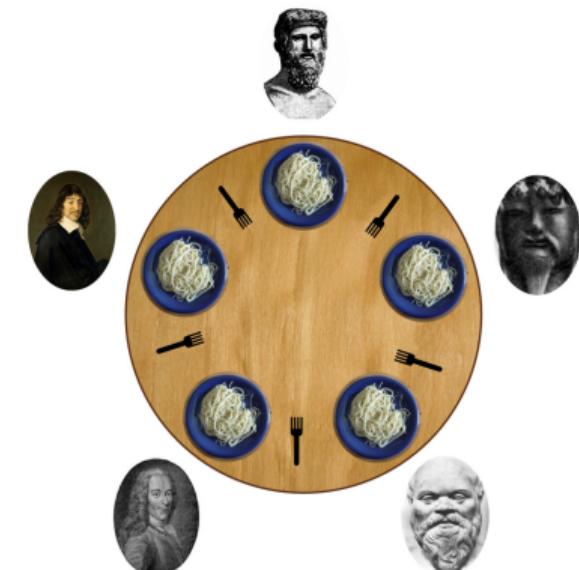


Image source: Wikipedia  
(<https://frama.link/xg3WnD0F>)



## Practical lab - Locking



- ▶ Add locking to the driver to prevent concurrent accesses to shared resources

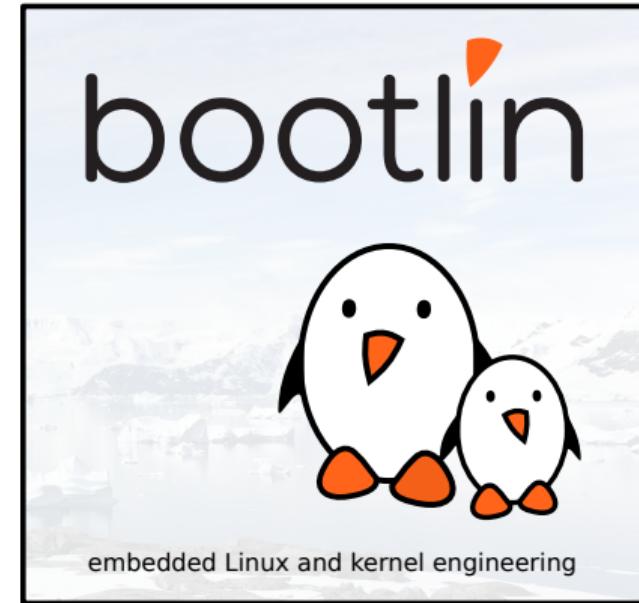


# Direct Memory Access

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



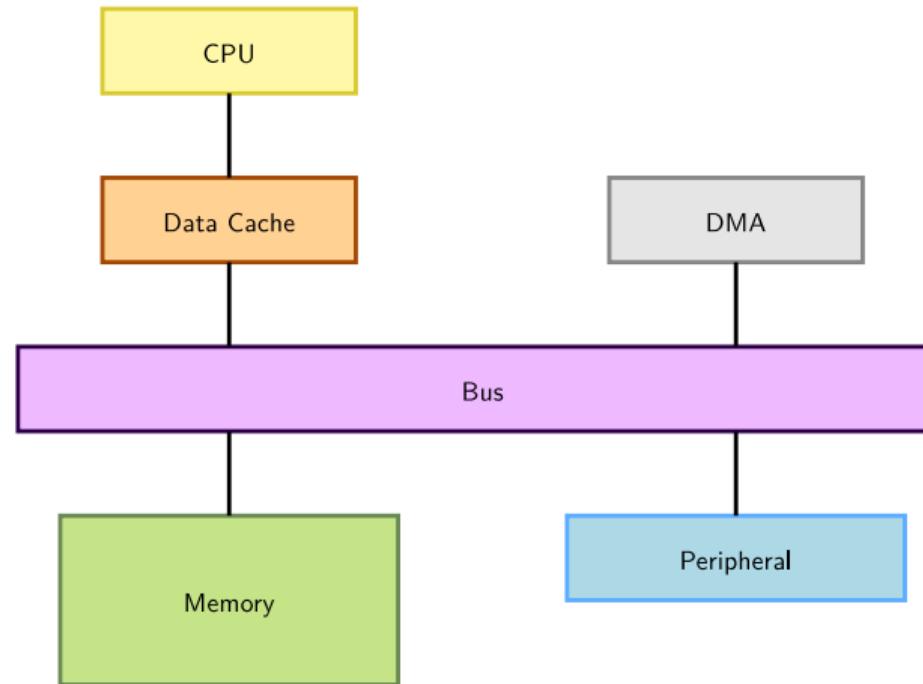


## DMA main principles



# DMA integration

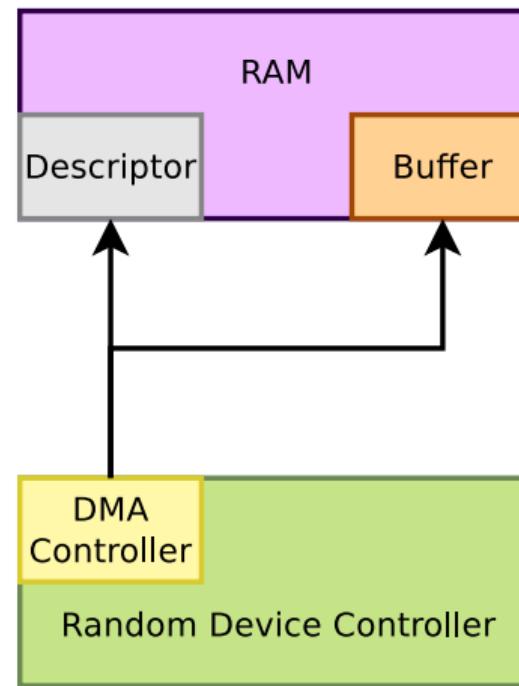
DMA (*Direct Memory Access*) is used to copy data directly between devices and RAM, without going through the CPU.





## Peripheral DMA

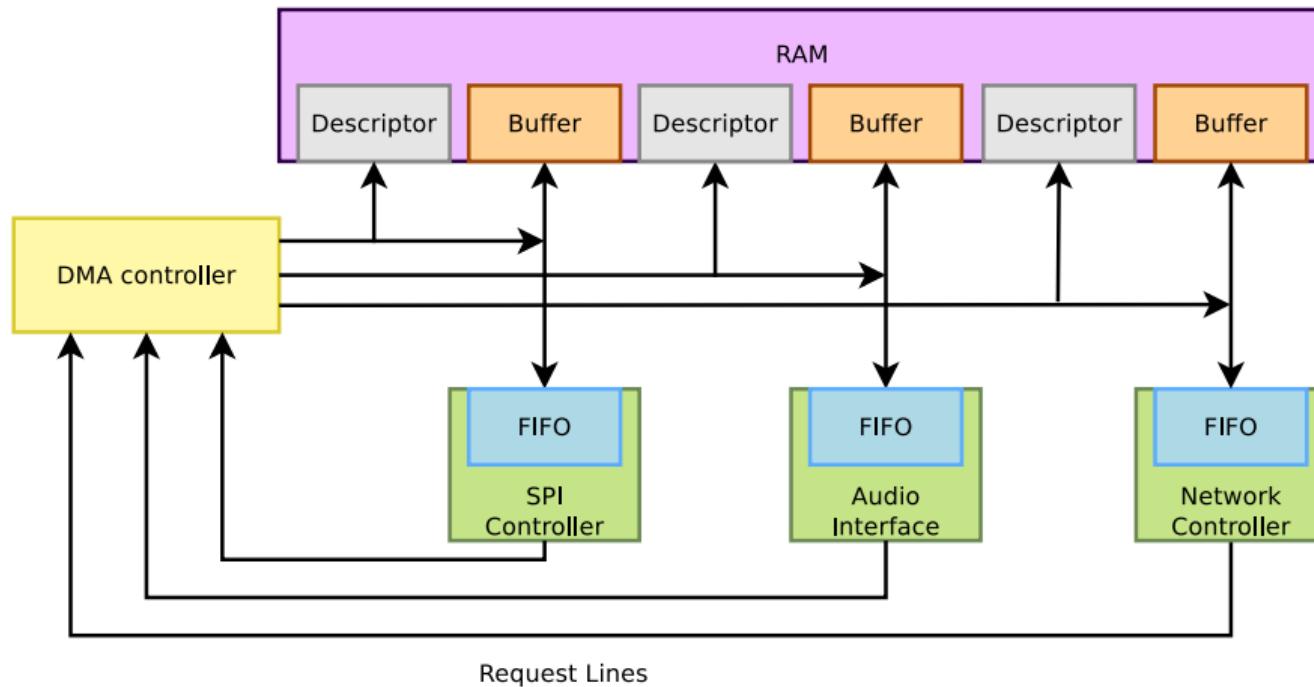
Some device controllers embedded their own DMA controller and therefore can do DMA on their own.





