



# Linux Kernel and Driver Development Training

## Linux Kernel and Driver Development Training

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.  
Latest update: February 6, 2019.

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

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





# Rights to copy

© Copyright 2004-2019, Bootlin

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

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

You are free:

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

Under the following conditions:

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

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

**Document sources:** <https://git.bootlin.com/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](#)



- ▶ Engineering company created in 2004,  
named "Free Electrons" until February 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.
- ▶ Activities: development, training, consulting, technical support.
- ▶ Added value: get the best of the user and development community and the resources it offers.



## Bootlin on-line resources

- ▶ All our training materials and technical presentations:  
<https://bootlin.com/docs/>
- ▶ Technical blog:  
<https://bootlin.com/blog/>
- ▶ News and discussions (LinkedIn):  
<https://www.linkedin.com/groups/4501089>
- ▶ Quick news (Twitter):  
<https://twitter.com/bootlincom>
- ▶ Elixir - browse Linux kernel sources on-line:  
<https://elixir.bootlin.com>



## Generic course information

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!

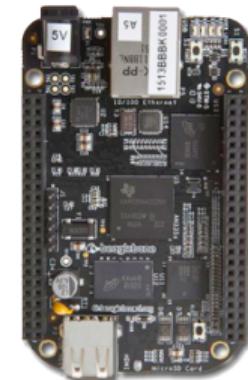




# Hardware used in this training session

## BeagleBone Black, from CircuitCo

- ▶ 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
- ▶ Ethernet, USB host and USB device, microSD, micro HDMI
- ▶ 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- ▶ A huge number of expansion boards, called *capes*. See [https://elinux.org/Beagleboard:BeagleBone\\_Capes](https://elinux.org/Beagleboard:BeagleBone_Capes).





# Shopping list: hardware for this course

- ▶ BeagleBone Black - Multiple distributors:  
See <http://beagleboard.org/Products/>. We also support the BeagleBone Black Wireless.
- ▶ Nintendo Nunchuk with UEXT connector:  
Olimex: <http://j.mp/1dTYLfs>
- ▶ Breadboard jumper wires - Male ends:  
Olimex: <http://bit.ly/2pSiIPs>
- ▶ USB Serial Cable - Male ends:  
Olimex: <http://j.mp/1eUuY2K>
- ▶ USB Serial Cable - Female ends:  
Olimex: <http://j.mp/18Hk8yF>
- ▶ 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 particularly 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

## Lab commands

Cross-compiling kernel:  
export ARCH=arm  
export CROSS\_COMPILE=arm-linux-  
make sama5\_defconfig

Booting kernel through tftp:  
setenv bootargs console=ttyS0 root=/dev/nfs  
setenv bootcmd tftp 0x21000000 zImage; tftp  
0x22000000 dtb; bootz 0x21000000 - 0x2200...

Making ubifs images:  
mkfs.ubifs -d rootfs -o root.ubifs -e 124KiB  
-m 2048 -c 1024

Encountered issues:  
Restart NFS server after editing /etc/exports!



# Cooperate!

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

- ▶ If you complete your labs before other people, don't hesitate to help other people 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](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](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

For more information see the [man page](#) or the [online help](#).

This is a very brief introduction to the most common vi commands.

**Entering command mode**

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

**Moving the cursor**

h for left arrow key move the cursor left.  
l for right arrow key move the cursor right.  
k for up arrow key move the cursor up.  
j for down arrow key move the cursor down.  
gg[lf] move the cursor one page forward.  
gG[bf] 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.  
o go to the start of the file.  
n go to line number n.  
[ctrl] n display the name of the current file and the cursor position in it.

**Entering editing mode**

i insert character under the cursor.  
a append one line after the cursor.  
o start to add a new line before the current one.  
c start to edit one line before the current one.

**Replacing characters, lines and words**

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

**COPYING and PASTING**

y copy the current line to the copy/paste buffer.  
yy copy the entire buffer after the current line.  
p Paste the copy/paste buffer before the current line.

**Deleting characters, words and lines**

dd delete the current line.  
d delete the character at the cursor location.  
dw delete the current word.

**Repeating commands**

^ repeat the last insertion, replacement or delete command.

**Looking for strings**

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

**Replacing strings**

s can also be done manually, search and replace once, and then using :%s/old/new/g to search and replace all.  
e.g. g/a/b/g between line numbers n and p, substitute all 'b' global occurrences of 'a' to 'b'.  
i, i<file>/<file> in the whole file (it will read the whole file)  
y, y<file>/<file> in the whole file (it will read the whole file)

**Applying a command several times - Examples**

15 move the cursor 15 lines down.  
15d delete 15 lines.  
15g move 15 lines from the cursor.  
15e go to the first line in the file.

**Misc**

[ctrl] l reduce the screen.

**Exiting and saving**

zz save current file and exit.  
x write current buffer to the current file.  
or zZ write [some] buffer to the file file.  
q! quit vi without saving changes.

**Going further**

vi is a very flexible and easy to learn editor for power users!  
It can make you extremely productive in the command line.  
Learn more by taking the quick tutorials 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

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## 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.

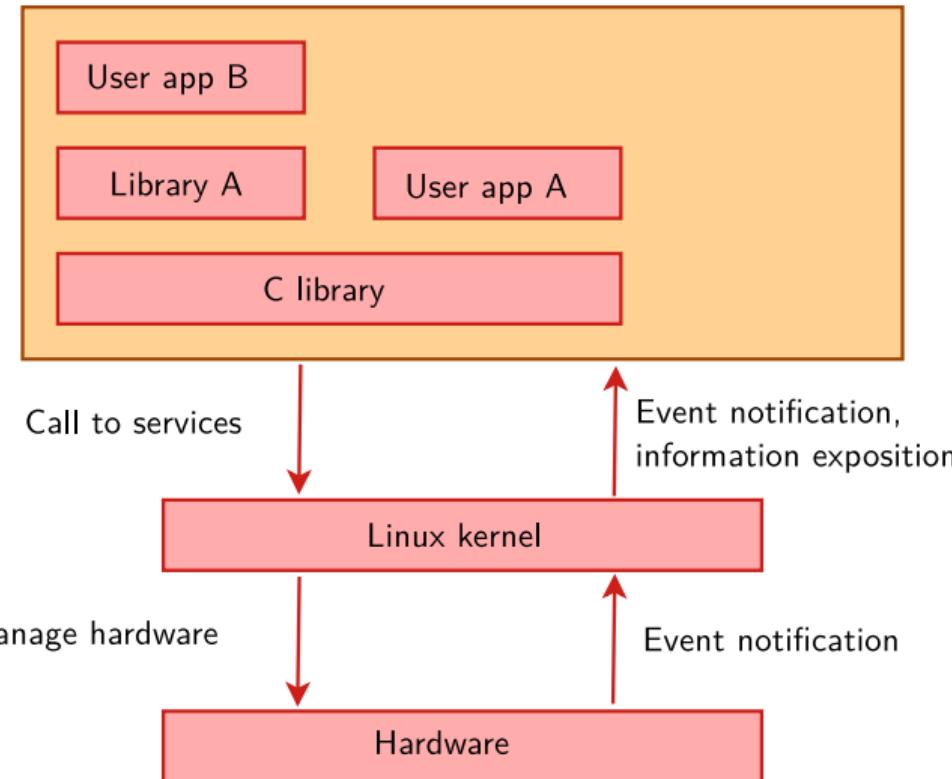


# Linux kernel key features

- ▶ Portability and hardware support.  
Runs on most architectures.
- ▶ 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 to “multiplex” 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

