Linux Kernel and Driver Development Training

Latest update: June 29, 2021.

Document updates and sources:
https://bootlin.com/doc/training/linux-kernel

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

embedded Linux and kernel engineering

Rights to copy

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

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  `https://kernel.org/`
- Kernel documentation links:
  `dev-tools/kasan`
- Links to kernel source files and directories:
  `drivers/input/`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

Company at a glance

- Engineering company created in 2004, named "Free Electrons" until Feb. 2018.
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
- Head count: 13 - Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, build systems and low level Free and Open Source Software for embedded and real-time systems.
- Feb. 2021: Bootlin is the 20th all-time Linux kernel contributor
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

bootlin

No.1	Unknown	140019(15.26%)
No.2	Intel	94806(10.33%)
No.3	Red Hat	78140(8.52%)
No.4	Hobbyists	73603(8.02%)
No.5	Novell	39218(4.27%)
No.6	IBM	35085(3.82%)
No.7	Linaro	28288(3.08%)
No.8	AMD	22426(2.44%)
No.9	Google	20489(2.23%)
No.10	Renesas Electronics	18443(2.01%)
No.11	Oracle	17729(1.93%)
No.12	Samsung	17514(1.91%)
No.13	Texas Instruments	16372(1.78%)
No.14	HuaWei	13377(1.46%)
No.15	Mellanox Technologies	11477(1.25%)
No.16	ARM	8919(0.97%)
No.17	Academics	8560(0.93%)
No.18	Consultants	8073(0.88%)
No.19	Broadcom	8011(0.87%)
No.20	Bootlin	7611(0.83%)
No.21	NXP	7549(0.82%)
No.22	Linutronix	7430(0.81%)
No.23	NVIDIA	6951(0.76%)
No.24	Canonical	6855(0.75%)
No.25	Linux Foundation	6369(0.69%)
No.26	Code Aurora Forum	6280(0.68%)
No.27	Pengutronix	6201(0.68%)
No.28	VISION Engraving and Routing Systems	6045(0.66%)
No.29	Analog Devices	5944(0.65%)
No.30	Fujitsu	5120(0.56%)
No.31	QUALCOMM	4903(0.53%)
No.32	Freescale	4694(0.51%)
No.33	Wolfson Microelectronics	4180(0.46%)
No.34	Marvell	4178(0.46%)
No.35	Nokia	4097(0.45%)
No.36	Cisco	4071(0.44%)
No.37	Parallels	3841(0.42%)
No.38	Imagination Technologies	3774(0.41%)
No.39	Facebook	3484(0.38%)
No.40	QLogic	3394(0.37%)
No.41	ST Microelectronics	3188(0.35%)
No.42	Astaro	2981(0.32%)
No.43	NetApp	2860(0.31%)

Top Linux contributors since `git` (2005)

Bootlin on-line resources

- All our training materials and technical presentations: https://bootlin.com/docs/
- Technical blog: https://bootlin.com/
- Quick news (Mastodon): https://fosstodon.org/@bootlin
- Quick news (Twitter): https://twitter.com/bootlincom
- Quick news (LinkedIn): https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line: https://elixir.bootlin.com

Mastodon is a free and decentralized social network created in the best interests of its users.

Image credits: Jin Nguyen - https://frama.link/bQwcWHTP

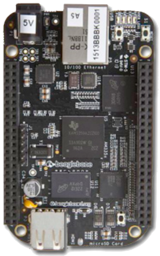
Generic course information

Corrections, suggestions, contributions and translations are welcome!

Supported hardware

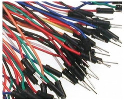
BeagleBone Black or BeagleBone Black Wireless, from BeagleBoard.org

- 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`.



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):
  Olimex: https://frama.link/zWJDToXP
- Nintendo Nunchuk with UEXT connector:
  Olimex: https://j.mp/1dTYLfs
- Breadboard jumper wires - Male ends (to connect the Nunchuk):
  Olimex: https://bit.ly/2pSiIPs
- 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.



Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

Practical lab guidelines

During practical labs...

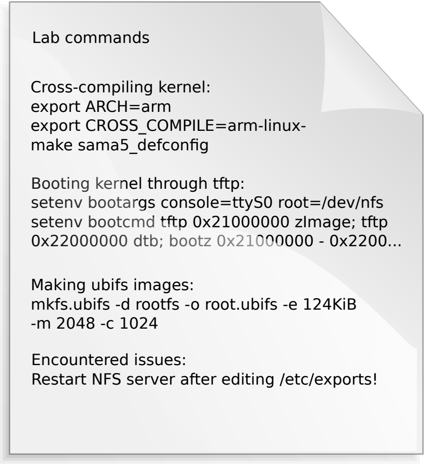
- We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- So, if you are more than 8 participants, please form up to 8 working groups.
- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't hesitate to copy and paste commands from the PDF slides and labs.

Advise: write down your commands!

During practical labs, write down all your commands in a text file.

- You can save a lot of time re-using commands in later labs.
- This helps to replay your work if you make significant mistakes.
- You build a reference to remember commands in the long run.
- That's particular useful to keep kernel command line settings that you used earlier.
- Also useful to get help from the instructor, showing the commands that you run.

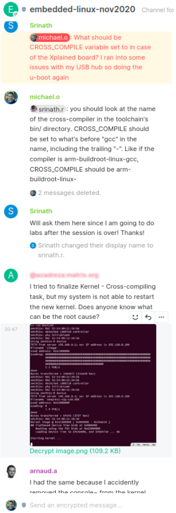
```
gedit ~/lab-history.txt
```



Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

- Use the dedicated Matrix channel for this session
- If you complete your labs before other people, don't hesitate to help them and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

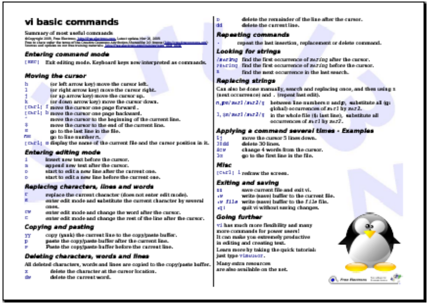


Command memento sheet

- This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- It saves us 1 day of UNIX / Linux command line training.
- Our best tip: in the command line shell, always hit the Tab key to complete command names and file paths. This avoids 95% of typing mistakes.
- Get an electronic copy on https://bootlin.com/doc/legacy/command-line/command_memento.pdf

vi basic commands

- The `vi` editor is very useful to make quick changes to files in an embedded target.
- Though not very user friendly at first, `vi` is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- Get an electronic copy on https://bootlin.com/doc/legacy/command-line/vi_memento.pdf
- You can also take the quick tutorial by running `vimtutor`. This is a worthy investment!



vi basic commands

Summary of most useful commands

Entering command mode

[ESC]	Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

h	(or left arrow key) move the cursor left.
l	(or right arrow key) move the cursor right.
j	(or up arrow key) move the cursor up.
k	(or down arrow key) move the cursor down.
[Ctrl] f	move the cursor one page forward.
[Ctrl] b	move the cursor one page backward.
^	move the cursor to the beginning of the current line.
$	move the cursor to the end of the current line.
G	go to the last line in the file.
nG	go to line number n.
[Ctrl] G	display the name of the current file and the cursor position in it.

Entering editing mode

i	insert new text before the cursor.
a	append new text after the cursor.
o	start to edit a new line after the current one.
O	start to edit a new line before the current one.

Replacing characters, lines and words

r	replace the current character (does not enter edit mode).
s	enter edit mode and substitute the current character by several ones.
cw	enter edit mode and change the word after the cursor.
C	enter edit mode and change the rest of the line after the cursor.

Copying and pasting

yy	copy (yank) the current line to the copy/paste buffer.
p	paste the copy/paste buffer after the current line.
P	Paste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.

x	delete the character at the cursor location.
dw	delete the current word.
D	delete the remainder of the line after the cursor.
dd	delete the current line.

Repeating commands

.	repeat the last insertion, replacement or delete command.

Looking for strings

/string	find the first occurrence of string after the cursor.
?string	find the first occurrence of string before the cursor.
n	find the next occurrence in the last search.

Replacing strings

Can also be done manually, search and replacing once, and then using n (next occurrence) and . (repeat last edit).

n,p s/str1/str2/g	between line numbers n and p, substitute all (g: global) occurrences of str1 by str2.
1,$ s/str1/str2/g	in the whole file ($: last line), substitute all occurrences of str1 by str2.

Applying a command several times - Examples

5j	move the cursor 5 lines down.
30dd	delete 30 lines.
4cw	change 4 words from the cursor.
1G	go to the first line in the file.

Misc

[Ctrl] l	redraw the screen.

Exiting and saving

ZZ	save current file and exit vi.
:w	write (save) buffer to the current file.
:w file	write (save) buffer to the file file.
:q!	quit vi without saving changes.

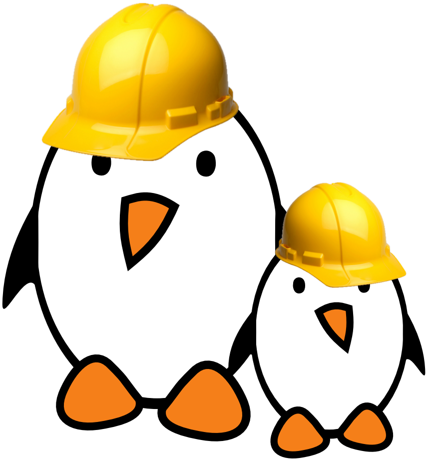
Going further

vi has much more flexibility and many more commands for power users!
It can make you extremely productive in editing and creating text.

Learn more by taking the quick tutorial: just type vimtutor.

Many extra resources are also available on the net.

Practical lab - Training Setup



Prepare your lab environment

- Download and extract the lab archive

Linux Kernel Introduction

embedded Linux and kernel engineering

Linux features

History

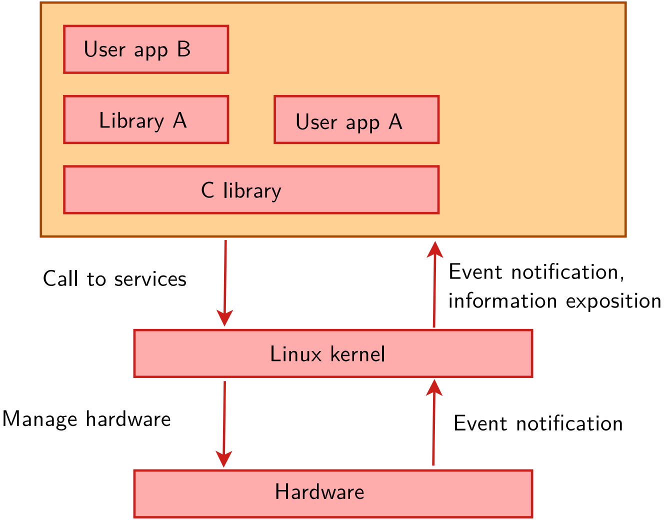
- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

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

Linux kernel key features

- Portability and hardware support. Runs on most architectures (see `arch/` in the source code).
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security. It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

Linux kernel in the system



Linux kernel main roles

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

System calls

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

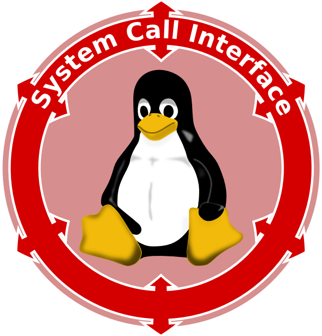


Image credits (Wikipedia):
https://bit.ly/2U2rdGB

Pseudo filesystems

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

Linux Kernel

Memory management

Device drivers + driver frameworks

Scheduler Task management

Low level architecture specific code

Device Trees (HW description), on some architectures

Filesystem layer and drivers

Network stack

Linux license

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

Supported hardware architectures

See the arch/ directory in the kernel sources

- Minimum: 32 bit processors, with or without MMU, supported by gcc
- 32 bit architectures (arch/ subdirectories)
  Examples: arm, arc, m68k, microblaze (soft core on FPGA)...
- 64 bit architectures:
  Examples: alpha, arm64, ia64...
- 32/64 bit architectures
  Examples: mips, powerpc, riscv, sh, sparc, x86...
- Note that unmaintained architectures can also be removed when they have compiling issues and nobody fixes them.
- Find details in kernel sources: arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/

Embedded Linux Kernel Usage

Corrections, suggestions, contributions and translations are welcome!

bootlin

embedded Linux and kernel engineering

Linux kernel sources

Location of kernel sources

- The official (*mainline*) versions of the Linux kernel, as released by Linus Torvalds, are available at https://kernel.org
  - These versions follow the development model of the kernel
  - However, they may not contain the latest development from a specific area yet. Some features in development might not be ready for mainline inclusion yet.
- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Useful only when mainline hasn't caught up yet. Many vendors invest in the mainline kernel at the same time.
- Many kernel sub-communities maintain their own kernel, with usually newer but fewer stable features
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - No official releases, only meant for sharing work and contributing to the mainline version.

Getting Linux sources

- The kernel sources are available from https://kernel.org/pub/linux/kernel as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- However, more and more people use the `git` version control system. Absolutely needed for kernel development!
  - Fetch the entire kernel sources and history
    `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
  - Create a branch that starts at a specific stable version
    `git checkout -b <name-of-branch> v5.6`
  - Web interface available at
    https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/
  - Read more about Git at https://git-scm.com/

Linux kernel size (1)

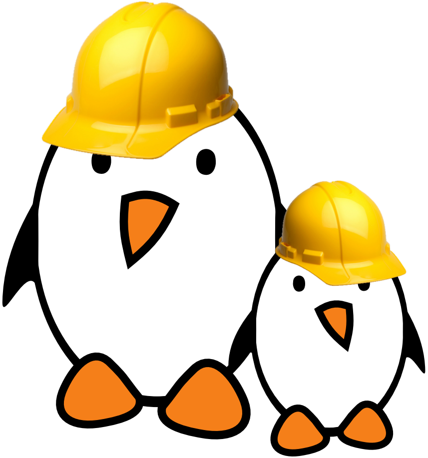
- Linux 5.10.11 sources:
  - 70,639 files (`git ls-files | wc -l`)
  - 29,746,102 lines (`git ls-files | xargs cat | wc -l`)
  - 962,810,769 bytes (`git ls-files | xargs cat | wc -c`)
- A minimum uncompressed Linux kernel just sizes 1-2 MB
- Why are these sources so big?
  Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...
- The Linux core (scheduler, memory management...) is pretty small!

Linux kernel size (2)

As of kernel version 5.7 (in percentage of total number of lines).

- drivers/: 60.1%
- arch/: 12.9%
- fs/: 4.7%
- sound/: 4.2%
- net/: 4.0%
- include/: 3.6%
- tools/: 3.2%
- Documentation/: 3.2%
- kernel/: 1.3%
- lib/: 0.6%
- mm/: 0.5%
- scripts/: 0.4%
- crypto/: 0.4%
- security/: 0.3%
- block/: 0.2%
- samples/: 0.1%
- virt/: 0.1%
- ...

Practical lab - Downloading kernel source code



- Clone the mainline Linux source tree with git

Kernel Source Code

Linux Code and Device Drivers

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`
- Ongoing work to compile the kernel with the LLVM C compiler (Clang) too: `https://clangbuiltlinux.github.io/`
- There are also plans to create new code in Rust too: `https://lwn.net/Articles/829858/`

No C library

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

Portability

- The Linux kernel code is designed to be portable
- All code outside `arch/` should be portable
- To this aim, the kernel provides macros and functions to abstract the architecture specific details
  - Endianness
    - `cpu_to_be32()`
    - `cpu_to_le32()`
    - `be32_to_cpu()`
    - `le32_to_cpu()`
  - 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.

No stable Linux internal API

- The internal kernel API to implement kernel code can undergo changes between two releases.
- 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 in kernel sources for reasons why.
- Of course, the kernel to user space API does not change (system calls, /proc, /sys), as it would break existing programs.

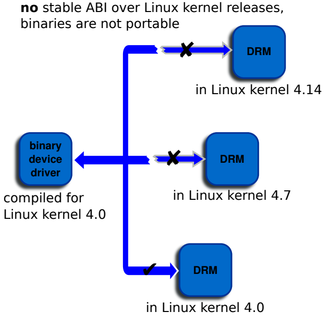


Image credits (Wikipedia):
https://bit.ly/2U2rdGB

Kernel memory constraints

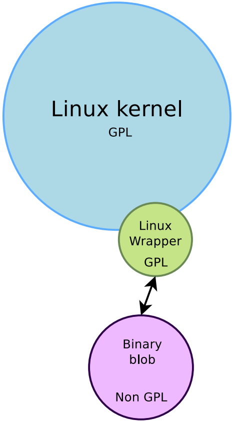
- No memory protection
- The kernel doesn't try to recover from attemps to access illegal memory locations. It just dumps *oops* messages on the system console.
- Fixed size stack (8 or 4 KB). Unlike in user space, no mechanism was implemented to make it grow. Don't use recursion!
- Swapping is not implemented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

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 validity of the GPL on this point has already been verified in courts
- 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: are they derived works of the kernel 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
  - Is it really useful to keep your drivers secret?
- There are some examples of proprietary drivers, like the Nvidia graphics drivers
  - They use a wrapper between the driver and the kernel
  - Unclear whether it makes it legal or not



The same binary blob could be used with a different OS kernel, through a different wrapper. This way, you cannot argue that the binary blog is an extension of the Linux kernel and that the GPL should apply to it too.

Advantages of GPL drivers

- You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- You could get free community contributions, support, code review and testing, though this generally only happens with code submitted for the mainline kernel.
- Your drivers can be freely and easily shipped by others (for example by Linux distributions or embedded Linux build systems).
- Pre-compiled drivers work with only one kernel version and one specific configuration, making life difficult for users who want to change the kernel version.
- Legal certainty, you are sure that a GPL driver is fine from a legal point of view.

Advantages of in-tree kernel drivers

Once your sources are accepted in the mainline tree...

- There are many more people reviewing your code, allowing to get cost-free security fixes and improvements.
- You can also get changes from people modifying internal kernel APIs.
- Accessing your code is easier for users.
- You can get contributions from your own customers.

This will for sure reduce your maintenance and support work

User space device drivers 1/3

- In some cases, it is possible to implement device drivers in user space!
- Can be used when
  - The kernel provides a mechanism that allows user space applications to directly access the hardware.
  - 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.