# DMA controllers

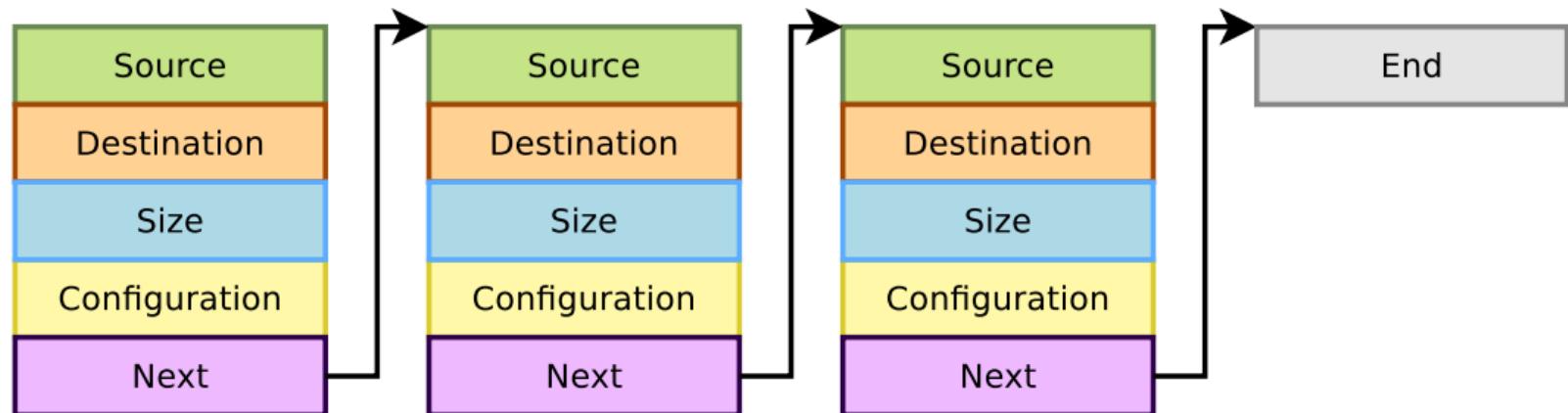
Other device controllers rely on an external DMA controller (on the SoC). Their drivers need to submit DMA descriptors to this controller.





# DMA descriptors

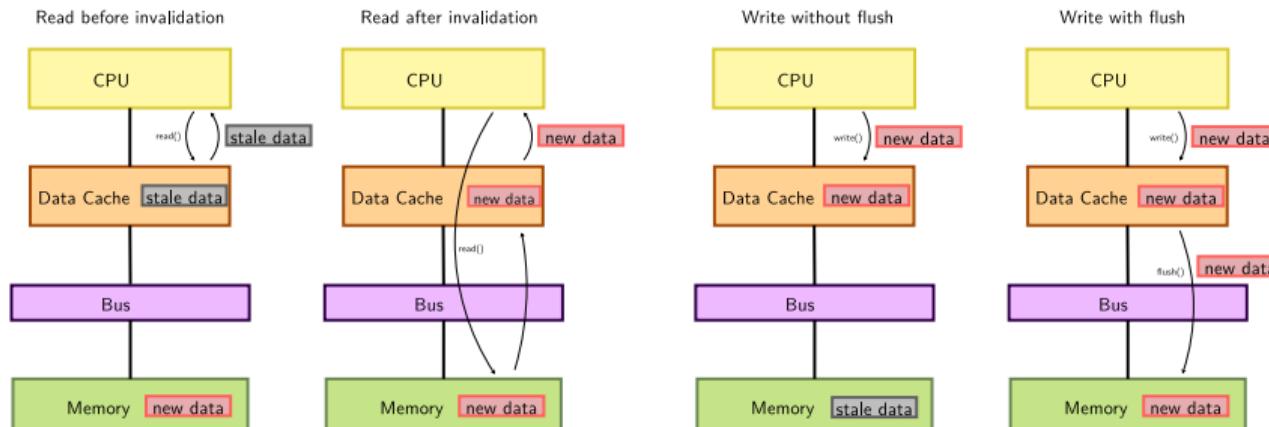
DMA descriptors describe the various attributes of a DMA transfer, and are chained.





# Cache constraints

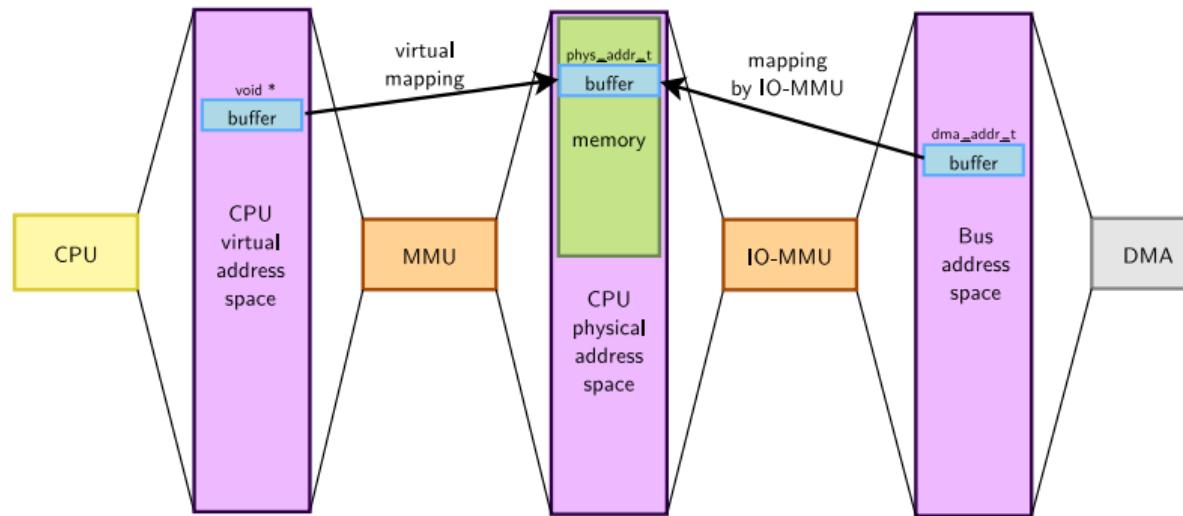
- ▶ The CPU can access memory through a data cache
  - Using the cache can be more efficient (faster accesses to the cache than the bus)
- ▶ But the DMA does not access the CPU cache, so one needs to take care of cache coherency (cache content vs. memory content):
  - When the CPU reads from memory accessed by DMA, the relevant cache lines must be invalidated to force reading from memory again
  - When the CPU writes to memory before starting DMA transfers, the cache lines must be flushed/cleaned in order to force the data to reach the memory