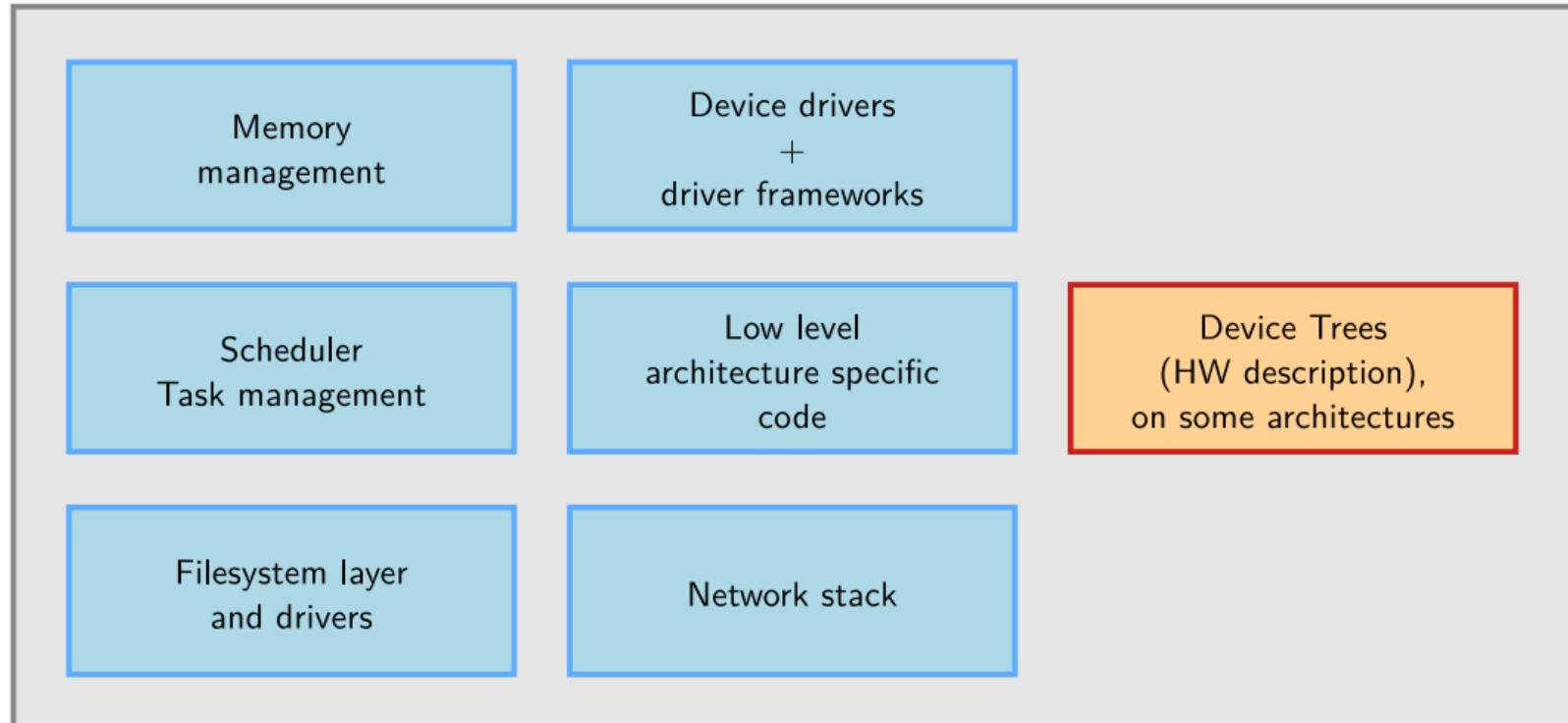


# Pseudo filesystems

- ▶ 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 set of devices and buses. Information about these devices.



## Linux Kernel





# Supported hardware architectures

- ▶ See the arch/ directory in the kernel sources
- ▶ Minimum: 32 bit processors, with or without MMU, and gcc support
- ▶ 32 bit architectures (arch/ subdirectories)  
Examples: arm, arc, c6x, m68k, microblaze
- ▶ 64 bit architectures:  
Examples: alpha, arm64, ia64...
- ▶ 32/64 bit architectures  
Examples: mips, powerpc, riscv, sh, sparc, x86
- ▶ Find details in kernel sources: arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/



# Embedded Linux Kernel Usage

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Linux kernel sources



# Location of kernel sources

- ▶ The official versions of the Linux kernel, as released by Linus Torvalds, are available at <http://www.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 kernel sub-communities maintain their own kernel, with usually newer but less 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 development trees are available.