User space device drivers 2/3

- Possibilities for user space device drivers:
  - USB with *libusb*, https://libusb.info/
  - SPI with *spidev*, spi/spidev
  - I2C with *i2cdev*, i2c/dev-interface
  - Memory-mapped devices with *UIO*, including interrupt handling, driver-api/uio-howto
- Certain classes of devices (printers, scanners, 2D/3D graphics acceleration) are typically handled partly in kernel space, partly in user space.

User space device drivers 3/3

- Advantages
  - No need for kernel coding skills. Easier to reuse code between devices.
  - 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 be swapped out (kernel code cannot be).
  - Can use floating-point computation.
  - Less in-kernel complexity.
  - Potentially higher performance, especially for memory-mapped devices, thanks to the avoidance of system calls.
- Drawbacks
  - Missing hardware abstraction provided by the kernel, need to adapt applications when replacing one device by another.
  - Less straightforward to handle interrupts.
  - Increased interrupt latency vs. kernel code.

Linux sources

Linux sources structure 1/5

- `arch/<ARCH>`
  - Architecture specific code
  - `arch/<ARCH>/mach-<machine>`, SoC family specific code
  - `arch/<ARCH>/include/asm`, architecture-specific headers
  - `arch/<ARCH>/boot/dts`, Device Tree source files, for some architectures
- `block/`
  - Block layer core
- `certs/`
  - Management of certificates for key signing
- `COPYING`
  - Linux copying conditions (GNU GPL)
- `CREDITS`
  - Linux main contributors

Linux sources structure 2/5

- `crypto/`
  - Cryptographic libraries
- `Documentation/`
  - Kernel documentation sources
    Generated documentation available on https://kernel.org/doc/
    (includes functions prototypes and comments extracted from source code).
- `drivers/`
  - All device drivers except sound ones (usb, pci...)
- `fs/`
  - Filesystems (`fs/ext4/`, etc.)
- `include/`
  - Kernel headers
- `include/linux/`
  - Linux kernel core headers

Linux sources structure 3/5

- `include/uapi/`
  - User space API headers
- `init/`
  - Linux initialization (including `init/main.c`)
- `ipc/`
  - Code used for Inter Process Communication
- `Kbuild`
  - Part of the kernel build system
- `Kconfig`
  - Top level description file for configuration parameters
- `kernel/`
  - Linux kernel core (very small!)
- `lib/`
  - Misc library routines (zlib, crc32...)

Linux sources structure 4/5

- MAINTAINERS
  - Maintainers of each kernel part. Very useful!
- Makefile
  - Top Linux Makefile (sets version information)
- mm/
  - Memory management code (small too!)
- net/
  - Network support code (not drivers)
- README
  - Description of kernel documentation
- samples/
  - Sample code (markers, kprobes, kobjects, bpf...)

Linux sources structure 5/5

- `scripts/`
  - Executables for kernel building and debugging
- `security/`
  - Security model implementations (SELinux...)
- `sound/`
  - Sound support code and drivers
- `tools/`
  - Code for various user space tools (mostly C, example: `perf`)
- `usr/`
  - Code to generate an initramfs cpio archive
- `virt/`
  - Virtualization support (KVM)

Kernel source management tools

Cscope

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

Cscope screenshot

```
C symbol: request_irq

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

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

- [Tab]: move the cursor between search results and commands
- [Ctrl] [D]: exit cscope

Elixir: browsing the Linux kernel sources

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

https://elixir.bootlin.com

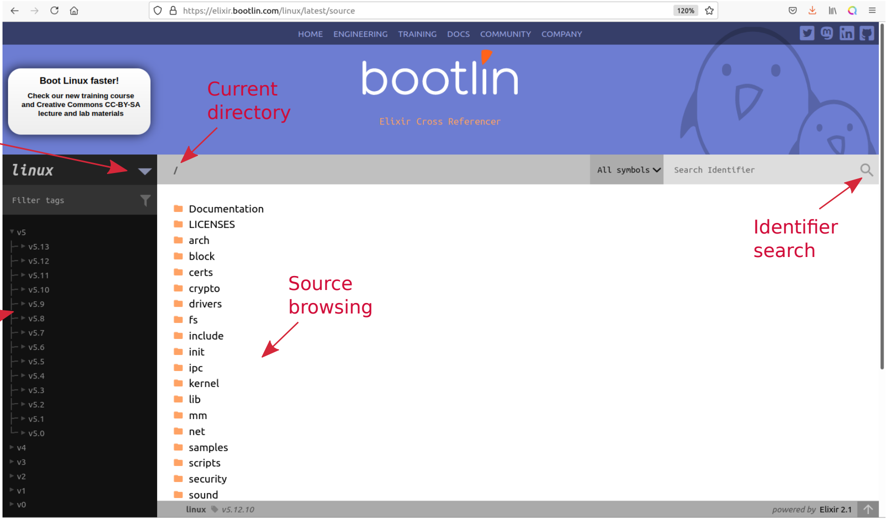
Project selection (U-Boot, Linux, BusyBox...)

Current directory

Identifier search

All versions available

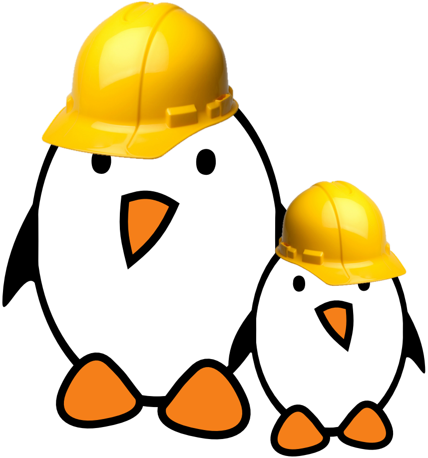
Source browsing



Text editors and IDEs for kernel development

- You can use text editors (Emacs, Vim...) to work on kernel code.
- At least Vim and Emacs support ctags and cscope and therefore can help with symbol lookup and auto-completion.
- It's also possible to use more elaborate IDEs to develop kernel code, such as Eclipse, QtCreator and most often Visual Studio Code: See Michael Opdenacker's presentation ELCE 2020:
  - Title: Using Visual Studio Code for Embedded Linux Development
  - Slides: https://tinyurl.com/y6d8yje7
  - Video: https://youtu.be/YGOZIIOWujc

Practical lab - Kernel Source Code - Exploring



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

Building the kernel

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-4.14.x/`
  - `make <target>`

Specifying the target architecture

First, specify the architecture for the kernel to build

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

Choose a compiler

The compiler invoked by the kernel Makefile is `$(CROSS_COMPILE)gcc`

- Specifying the compiler is already needed at configuration time, as some kernel configuration options depend on the capabilities of the compiler.
- When compiling natively
  - Leave `CROSS_COMPILE` undefined and the kernel will be natively compiled for the host architecture using `gcc`.
- When using a cross-compiler
  - To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:
    `mips-linux-gcc`: the prefix is `mips-linux-`
    `arm-linux-gnueabi-gcc`: the prefix is `arm-linux-gnueabi-`
  - So, you can specify your cross-compiler as follows:
    `export CROSS_COMPILE=arm-linux-gnueabi-`

`CROSS_COMPILE` is actually the prefix of the cross compiling tools (`gcc`, `as`, `ld`, `objcopy`, `strip`...).

Specifying ARCH and CROSS_COMPILE

There are actually two ways of defining `ARCH` and `CROSS_COMPILE`:

- Pass `ARCH` and `CROSS_COMPILE` on the `make` command line:
  ```
  make ARCH=arm CROSS_COMPILE=arm-linux- ...
  ```
  Drawback: it is easy to forget to pass these variables when you run any `make` command, causing your build and configuration to be screwed up.
- Define `ARCH` and `CROSS_COMPILE` as environment variables:
  ```
  export ARCH=arm
  export CROSS_COMPILE=arm-linux-
  ```
  Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your `~/.bashrc` file to make them permanent and visible from any terminal.

Kernel configuration details

- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `CONFIG_PARAM=value` (included by the kernel Makefile)
- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:
  - `make xconfig`, `make gconfig` (graphical)
  - `make menuconfig`, `make nconfig` (text)
  - You can switch from one to another, they all load/save the same `.config` file, and show the same set of options

Initial configuration

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

- Desktop or server case:
  - Advisable to start with the configuration of your running kernel, usually available in `/boot`:
    `cp /boot/config-`uname -r` .config`
- Embedded platform case (at least on ARM 32 bit):
  - Default configuration files are available, usually for each CPU family.
  - They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files (only settings different from default ones).
  - Run `make help` to find if one is available for your platform
  - To load a default configuration file, just run
    `make cpu_defconfig`
  - This will overwrite your existing `.config` file!

Now, you can make configuration changes (`make menuconfig`...).

Create your own default configuration

To create your own default configuration file:

- `make savedefconfig`
  This creates a minimal configuration (non-default settings)
- `mv defconfig arch/<arch>/configs/myown_defconfig`
  This way, you can share a reference configuration inside the kernel sources.

Kernel or module?

- The **kernel image** is a **single file**, resulting from the linking of all object files that correspond to features enabled in the configuration
  - This is the file that gets loaded in memory by the bootloader
  - All included 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

There are dependencies between kernel options

- For example, enabling a network driver requires the network stack to be enabled
- Two types of dependencies:
  - `depends on` dependencies. In this case, option B that depends on option A is not visible until option A is enabled
  - `select` dependencies. In this case, with option B depending on option A, when option A is enabled, option B is automatically enabled. In particular, such dependencies are used to declare what features a hardware architecture supports.

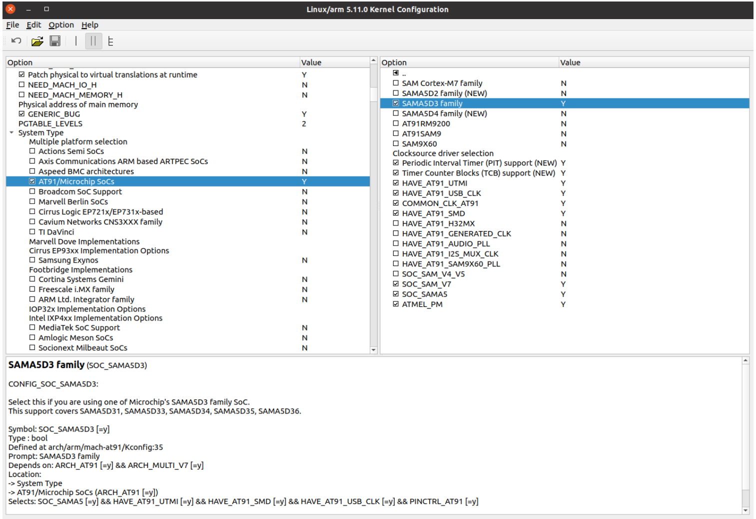
```
menuconfig ATA
	tristate "Serial ATA and Parallel ATA drivers (libata)"
	depends on HAS_IOMEM
	depends on BLOCK
	select SCSI
	select GLOB
	---help---
	  If you want to use an ATA hard disk, ATA tape drive, ATA CD-ROM or
	  any other ATA device under Linux, say Y and make sure that you know
	  the name of your ATA host adapter (the card inside your computer
	  that "speaks" the ATA protocol, also called ATA controller),
	  because you will be asked for it.
```

Kconfig file excerpt

`make xconfig`

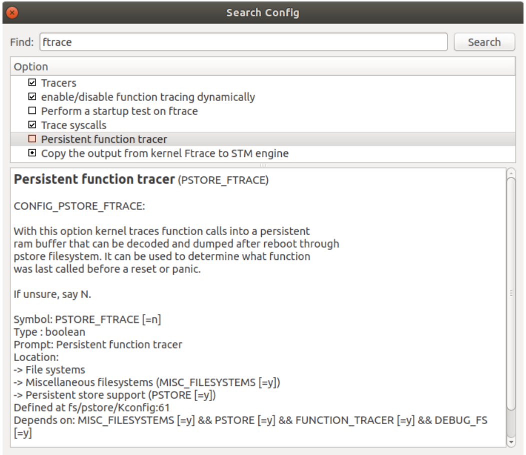
- The most common graphical interface to configure the kernel.
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `qt5-default`

make xconfig screenshot



make xconfig search interface

Looks for a keyword in the parameter name (shortcut: `[Ctrl]` + `[f]`).
Allows to set values to found parameters.



Kernel configuration options

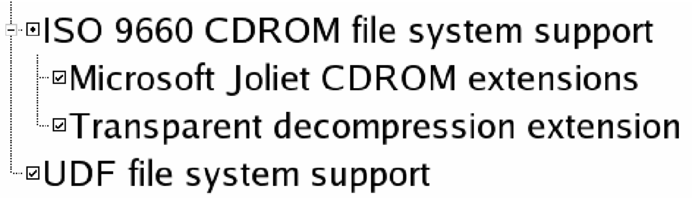
Compiled as a module (separate file)

CONFIG_ISO9660_FS=m

Driver options

CONFIG_JOLIET=y

CONFIG_ZISOFS=y



Compiled statically into the kernel

CONFIG_UDF_FS=y

Values in resulting .config file

Parameter values as displayed in make xconfig

Corresponding .config file excerpt

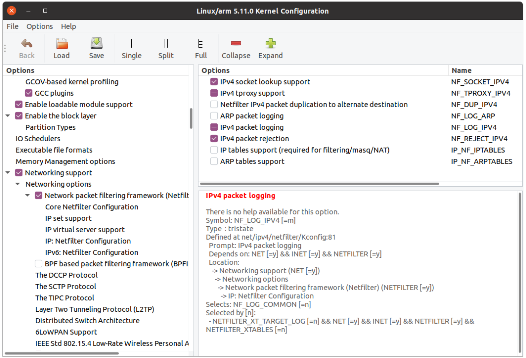
Options are grouped by sections and are prefixed with `CONFIG_`.

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

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

make gconfig

`make gconfig`

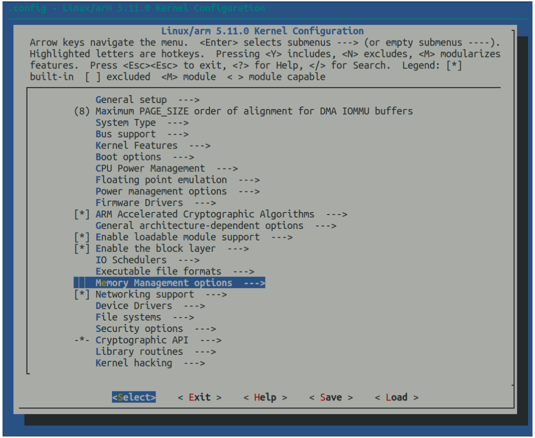
- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`



make menuconfig

`make menuconfig`

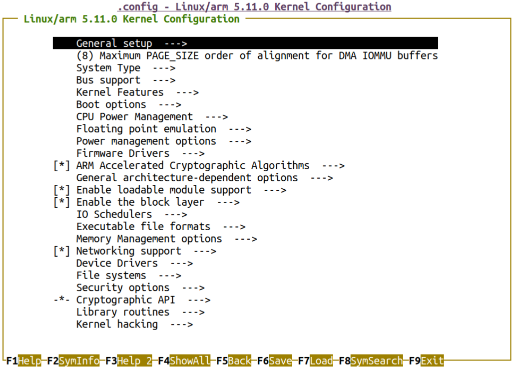
- Useful when no graphics are available. Very efficient interface.
- Same interface found in other tools: BusyBox, Buildroot...
- Convenient number shortcuts to jump directly to search results.
- Required Debian packages: `libncurses-dev`



make nconfig

`make nconfig`

- A newer, similar text interface
- More user friendly (for example, easier to access help information).
- However, lacking the shortcuts that `menuconfig` offers in search results. Therefore, much less convenient than `menuconfig`.
- Required Debian packages: `libncurses-dev`



make oldconfig

`make oldconfig`

- Needed very often!
- 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 interfaces of the kernel (`xconfig`, `menuconfig`, `oldconfig`...) keep this `.config.old` backup copy.

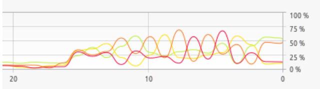
Compiling and installing the kernel

make

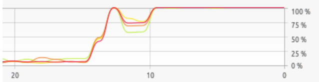
- Run it in the main kernel source directory!
- Remember to run multiple jobs in parallel if you have multiple CPU cores / threads. Our advice: `ncpus * 2` or `ncpus + 2`, to fully load the CPU and I/Os at all times.
  Example: `make -j 8`
- No need to run as root!
- To **re**compile faster (7x according to some benchmarks), use the `ccache` compiler cache:
  `export CROSS_COMPILE="ccache riscv64-linux-"`

Benefits on parallel make

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



`make`
total time: 129 s



`make -j8`
total time: 67 s

Kernel compilation results

- `vmlinux`, the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted
- `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can 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 files (on some architectures)
- All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.

Kernel installation: native case

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

Kernel installation: embedded case

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

Module installation: native case

- `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. Used to find drivers for devices. Example line:
    `alias usb:v066Bp20F9d*dc*dsc*dp*ic*isc*ip*in* asix`
  - `modules.dep`, `modules.dep.bin`
    Module dependencies
  - `modules.symbols`, `modules.symbols.bin`
    Tells which module a given symbol belongs to.

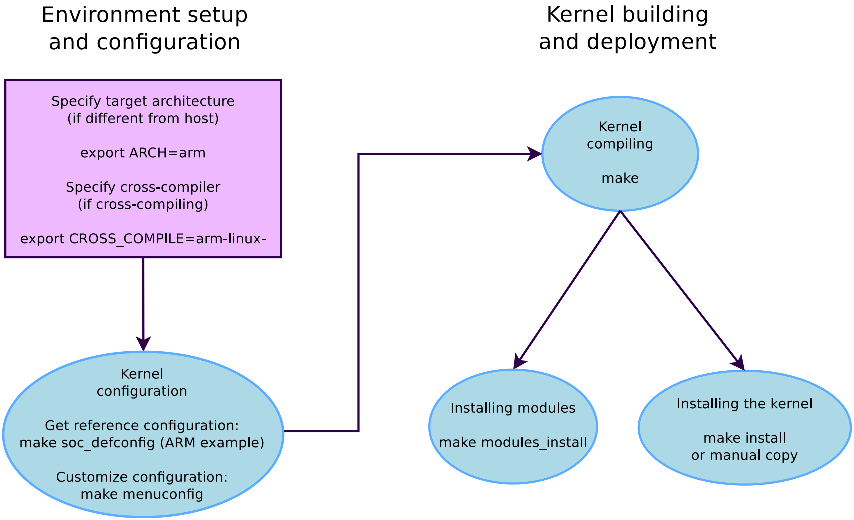
Module installation: embedded case

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

Kernel cleanup targets

- Clean-up generated files (to force re-compilation):
  `make clean`
- Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your `.config` file!
  `make mrproper`
- Also remove editor backup and patch reject files (mainly to generate patches):
  `make distclean`
- If you are in a git tree, remove all files not tracked (and ignored) by git:
  `git clean -fdx`

Kernel building overview



Booting the kernel

Device Tree (1)

- Many embedded architectures have a lot of non-discoverable hardware (serial, Ethernet, I2C, Nand flash, USB controllers...)
- Depending on the architecture, such hardware is either described in BIOS ACPI tables (x86), using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- The Device Tree (DT) was created for PowerPC, and later was adopted by other architectures (ARM, ARC...). Now Linux has DT support in most architectures, at least for specific systems (for example for the OLPC on x86).

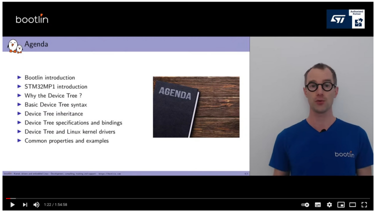
Device Tree (2)

- A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, and needs to be passed to the kernel at boot time.
  - There is one different Device Tree for each board/platform supported by the kernel, available in `arch/arm/boot/dts/<board>.dtb`.
  - See `arch/arm/boot/dts/at91-sama5d3_xplained.dts` for example.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Customize your board device tree!

Often needed for embedded board users:

- To describe external devices attached to non-discoverable busses (such as I2C) and configure them.
- To configure pin muxing: choosing what SoC signals are made available on the board external connectors. See http://linux.tanzilli.com/ for a web service doing this interactively.
- To configure some system parameters: flash partitions, kernel command line (other ways exist)
- Device Tree 101 webinar, Thomas Petazzoni (2021):
  Slides: https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/
  Video: https://youtu.be/a9CZ1Uk3OYQ



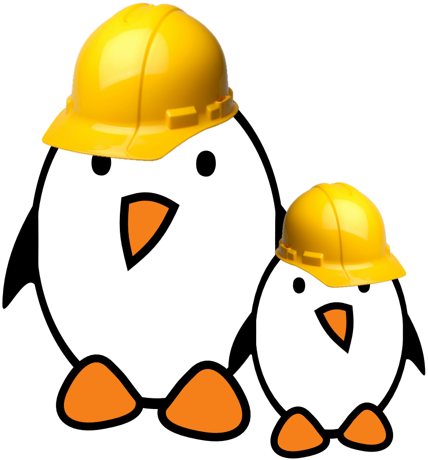
Booting with U-Boot

- Recent versions of U-Boot can boot the `zImage` binary.
- Older versions require a special kernel image format: `uImage`
  - `uImage` is generated from `zImage` using the `mkimage` tool. It is done automatically by the kernel `make uImage` target.
  - On some ARM platforms, `make uImage` requires passing a `LOADADDR` environment variable, which indicates at which physical memory address the kernel will be executed.
- In addition to the kernel image, U-Boot can also pass a *Device Tree Blob* to the kernel.
- The typical boot process is therefore:
  1. Load `zImage` or `uImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `bootz X - Y` (`zImage` case), or `bootm X - Y` (`uImage` case)
     The `-` in the middle indicates no *initramfs*

Kernel command line

- In addition to the compile time configuration, the kernel behavior can be adjusted with no recompilation using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root filesystem (covered later)
  - `console=` for the destination of kernel messages
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 rootwait`
  - Many more exist. The most important ones are documented in `admin-guide/kernel-parameters` in kernel documentation.
- This kernel command line can be, in order of priority (highest to lowest):
  - Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel.
  - Specified in the Device Tree (for architectures which use it)
  - Built into the kernel, using the `CONFIG_CMDLINE` option.
  - A combination of the above depending on the kernel configuration.

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:

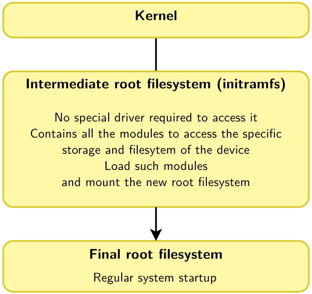
- Set up a cross-compiling environment
- Cross-compile a kernel for an ARM target platform
- Boot this kernel from a directory on your workstation, accessed by the board through NFS

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 dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the `ubifs` module depends on the `ubi` and `mtd` modules.
- Dependencies are described both in `/lib/modules/<kernel-version>/modules.dep` and in `/lib/modules/<kernel-version>/modules.dep.bin` (binary hashed format) These files are generated when you run `make modules_install`.

Kernel log

When a new module is loaded, related information is available in the kernel log.

- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the `dmesg` command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel command line parameter, or completely disabled with the `quiet` parameter). Example:
  `console=ttyS0 root=/dev/mmcblk0p2 loglevel=5`
- Note that you can write to the kernel log from user space too:
  `echo "<n>Debug info" > /dev/kmsg`

Module utilities (1)

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

- `modinfo <module_name>` (for modules in `/lib/modules`)
  `modinfo <module_path>.ko`
  Gets information about a module without loading it: parameters, license, description and dependencies.
- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.

Understanding module loading issues

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

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

Module utilities (2)

- `sudo modprobe <module_name>`
  Most common usage of `modprobe`: tries to load all the modules the given module depends on, and then this module. Lots of other options are available. `modprobe` automatically looks in `/lib/modules/<version>/` for the object file corresponding to the given module name.
- `lsmod`
  Displays the list of loaded modules
  Compare its output with the contents of `/proc/modules`!

Module utilities (3)

- sudo rmmod <module_name>
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- sudo modprobe -r <module_name>
  Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules

- Find available parameters:
  `modinfo usb-storage`
- Through `insmod`:
  `sudo insmod ./usb-storage.ko delay_use=0`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options usb-storage delay_use=0`
- Through the kernel command line, when the driver is built statically into the kernel:
  `usb-storage.delay_use=0`
  - `usb-storage` is the *driver name*
  - `delay_use` is the *driver parameter name*. It specifies a delay before accessing a USB storage device (useful for rotating devices).
  - `0` is the *driver parameter value*

Check module parameter values

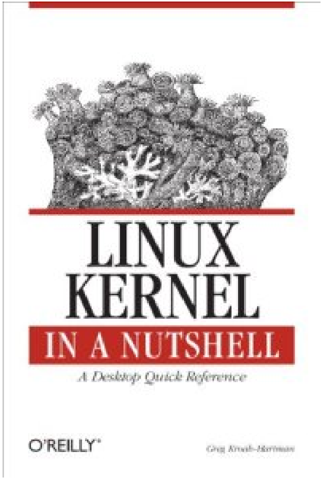
How to find/edit the current values for the parameters of a loaded module?

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

Useful reading

Linux Kernel in a Nutshell, Dec. 2006

- By Greg Kroah-Hartman, O'Reilly
  http://www.kroah.com/lkn/
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  https://bootlin.com/community/kernel/lkn/
- Getting old but still containing useful content.



Developing kernel modules

Corrections, suggestions, contributions and translations are welcome!

Hello module 1/2

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

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

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

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

Hello module 2/2

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

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

- Code marked as `__exit`:
  - Discarded when module compiled statically into the kernel, or when module unloading support is not enabled.
- Code of this example module available on https://frama.link/Q3CNXnom

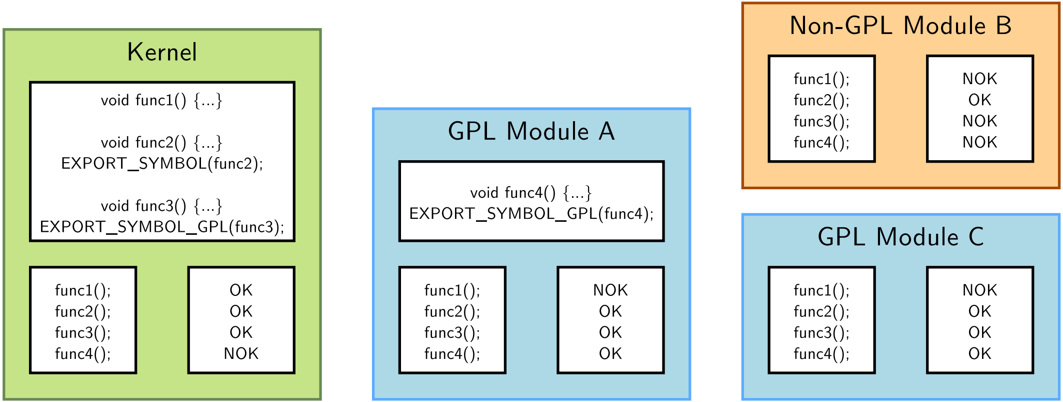
Hello module explanations

- 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



Module license

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

Compiling a module

Two solutions

- *Out of tree*
  - When the code is outside of the kernel source tree, in a different directory
  - Advantage: Might be easier to handle than modifications to the kernel itself
  - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
- Inside the kernel tree
  - Well integrated into the kernel configuration/compilation process
  - Driver can be built statically if needed

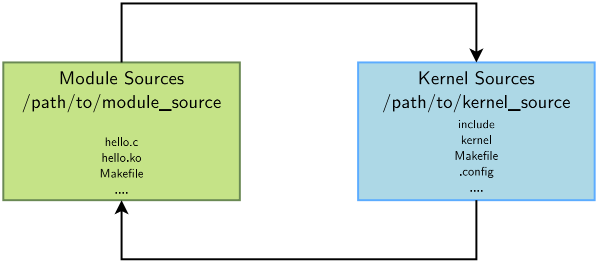
Compiling an out-of-tree module 1/2

- 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.

Modules and kernel version

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

New driver in kernel sources 1/2

- 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

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

MODULE_LICENSE("GPL");

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

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

Hello module with parameters 2/2