# DMA addressing constraints

- ▶ Memory and devices have physical addresses: `phys_addr_t`
- ▶ CPUs usually access memory through an MMU, using virtual pointers: `void *`
- ▶ DMA controllers do not access memory through the MMU and thus cannot manipulate virtual addresses, instead they access a `dma_addr_t` through either:
  - physical addresses directly
  - an IOMMU, in which case a specific mapping must be created





# DMA memory allocation constraints

---

The APIs must remain generic and handle all cases transparently, hence:

- ▶ Each memory chunk accessed by the DMA shall be physically contiguous, which means one can use:
  - any memory allocated by `kmalloc()` (up to 128 KB)
  - any memory allocated by `__get_free_pages()` (up to 8MB)
  - block I/O and networking buffers, designed to support DMA
- ▶ Unless the buffer is smaller than one page, one cannot use:
  - kernel memory allocated with `vmalloc()`
  - user memory allocated with `malloc()`
    - Almost all the time userspace relies on the kernel to allocate the buffers and `mmap()` them to be usable from userspace (requires a dedicated user API)



## Kernel APIs for DMA



## dma-mapping vs. dmaengine vs. dma-buf

---

The dma-mapping API:

- ▶ Allocates and manages DMA buffers
- ▶ Offers generic interfaces to handle coherency
- ▶ Manages IO-MMU DMA mappings when relevant
- ▶ See [core-api/dma-api](#) and [core-api/dma-api-howto](#)

The dmaengine API:

- ▶ Abstracts the DMA controller
- ▶ Offers generic functions to configure, queue, trigger, stop transfers
- ▶ Unused when dealing with peripheral DMA
- ▶ See [driver-api/dmaengine/client](#) and

The dma-buf API:

- ▶ Enables sharing DMA buffers between devices within the kernel
- ▶ Not covered in this training



## ► Coherent mappings

- The kernel allocates a suitable buffer and sets the mapping for the driver
- Can simultaneously be accessed by the CPU and device
- So, has to be in a cache coherent memory area
- Usually allocated for the whole time the module is loaded
  - Can be expensive to setup and use on some platforms
  - Typically implemented by disabling cache on ARM

## ► Streaming mappings

- Use an already allocated buffer
- The driver provides a buffer, the kernel just sets the mapping
- Mapping set up for each transfer (keeps DMA registers free on the hardware)



## dma-mapping: memory addressing constraints

- ▶ The default addressing capability of the DMA controllers is assumed to be 32-bit.
- ▶ If the platform supports it, the DMA addressing capability can be:
  - increased (eg. need to access highmem)
  - decreased (eg. ISA devices, where `kmalloc()` buffers can also be allocated in the first part of the RAM with `GFP_DMA`)
- ▶ Linux stores this capability in a per-device mask, DMA mappings can fail because a buffer is out of reach
- ▶ In all cases, the DMA mask shall be consistent before allocating buffers

```
int dma_set_mask_and_coherent(struct device *dev, u64 mask)
```

- ▶ Maximum and optimal buffer sizes can also be retrieved to optimize allocations/buffer handling

```
size_t dma_max_mapping_size(struct device *dev);  
size_t dma_opt_mapping_size(struct device *dev);
```



# dma-mapping: Allocating coherent memory mappings

The kernel takes care of both buffer allocation and mapping:

```
#include <linux/dma-mapping.h>

void * dma_alloc_coherent( /* Output: buffer address */
    struct device *dev, /* device structure */
    size_t size, /* Needed buffer size in bytes */
    dma_addr_t *handle, /* Output: DMA bus address */
    gfp_t gfp /* Standard GFP flags */
);
void dma_free_coherent(struct device *dev,
    size_t size, void *cpu_addr, dma_addr_t handle);
```

Note: called *consistent mappings* on PCI  
(`pci_alloc_consistent()` and `pci_free_consistent()`)



# dma-mapping: Setting up streaming memory mappings (single)

Works on already allocated buffers:

```
#include <linux/dma-mapping.h>

dma_addr_t dma_map_single(
    struct device *,          /* device structure */
    void *,                  /* input: buffer to use */
    size_t,                  /* buffer size */
    enum dma_data_direction /* Either DMA_BIDIRECTIONAL,
                           * DMA_TO_DEVICE or
                           * DMA_FROM_DEVICE */
);
void dma_unmap_single(struct device *dev, dma_addr_t handle,
                      size_t size, enum dma_data_direction dir);
```



## dma-mapping: Setting up streaming memory mappings (multiples)

A scatterlist using the scatter-gather library can be used to map several buffers and link them together

```
#include <linux/dma-mapping.h>
#include <linux/scatterlist.h>

struct scatterlist sglist[NENTS], *sg;
int i, count;

sg_init_table(sglist, NENTS);
sg_set_buf(&sglist[0], buf0, len0);
sg_set_buf(&sglist[1], buf1, len1);

count = dma_map_sg(dev, sglist, NENTS, DMA_TO_DEVICE);
for_each_sg(sglist, sg, count, i) {
    dma_address[i] = sg_dma_address(sg);
    dma_len[i] = sg_dma_len(sg);
}
...
dma_unmap_sg(dev sglist, count, DMA_TO_DEVICE);
```



# dma-mapping: Setting up streaming I/O mappings

Physical addresses with MMIO registers shall always be remapped (otherwise it would not work when they are accessed through an IO-MMU)

```
#include <linux/dma-mapping.h>

dma_addr_t dma_map_resource(
    struct device *,          /* device structure */
    phys_addr_t,              /* input: resource to use */
    size_t,                   /* buffer size */
    enum dma_data_direction,  /* Either DMA_BIDIRECTIONAL,
                               * DMA_TO_DEVICE or
                               * DMA_FROM_DEVICE */
    unsigned long attrs,       /* optional attributes */
);
void dma_unmap_resource(struct device *dev, dma_addr_t handdle,
    size_t size, enum dma_data_direction dir, unsigned long attrs);
```



## dma-mapping: Verifying DMA memory mappings

---

- ▶ All mapping helpers can fail and return errors
- ▶ The right way to check the validity of the returned `dma_addr_t` is to call:  
`int dma_mapping_error(struct device *dev, dma_addr_t dma_addr)`
  - May give additional clues if `CONFIG_DMA_API_DEBUG` is enabled.



# dma-mapping: Syncing streaming DMA mappings

- ▶ In general streaming mappings are:
  - mapped right before use with DMA
    - MEM\_TO\_DEV: caches are flushed
  - unmapped right after
    - DEV\_TO\_MEM: cache lines are invalidated
- ▶ The CPU shall only access the buffer after unmapping!
- ▶ If however the same memory region has to be used for several DMA transfers, the same mapping can be kept in place. In this case the data must be synchronized before CPU access:
  - The CPU needs to access the data:

```
dma_sync_single_for_cpu(dev, dma_handle, size, direction);
dma_sync_sg_for_cpu(dev, sglist, nents, direction);
```
  - The device needs to access the data:

```
dma_sync_single_for_device(dev, dma_handle, size, direction);
dma_sync_sg_for_device(dev, sglist, nents, direction);
```