## Getting Linux sources

- ▶ The kernel sources are available from <http://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> v3.11
```
  - ▶ Web interface available at  
<http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/>.
  - ▶ Read more about Git at <http://git-scm.com/>



# Working with git: SSD storage needed for serious work

For all work with `git`, but especially on big projects such as the Linux kernel...

- ▶ Having a fast disk will dramatically speed up most `git` operations.
- ▶ Ask your boss to order an SSD disk for your laptop. It will make you more productive.
- ▶ If you are in a Bootlin public session, you already have an SSD disk!





# Linux kernel size (1)

- ▶ Linux 4.11 sources:
  - ▶ 57994 files (`git ls-files | wc -l`)
  - ▶ 23144003 lines (`wc -l $(git ls-files)`)
  - ▶ 675576310 bytes (`wc -c $(git ls-files)`)
- ▶ Minimum Linux 4.11 compiled kernel size, booting to a shell on the ARM Versatile board: 405,464 bytes (compressed), 1,112,264 bytes (raw)
- ▶ 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 4.6 (in lines).

- ▶ drivers/: 57.0%
- ▶ arch/: 16.3%
- ▶ fs/: 5.5%
- ▶ sound/: 4.4%
- ▶ net/: 4.3%
- ▶ include/: 3.5%
- ▶ Documentation/: 2.8%
- ▶ tools/: 1.3%
- ▶ kernel/: 1.2%
- ▶ firmware/: 0.6%
- ▶ lib/: 0.5%
- ▶ mm/: 0.5%
- ▶ scripts/: 0.4%
- ▶ crypto/: 0.4%
- ▶ security/: 0.3%
- ▶ block/: 0.1%
- ▶ ...



## Practical lab - Downloading kernel source code



- ▶ Clone the mainline Linux source tree with git



# Kernel Source Code

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Linux Code and Device Drivers



# Programming language

- ▶ Implemented in C like all Unix systems. (C was created to implement the first 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-6.2.0/gcc/C-Extensions.html>
- ▶ Ongoing work to compile the kernel with the LLVM compiler.



## 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



# No floating point computation

- ▶ Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on certain ARM CPUs).
- ▶ Don't be confused with floating point related configuration options
  - ▶ They are related to the emulation of floating point operation performed by the user space applications, triggering an exception into the kernel.
  - ▶ Using soft-float, i.e. emulation in user space, is however recommended for performance reasons.



# 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.



## Kernel memory constraints

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



# 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:  
<http://j.mp/fbyuuH>
  - ▶ 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



## 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*, <http://www.libusb.info/>
  - ▶ SPI with *spidev*, Documentation/spi/spidev
  - ▶ I2C with *i2cdev*, Documentation/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
  - ▶ 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>, machine/board specific code
  - ▶ arch/<ARCH>/include/asm, architecture-specific headers
  - ▶ arch/<ARCH>/boot/dts, Device Tree source files, for some architectures
- ▶ block/
  - ▶ Block layer core
- ▶ COPYING
  - ▶ Linux copying conditions (GNU GPL)
- ▶ CREDITS
  - ▶ Linux main contributors
- ▶ crypto/
  - ▶ Cryptographic libraries



# Linux sources structure 2/5

- ▶ Documentation/
  - ▶ Kernel documentation sources  
Also available on <https://www.kernel.org/doc/>  
(includes functions prototypes and comments extracted from source code).
- ▶ drivers/
  - ▶ All device drivers except sound ones (usb, pci...)
- ▶ firmware/
  - ▶ Legacy: firmware images extracted from old drivers
- ▶ 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 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 arch and version)
- ▶ mm/
  - ▶ Memory management code (small too!)
- ▶ net/
  - ▶ Network support code (not drivers)
- ▶ README
  - ▶ Overview and building instructions
- ▶ samples/
  - ▶ Sample code (markers, kprobes, kobjects, bpf...)



## Linux sources structure 5/5

- ▶ scripts/
  - ▶ Executables for internal or external use
- ▶ 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



- ▶ 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`.
- ▶ Dedicated graphical front-end: KScope