```
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

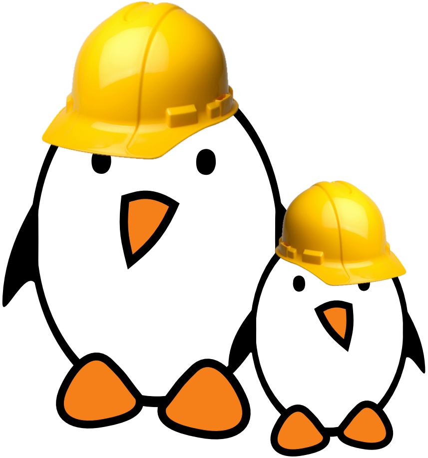
Source code available on: https://github.com/bootlin/training-materials/blob/master/code/hello-param/hello_param.c

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

Useful general-purpose kernel APIs

Memory/string utilities

- 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: kmemdup()
- In include/linux/kernel.h
  - String to int conversion: simple_strtoul(), simple_strtol(), simple_strtoull(), simple_strtoll()
  - Other string functions: sprintf(), sscanf()

Linked lists

- 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 */
    }
    [...]
}
```

Linux device and driver model

embedded Linux and kernel engineering

Introduction

The need for a device model?

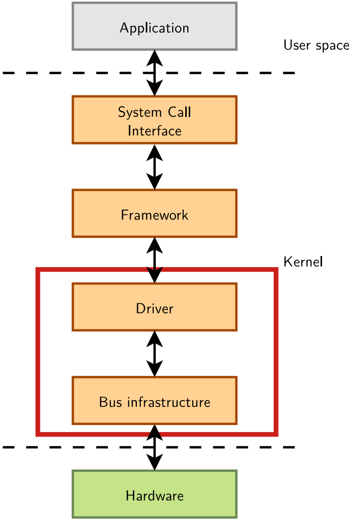
- The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to **maximize the reusability** of code between platforms.
- For example, we want the same *USB device driver* to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these 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.



Device Model data structures

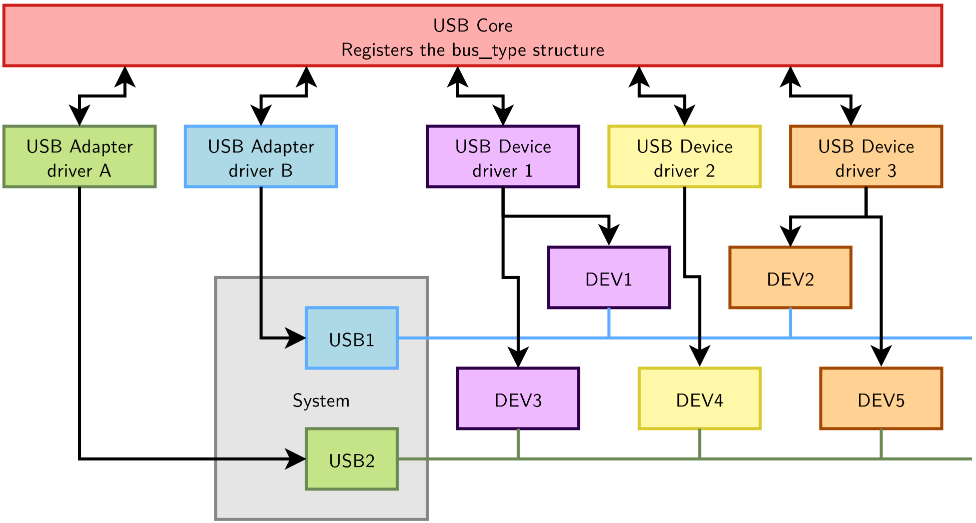
- The *device model* is organized around three main data structures:
  - The `struct bus_type` structure, which represents one type of bus (USB, PCI, I2C, etc.)
  - The `struct device_driver` structure, which represents one driver capable of handling certain devices on a certain bus.
  - The `struct 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.

Bus Drivers

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

Example of the USB bus

Example: USB Bus 1/2

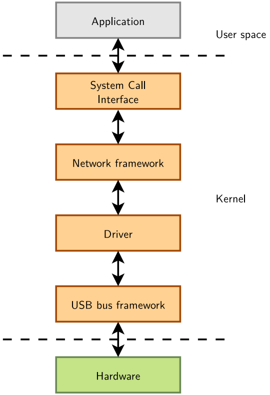


Example: USB Bus 2/2

- 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.

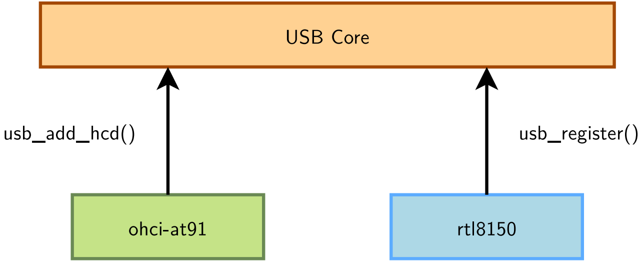
```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

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);
```

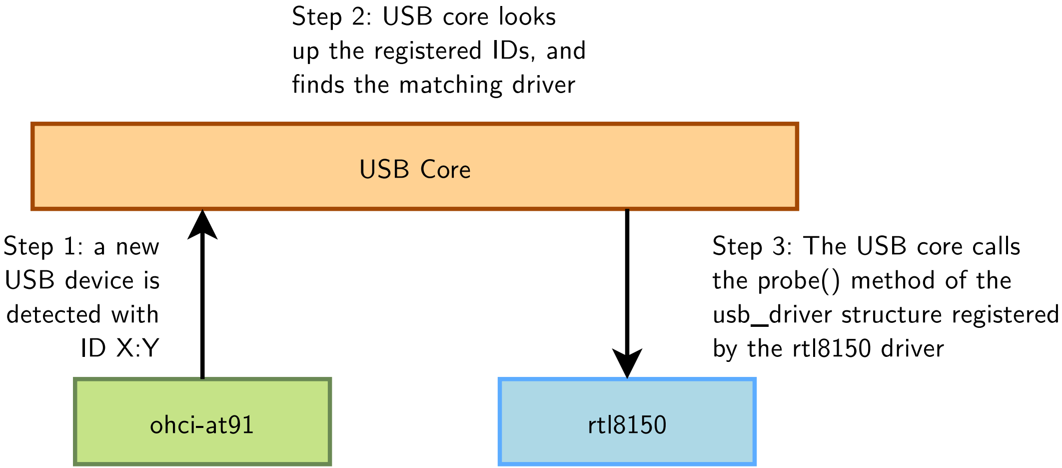
At Initialization

- 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



Probe Method

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

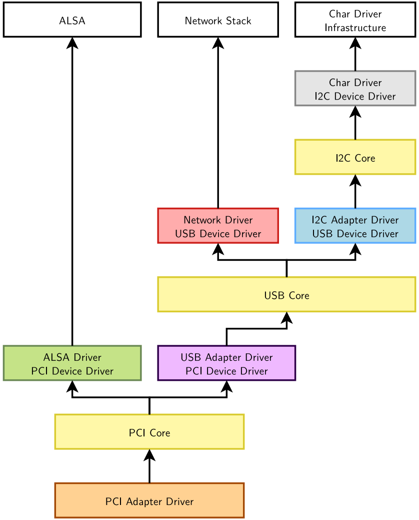
Example: probe() and disconnect() methods

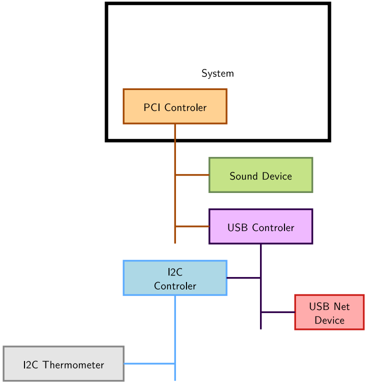
```c
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

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

    return 0;
}
```

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

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

Source: drivers/net/usb/rtl8150.c

The Model is Recursive





Platform drivers

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 in either:
  - The kernel source code
  - The *Device Tree*, a hardware description file used on some architectures.
  - BIOS ACPI tables (x86/PC architecture)

Platform devices

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

Implementation of a Platform Driver (1)

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

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

Implementation of a Platform Driver (2)

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

```
static int __init imx_serial_init(void) {
    ret = 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);
```

Platform Device Instantiation: old style (1/2)

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of `struct platform_device` structures, as done on a few old ARM platforms. Definition done in the board-specific or SoC specific code.
  - By using a *device tree*, as done on Power PC (and on most ARM platforms) from which `struct platform_device` structures are created
- Example on ARM, where the instantiation was done in `arch/arm/mach-imx/mx1ads.c`