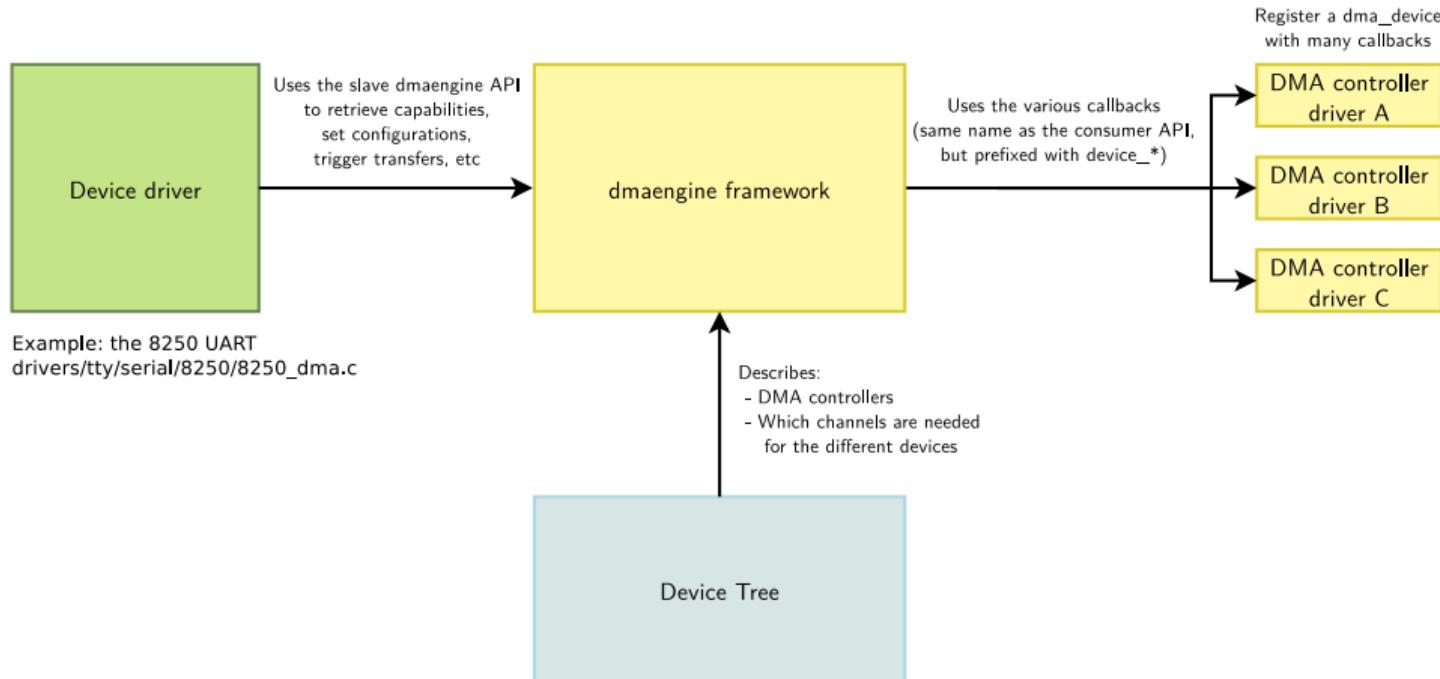
## Starting DMA transfers

---

- ▶ If the device you're writing a driver for is doing peripheral DMA, no external API is involved.
- ▶ If it relies on an external DMA controller, you'll need to
  1. Ask the hardware to use DMA, so that it will drive its request line
  2. Use Linux dmaengine framework, especially its slave API



# The dmaengine framework





# dmaengine: Slave API: Initial configuration

Steps to start a DMA transfer with dmaengine:

1. Request a channel for exclusive use with `dma_request_chan()`, or one of its variants
  - This channel pointer will be used all along
  - Returns a pointer over a `struct dma_chan` which can also be an error pointer
2. Configure the engine by filling a `struct dma_slave_config` structure and passing it to `dmaengine_slave_config()`:

```
struct dma_slave_config txconf = {};  
  
/* Tell the engine what configuration we want on a given channel:  
 * direction, access size, burst length, source and destination).  
 * Source being memory, there is no buswidth or maxburst limitation  
 * and each buffer will be different. */  
txconf.direction = DMA_MEM_TO_DEV;  
txconf.dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;  
txconf.dst_maxburst = TX_TRIGGER;  
txconf.dst_addr = fifo_dma_addr;  
ret = dmaengine_slave_config(dma->txchan, &txconf);
```



# dmaengine: Slave API: Per-transfer configuration (1/2)

1. Create a descriptor with all the required configuration for the next transfer with:

```
struct dma_async_tx_descriptor *
dmaengine_prep_slave_single(struct dma_chan *chan, dma_addr_t buf,
                           size_t len, enum dma_transfer_direction dir,
                           unsigned long flags);

struct dma_async_tx_descriptor *
dmaengine_prep_slave_sg(struct dma_chan *chan, struct scatterlist *sgl,
                        unsigned int sg_len, enum dma_transfer_direction dir,
                        unsigned long flags);

struct dma_async_tx_descriptor *
dmaengine_prep_dma_cyclic(struct dma_chan *chan, dma_addr_t buf, size_t buf_len,
                          size_t period_len, enum dma_data_direction dir);
```

- ▶ Common flags are:
  - DMA\_PREP\_INTERRUPT: Generates an interrupt once done
  - DMA\_CTRL\_ACK: No need for a manual ack of the transaction
- ▶ The descriptor returned can be used to fill-in a callback:

```
desc->callback = foo_dma_complete;
desc->callback_param = foo_dev;
```



## dmaengine: Slave API: Per-transfer configuration (2/2)

### 2. Queue the next operation:

```
dma_cookie_t cookie;  
  
cookie = dmaengine_submit(desc);  
ret = dma_submit_error(cookie);  
if (ret)  
    ...
```

### 3. Trigger the queued transfers

```
dma_async_issue_pending(chan);
```

### 3bis. In case anything went wrong or the device should stop being used, it is possible to terminate all ongoing transactions with:

```
dmaengine_terminate_sync(chan);
```



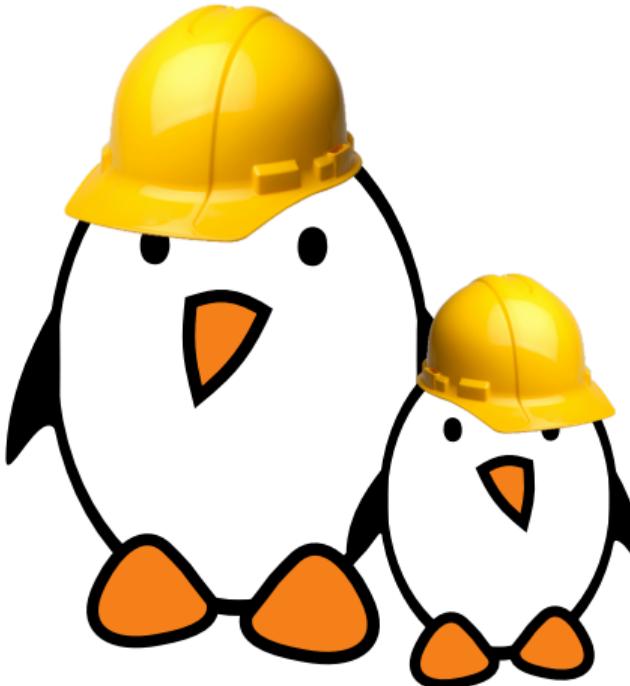
## Examples

---

- ▶ Commented network driver, whith both streaming and coherent mappings:  
<https://bootlin.com/pub/drivers/r6040-network-driver-with-comments.c>
- ▶ Example of usage of the slave API: look at the code for `stm32_i2c_prep_dma_xfer()`.



## Practical lab - DMA



- ▶ Setup streaming mappings with the dma-mapping API
- ▶ Configure a DMA controller with the dmaengine API
- ▶ Configure the hardware to trigger DMA transfers
- ▶ Wait for DMA completion



# Kernel debugging

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# Debugging using messages (1/3)

---

Three APIs are available

- ▶ The old `printf()`, no longer recommended for new debugging messages
- ▶ The `pr_*`() family of functions: `pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_err()`, `pr_warning()`, `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)

- ▶ 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 print if called too much based on `/proc/sys/kernel/printk_ratelimit{_burst}` values



## Debugging using messages (3/3)

- ▶ 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 uses %pK
- ▶ See [core-api/printk-formats](#) for an exhaustive list of supported format specifiers



## pr\_debug() and dev\_dbg()

---

- ▶ 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
  - 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](#)



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](#) (section *The debugfs filesystem*)



# DebugFS API

- ▶ 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.