- ▶ <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!



The screenshot shows the Bootlin website interface. On the left, there is a sidebar for 'Project selection' containing a dropdown menu with 'linux' selected, and a list of projects: 'busybox', 'linux', and 'u-boot'. Below this is a tree view of kernel versions from v4.10 to v4.16. A red arrow points to the 'busybox' item with the label 'All versions available'. In the center, there is a 'Search Identifier' input field with a magnifying glass icon, and a list of kernel subsystems: Documentation, arch, block, certs, crypto, drivers, firmware, fs, include, init, ipc, kernel, lib, mm, net, samples, and scripts. A red arrow points to the search field with the label 'Identifier search'. A red arrow also points to the list of subsystems with the label 'Source browsing'. At the bottom right, there is a footer bar with the text 'powered by Elixir 0.2'.



# 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.



## Specifying the target architecture

First, specify the architecture for the kernel to build

- ▶ Set it 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.



# 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>`



## Kernel configuration details

- ▶ The configuration is stored in the `.config` file at the root of kernel sources
  - ▶ Simple text file, key=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:
  - ▶ 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

- ▶ `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
- ▶ `string` options, to specify string values



# 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 A that depends on option B is not visible until option B is enabled
  - ▶ select dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
- ▶ With the Show All Options option, make xconfig allows to see all options, even the ones that cannot be selected because of missing dependencies. Values for dependencies are shown.



## make xconfig

### make xconfig

- ▶ The most common graphical interface to configure the kernel.
- ▶ Make sure you read
  - help -> introduction: useful options!
- ▶ File browser: easier to load configuration files
- ▶ Search interface to look for parameters
- ▶ Required Debian / Ubuntu packages: qt5-default



make xconfig screenshot

The screenshot shows the 'Option' menu open in the kernel configuration tool. The 'Kernel compression mode' section is currently selected, with 'LZ4' chosen as the compression method. The 'LZ4 (KERNEL\_LZ4)' section is expanded, displaying configuration details for the LZ4 compressor.

**Cross-compiler tool prefix:**

- Compile also drivers which will not load
- Local version - append to kernel release:
- Automatically append version information to the version string

**Kernel compression mode**

- Gzip
- LZMA
- XZ
- LZO
- LZ4

**Default hostname: (none)**

- Support for paging of anonymous memory (swap)
- System V IPC
- POSIX Message Queues
- Enable process vm ready/writev syscalls

**LZ4 (KERNEL\_LZ4)**

**CONFIG\_KERNEL\_LZ4:**

LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding. A preliminary version of LZ4 de/compression tool is available at <<https://code.google.com/p/lz4/>>.

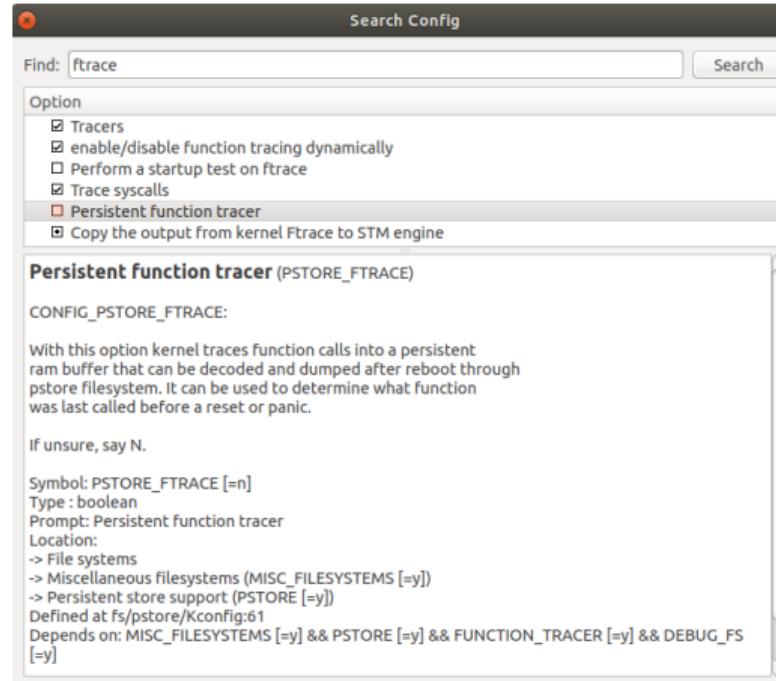
Its compression ratio is worse than LZO. The size of the kernel is about 8% bigger than LZO. But the decompression speed is faster than LZO.

**Symbol: KERNEL\_LZ4 [=n]**  
Type : boolean  
Prompt: LZ4  
Location:  
-> General setup  
-> Kernel compression mode (<choice> [=y])  
Defined at init/Kconfig:200  
Depends on: <choice> && HAVE\_KERNEL\_LZ4 [=y]



# make xconfig search interface

Looks for a keyword in the parameter name. Allows to select or unselect found parameters.





# Kernel configuration options

Compiled as a module (separate file)

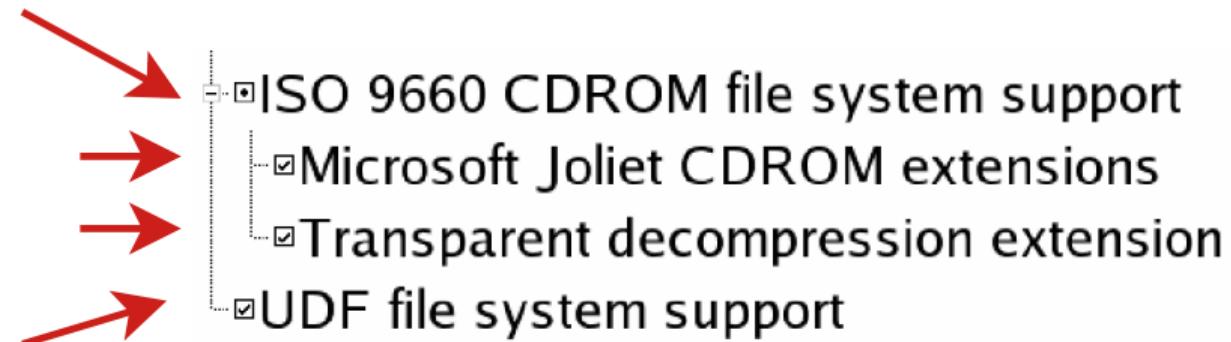
`CONFIG_ISO9660_FS=m`

Driver options

`CONFIG_JOLIET=y`

`CONFIG_ZISOFS=y`

Compiled statically into the kernel  
`CONFIG_UDF_FS=y`





## Corresponding .config file excerpt

Options are grouped by sections and are prefixed with CONFIG\_.

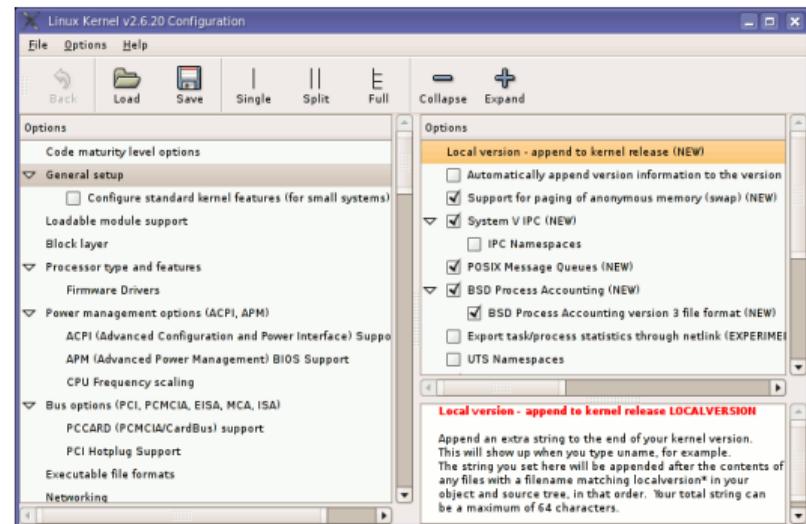
```
#  
# CD-ROM/DVD Filesystems  
#  
CONFIG_IS09660_FS=m  
CONFIG_JOLIET=y  
CONFIG_ZISOFS=y  
CONFIG_UDF_FS=y  
CONFIG_UDF_NLS=y  
  