```
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

Platform device instantiation: old style (2/2)

- The device was part of a list

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device,
};
```

- And the list of devices was added to the system during board initialization

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

The Resource Mechanism

- Each device managed by a particular driver typically uses different hardware resources: addresses for the I/O registers, DMA channels, IRQ lines, etc.
- Such information can be represented using `struct resource`, and an array of `struct resource` is associated to a `struct platform_device`
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

Declaring resources (old style)

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

Using Resources (old style)

- When a `struct platform_device` was added to the system using `platform_add_devices()`, the `probe()` method of the platform driver was called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources:

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

platform_data Mechanism (old style)

- In addition to the well-defined resources, many drivers require driver-specific information for each platform device
- Such information could be passed using the `platform_data` field of `struct device` (from which `struct platform_device` inherits)
- As it is a `void *` pointer, it could be used to pass any type of information.
  - Typically, each driver defines a structure to pass information through `struct platform_data`

platform_data example 1/2

- The i.MX serial port driver defines the following structure to be passed through `struct platform_data`

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- The MX1ADS board code instantiated such a structure

```
static struct imxuart_platform_data uart_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

platform_data Example 2/2

- The `uart_pdata` structure was associated to the `struct platform_device` structure in the MX1ADS board file (the real code was slightly more complicated)

```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
         .platform_data = &uart_pdata,
    },
    .resource = imx_uart1_resources,
    [...]
};
```

- The driver can access the platform data:

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
    [...]
```

Device Tree

- On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable.
- Such architectures have moved, to use the *Device Tree*.
- It is a **tree of nodes** that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board.
- Each node can have a number of **properties** describing various properties of the devices: addresses, interrupts, clocks, etc.
- At boot time, the kernel is given a compiled version, the **Device Tree Blob**, which is parsed to instantiate all the devices described in the DT.
- On ARM, they are located in `arch/arm/boot/dts/`.

Device Tree example

```
uart0: serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
        status = "disabled";
};
```

- `serial@44e09000` is the **node name**
- `uart0` is a **label**, that can be referred to in other parts of the DT as `&uart0`
- other lines are **properties**. Their values are usually strings, list of integers, or references to other nodes.

Device Tree inheritance (1/2)

- Each particular hardware platform has its own *device tree*.
- However, several hardware platforms use the same processor, and often various processors in the same family share a number of similarities.
- To allow this, a *device tree* file can include another one. The trees described by the including file overlays the tree described by the included file. This can be done:
  - Either by using the `/include/` statement provided by the Device Tree language.
  - Either by using the `#include` statement, which requires calling the C preprocessor before parsing the Device Tree.

  Linux currently uses either one technique or the other (different from one ARM subarchitecture to another, for example).

Device Tree inheritance (2/2)

```
/ {
    compatible = "ti,am33xx";
    [...]
    ocp {
        [...]
        uart0: serial@0 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x0 0x1000>;
            interrupts = <72>;
            status = "disabled";
            [...]
        };
    };
};
```

am33xx-l4.dtsi (included by am33xx.dtsi)

Definition of the AM33xx SoC

+

```
[...]
&uart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins>;
        status = "okay";
};
```

am335x-bone-common.dtsi

Common definitions for BeagleBone boards

+

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include "am335x-boneblack-common.dtsi"

/ {
        model = "TI AM335x BeagleBone Black";
        compatible = "ti,am335x-bone-black",
                     "ti,am335x-bone",
                     "ti,am33xx";
};

[...]
```

am335x-boneblack.dts

Definition for BeagleBone Black

=

```
/ {
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
    model = "TI AM335x BeagleBone Black";
    [...]
    ocp {
        [...]
        uart0: serial@0 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x0 0x1000>;
            interrupts = <72>;
            pinctrl-names = "default";
            pinctrl-0 = <&uart0_pins>;
            status = "okay";
        };
    };
};
```

am335x-boneblack.dtb

Compiled DTB

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Device Tree: compatible string

- With the *device tree*, a *device* is bound to the corresponding *driver* using the **compatible** string.
- The `of_match_table` field of `struct device_driver` lists the compatible strings supported by the driver. `drivers/tty/serial/omap-serial.c` example:

```
#if defined(CONFIG_OF)
static const struct of_device_id omap_serial_of_match[] = {
        { .compatible = "ti,omap2-uart" },
        { .compatible = "ti,omap3-uart" },
        { .compatible = "ti,omap4-uart" },
        {},
};
MODULE_DEVICE_TABLE(of, omap_serial_of_match);
#endif
static struct platform_driver serial_omap_driver = {
        .probe          = serial_omap_probe,
        .remove         = serial_omap_remove,
        .driver         = {
                .name   = DRIVER_NAME,
                .pm     = &serial_omap_dev_pm_ops,
                .of_match_table = of_match_ptr(omap_serial_of_match),
        },
};
```

- Note: the `of_match_ptr()` macro instantiates to `NULL` when `CONFIG_OF` is not set.

Device Tree Resources

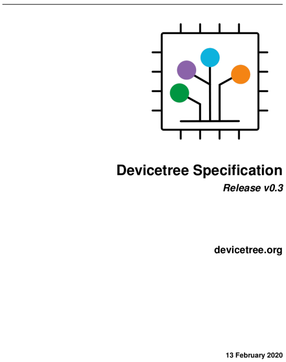
- The drivers will use the same mechanism that we saw previously to retrieve basic information: interrupts numbers, physical addresses, etc.
- The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver.
- Any additional information will be specific to a driver or the class it belongs to, defining the *bindings*.

Device Tree design principles

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

Device Tree specifications

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


Devicetree Specification
Release v0.3

devicetree.org

13 February 2020

Device Tree binding: old style

Documentation/devicetree/bindings/mtd/spear_smi.txt

```
* SPEAr SMI

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

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

```
Example:

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

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

Device Tree binding: YAML style

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

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

title: I2C controller embedded in STMicroelectronics STM32 I2C platform

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

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

  reg:
    maxItems: 1

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

  resets:
    maxItems: 1
```

```yaml
  clocks:
    maxItems: 1

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

  ...

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

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

Device Tree binding: YAML style example

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

Validating Device Tree in Linux

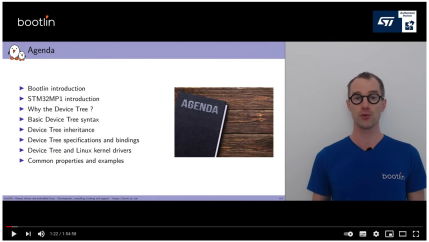
- `dtc` only does syntaxic validation
- YAML bindings allow to do semantic validation
- Linux kernel `make` rules:
  - `make dt_binding_check`
    verify that YAML bindings are valid
  - `make dtbs_check`
    validate DTs currently enabled against YAML bindings
  - `make DT_SCHEMA_FILES=Documentation/devicetree/bindings/trivial-devices.yaml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

sysfs

- 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!
- Take your time to explore `/sys` on your workstation.

References

- Device Tree 101 webinar, Thomas Petazzoni (2021):
  Slides: https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/
  Video: https://youtu.be/a9CZ1Uk3OYQ
- Kernel documentation
  - driver-api/driver-model/
  - devicetree/
  - filesystems/sysfs
- https://devicetree.org
- The kernel source code
  - Full of examples of other drivers!



Introduction to the I2C subsystem

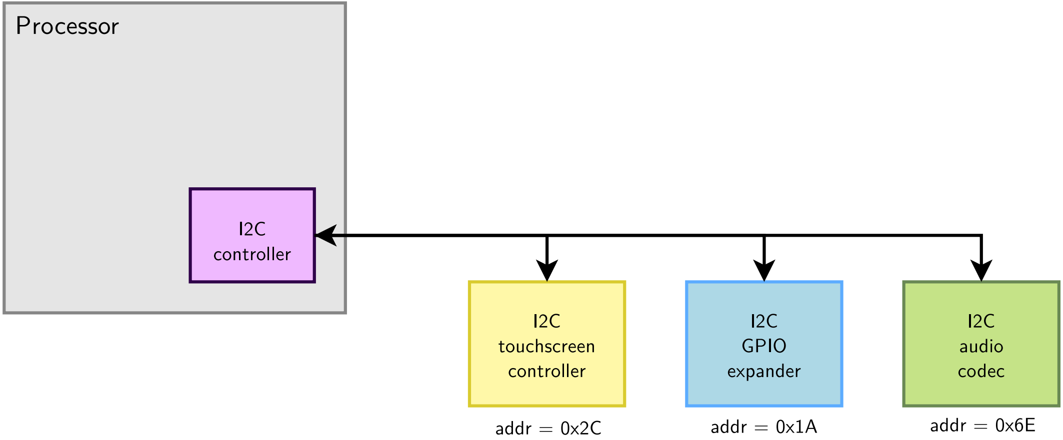
Corrections, suggestions, contributions and translations are welcome!

embedded Linux and kernel engineering

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`

`#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`

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

```c
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;
}
```

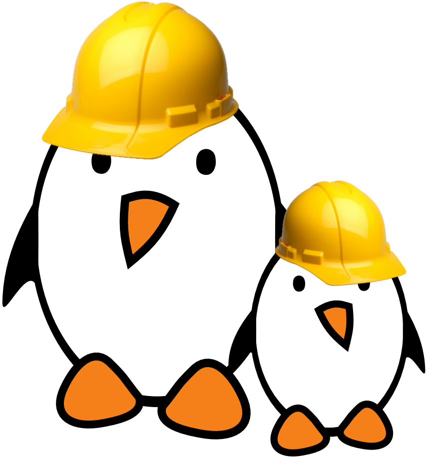
From drivers/iio/accel/da311.c

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

Practical lab - Linux device model for an I2C driver



- Modify the Device Tree to instantiate an I2C device.
- Implement a driver that registers as an I2C driver.
- Make sure that the probe/remove functions are called when there is a device/driver match.
- Explore the *sysfs* entries related to your driver and device.

Communicating with the I2C device: raw API

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

- `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, for example to read or write a register into a device.
- Linux provides SMBus functions that *should be used* instead of the raw API, if the I2C device supports this standard type of transactions. The driver can then be used on both SMBus and I2C adapters (can't use I2C commands on SMBus adapters).
- 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] S Addr Rd [A] [Data] NA P`
  - Which means it first writes a one byte data command (*Comm*), 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

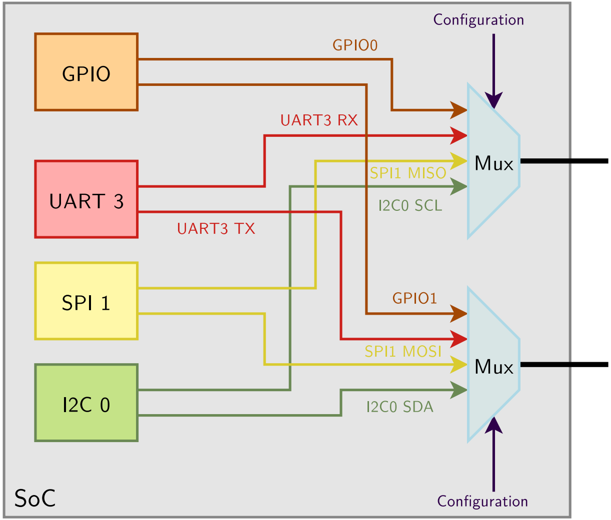
- https://en.wikipedia.org/wiki/I2C, general presentation of the I2C protocol
- i2c/, details about Linux support for I2C
  - i2c/writing-clients
    How to write I2C kernel device drivers
  - i2c/dev-interface
    How to write I2C user-space device drivers
  - i2c/instantiating-devices
    How to instantiate devices
  - i2c/smbus-protocol
    Details on the SMBus functions
  - i2c/functionality
    How the functionality mechanism works
- https://bootlin.com/pub/video/2012/elce/elce-2012-anders-board-bringup-i2c.webm, excellent talk: *You, me and I2C* from David Anders at ELCE 2012.

Introduction to pin muxing

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.

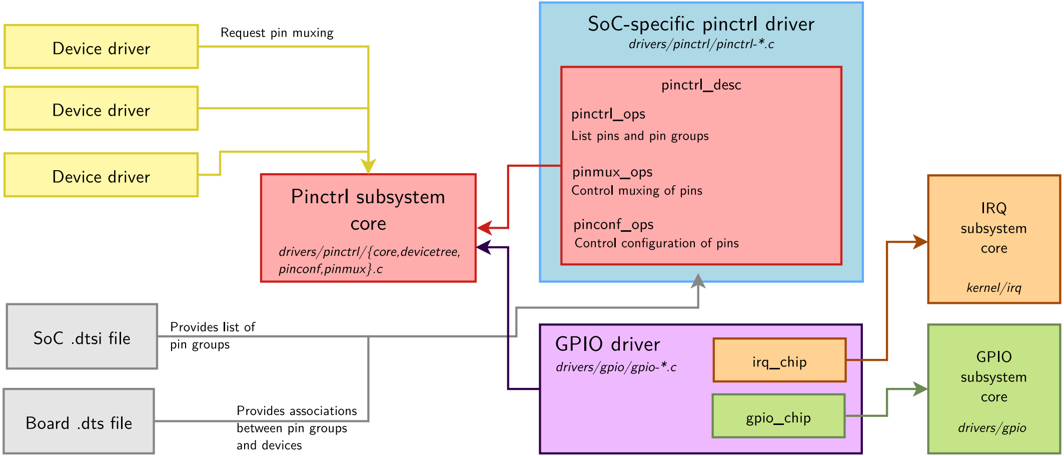
Pin muxing diagram



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 certains 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: pinctrl@01c20800 {
        compatible = "allwinner,sun7i-a20-pinctrl";
        reg = <0x01c20800 0x400>;
        ...

        uart0_pb_pins: uart0-pb-pins {
          pins = "PB22", "PB23";
          function = "uart0";
        };
        ...
      };
    };
};
```

UART0 pin mux config

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

Board level

```
/ {
...
        leds {
                compatible = "gpio-leds";
                pinctrl-names = "default";
                pinctrl-0 = <&led_pins_olinuxino>;

                led {
                        label = "a20-olinuxino-micro:green:usr";
                        gpios = <&pio 7 2 GPIO_ACTIVE_HIGH>;
                        default-state = "on";
                };
        };
...
};

&pio {
        ....

        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";
};
```

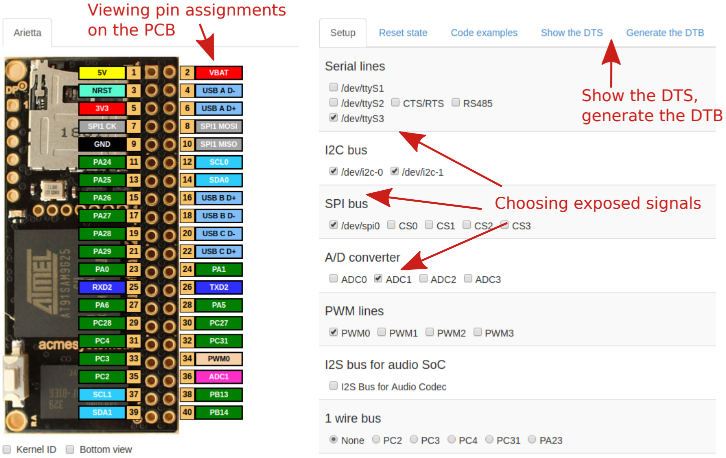
Declare LED device and associate pin mux config

LED pin mux config

Enable UART0 and associate pin mux config

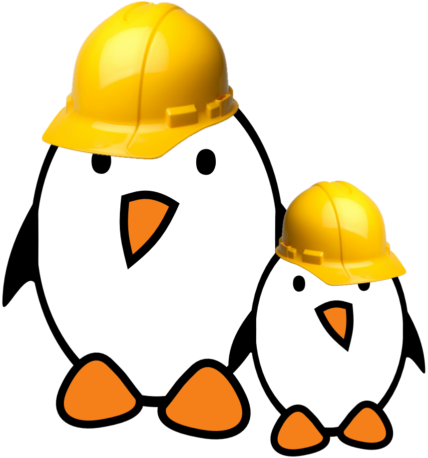
arch/arm/boot/dts/sun7i-a20-olinuxino-micro.dts

Illustration: live pin muxing configuration



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

Practical lab - Communicate with the Nunchuk



- Configure the pinmuxing for the I2C bus used to communicate with the Nunchuk
- Validate that the I2C communication works with user space tools.
- Extend the I2C driver started in the previous lab to communicate with the Nunchuk.

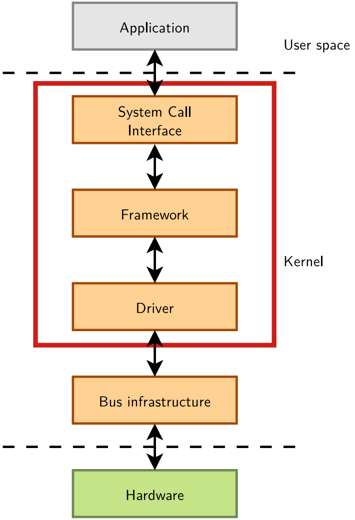
Kernel frameworks for device drivers

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

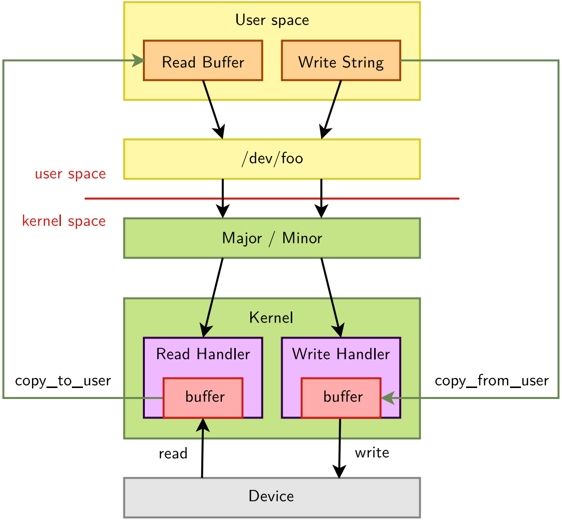
- Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needed root privileges
  - Coherency between device files and devices handled by the kernel was left to the system developer
- The `devtmpfs` 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.

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.

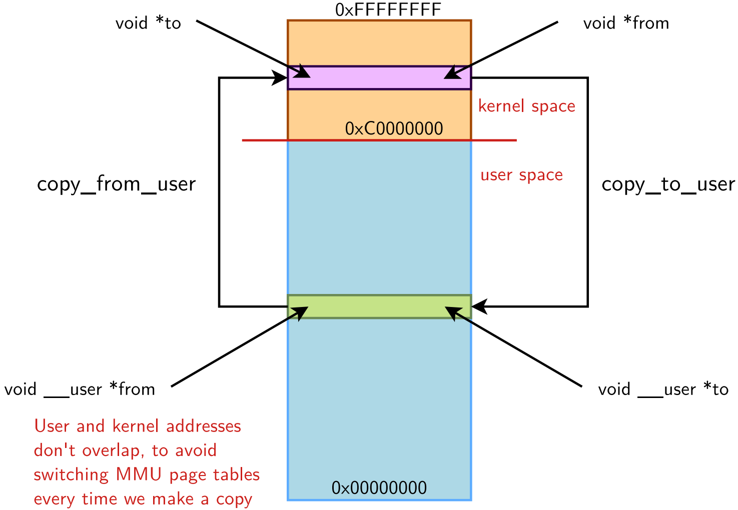
Exchanging data with user space 1/3

- Kernel code isn't allowed to directly access user space memory, using `memcpy()` or direct pointer dereferencing
  - Doing so does not work on some architectures
  - If the address passed by the application was invalid, the application would 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).
- 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



User and kernel addresses don't overlap, to avoid switching MMU page tables every time we make a copy

Zero copy access to user memory

- 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

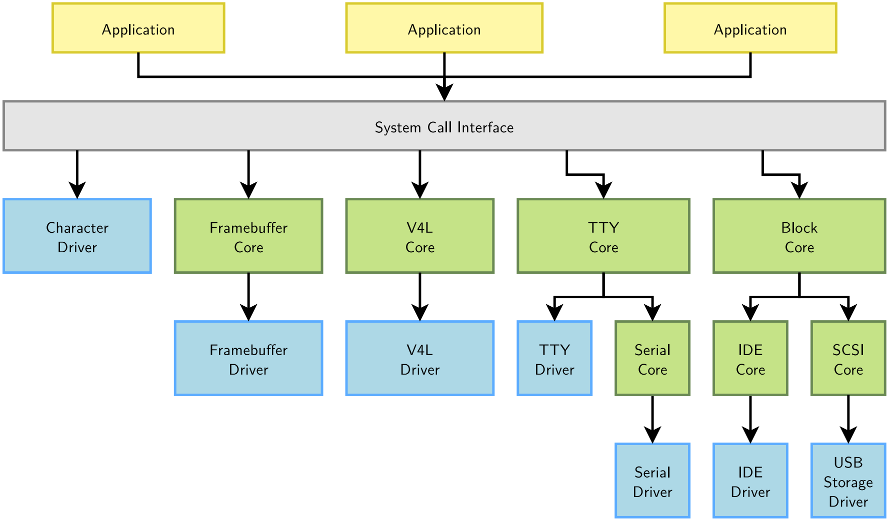
```
#include <linux/phantom.h>