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](#)



- ▶ [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 [dev-tools/gdb-kernel-debugging](#) for more information



## Using kgdb (1/2)

---

- ▶ Details available in the kernel documentation: [dev-tools/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)

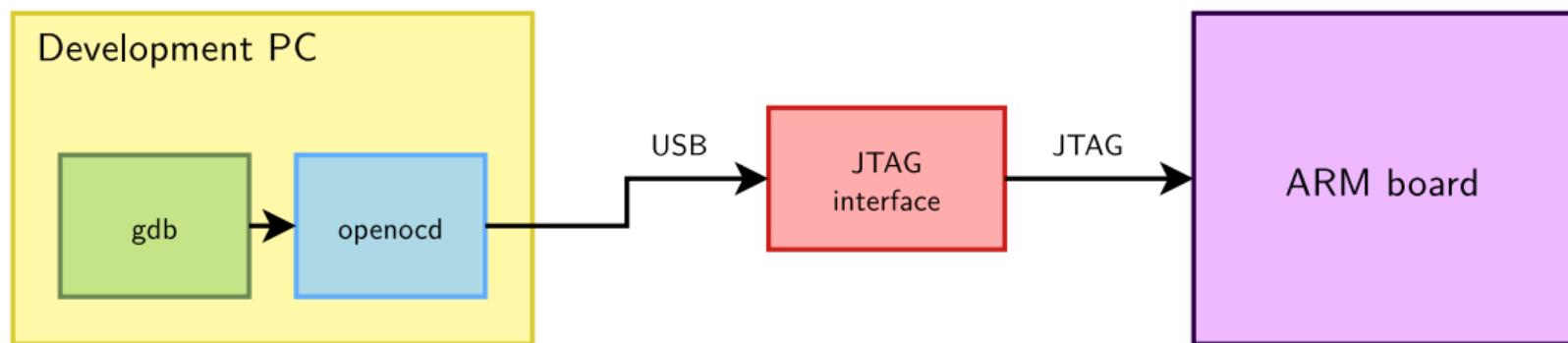
- ▶ Then also pass kgdbwait to the kernel: it makes kgdb wait for a debugger connection.
- ▶ Boot your kernel, and when the console is initialized, interrupt the kernel with a break character and then g in the serial console (see our *Magic SysRq* explanations).
- ▶ On your workstation, start gdb as 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.



# Debugging with a JTAG interface

Two types of JTAG dongles

- ▶ The ones offering a gdb compatible interface, over a serial port or an Ethernet connection. gdb can directly connect to them.
- ▶ The ones not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger): <http://openocd.sourceforge.net/>
  - OpenOCD is the bridge between the gdb debugging language and the JTAG interface of the target CPU.
  - See the very complete documentation: <http://openocd.org/documentation/>
  - For each board, you'll need an OpenOCD configuration file (ask your supplier)





- ▶ 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 `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.
- ▶ If your device is not probed, try enabling `CONFIG_DRIVER_DEBUG`
  - Extremely verbose!
  - Will enable all the debug logs in the device-driver core section



## Getting help and reporting bugs

---

- ▶ If you are using a custom kernel from a hardware vendor, contact that company. The community will have less interest supporting a custom kernel.
- ▶ Otherwise, or if this doesn't work, try to reproduce the issue on the latest version of the kernel.
- ▶ Make sure you investigate the issue as much as you can: see [admin-guide/bug-bisect](#)
- ▶ Check for previous bugs reports. Use web search engines, accessing public mailing list archives.
- ▶ If you're the first to face the issue, it's very useful for others to report it, even if you cannot investigate it further.
- ▶ If the subsystem you report a bug on has a mailing list, use it. Otherwise, contact the official maintainer (see the [MAINTAINERS](#) file). Always give as many useful details as possible.



# Practical lab - Kernel debugging



- ▶ 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.



# Power Management

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# PM building blocks

---

- ▶ Several power management *building blocks*
  - Clock framework
  - Suspend and resume
  - CPUidle
  - Runtime power management
  - Power domains
  - Frequency and voltage scaling
- ▶ Independent *building blocks* that can be improved gradually during development



# Clock framework (1)

---

- ▶ Generic framework to manage clocks used by devices in the system
- ▶ Allows to reference count clock users and to shutdown the unused clocks to save power
- ▶ Simple API described in [include/linux/clk.h](#).
  - `clk_get()` to lookup and obtain a reference to a clock producer
  - `clk_enable()` to inform the system when the clock source should be running
  - `clk_disable()` to inform the system when the clock source is no longer required.
  - `clk_put()` to free the clock source
  - `clk_get_rate()` to obtain the current clock rate (in Hz) for a clock source
  - `clk_set_rate()` to set the current clock rate (in Hz) of a clock source



## Clock framework (2)

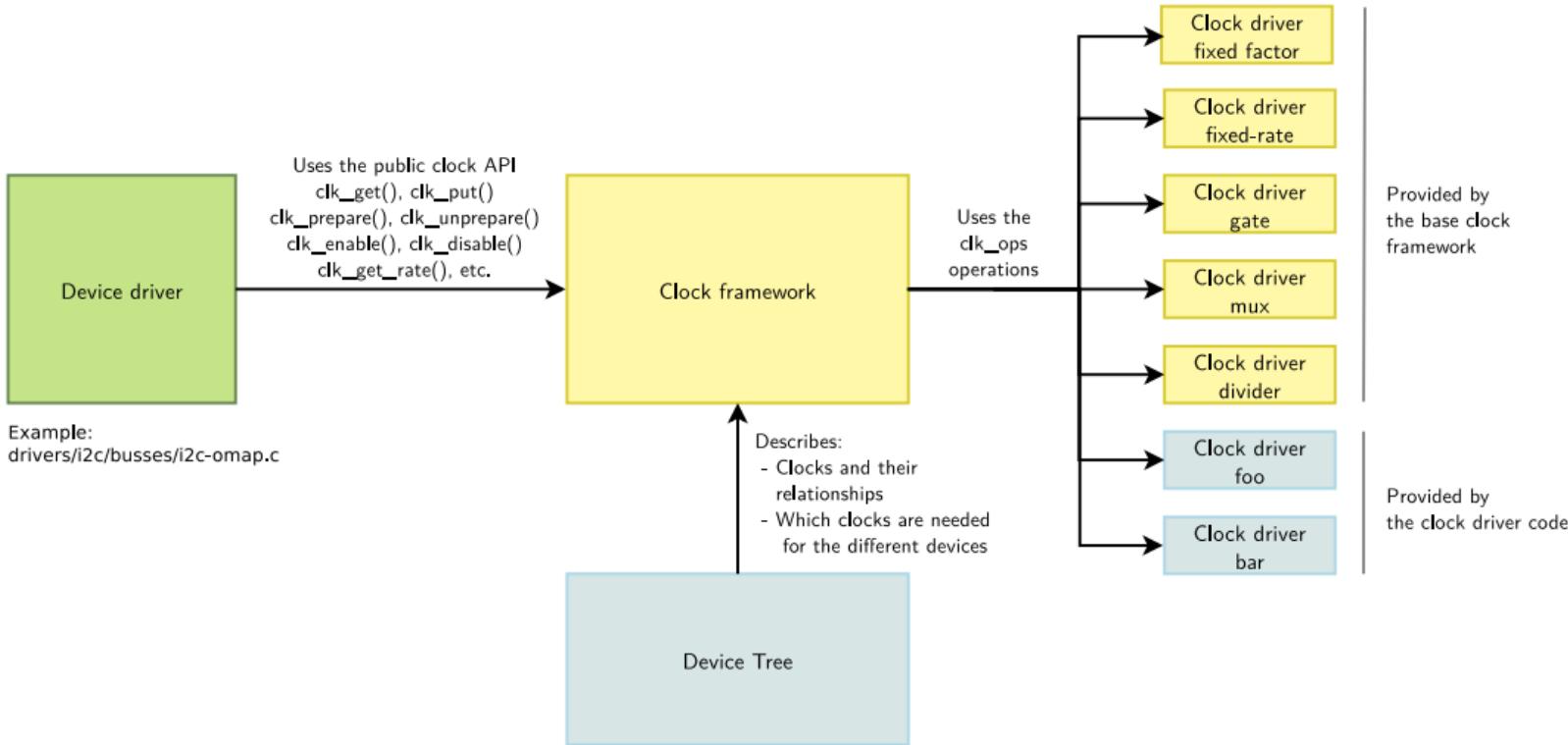
---

### The common clock framework

- ▶ Allows to declare the available clocks and their association to devices in the Device Tree
- ▶ Provides a *debugfs* representation of the clock tree
- ▶ Is implemented in [drivers/clk/](#)