#  
# DOS/FAT/NT Filesystems  
#  
# CONFIG_MSdos_FS is not set  
# CONFIG_VFAT_FS is not set  
CONFIG_NTFS_FS=m  
# CONFIG_NTFS_DEBUG is not set  
CONFIG_NTFS_RW=y
```



# 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

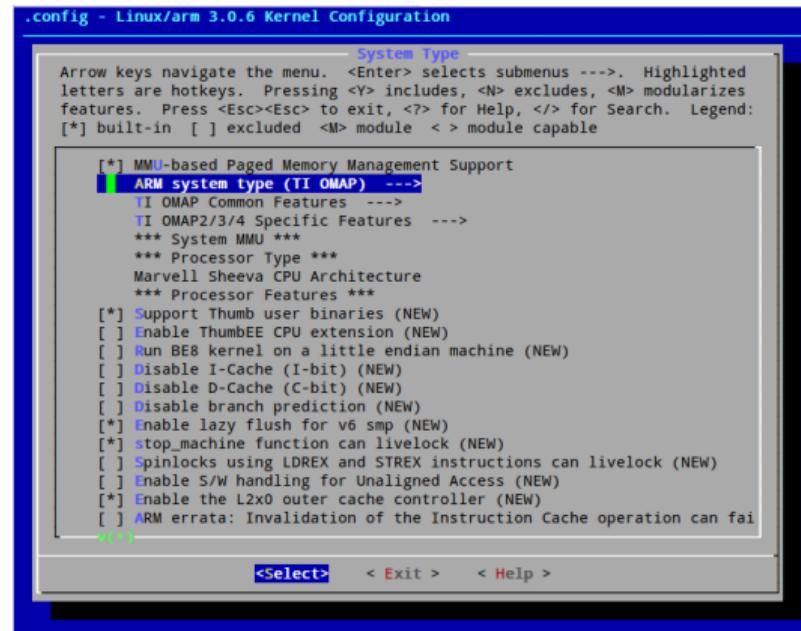




# make menuconfig

## make menuconfig

- ▶ Useful when no graphics are available.  
Pretty convenient too!
- ▶ Same interface found in other tools:  
BusyBox, Buildroot...
- ▶ Required Debian packages:  
`libncurses-dev`





# make nconfig

## make nconfig

- ▶ A newer, similar text interface
- ▶ More user friendly (for example, easier to access help information).
- ▶ Required Debian packages:  
libncurses-dev

```
.config - Linux/x86_64 3.0.0 Kernel Configuration
Linux/x86_64 3.0.0 Kernel Configuration

[ ] General setup --->
  [ ] Enable loadable module support --->
  [*] Enable the block layer --->
    Processor type and features --->
    Power management and ACPI options --->
    Bus options (PCI etc.) --->
    Executable file formats / Emulations --->
  [ ] Networking support --->
    Device Drivers --->
    Firmware Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
  [ ] Cryptographic API --->
  [ ] Virtualization --->
    Library routines --->