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

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

Driver-specific Data Structure

- 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.

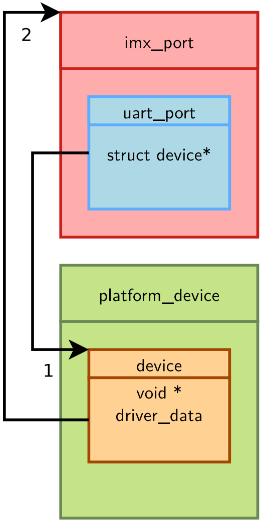
```
struct rtl8150 {
    unsigned long flags;
    struct usb_device *udev;
    struct tasklet_struct tl;
    struct net_device *netdev;
    [...]
};
```

Links between structures 1/4

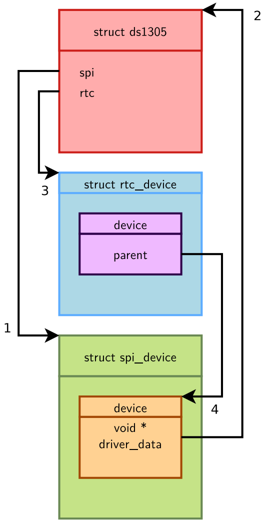
- The framework structure typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
  - It's the relationship between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- The device structure 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

```c
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);
    [...]
}
```

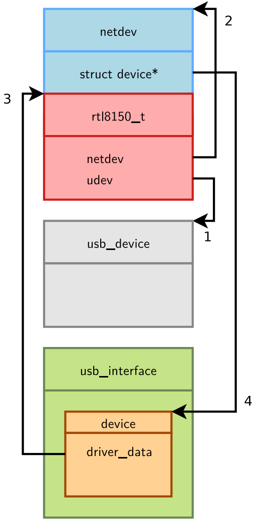


```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305                   *ds1305;

    [...]

    /* 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);

    [...]
}
```

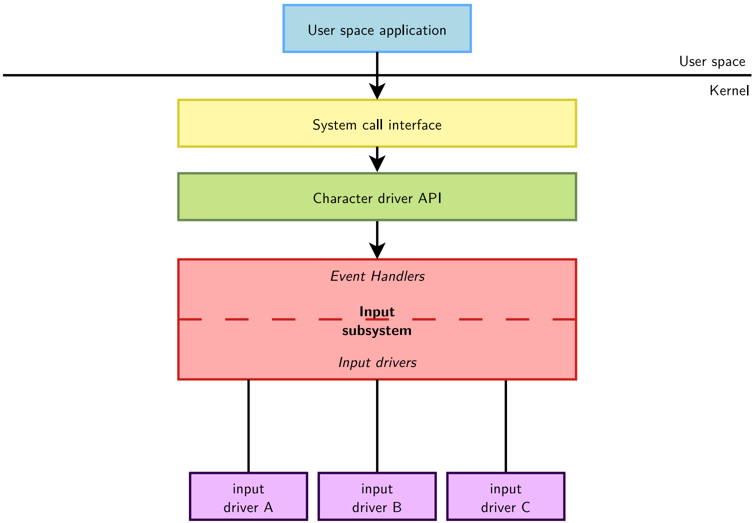


The input subsystem

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);`

Input subsystem API 2/3

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

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

- `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 *);`

Input subsystem API 3/3

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

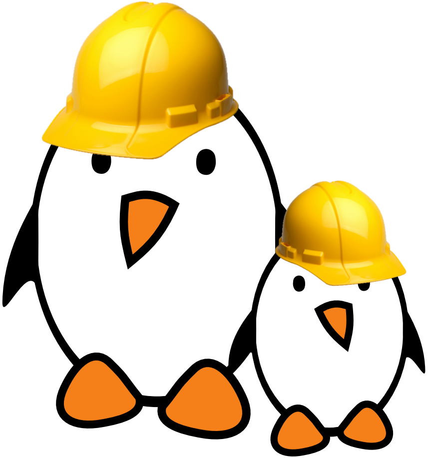
- The input subsystem provides an API to support simple input devices that *do not raise interrupts* but have to be *periodically scanned or polled* to detect changes in their state.
- Setting up polling is done using `input_setup_polling()`:
  `int input_setup_polling(struct input_dev *dev, void (*poll_fn)(struct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

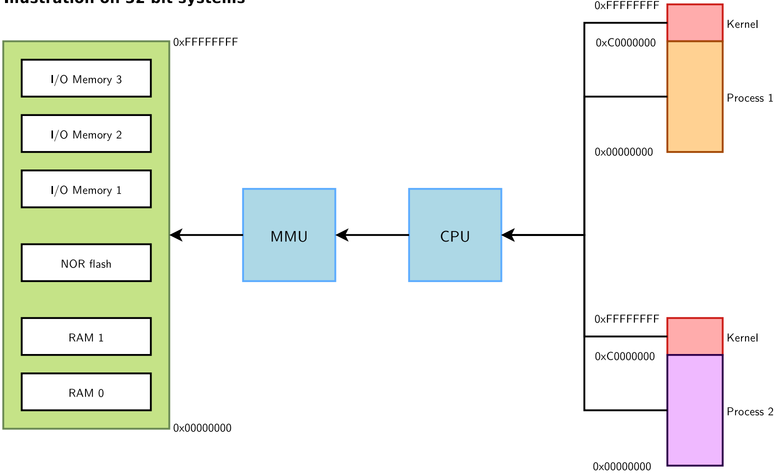


- Extend the Nunchuk driver to expose the Nunchuk features to user space applications, as an *input* device.
- Test the operation of the Nunchuk using `evtest`

Memory Management

Physical and virtual memory

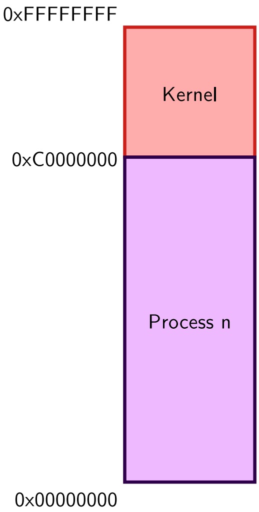
Illustration on 32 bit systems



Physical addresses

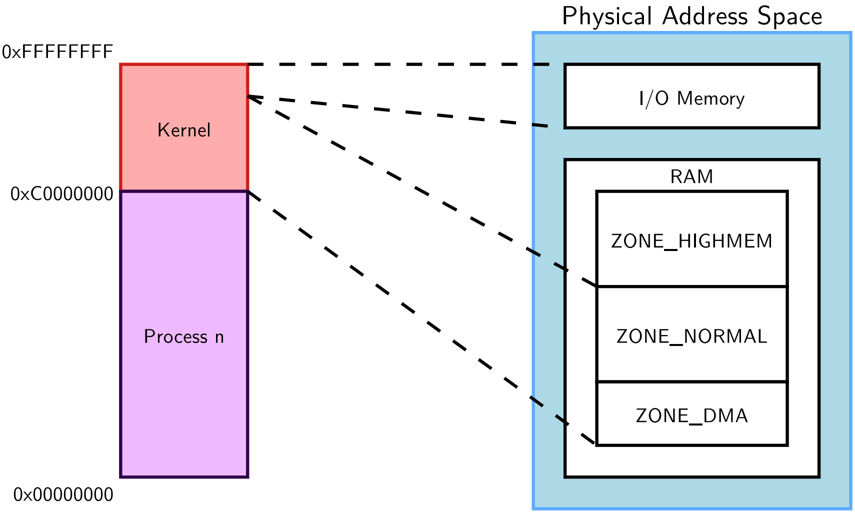
Virtual addresses

Virtual memory organization (on 32 bit)



- 1GB reserved for kernel-space
  - Contains kernel code and core data structures, identical in all address spaces
  - Most memory can be a direct mapping of physical memory at a fixed offset
- Complete 3GB exclusive mapping available for each user space process
  - Process code and data (program, stack, ...)
  - Memory-mapped files
  - Not necessarily mapped to physical memory (demand fault paging used for dynamic mapping to physical memory pages)
  - Differs from one address space to another

Physical / virtual memory mapping (on 32 bit)



Accessing more physical memory on 32 bit

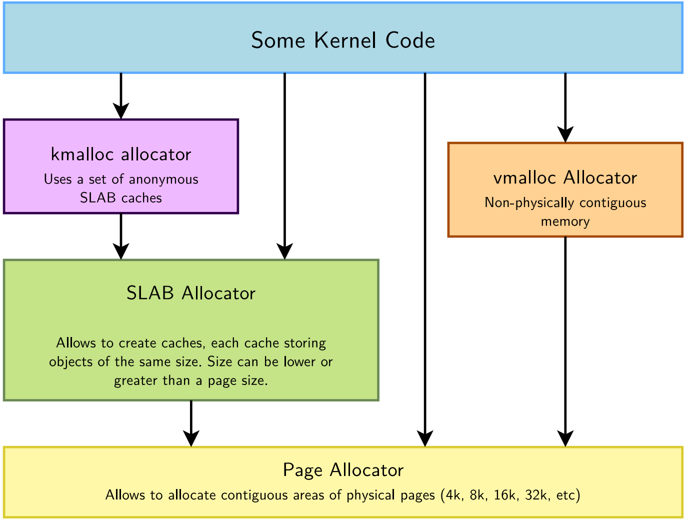
If you cannot use a 64 bit system (see x86/x86_64/mm for example)

- 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

Notes on user space memory

- New user space memory is allocated either from the already allocated process memory, or using the `mmap` system call
- Note that memory allocated 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
- User space memory allocation is allowed to over-commit memory (more than available physical memory) $\Rightarrow$ can lead to out of memory
- OOM killer kicks in and selects a process to kill to retrieve some memory. That's better than letting the system freeze.

Allocators in the kernel



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* is a solution to satisfy requests for large contiguous areas (see `https://lwn.net/Articles/486301/`).

Page allocator API: get free pages

- `unsigned long get_zeroed_page(int flags)`
  - Returns the virtual address of a free page, initialized to zero
  - `flags`: see the next pages for details.
- `unsigned long __get_free_page(int flags)`
  - Same, but doesn't initialize the contents
- `unsigned long __get_free_pages(int flags, unsigned int order)`
  - Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being `log2(number_of_pages)`.Can be computed from the size with the `get_order()` function.

Page allocator API: free pages

- `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

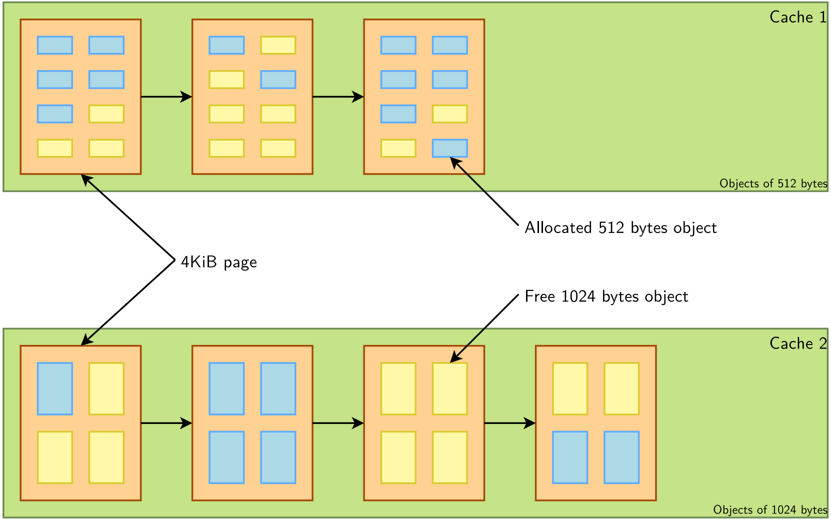
The most common ones are:

- 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.
- GFP_DMA
  - Allocates memory in an area of the physical memory usable for DMA transfers. See our DMA chapter.
- 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
    <X> 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`, `GFP_DMA`, 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, int 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_update_work *work;
work = kmalloc(sizeof *work, GFP_ATOMIC);
...
kfree(work);
```

kmalloc API 2/2

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

devm_ kmalloc functions

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

- `void *devm_kmalloc(struct device *dev, size_t size, int flags);`
- `void *devm_kzalloc(struct device *dev, size_t size, int flags);`
- `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.
- 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, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- Example use: to allocate kernel buffers to load module code.
- API in `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

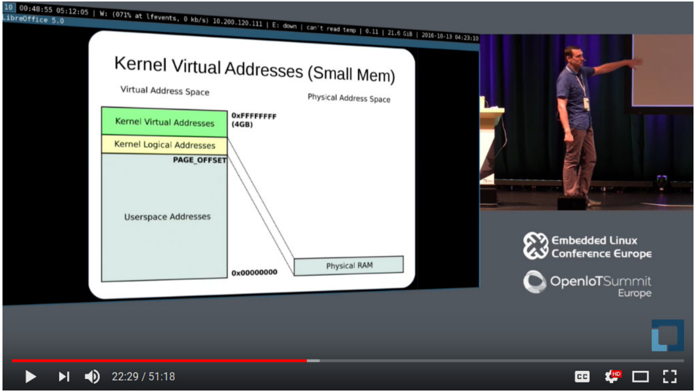
Kernel memory debugging

- 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

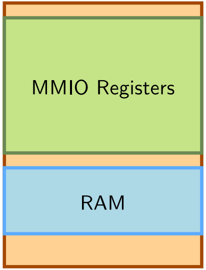
Virtual memory and Linux, Alan Ott and Matt Porter, 2016
Great and much more complete presentation about this topic
https://bit.ly/2Af1G2i (video: https://bit.ly/2Bwwv0C)



I/O Memory

Memory-Mapped I/O

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



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

Requesting I/O memory

- Tells the kernel which driver is using which I/O registers
- ```
  struct resource *request_mem_region(unsigned long start,
                                      unsigned long len, char *name);
  ```
- ```
  void release_mem_region(unsigned long start, unsigned long len);
  ```
- Allows to prevent other drivers 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
4804c000-4804cfff : 4804c000.gpio gpio@0
48060000-48060fff : 48060000.mmc mmc@0
4819c000-4819cfff : 4819c000.i2c i2c@0
481a8000-481a8fff : 481a8000.serial serial@0
481ac000-481acfff : 481ac000.gpio gpio@0
481ae000-481aefff : 481ae000.gpio gpio@0
481d8000-481d8fff : 481d8000.mmc mmc@0
49000000-4900ffff : 49000000.dma edma3_cc
4a100000-4a1007ff : 4a100000.ethernet ethernet@0
4a101200-4a1012ff : 4a100000.ethernet ethernet@0
80000000-9fdfffff : System RAM
80008000-80cfffff : Kernel code
80e00000-80f3d807 : Kernel data
```

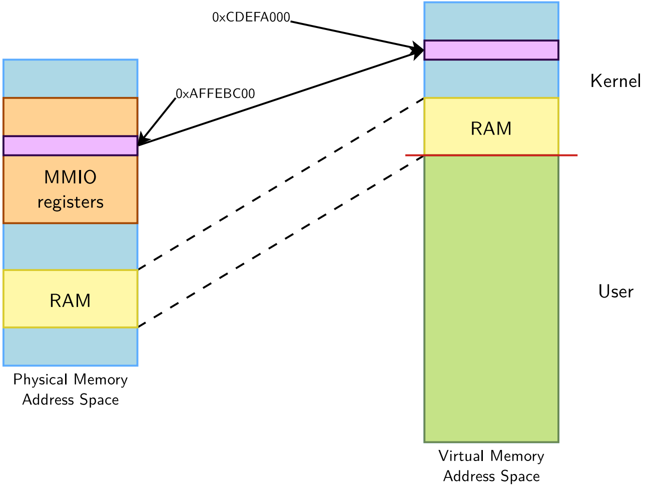
Mapping I/O memory in virtual memory

- 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`

Managed API

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

- devm_ioremap(), devm_iounmap()
- devm_ioremap_resource()
  - Takes care of both the request and remapping operations!
- devm_platform_ioremap_resource()
  - Takes care of platform_get_resource(), request_mem_region() and ioremap()
  - Caution: unlike the other devm_ functions, its first argument is of type struct 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

- Directly reading from or writing to addresses returned by `ioremap()` (*pointer dereferencing*) may not work on some architectures.
- To do PCI-style, little-endian accesses (byte swapping being done automatically assuming a little-endian device):

```
unsigned read[bwlq](void *addr);
void write[bwlq](unsigned val, void *addr);
```

- To do raw access, without endianness conversion

```
unsigned __raw_read[bwlq](void *addr);
void __raw_write[bwlq](unsigned val, void *addr);
```

- Little-endian is more frequent and also easier to use in drivers. Even if you just read the least significant byte of a 32-bit register, it's still at the same address.
- Example
  - 32 bit write (`drivers/tty/serial/uartlite.c`):

```
writel(c & 0xff, port->membase + 4);
```

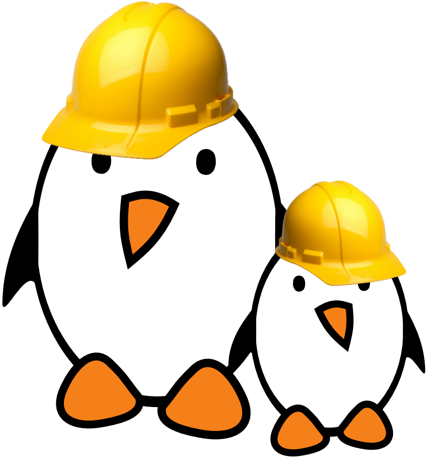
Avoiding I/O access issues

- Caching on I/O memory already disabled
- Use the `writel()`/`readl()` macros, they do the right thing for your architecture
- The compiler and/or CPU can reorder memory accesses, which might cause trouble for your devices is they expect one register to be read/written before another one.
  - Memory barriers are available to prevent this reordering
  - `rmb()` is a read memory barrier, prevents reads to cross the barrier
  - `wmb()` is a write memory barrier
  - `mb()` is a read-write memory barrier
  - Starts to be a problem with CPUs that reorder instructions and with SMP. See `Documentation/memory-barriers.txt` for details.
- Note that `readl()`, `writel()` and similar functions already contain barriers (safer), while the raw ones don't.