# Diagram overview of the common clock framework





## Clock framework (3)

The interface of the CCF divided into two halves:

- ▶ Common Clock Framework core
  - Common definition of `struct clk`
  - Common implementation of the `clk.h` API (defined in `drivers/clk/clk.c`)
  - `struct clk_ops`: operations invoked by the `clk` API implementation
  - Not supposed to be modified when adding a new driver
- ▶ Hardware-specific
  - Callbacks registered with `struct clk_ops` and the corresponding hardware-specific structures
  - Has to be written for each new hardware clock
  - Example: `drivers/clk/mvebu/clk-cpu.c`



## Clock framework (4)

Hardware clock operations: device tree

- ▶ The **device tree** is the **mandatory way** to declare a clock and to get its resources, as for any other driver using DT we have to:
  - Parse the device tree to **setup** the clock: the resources but also the properties are retrieved.
  - Declare the **compatible** clocks and associate each to an **initialization** function using `CLK_OF_DECLARE()`
  - Example: `arch/arm/boot/dts/armada-xp.dtsi` and `drivers/clk/mvebu/armada-xp.c`

See our presentation about the Clock Framework for more details:

<https://bootlin.com/pub/conferences/2013/elce/common-clock-framework-how-to-use-it/>



# Suspend and resume (to / from RAM)

- ▶ Infrastructure in the kernel to support suspend and resume
- ▶ System on Chip hooks
  - Define operations (at least `valid()` and `enter()`) `struct platform_suspend_ops` structure. See the documentation for this structure for details about possible operations and the way they are used.
  - Registered using the `suspend_set_ops()` function
  - See `arch/arm/mach-at91/pm.c`
- ▶ Device driver hooks
  - pm operations (`suspend()` and `resume()` hooks) in the `struct device_driver` as a `struct dev_pm_ops` structure in (`struct platform_driver`, `struct usb_driver`, etc.)
  - See `drivers/net/ethernet/cadence/macb_main.c`
- ▶ *Hibernate to disk* is based on suspend to RAM, copying the RAM contents (after a simulated suspend) to a swap partition.



## Triggering suspend / hibernate

---

- ▶ `struct suspend_ops` functions are called by the `enter_state()` function.  
`enter_state()` also takes care of executing the suspend and resume functions for your devices.
- ▶ Read `kernel/power/suspend.c`
- ▶ The execution of this function can be triggered from user space:
  - `echo mem > /sys/power/state` (suspend to RAM)
  - `echo disk > /sys/power/state` (hibernate to disk)
- ▶ Systemd can also manage suspend and hibernate for you, and offers customizations
  - `systemctl suspend` or `systemctl hibernate`.
  - See <https://www.man7.org/linux/man-pages/man8/systemd-suspend.service.8.html>



## Saving power in the idle loop

---

- ▶ The idle loop is what you run when there's nothing left to run in the system.
- ▶ `arch_cpu_idle()` implemented in all architectures in  
`arch/<arch>/kernel/process.c`
- ▶ Example: `arch/arm/kernel/process.c`
- ▶ The CPU can run power saving HLT instructions, enter NAP mode, and even disable the timers (tickless systems).
- ▶ See also [https://en.wikipedia.org/wiki/Idle\\_loop](https://en.wikipedia.org/wiki/Idle_loop)



## Adding support for multiple idle levels

- ▶ Modern CPUs have several sleep states offering different power savings with associated wake up latencies
- ▶ The *dynamic tick* feature allows to remove the periodic timer tick to save power, and to know when the next event is scheduled, for smarter sleeps.
- ▶ CPUidle infrastructure to change sleep states
  - Platform-specific driver defining sleep states and transition operations
  - Platform-independent governors
  - Available in particular for x86/ACPI and most ARM SoCs
  - See [admin-guide/pm/cpuidle](#) in kernel documentation.



<https://01.org/powertop/>

- ▶ With dynamic ticks, allows to fix parts of kernel code and applications that wake up the system too often.
- ▶ PowerTOP allows to track the worst offenders
- ▶ Now available on ARM cpus implementing CPUidle
- ▶ Also gives you useful hints for reducing power.
- ▶ Try it on your x86 laptop:  
`sudo powertop`



## Runtime power management

- ▶ Managing per-device idle, each device being managed by its device driver independently from others.
- ▶ According to the kernel configuration interface: *Enable functionality allowing I/O devices to be put into energy-saving (low power) states at run time (or autosuspended) after a specified period of inactivity and woken up in response to a hardware-generated wake-up event or a driver's request.*
- ▶ New hooks must be added to the drivers: `runtime_suspend()`, `runtime_resume()`, `runtime_idle()` in the `struct dev_pm_ops` structure in `struct device_driver`.
- ▶ API and details on `power/runtime_pm`
- ▶ See `drivers/net/ethernet/cadence/macb_main.c` again.



## Generic PM Domains (genpd)

- ▶ Generic infrastructure to implement power domains based on Device Tree descriptions, allowing to group devices by the physical power domain they belong to. This sits at the same level as bus type for calling PM hooks.
- ▶ All the devices in the same PD get the same state at the same time.
- ▶ Specifications and examples available at  
[Documentation/devicetree/bindings/power/power\\_domain.txt](Documentation/devicetree/bindings/power/power_domain.txt)
- ▶ Driver example: [drivers/soc/rockchip/pm\\_domains.c](drivers/soc/rockchip/pm_domains.c)  
(`rockchip_pd_power_on()`, `rockchip_pd_power_off()`,  
`rockchip_pm_add_one_domain()`...)
- ▶ DT example: look for `rockchip,px30-power-controller`  
(<arch/arm64/boot/dts/rockchip/px30.dtsi>) and find PD definitions and corresponding devices.
- ▶ See Kevin Hilman's talk at Kernel Recipes 2017:  
<https://youtu.be/SctfvoskABM>



# Frequency and voltage scaling (1)

Frequency and voltage scaling possible through the cpufreq kernel infrastructure.

- ▶ Generic infrastructure: [drivers/cpufreq/cpufreq.c](#) and [include/linux/cpufreq.h](#)
- ▶ Generic governors, responsible for deciding frequency and voltage transitions
  - performance: maximum frequency
  - powersave: minimum frequency
  - ondemand: measures CPU consumption to adjust frequency
  - conservative: often better than ondemand. Only increases frequency gradually when the CPU gets loaded.
  - userspace: leaves the decision to a user space daemon.
- ▶ This infrastructure can be controlled from  
`/sys/devices/system/cpu/cpu<n>/cpufreq/`



## Frequency and voltage scaling (2)

- ▶ CPU frequency drivers are in [drivers/cpufreq/](#). Example:  
[drivers/cpufreq/omap-cpufreq.c](#)
- ▶ Must implement the operations of the cpufreq\_driver structure and register them using [cpufreq\\_register\\_driver\(\)](#)
  - `init()` for initialization
  - `exit()` for cleanup
  - `verify()` to verify the user-chosen policy
  - `setpolicy()` or `target()` to actually perform the frequency change
- ▶ See documentation in [cpu-freq/](#) for useful explanations



- ▶ Modern embedded platforms have hardware responsible for voltage and current regulation
- ▶ The regulator framework allows to take advantage of this hardware to save power when parts of the system are unused
  - A consumer interface for device drivers (i.e. users)
  - Regulator driver interface for regulator drivers
  - Machine interface for board configuration
  - sysfs interface for user space
- ▶ See [power/regulator/](#) in kernel documentation.