F1 Help F2 Sym Info F3 Insts F4 Config F5 Back F6 Save F7 Load F8 Sym Search F9 Exit
```



## make oldconfig

make oldconfig

- ▶ Needed very often!
- ▶ Useful to upgrade a `.config` file from an earlier kernel release
- ▶ Issues warnings for configuration parameters that no longer exist in the new kernel.
- ▶ Asks for values for new parameters (while `xconfig` and `menuconfig` silently set default values for new parameters).

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!



# 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



# Choose a compiler

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

- ▶ 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 compilation

- ▶ make
  - ▶ In the main kernel source directory!
  - ▶ Remember to run multiple jobs in parallel if you have multiple CPU cores. Example:  
`make -j 8`
  - ▶ No need to run as root!
- ▶ Generates
  - ▶ `vmlinu`x, 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, `vmlinu`.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` behaviour 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.aliases.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

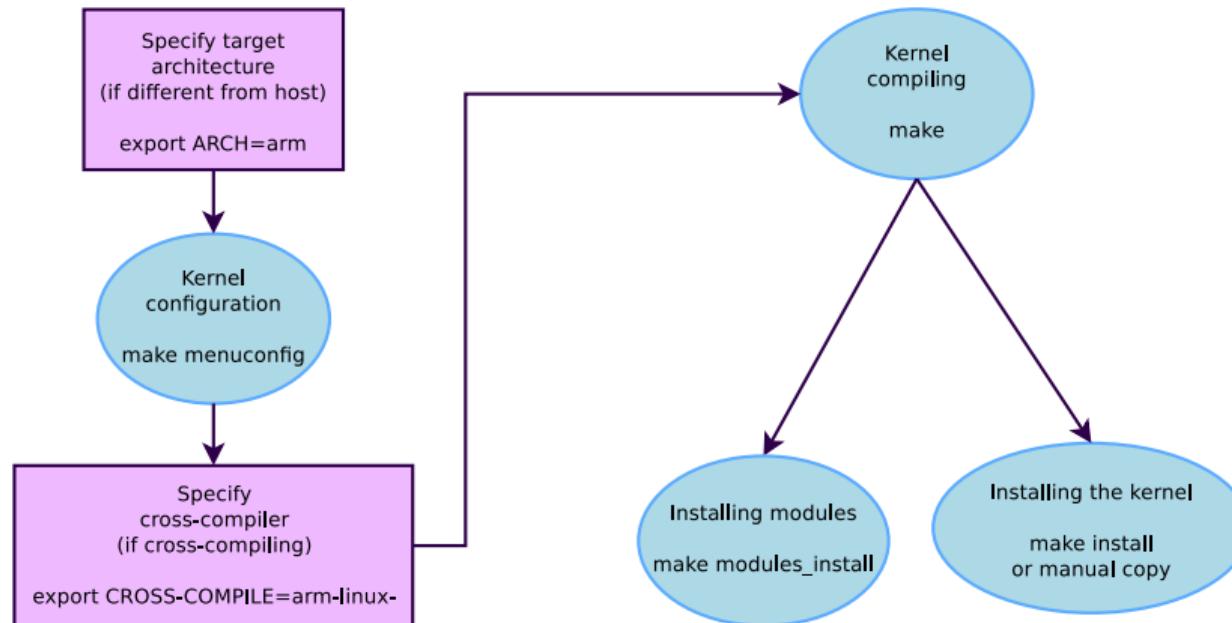
- ▶ Clean-up generated files (to force re-compilation):  
`make clean`
- ▶ Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your `.config` file!  
`make mrproper`
- ▶ Also remove editor backup and patch reject files (mainly to generate patches):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





# Kernel building overview

Environment setup  
and configuration





## Booting the kernel



# Device Tree (DT)

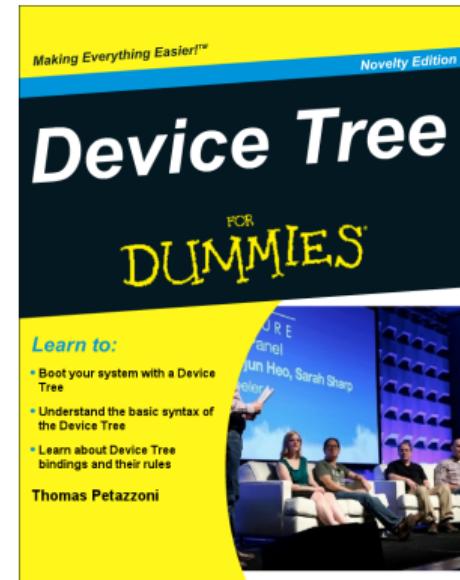
- ▶ Many embedded architectures have a lot of non-discoverable hardware.
- ▶ Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ▶ The DT was created for PowerPC, and later was 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).
- ▶ 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`.
  - ▶ 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.
- ▶ To configure some system parameters: flash partitions, kernel command line (other ways exist)
- ▶ Useful reference: Device Tree for Dummies, Thomas Petazzoni (Apr. 2014):  
<http://j.mp/1jQU6NR>





# 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`

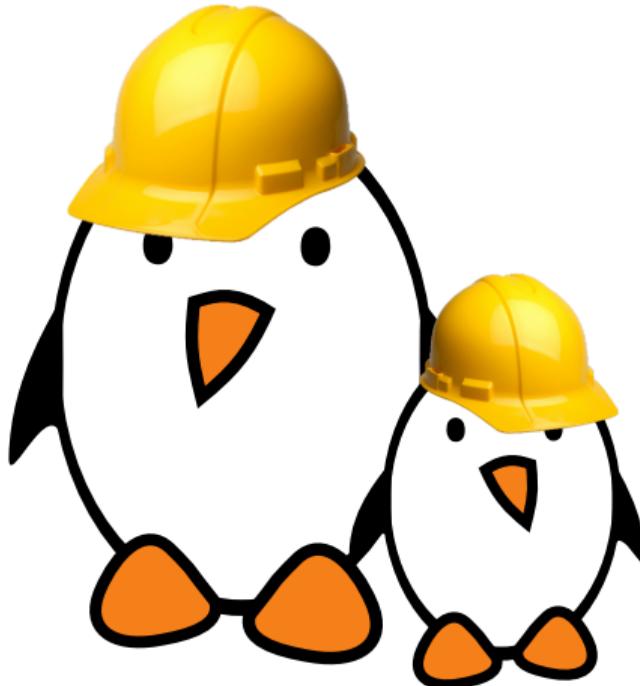


# Kernel command line

- ▶ In addition to the compile time configuration, the kernel behaviour 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
  - ▶ Many more exist. The most important ones are documented in `admin-guide/kernel-parameters` in kernel documentation.
- ▶ This kernel command line is either
  - ▶ 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.



# 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.



## 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`  
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 (**diagnostic message**)
- ▶ 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 the current values for the parameters of a loaded module?

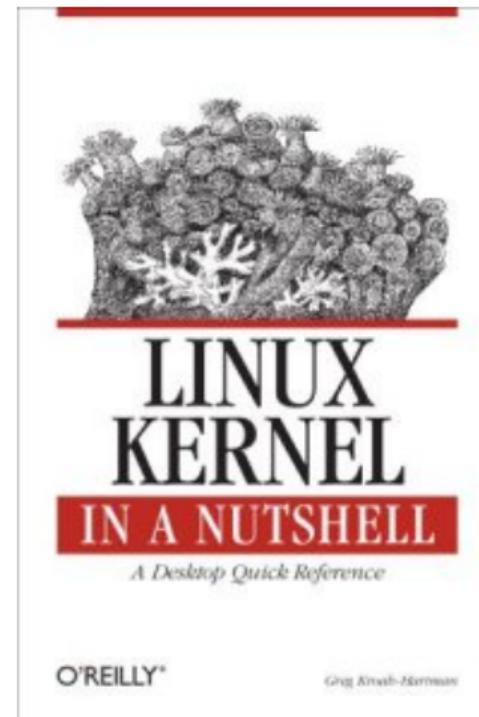
- ▶ Check `/sys/module/<name>/parameters`.
- ▶ There is one file per parameter, containing the parameter value.



## 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/>
- ▶ Our rating: 2 stars





# Developing Kernel Modules

# Developing Kernel Modules

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# Hello Module 1/2

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

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

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

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



## Hello Module 2/2

- ▶ `__init`
  - ▶ removed after initialization (static kernel or module.)
- ▶ `__exit`
  - ▶ discarded when module compiled statically into the kernel, or when module unloading support is not enabled.
- ▶ Example available on  
<https://git.bootlin.com/training-materials/plain/code/hello/hello.c>



# Hello Module Explanations

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

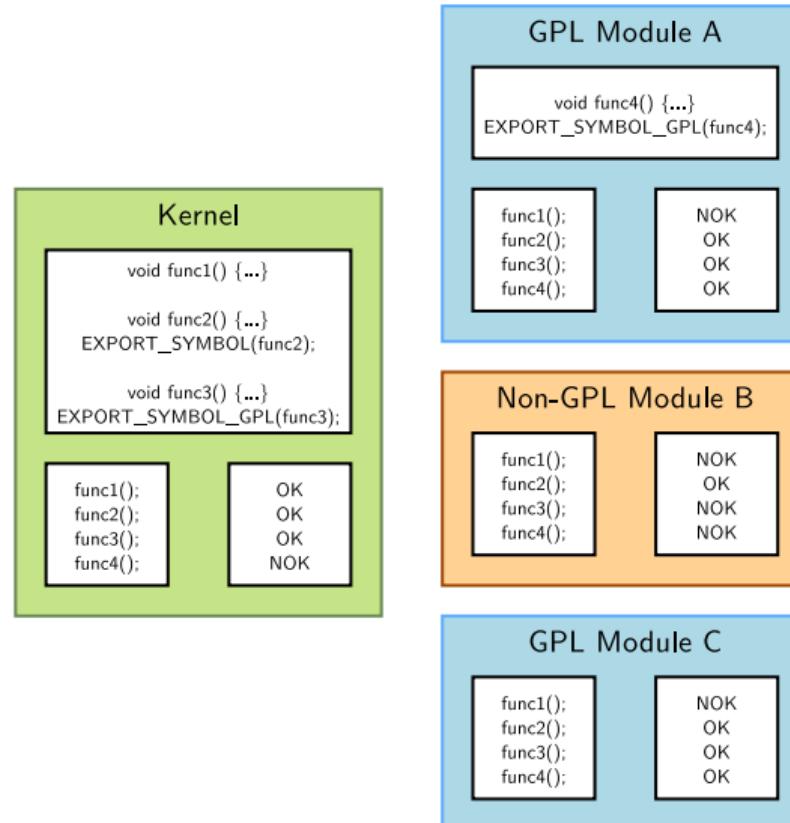


## Symbols Exported to Modules 1/2

- ▶ 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
- ▶ 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).
  - ▶ Useful for users to check that their system is 100% free (check `/proc/sys/kernel/tainted`)
- ▶ 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

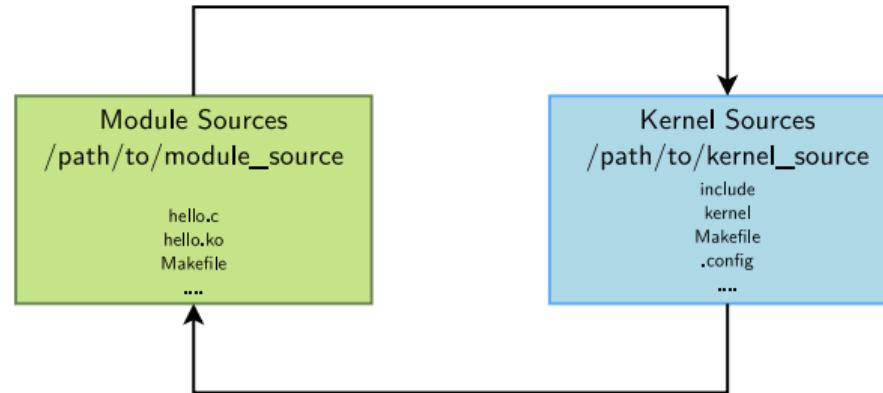
```
ifeq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