/dev/mem

- Used to provide user space applications with direct access to physical addresses.
- Usage: open `/dev/mem` and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- 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



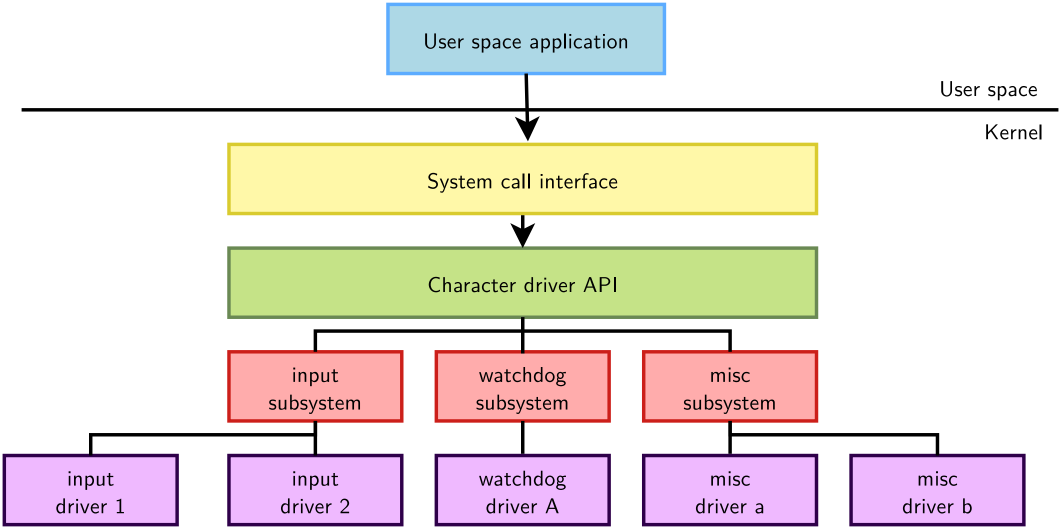
- Add UART devices to the board device tree
- Access I/O registers to control the device and send first characters to it.

The misc subsystem

Why a *misc* subsystem?

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

Misc subsystem diagram



Misc subsystem API (1/2)

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

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

Misc subsystem API (2/2)

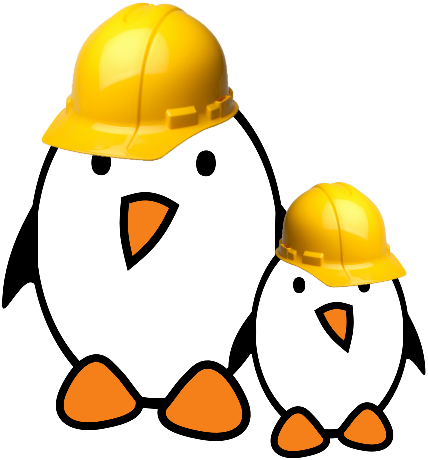
The main fields to be filled in `struct miscdevice` are:

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

User space API for misc devices

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

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

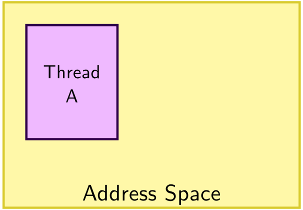
Processes and scheduling

Process, thread?

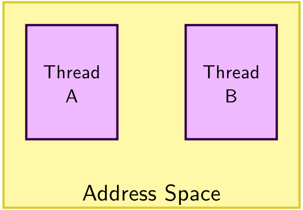
- Confusion about the terms *process*, *thread* and *task*
- In UNIX, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - A single thread, which is the only entity known by the scheduler.
- Additional threads 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()

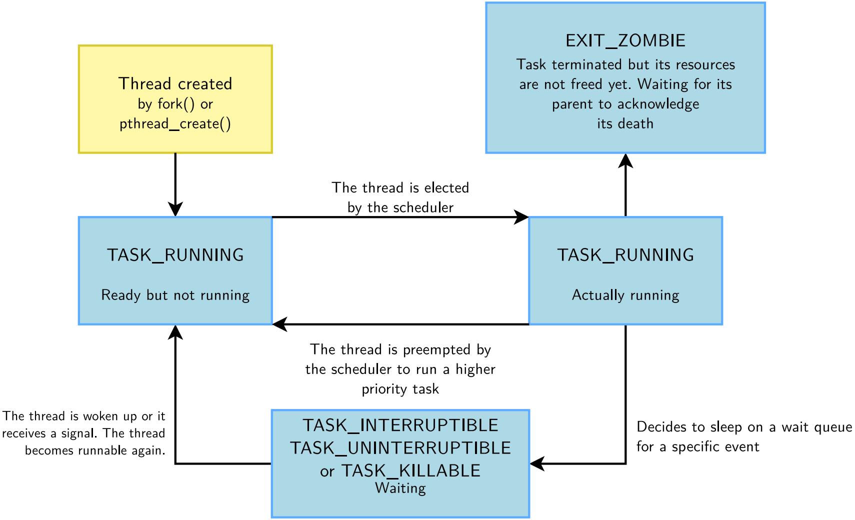


Same process after pthread_create()

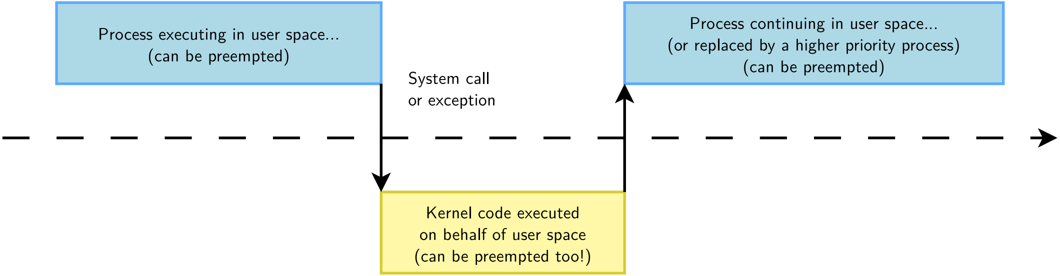
Relation between execution mode, address space and context

- 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 were 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



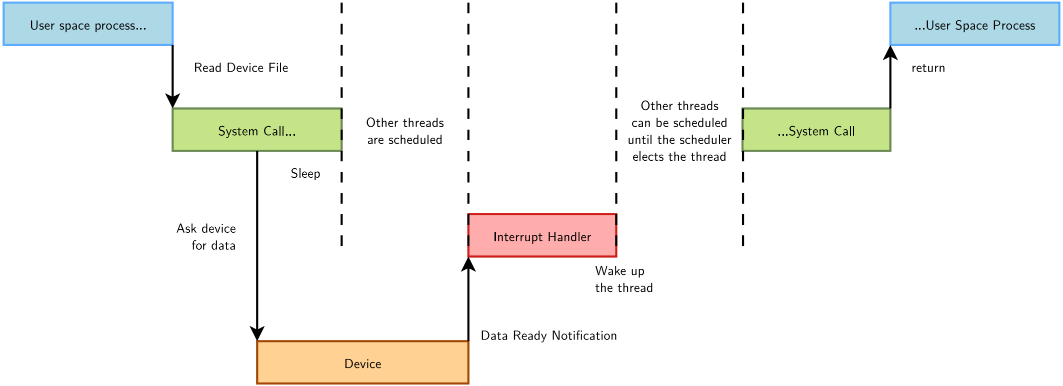
Execution of system calls



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 char/tpm/tpm_ibmvtpm.c

Waking up!

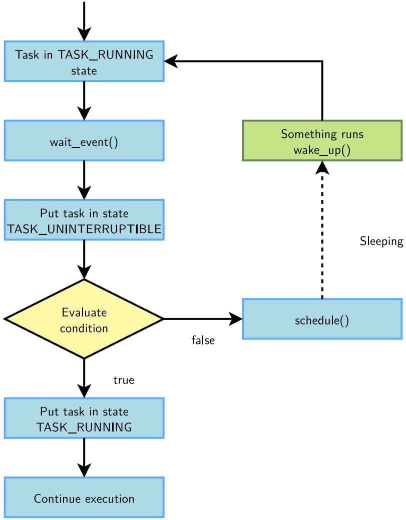
Typically done by interrupt handlers when data sleeping processes are waiting for become available.

- `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 no 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_s`: Timeout delay after which the operation fails and returns -ETIMEDOUT

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);
```

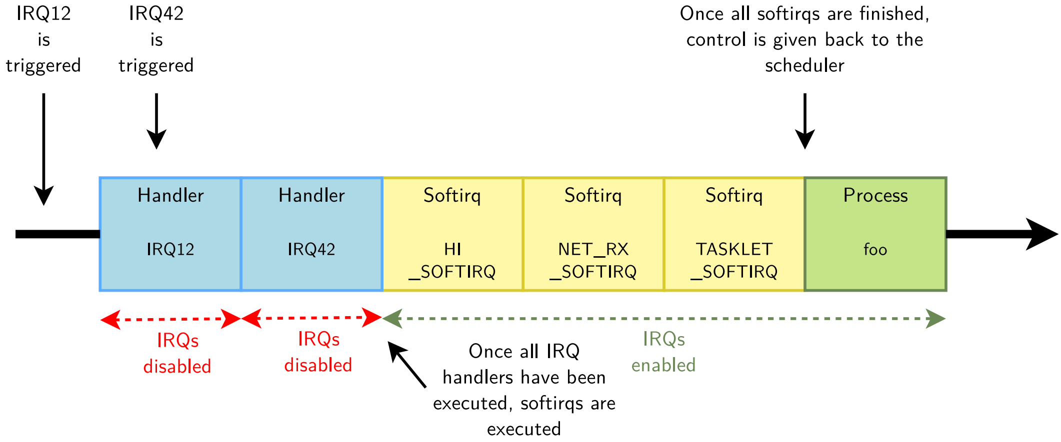
- `handler`, "hard IRQ" handler
- `thread_fn`, executed in a thread

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

- 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 complete kernel subsystems (network, etc.)
- The list of softirqs is defined in `include/linux/interrupt.h`: `HI_SOFTIRQ`, `TIMER_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ`, `BLOCK_SOFTIRQ`, `IRQ_POLL_SOFTIRQ`, `TASKLET_SOFTIRQ`, `SCHED_SOFTIRQ`, `HRTIMER_SOFTIRQ`, `RCU_SOFTIRQ`
- `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` are used to execute tasklets

Example usage of softirqs - NAPI

NAPI = *New API*

- Interface in the Linux kernel used for interrupt mitigation in network drivers
- Principle: when the network traffic exceeds a given threshhold ("budget"), disable network interrupts and consume incoming packets through a polling function, instead of processing each new packet with an interrupt.
- This reduces overhead due to interrupts and yields better network throughput.
- The polling function is run by `napi_schedule()`, which uses `NET_RX_SOFTIRQ`.
- See `https://en.wikipedia.org/wiki/New_API` for details
- See also our commented network driver on `https://frama.link/qCaWu1-U`

Tasklets

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

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

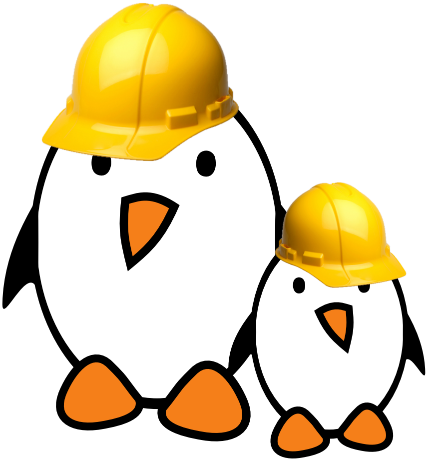
- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts. It can typically be used for background work which can be scheduled.
- The function registered as workqueue is executed in a thread, which means:
  - All interrupts are enabled
  - Sleeping is allowed
- A workqueue, usually allocated in a per-device structure, is registered with `INIT_WORK()` and typically triggered with `queue_work()`
- The complete API, in `include/linux/workqueue.h`, provides many other possibilities (creating its own workqueue threads, etc.)
- 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



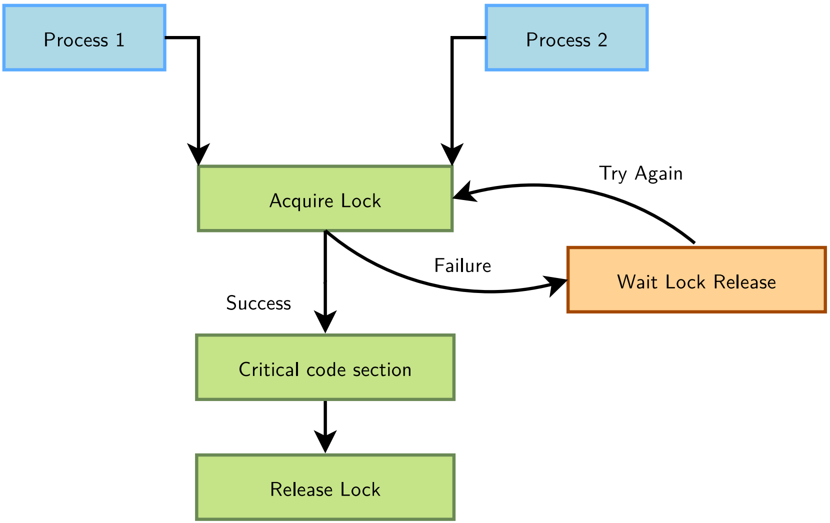
- Adding read capability to the character driver developed earlier.
- Register an interrupt handler for each device.
- Waiting for data to be available in the read file operation.
- Waking up the code when data are available from the devices.

Concurrent Access to Resources: Locking

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



Linux mutexes

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

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

Locking and unlocking mutexes 1/2

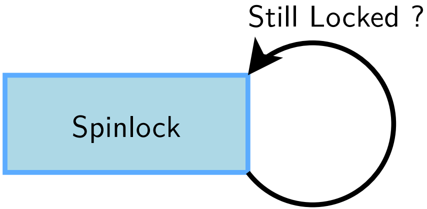
- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: can't be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a 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.



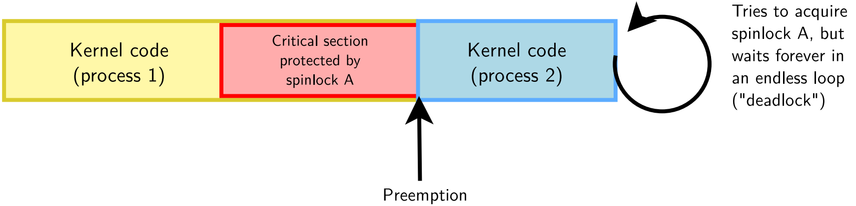
Initializing spinlocks

- 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

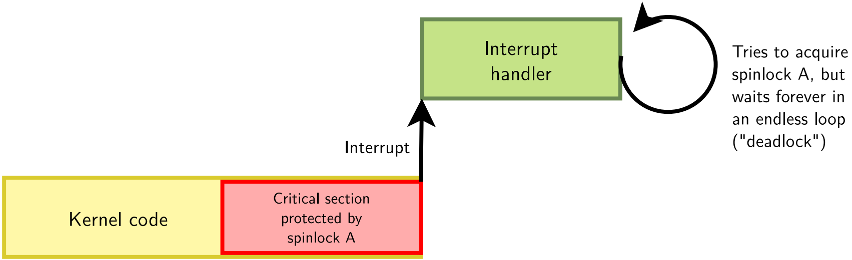
Several variants, depending on where the spinlock is called:

- `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).
    - Kernel preemption on the local CPU is disabled. We need to avoid deadlocks because of preemption from processes that want to get the same lock:



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 */
};
```

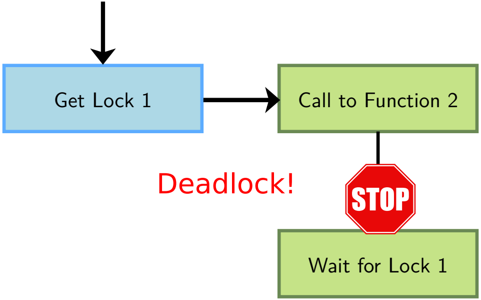
- Spinlock taken/released with protection against interrupts

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

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

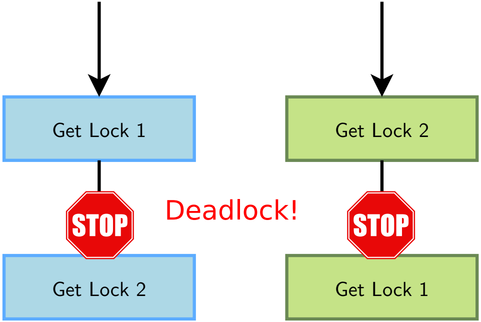
More deadlock situations

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

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



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



Debugging locking and concurrency issues

- 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
- Kernel Concurrency SANitizer framework
  - CONFIG_KCSAN, introduced in Linux 5.8.
  - Can find concurrency issues in your system.
  - See 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).
- When available, use atomic operations.