## BSP work for a new board

---

In case you just need to create a BSP for your board, and your CPU already has full PM support, you should just need to:

- ▶ Create clock definitions and bind your devices to them.
- ▶ Implement PM handlers (suspend, resume) in the drivers for your board specific devices.
- ▶ Implement runtime PM handlers in your drivers.
- ▶ Implement board specific power management if needed (mainly battery management)
- ▶ Implement regulator framework hooks for your board if needed.
- ▶ Attach on-board devices to PM domains if needed
- ▶ All other parts of the PM infrastructure should be already there: suspend / resume, cpuidle, cpu frequency and voltage scaling, PM domains.



## Useful resources

---

- ▶ *power/* in kernel documentation.
  - Will give you many useful details.
- ▶ Introduction to kernel power management, Kevin Hilman (Kernel Recipes 2015)
  - <https://www.youtube.com/watch?v=juJJZ0RgVwI>

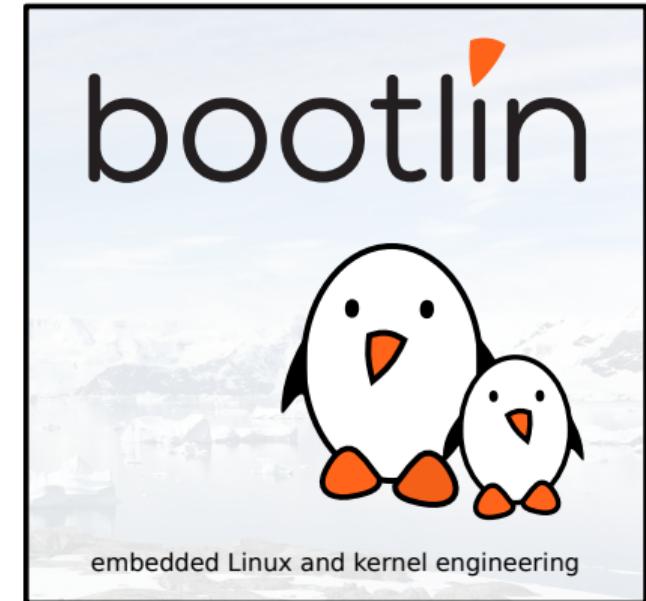


# Kernel Resources

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## 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 (\$7 / month)
- ▶ Articles available for non subscribers after 1 week.





# Useful Online Resources

---

- ▶ Kernel documentation
  - <https://kernel.org/doc/>
- ▶ Linux kernel mailing list FAQ
  - <http://vger.kernel.org/lkml/>
  - Complete Linux kernel FAQ
  - Read this before asking a question to the mailing list
- ▶ Linux kernel mailing lists
  - <http://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



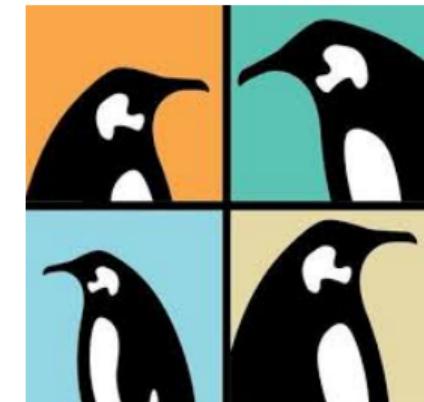
# International Conferences (1)

## ► 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](https://elinux.org/ELC_Presentations)

## ► Linux Plumbers

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





## International Conferences (2)

- ▶ 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.
- ▶ linux.conf.au: <https://linux.org.au/conf/>
  - In Australia / New Zealand
  - Features a few presentations by key kernel hackers.
- ▶ Currently, most conferences are available on-line. They are much more affordable and often free.





## After the course

Continue to learn:

- ▶ Run your labs again on your own hardware. The Nunchuk 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/](https://git.kernel.org/?p=drivers/staging/)
- ▶ Investigating and do the triage of issues reported by Coverity Scan:  
<https://scan.coverity.com/projects/linux>



# Contribute your changes

---

## 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>
- ▶ How to Participate in the Linux Community (by Jonathan Corbet). A guide to the kernel development process <https://j.mp/tX2Ld6>

## General advices:

- ▶ Play with `git commit --fixup=` and `git rebase -i`
- ▶ Send new versions of your changes formatted with `git format-patch`
  - Include a cover letter if you have more than one patch (`--cover-letter`)
  - Don't forget to add a changelog and use `-vX` to generate your next iteration
- ▶ Don't be discouraged by the feedback you get



## Last slides

# Last slides

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Thank you!  
And may the Source be with you



# Rights to copy

---

© Copyright 2004-2023, Bootlin

**License: Creative Commons Attribution - Share Alike 3.0**

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

You are free:

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

Under the following conditions:

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

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

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

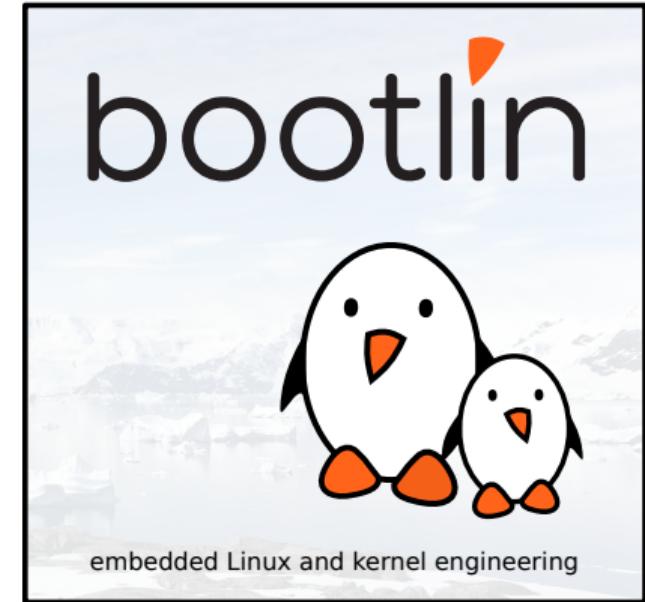


# Backup slides

© Copyright 2004-2023, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## mmap