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 the 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
  - ▶ Many macros or functions depend on the configuration
- ▶ 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 `Documentation/kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.



# Hello Module with Parameters 1/2

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

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how many
   times we say hello, and to whom */

static char *whom = "world";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);
```



## 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 example!

Source code available on:

[https://git.bootlin.com/training-materials/plain/code/hello-param/hello\\_param.c](https://git.bootlin.com/training-materials/plain/code/hello-param/hello_param.c)



## Declaring a module parameter

```
module_param(  
    name, /* name of an already defined variable */  
    type, /* either byte, short, ushort, int, uint, long, ulong,  
            charp, bool or invbool. (checked at run time!) */  
    perm /* for /sys/module/<module_name>/parameters/<param>,  
          0: no such module parameter value file */  
);  
  
/* Example */  
static int irq=5;  
module_param(irq, int, S_IRUGO);
```

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

## Useful general-purpose kernel APIs

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# 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)

From include/linux/atmel\_tc.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)

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

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Introduction



## The need for a device model?

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

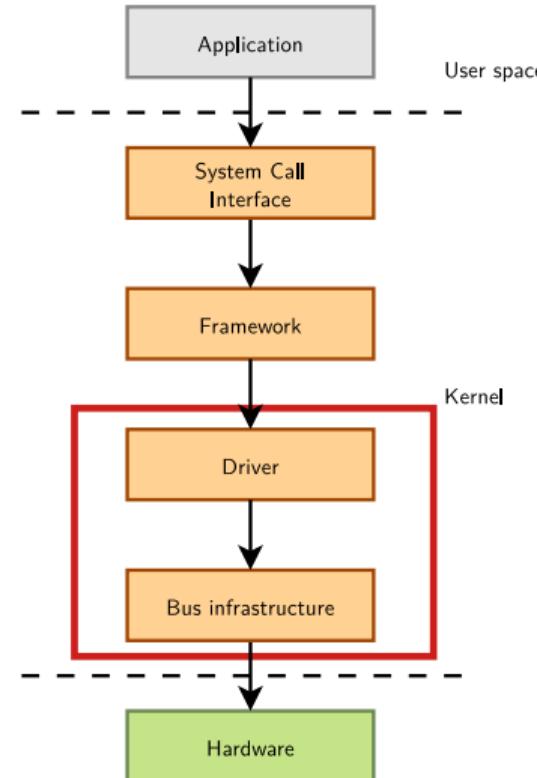


# Kernel and Device Drivers

In Linux, a driver is always interfacing with:

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

This section focuses on the *device model*, 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 enumerations are handled.



# Bus Drivers

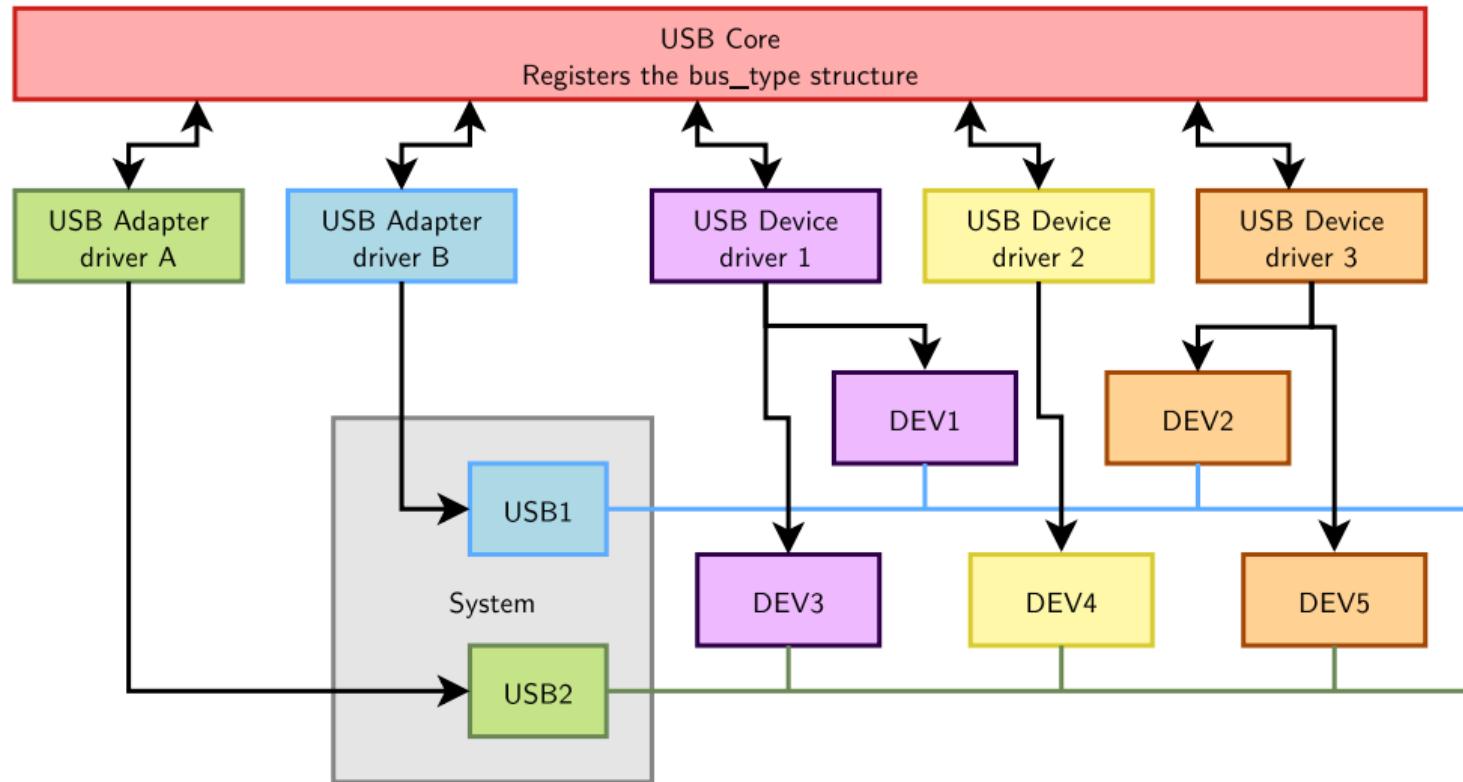
- ▶ The first component of the device model is the bus driver
  - ▶ One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- ▶ It is responsible for
  - ▶ Registering the bus type (`struct bus_type`)
  - ▶ Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices, 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 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` to extract at compile time the relation 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 (Un)Registration

- ▶ When the driver is loaded or unloaded, it must register or unregister itself 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);
}

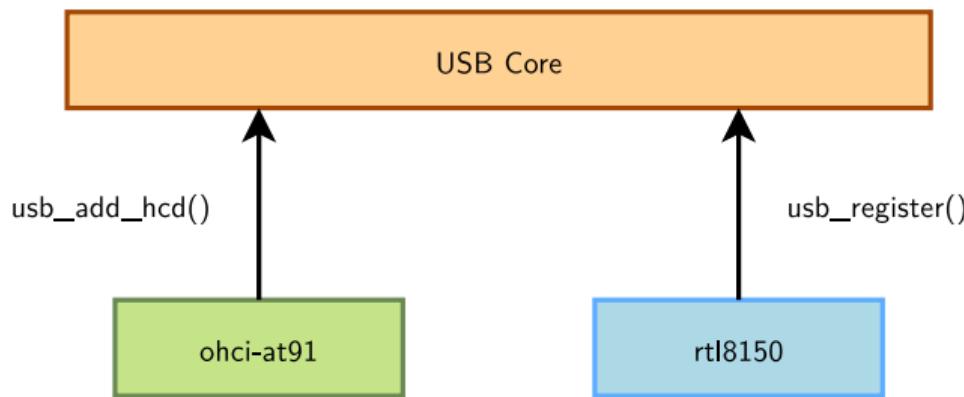
module_init(usb rtl8150_init);
module_exit(usb rtl8150_exit);
```

- ▶ Note: this code has now been replaced by a shorter `module_usb_driver()` macro call.



## At Initialization

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

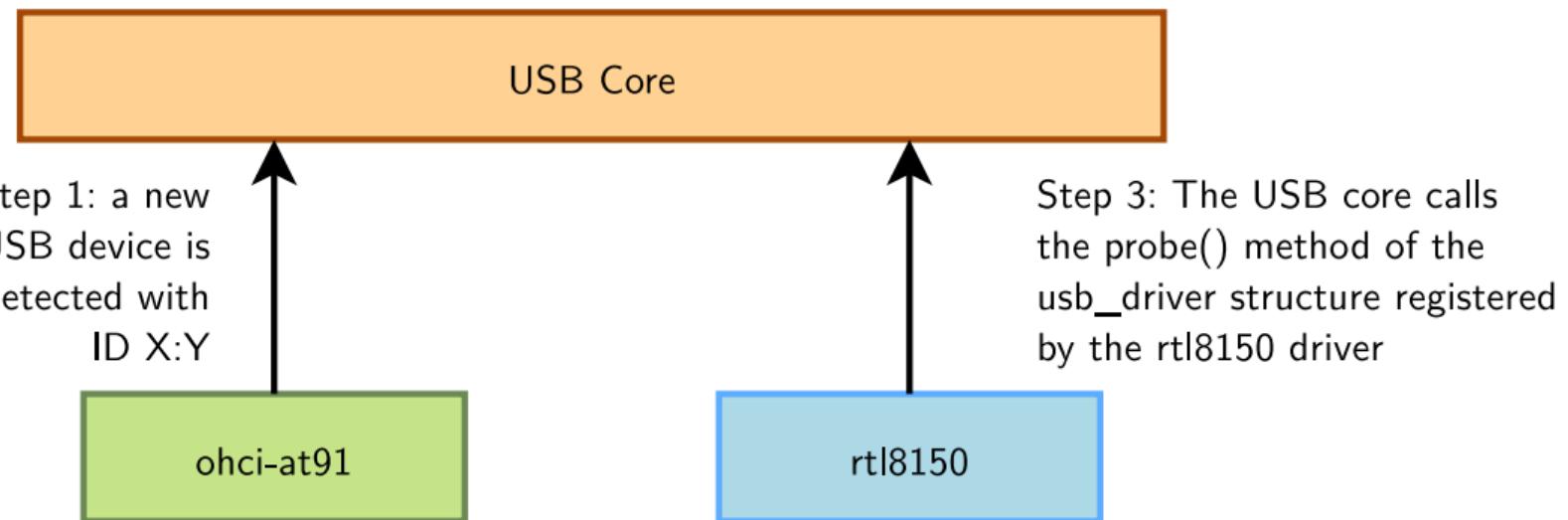


- ▶ The USB core now knows the association between the vendor/product IDs of `rtl8150` and the `struct usb_driver` structure of this driver



## When a device is detected

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





## Probe Method

- ▶ 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.