Atomic variables 1/2

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Atomic operations definitions
  - `#include <asm/atomic.h>`
- `atomic_t`
  - Contains a signed integer (at least 24 bits)
- Atomic operations (main ones)
  - Set or read the counter:
    - `void atomic_set(atomic_t *v, int i);`
    - `int atomic_read(atomic_t *v);`
  - Operations without return value:
    - `void atomic_inc(atomic_t *v);`
    - `void atomic_dec(atomic_t *v);`
    - `void atomic_add(int i, atomic_t *v);`
    - `void atomic_sub(int i, atomic_t *v);`

Atomic variables 2/2

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

Atomic bit operations

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long *` type.
- Apply to a `void *` type on a few others.
- 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.

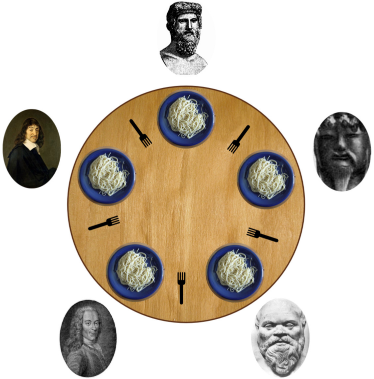
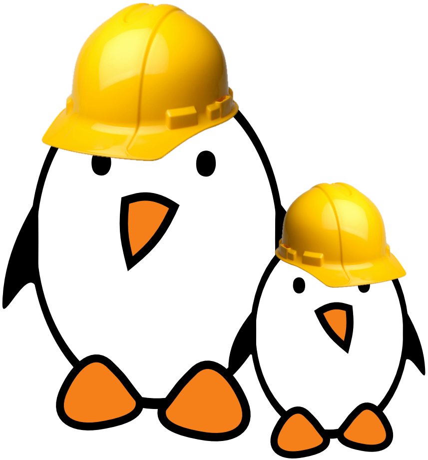


Image source: Wikipedia (https://frama.link/xg3Wnd0F)

Practical lab - Locking



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

Kernel debugging

Debugging using messages (1)

Three APIs are available

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

Debugging using messages (2)

- 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
    ```

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`

DebugFS

A virtual filesystem to export debugging information to user space.

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

Deprecated debugging mechanisms

Some additional debugging mechanisms, whose usage is now considered deprecated

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

Using Magic SysRq

Functionnality provided by serial drivers

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

kgdb - A kernel debugger

- 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

Using kgdb 1/2

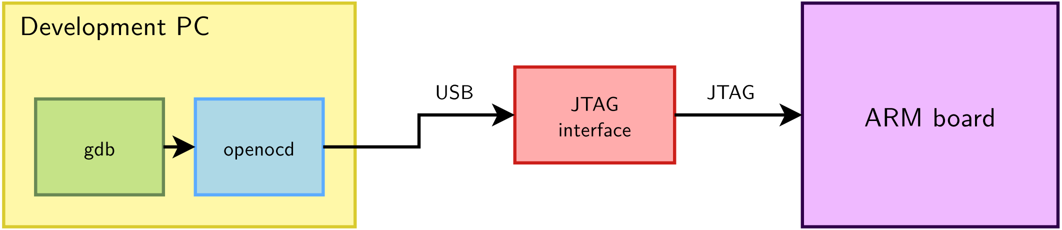
- Details available in the kernel documentation: dev-tools/kgdb
- Recommended to turn on CONFIG_FRAME_POINTER to aid in producing more reliable stack backtraces in gdb.
- You must include a kgdb I/O driver. One of them is kgdb over serial console (kgdboc: kgdb over console, enabled by CONFIG_KGDB_SERIAL_CONSOLE)
- Configure kgdboc at boot time by passing to the kernel:
  - kgdboc=<tty-device>,<bauds>.
  - For example: kgdboc=ttyS0,115200

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.

Two types of JTAG dongles

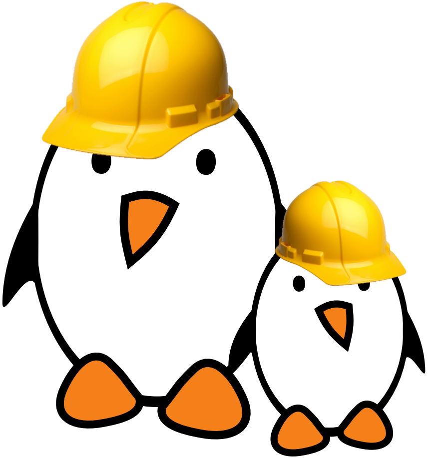
- The ones offering a gdb compatible interface, over a serial port or an Ethernet connection. gdb can directly connect to them.
- The ones not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger): http://openocd.sourceforge.net/
  - OpenOCD is the bridge between 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)



More kernel debugging tips

- Make sure CONFIG_KALLSYMS_ALL is enabled
  - Is turned on by default
  - To get oops messages with symbol names instead of raw addresses
- On ARM, if your kernel doesn't boot or hangs without any message, you can activate early debugging options (CONFIG_DEBUG_LL and CONFIG_EARLYPRINTK), and add earlyprintk to the kernel command line.

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.

Porting the Linux kernel to an ARM board

embedded Linux and kernel engineering

Porting the Linux kernel

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

Architecture, CPU and Machine

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

Before the Device Tree and ARM cleanup

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

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

The Device Tree and the ARM cleanup

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

Adding the support for a new ARM board

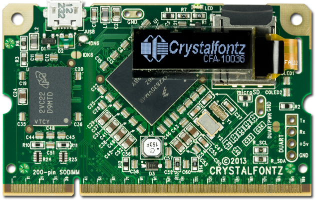
Provided the SoC used on your board is supported by the Linux kernel:

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

Studying the Crystalfontz CFA-10036 platform

After using a platform based on the AM335x processor from Texas Instruments, let's study another platform Bootlin has worked on specifically.

- Crystalfontz CFA-10036
- Uses the Freescale iMX28 SoC, from the MXS family.
- 128MB of RAM
- 1 serial port, 1 LED
- 1 I2C bus, equipped with an OLED display
- 1 SD-Card slot



Disclaimer: while the way of describing a board has slightly evolved over the past years, the official Crystalfontz support has not. As our incentive is to show up-to-date code and share best practices, the next snippets of code may diverge a little compared to the upstream files.

Crystalfontz CFA-10036 Device Tree, header

- SPDX license tag
- Mandatory Device Tree language definition

```
/dts-v1/;
```

- Include the `.dtsi` file describing the SoC

```
#include "imx28.dtsi"
```

- Start the root of the tree (named `/`) then describe the board
  - A human-readable string to describe the machine (shown at boot time)

```
model = "Crystalfontz CFA-10036 Board";
```

  - A list of *compatible* strings, from the most specific one to the most general one. Mandatory to execute the right SoC specific initializations and board specific code.

```
compatible = "crystalfontz,cfa10036", "fsl,imx28";
```

Crystalfontz CFA-10036, backbone

- Definition of the buses and peripherals

```
/ {
    /* Define here 'standalone' peripherals and internal buses */
    memory {
        device_type = "memory";
        reg = <0x40000000 0x8000000>; /* 128 MB */
    };
    apb@80000000 {
        apbh@80000000 {
            /* Define apbh peripherals here */
            apbx@80040000 {
                /* Define apbx peripherals here */
            };
        };
    };
};
/* Reference here existing nodes with their labels */
```

Crystalfontz CFA-10036 Device Tree, enable already described devices

- The CFA-10036 has one debug UART. It is described in the iMX28 DTSI file, so the corresponding controller should be referenced in the board DTS and enabled:

```
&duart {
    pinctrl-names = "default";
    pinctrl-0 = <&duart_pins_b>;
    status = "okay";
};
```

- It also features an USB port which is described in the SoC DTSI but needs to be enabled:

```
&usb0 {
    pinctrl-names = "default";
    pinctrl-0 = <&usb0_otg_cfa10036>;
    status = "okay";
};
```

Crystalfontz CFA-10036 Device Tree, fully describe additional devices

- The I2C bus with a Solomon SSD1306 OLED display connected on it must be described entirely at the location where it belongs:

```
apbc@80040000 {
    i2c0: i2c@18000 { /* This means physical offset 0x80058000 */
        reg = <0x18000 0x1000>;
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins_b>;
        status = "okay";
        clock-frequency = <400000>;

        ssd1306: oled@3c {
            compatible = "solomon,ssd1306fb-i2c";
            pinctrl-names = "default";
            pinctrl-0 = <&ssd1306_cfa10036>;
            reg = <0x3c>;
            reset-gpios = <&gpio2 7 0>;
            solomon,height = <32>;
            solomon,width = <128>;
            solomon,page-offset = <0>;
        };
    };
```

- Mind the display's pin configuration that has not yet been described

Crystalfontz CFA-10036 Device Tree, LEDs

- One LED is connected to this platform, let's describe it as well

```
/ {
    leds {
        compatible = "gpio-leds";
        pinctrl-names = "default";
        pinctrl-0 = <&led_pins_cfa10036>;

        power {
            gpios = <&gpio3 4 1>;
            default-state = "on";
        };
    };
```

- Also mind the pin configuration that we can define at any place

Crystalfontz CFA-10036 Device Tree, muxing

- Definition of a few pins that will be muxed as GPIO, for LEDs and reset.

```
&pinctrl {
    ssd1306_cfa10036: ssd1306-10036@0 {
        reg = <0>;
        fsl,pinmux-ids = <0x2073>; /* MX28_PAD_SSP0_D7__GPIO_2_7 */
        fsl,drive-strength = <0>;
        fsl,voltage = <1>;
        fsl,pull-up = <0>;
    };

    led_pins_cfa10036: leds-10036@0 {
        reg = <0>;
        fsl,pinmux-ids = <0x3043>; /* MX28_PAD_AUART1_RX__GPIO_3_4 */
        fsl,drive-strength = <0>;
        fsl,voltage = <1>;
        fsl,pull-up = <0>;
    };
};
```

Crystalfontz CFA-10036 Device Tree, Breakout Boards

- The CFA-10036 can be plugged in other breakout boards, and the device tree also allows us to describe this, using includes. For example, the CFA-10057:

```
#include "imx28-cfa10036.dts"
```

- This allows to have a layered description. This can also be done for boards that have a lot in common, like the BeagleBone and the BeagleBone Black, or the AT91 SAMA5D3-based boards.

Crystalfontz CFA-10036: build the DTB

- To ensure that the Device Tree Blob gets built for this board Device Tree Source, one need to ensure it is listed in arch/arm/boot/dts/Makefile:

```
dtb-$(CONFIG_ARCH_MXS) +=
        imx28-cfa10036.dtb \
        imx28-cfa10037.dtb \
        imx28-cfa10049.dtb \
        imx28-cfa10055.dtb \
        imx28-cfa10056.dtb \
        imx28-cfa10057.dtb \
        imx28-cfa10058.dtb \
        imx28-evk.dtb
```

Understanding the SoC support

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

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

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

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

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

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

Components of the minimal SoC support

The minimal SoC support consists of

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

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

Extending the minimal SoC support

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

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

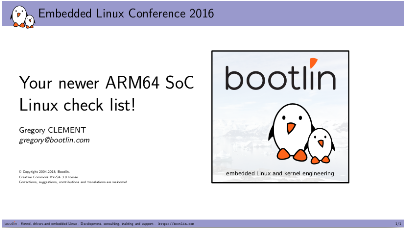
Adding controller drivers

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

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

Porting the Linux kernel: further reading

- Gregory Clement, Your newer ARM64 SoC Linux support check-list!
  https://bit.ly/2r8lHnE
- Thomas Petazzoni, Your new ARM SoC Linux support check-list!
  https://bit.ly/2ivqtDD
- Our technical presentations on various kernel subsystems:
  https://bootlin.com/docs/



Power Management

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)

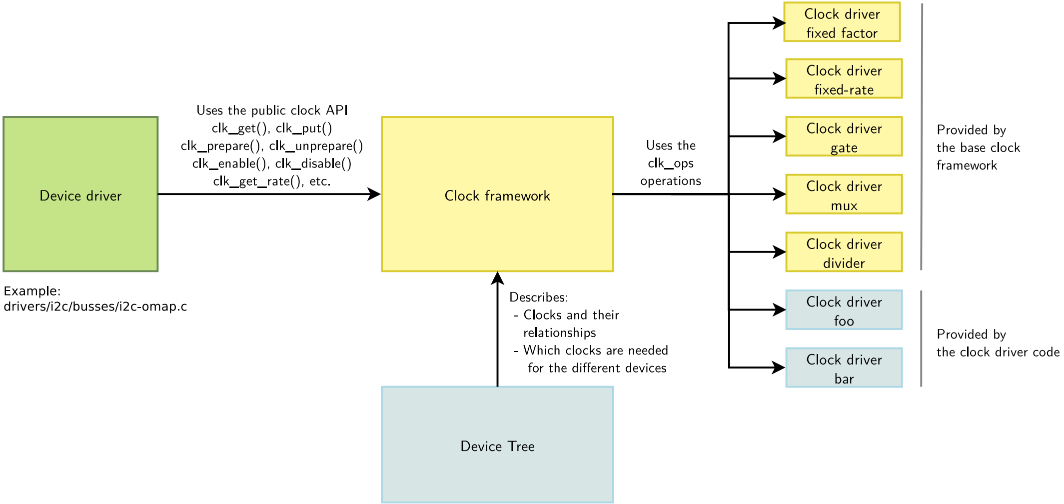
- Generic framework to manage clocks used by devices in the system
- Allows to reference count clock users and to shutdown the unused clocks to save power
- Simple API described in `include/linux/clk.h`.
  - `clk_get()` to lookup and obtain a reference to a clock producer
  - `clk_enable()` to inform 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

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

Managing idle

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.

PowerTOP

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`
- Driver example: `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

Regulator framework

- 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=juJJZORgVwI

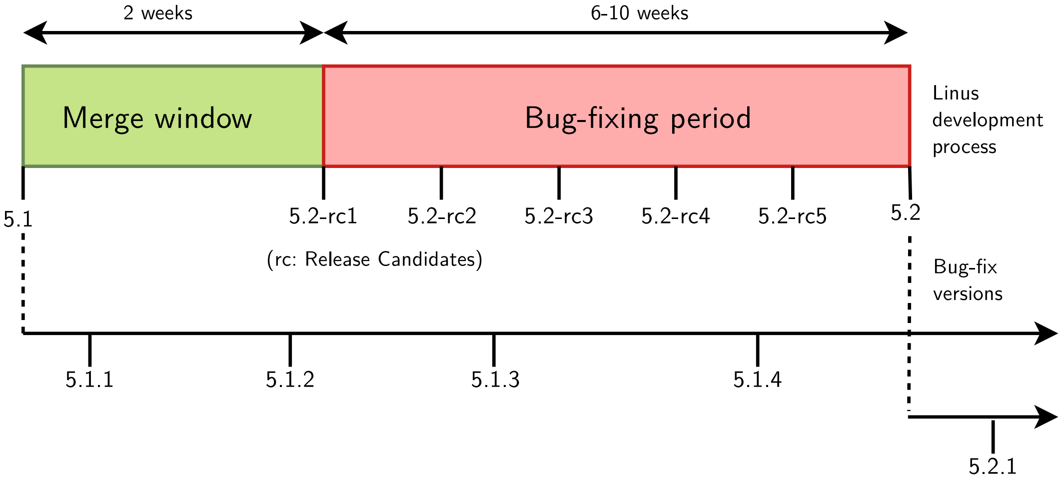
The kernel development and contribution process

Linux versioning scheme and development process

Linux versioning scheme

- Until 2003, there was a new stable release branch of Linux every 2 or 3 years (2.0, 2.2, 2.4). New development branches took 2-3 years to become stable (too slow!).
- Since 2003, there is a new stable release of Linux about every 10 weeks:
  - Versions `2.6` (Dec. 2003) to `2.6.39` (May 2011)
  - Versions `3.0` (Jul. 2011) to `3.19` (Feb. 2015)
  - Versions `4.0` (Apr. 2015) to `4.20` (Dec. 2018)
  - Version `5.0` was released in Mar. 2019.
- Features are added to the kernel in a progressive way. Since 2003, kernel developers have managed to do so without having to introduce a massively incompatible development branch.
- For each release, there are bugfix and security updates: 5.0.1, 5.0.2, etc.

Using merge and bug fixing windows



Need for long term support (1)

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

Version	Maintainer	Released	Projected EOL
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2025
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2024
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2024
4.9	Greg Kroah-Hartman & Sasha Levin	2016-12-11	Jan, 2023
4.4	Greg Kroah-Hartman & Sasha Levin	2016-01-10	Feb, 2022

Captured on `https://kernel.org` in Feb. 2021, following the *Releases* link.

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

Need for long term support (2)

- You could also get long term support from a commercial embedded Linux provider.
  - Wind River Linux can be supported for up to 15 years.
  - Ubuntu Core can be supported for up to 10 years.
- *"If you are not using a supported distribution kernel, or a stable / longterm kernel, you have an insecure kernel"* - Greg KH, 2019
  Some vulnerabilities are fixed in stable without ever getting a CVE.
- The *Civil Infrastructure Platform* project is an industry / Linux Foundation effort to support selected LTS versions (starting with 4.4) much longer ($>$ 10 years). See `https://bit.ly/2hy1QYC`.