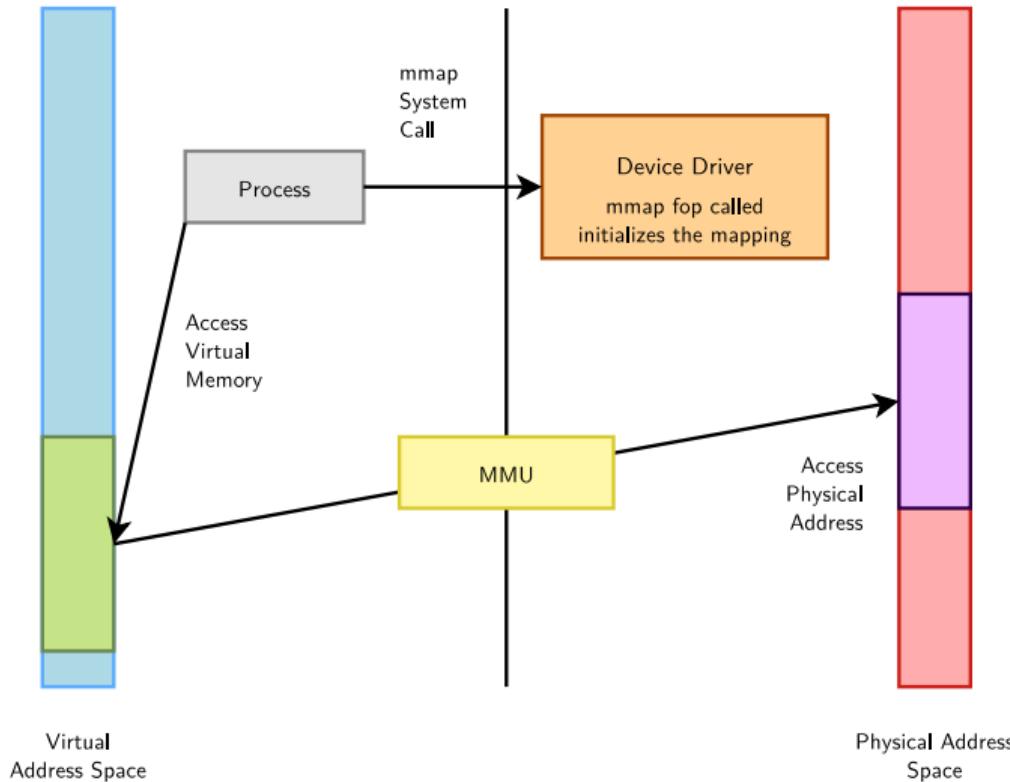
- ▶ Possibility to have parts of the virtual address space of a program mapped to the contents of a file
- ▶ Particularly useful when the file is a device file
- ▶ Allows to access device I/O memory and ports without having to go through (expensive) read, write or ioctl calls
- ▶ One can access to current mapped files by two means:
  - `/proc/<pid>/maps`
  - `pmap <pid>`



start-end	perm	offset	major:minor	inode	mapped file	name
...						
7f4516d04000-7f4516d06000	rw-s	1152a2000	00:05	8406	/dev/dri/card0	
7f4516d07000-7f4516d0b000	rw-s	120f9e000	00:05	8406	/dev/dri/card0	
...						
7f4518728000-7f451874f000	r-xp	00000000	08:01	268909	/lib/x86_64-linux-gnu/libexpat.so.1.5.2	
7f451874f000-7f451894f000	---p	00027000	08:01	268909	/lib/x86_64-linux-gnu/libexpat.so.1.5.2	
7f451894f000-7f4518951000	r--p	00027000	08:01	268909	/lib/x86_64-linux-gnu/libexpat.so.1.5.2	
7f4518951000-7f4518952000	rw-p	00029000	08:01	268909	/lib/x86_64-linux-gnu/libexpat.so.1.5.2	
...						
7f451da4f000-7f451dc3f000	r-xp	00000000	08:01	1549	/usr/bin/Xorg	
7f451de3e000-7f451de41000	r--p	001ef000	08:01	1549	/usr/bin/Xorg	
7f451de41000-7f451de4c000	rw-p	001f2000	08:01	1549	/usr/bin/Xorg	
...						



# mmap overview



Virtual  
Address Space

Physical Address  
Space



# How to Implement mmap - User space

- ▶ Open the device file
- ▶ Call the mmap system call (see man mmap for details):

```
void * mmap(  
    void *start, /* Often 0, preferred starting address */  
    size_t length, /* Length of the mapped area */  
    int prot, /* Permissions: read, write, execute */  
    int flags, /* Options: shared mapping, private copy... */  
    int fd, /* Open file descriptor */  
    off_t offset /* Offset in the file */  
);
```

- ▶ You get a virtual address you can write to or read from.



# How to Implement mmap - Kernel space

- ▶ Character driver: implement an mmap file operation and add it to the driver file operations:

```
int (*mmap) (
    struct file *,          /* Open file structure */
    struct vm_area_struct * /* Kernel VMA structure */
);
```

- ▶ Initialize the mapping.
  - Can be done in most cases with the `remap_pfn_range()` function, which takes care of most of the job.



## remap\_pfn\_range()

- ▶ *pfn*: page frame number
- ▶ The most significant bits of the page address (without the bits corresponding to the page size).

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

int remap_pfn_range(
    struct vm_area_struct *, /* VMA struct */
    unsigned long virt_addr, /* Starting user
                             * virtual address */
    unsigned long pfn,       /* pfn of the starting
                             * physical address */
    unsigned long size,      /* Mapping size */
    pgprot_t prot           /* Page permissions */
);
```



# Simple mmap implementation

```
static int acme_mmap
    (struct file * file, struct vm_area_struct *vma)
{
    size = vma->vm_end - vma->vm_start;

    if (size > ACME_SIZE)
        return -EINVAL;

    if (remap_pfn_range(vma,
                        vma->vm_start,
                        ACME_PHYS >> PAGE_SHIFT,
                        size,
                        vma->vm_page_prot))
        return -EAGAIN;

    return 0;
}
```



- ▶ <https://bootlin.com/pub/mirror/devmem2.c>, by Jan-Derk Bakker
- ▶ Very useful tool to directly peek (read) or poke (write) I/O addresses mapped in physical address space from a shell command line!
  - Very useful for early interaction experiments with a device, without having to code and compile a driver.
  - Uses mmap to /dev/mem.
  - Examples (b: byte, h: half, w: word)
    - devmem2 0x000c0004 h (reading)
    - devmem2 0x000c0008 w 0xffffffff (writing)
  - devmem is now available in BusyBox, making it even easier to use.



- ▶ The device driver is loaded. It defines an `mmap` file operation.
- ▶ A user space process calls the `mmap` system call.
- ▶ The `mmap` file operation is called.
- ▶ It initializes the mapping using the device physical address.
- ▶ The process gets a starting address to read from and write to (depending on permissions).
- ▶ The MMU automatically takes care of converting the process virtual addresses into physical ones.
- ▶ Direct access to the hardware without any expensive read or write system calls



## Useful general-purpose kernel APIs



- ▶ In `include/linux/string.h`
  - Memory-related: `memset()`, `memcpy()`, `memmove()`, `memscan()`, `memcmp()`, `memchr()`
  - String-related: `strcpy()`, `strcat()`, `strcmp()`, `strchr()`, `strrchr()`, `strlen()` and variants
  - Allocate and copy a string: `kstrdup()`, `kstrndup()`
  - Allocate and copy a memory area: `kmempdup()`
- ▶ In `include/linux/kernel.h`
  - String to int conversion: `simple strtoul()`, `simple strtol()`,  
`simple strtoull()`, `simple strtoll()`
  - Other string functions: `sprintf()`, `sscanf()`



- ▶ Convenient linked-list facility in `include/linux/list.h`
  - Used in thousands of places in the kernel
- ▶ Add a `struct list_head` member to the structure whose instances will be part of the linked list. It is usually named `node` when each instance needs to only be part of a single list.
- ▶ Define the list with the `LIST_HEAD()` macro for a global list, or define a `struct list_head` element and initialize it with `INIT_LIST_HEAD()` for lists embedded in a structure.
- ▶ Then use the `list_*`() API to manipulate the list
  - Add elements: `list_add()`, `list_add_tail()`
  - Remove, move or replace elements: `list_del()`, `list_move()`, `list_move_tail()`, `list_replace()`
  - Test the list: `list_empty()`
  - Iterate over the list: `list_for_each_*`() family of macros



## Linked lists examples 1/2

From include/soc/at91/atmel\_tcb.h

```
/*
 * Definition of a list element, with a
 * struct list_head member
 */
struct atmel_tc
{
    /* some members */
    struct list_head node;
};
```



## Linked lists examples 2/2

From drivers/misc/atmel\_tclib.c

```
/* Define the global list */
static LIST_HEAD(tc_list);

static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    /* Add an element to the list */
    list_add_tail(&tc->node, &tc_list);
}

struct atmel_tc *atmel_tc_alloc(unsigned block, const char *name)
{
    struct atmel_tc *tc;
    /* Iterate over the list elements */
    list_for_each_entry(tc, &tc_list, node) {
        /* Do something with tc */
    }
    [...]
}
```