What's new in each Linux release? (1)

The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
Date:   Wed Jul 13 11:29:17 2011 +0200

    at91: at91-ohci: support overcurrent notification

    Several USB power switches (AIC1526 or MIC2026) have a digital output
    that is used to notify that an overcurrent situation is taking
    place. This digital outputs are typically connected to GPIO inputs of
    the processor and can be used to be notified of these overcurrent
    situations.

    Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data
    structure so that boards can tell the AT91 OHCI driver which pins are
    used for the overcurrent notification, and an overcurrent_supported
    boolean to tell the driver whether overcurrent is supported or not.

    The code has been largely borrowed from ohci-da8xx.c and
    ohci-s3c2410.c.

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
    Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>
```

Very difficult to find out the key changes and to get the global picture out of individual changes.

What's new in each Linux release? (2)

Fortunately, there are some useful resources available

- https://kernelnewbies.org/LinuxChanges
  In depth coverage of the new features in each kernel release
- https://lwn.net/Kernel
  Coverage of the features accepted in each merge window

January 18, 2021	Resource limits in user namespaces
January 15, 2021	Fast commits for ext4
January 14, 2021	MAINTAINERS truth and fiction
January 11, 2021	Old compilers and old bugs
January 7, 2021	Restricted DMA
January 5, 2021	Portable and reproducible kernel builds with TuxMake
December 28, 2020	5.11 Merge window, part 2
December 18, 2020	5.11 Merge window, part 1

Contributing to the Linux kernel

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.

How to Become a Kernel Developer?

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

Contribute to the Linux Kernel (1)

- Clone Linus Torvalds' tree:
  - `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- Keep your tree up to date
  - `git pull`
- Look at the master branch and check whether your issue / change hasn't been solved / implemented yet. Also check the maintainer's git tree and mailing list (see the `MAINTAINERS` file).You may miss submissions that are not in mainline yet.
- If the maintainer has its own git tree, create a remote branch tracking this tree. This is much better than creating another clone (doesn't duplicate common stuff):
  - `git remote add linux-omap git://git.kernel.org/pub/scm/linux/kernel/git/tmlind/linux-omap.git`
  - `git fetch linux-omap`

Contribute to the Linux Kernel (2)

- Either create a new branch starting from the current commit in the master branch:
  - `git checkout -b feature`
- Or, if more appropriate, create a new branch starting from the maintainer's master branch:
  - `git checkout -b feature linux-omap/master` (remote tree / remote branch)
- In your new branch, implement your changes.
- Test your changes (must at least compile them).
- Run `git add` to add any new files to the index.

Configure git send-email

- Make sure you already have configured your name and e-mail address (should be done before the first commit).
  - `git config --global user.name 'My Name'`
  - `git config --global user.email me@mydomain.net`
- Configure your SMTP settings. Example for a Google Mail account:
  - `git config --global sendemail.smtpserver smtp.googlemail.com`
  - `git config --global sendemail.smtpserverport 587`
  - `git config --global sendemail.smtpencryption tls`
  - `git config --global sendemail.smtpuser jdoe@gmail.com`
  - `git config --global sendemail.smtppass xxx`

Contribute to the Linux Kernel (3)

- Group your changes by sets of logical changes, corresponding to the set of patches that you wish to submit.
- Commit and sign these groups of changes (signing required by Linux developers).
  - `git commit -s`
  - Make sure your first description line is a useful summary and starts with the name of the modified subsystem. This first description line will appear in your e-mails
- The easiest way is to look at previous commit summaries on the main file you modify
  - `git log --pretty=oneline <path-to-file>`
- Examples subject lines (`[PATCH]` omitted):

```
Documentation: prctl/seccomp_filter
PCI: release busn when removing bus
ARM: add support for xz kernel decompression
```

Contribute to the Linux Kernel (4)

- Remove previously generated patches
  - `rm 00*.patch`
- Have git generate patches corresponding to your branch (assuming it is the current branch)
  - If your branch is based on mainline
    - `git format-patch master`
  - If your branch is based on a remote branch
    - `git format-patch <remote>/<branch>`
- Make sure your patches pass `checkpatch.pl` checks:
  - `scripts/checkpatch.pl --strict 00*.patch`
- Now, send your patches to yourself
  - `git send-email --compose --to me@mydomain.com 00*.patch`
- If you have just one patch, or a trivial patch, you can remove the empty line after `In-Reply-To:`. This way, you won't add a summary e-mail introducing your changes (recommended otherwise).

Contribute to the Linux Kernel (5)

- Check that you received your e-mail properly, and that it looks good.
- Now, find the maintainers for your patches

```
scripts/get_maintainer.pl ~/patches/00*.patch
Russell King <linux@arm.linux.org.uk> (maintainer:ARM PORT)
Nicolas Pitre <nicolas.pitre@linaro.org>
(commit_signer:1/1=100%)
linux-arm-kernel@lists.infradead.org (open list:ARM PORT)
linux-kernel@vger.kernel.org (open list)
```

- Now, send your patches to each of these people and lists
  - `git send-email --compose --to linux@arm.linux.org.uk --to nicolas.pitre@linaro.org --cc linux-arm-kernel@lists.infradead.org --cc linux-kernel@vger.kernel.org 00*.patch`
- Wait for replies about your changes, take the comments into account, and resubmit if needed, until your changes are eventually accepted.

Contribute to the Linux Kernel (6)

- If you use `git format-patch` to produce your patches, you will need to update your branch and may need to group your changes in a different way (one patch per commit).
- Here's what we recommend
  - Update your master branch
    - `git checkout master; git pull`
  - Back to your branch, implement the changes taking community feedback into account. Commit these changes.
  - Still in your branch: reorganize your commits and commit messages
    - `git rebase --interactive origin/master`
    - `git rebase` allows to rebase (replay) your changes starting from the latest commits in master. In interactive mode, it also allows you to merge, edit and even reorder commits, in an interactive way.
  - Third, generate the new patches with `git format-patch`.

Kernel Resources

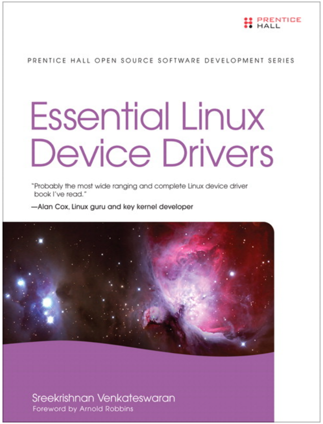
Linux Weekly News

- https://lwn.net/
- The weekly digest off all Linux and free software information sources
- In depth technical discussions about the kernel
- Subscribe to finance the editors ($7 / month)
- Articles available for non subscribers after 1 week.

Useful Reading (1)

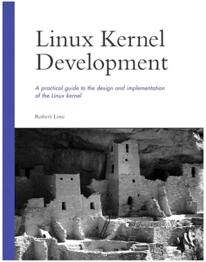
Essential Linux Device Drivers, April 2008

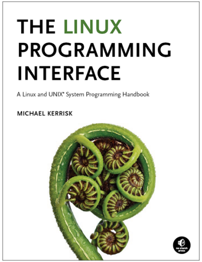
- https://elinuxdd.com/
- By Sreekrishnan Venkateswaran, an embedded IBM engineer with more than 10 years of experience
- Covers a wide range of topics not covered by LDD: serial drivers, input drivers, I2C, PCMCIA, PCI, USB, video drivers, audio drivers, block drivers, network drivers, Bluetooth, IrDA, MTD, drivers in user space, kernel debugging, etc.
- *Probably the most wide ranging and complete Linux device driver book I've read* – Alan Cox



Useful Reading (2)

- Linux Kernel Development, 3rd Edition, Jun 2010
  - Robert Love, Novell Press
  - https://rlove.org
  - A very synthetic and pleasant way to learn about kernel subsystems (beyond the needs of device driver writers)
- The Linux Programming Interface, Oct 2010
  - Michael Kerrisk, No Starch Press
  - https://man7.org/tlpi/
  - A gold mine about the kernel interface and how to use it



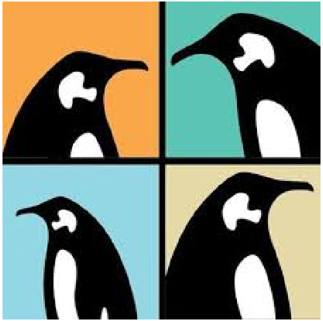


Useful Online Resources

- 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
- Kernel Newbies
  - https://kernelnewbies.org/
  - Glossary, articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.
- Kernel glossary
  - https://kernelnewbies.org/KernelGlossary

International Conferences (1)

- Embedded Linux Conference:
  - https://embeddedlinuxconference.com/
  - Organized by the Linux Foundation every year in North America and in Europe
  - Very interesting kernel and user space topics for embedded systems developers. Many kernel and embedded project maintainers are present.
  - Presentation slides and videos freely available on https://elinux.org/ELC_Presentations
- Linux Plumbers: https://linuxplumbersconf.org
  - About the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc. Not really a conventional conference with formal presentations, but rather a place where contributors on each topic meet, share their progress and make plans for work ahead.



International Conferences (2)

- 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.

Continue to learn after the course

Here are a few suggestions:

- 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.
- Help with tasks keeping the kernel code clean and up-to-date: https://kernelnewbies.org/KernelJanitors/Todo
- Propose fixes for issues reported by the *Coccinelle* tool: `make coccicheck`
- Participate to improving drivers in `drivers/staging/`
- Investigate and do the triage of issues reported by Coverity Scan: https://scan.coverity.com/projects/linux
- Learn by reading the kernel code by yourself, ask questions and propose improvements.
- Implement and share drivers for your own hardware, of course!

Last slides

Thank you!

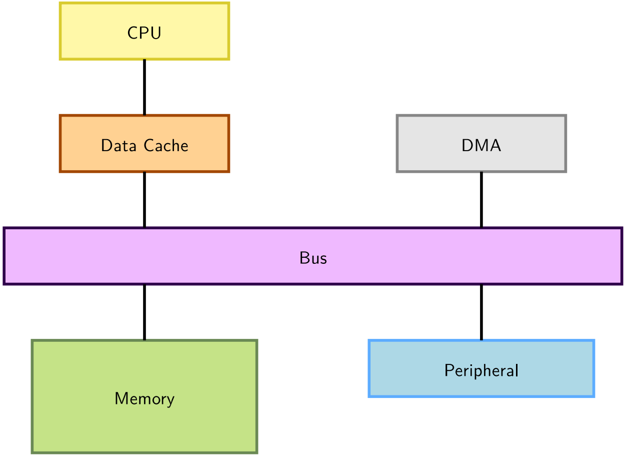
And may the Source be with you

Backup slides

DMA

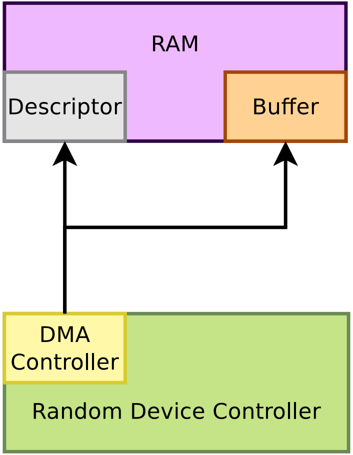
DMA integration

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



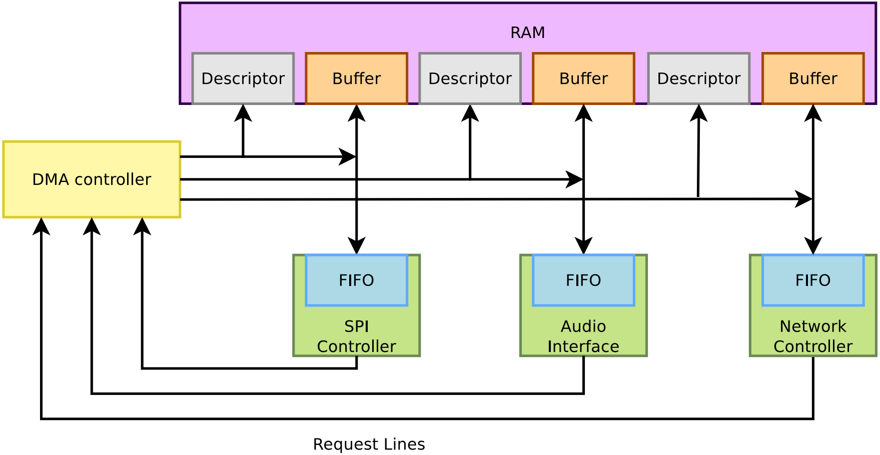
Peripheral DMA

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



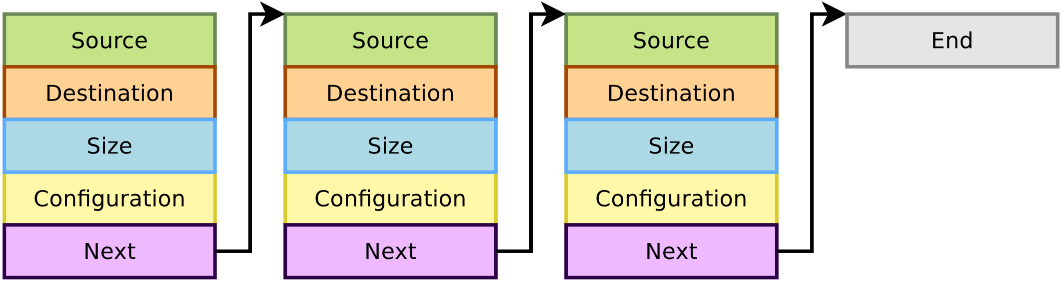
DMA controllers

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.



DMA usage

Constraints with a DMA

- A DMA deals with physical addresses, so:
  - Programming a DMA requires retrieving a physical address at some point (virtual addresses are usually used)
  - The memory accessed by the DMA shall be physically contiguous
- 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)
  - Either clean (write to memory) or invalidate the cache lines corresponding to the buffer accessed by DMA and processor at the right times

DMA memory constraints

- Need to use contiguous memory in physical space.
- Can use any memory allocated by `kmalloc()` (up to 128 KB) or `__get_free_pages()` (up to 8MB).
- Can use block I/O and networking buffers, designed to support DMA.
- Can not use `vmalloc()` memory (would have to setup DMA on each individual physical page).

Memory synchronization issues

Memory caching could interfere with DMA

- Before DMA to device
  - Need to make sure that all writes to the DMA buffer are done (corresponding cache lines cleaned)
- After DMA from device
  - Before drivers read from a DMA buffer, need to make sure that the corresponding cache lines are invalidated.
- Bidirectional DMA
  - Need to do both of the above operations

The kernel DMA utilities can take care of:

- Either allocating a buffer in a cache coherent area,
- Or making sure caches are handled when required,
- Managing the DMA mappings and IOMMU (if any).
- See `core-api/dma-api` for details about DMA and the Linux DMA generic API.
- Most subsystems (such as PCI or USB) supply their own DMA API, derived from the generic one. May be sufficient for most needs.

Coherent or streaming DMA mappings

- 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.
- Streaming mappings
  - The kernel just sets the mapping for a buffer provided by the driver.
  - Use an already allocated buffer
  - Mapping set up for each transfer. Keeps DMA registers free on the hardware.

Allocating coherent mappings

The kernel takes care of both buffer allocation and mapping

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

void *                          /* Output: buffer address */
    dma_alloc_coherent(
         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()`)

Setting up streaming mappings

Works on already allocated buffers

```
#include <linux/dmapool.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 handdle,
    size_t size, enum dma_data_direction dir);
```

DMA notes

Streaming mapping notes:

- When the mapping is active: only the device should access the buffer (potential cache issues otherwise).
- The CPU can access the buffer only after unmapping!
- Another reason: if required, this API can create an intermediate bounce buffer (used if the given buffer is not usable for DMA).
- The Linux API also supports scatter / gather DMA streaming mappings.

Commented network driver example:

- See `https://bootlin.com/pub/drivers/r6040-network-driver-with-comments.c` for a commmented network driver, which both streaming and coherent mappings.

DMA transfers

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
  - Ask the hardware to use DMA, so that it will drive its request line
  - Use Linux DMAEngine framework, especially its slave API

DMAEngine slave API 1/2

In order to start a DMA transfer with DMAEngine, you need to call the following functions from your driver

1. Request a channel for exclusive use with `dma_request_channel()`, or one of its variants
2. Configure it for our use case, by filling a `struct dma_slave_config` structure with various parameters (source and destination adresses, accesses width, etc.) and passing it as an argument to `dmaengine_slave_config()`
3. Start a new transaction with `dmaengine_prep_slave_single()` or `dmaengine_prep_slave_sg()`
4. Put the transaction in the driver pending queue using `dmaengine_submit()`
5. And finally ask the driver to process all pending transactions using `dma_async_issue_pending()`

DMAEngine slave API 2/2

- Of course, all this needs to be done in addition to the DMA mapping seen previously
- Some frameworks abstract it away, such as *SPI* and *ASoC*
- Example usage of the slave API: look at the code for `stm32_i2c_prep_dma_xfer()`.

Details in kernel documentation: `driver-api/dmaengine/client`

mmap

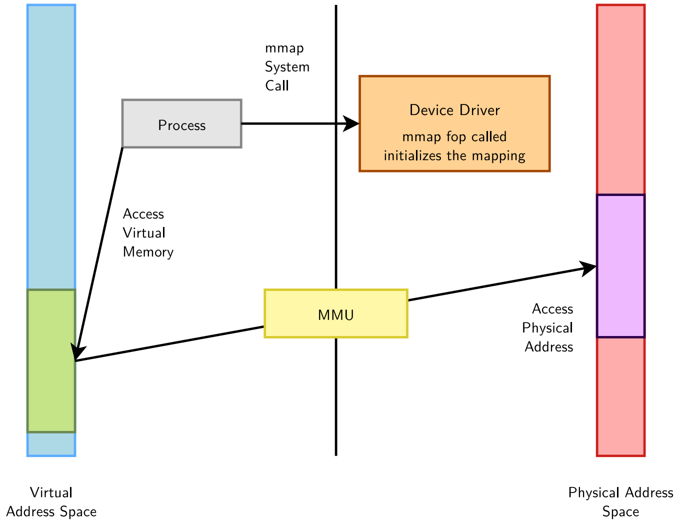
mmap

- 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>`

/proc/<pid>/maps

```
      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



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

```c
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;
    }
```

devmem2

- 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.

mmap summary

- 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