# Probe Method Example

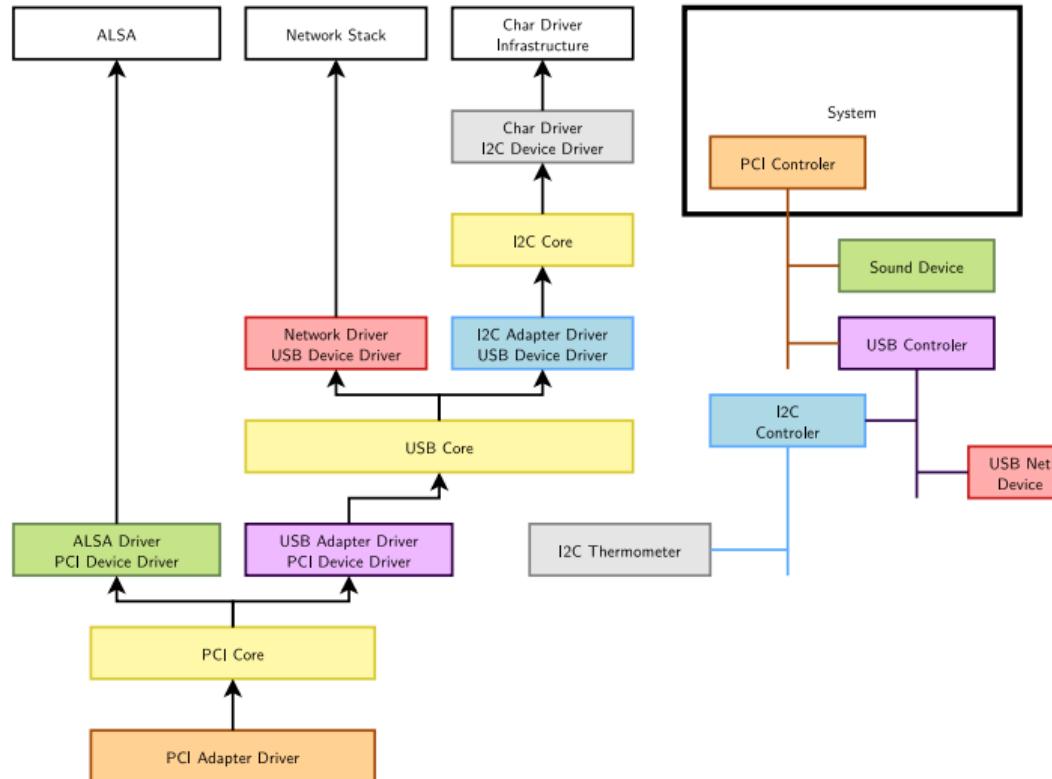
```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    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;
}
```



# 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.



## 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);
}
```



## 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 uart1_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 = &uart1_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 are moving, or 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)

Definition of the AM33xx SoC

```
/ {  
    compatible = "ti,am33xx";  
    [...]  
    ocp {  
        [...]  
        uart0: serial@044e09000 {  
            compatible = "ti,am3352-uart",  
                        "ti,omap3-uart";  
            reg = <0x44e09000 0x2000>;  
            interrupts = <72>;  
            status = "disabled";  
            [...]  
        };  
    };  
};  
am33xx.dtsi
```

Common definitions for  
BeagleBone boards

```
[...]  
&uart0 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&uart0_pins>;  
    status = "okay";  
};  
am335x-bone-common.dtsi
```

Definition for BeagleBone Black

```
#include "am33xx.dtsi"  
#include "am335x-bone-common.dtsi"  
/{  
    model = "TI AM335x BeagleBone Black";  
    compatible = "ti,am335x-bone-black",  
                "ti,am335x-bone",  
                "ti,am33xx";  
};  
[...]  
am335x-boneblack.dts
```



Compiled DTB

```
/ {  
    compatible = "ti,am335x-bone-black", "ti,am335x-bone",  
                "ti,am33xx";  
    model = "TI AMM335x BeagleBone Black";  
    [...]  
    ocp {  
        uart0: serial@044e09000 {  
            compatible = "ti,am3352-uart", "ti,omap3-uart";  
            reg = <0x44e09000 0x2000>;  
            interrupts = <72>;  
            pinctrl-names = "default";  
            pinctrl-0 = <&uart0_pins>;  
            status = "okay";  
        };  
    };  
};  
am335x-boneblack.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.

```
#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),
    },
};
```



# 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 bindings

- ▶ The compatible string and the associated properties define what is called a *device tree binding*.
- ▶ *Device tree bindings* are all documented in Documentation/devicetree/bindings.
- ▶ Since the Device Tree is normally part of the kernel ABI, the *bindings* must remain compatible over time.
  - ▶ A new kernel must be capable of using an old Device Tree.
  - ▶ This requires a very careful design of the bindings. They are all reviewed on the devicetree@vger.kernel.org mailing list.
  - ▶ See Thomas Petazzoni's presentation on this topic: *Device Tree as a stable ABI: a fairy tale?* (<http://bit.ly/1U1tYkT>).
- ▶ A Device Tree binding should contain only a *description of the hardware* and not *configuration*.
  - ▶ An interrupt number can be part of the Device Tree as it describes the hardware.
  - ▶ But not whether DMA should be used for a device or not, as it is a configuration choice.

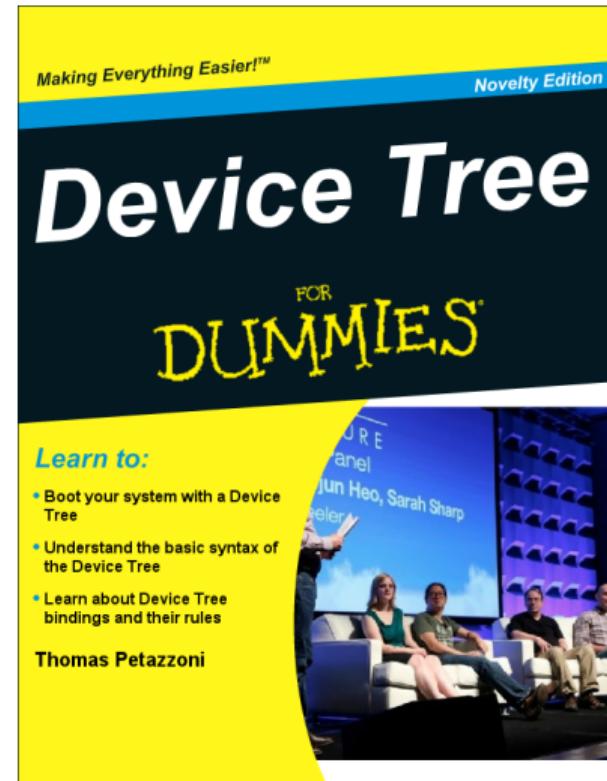


- ▶ The bus, device, drivers, etc. structures are internal to the kernel
- ▶ The `sysfs` virtual filesystem offers a mechanism to export such information to user space
- ▶ Used for example by `udev` to provide automatic module loading, firmware loading, device file creation, etc.
- ▶ `sysfs` is usually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



## References

- ▶ Device Tree for Dummies, Thomas Petazzoni (Apr. 2014):  
<http://j.mp/1jQU6NR>
- ▶ Kernel documentation
  - ▶ Documentation/driver-model/
  - ▶ Documentation/devicetree/
  - ▶ Documentation/filesystems/sysfs.txt
- ▶ <http://devicetree.org>
- ▶ The kernel source code
  - ▶ Full of examples of other drivers!





# Introduction to the I2C subsystem

# Introduction to the I2C subsystem

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



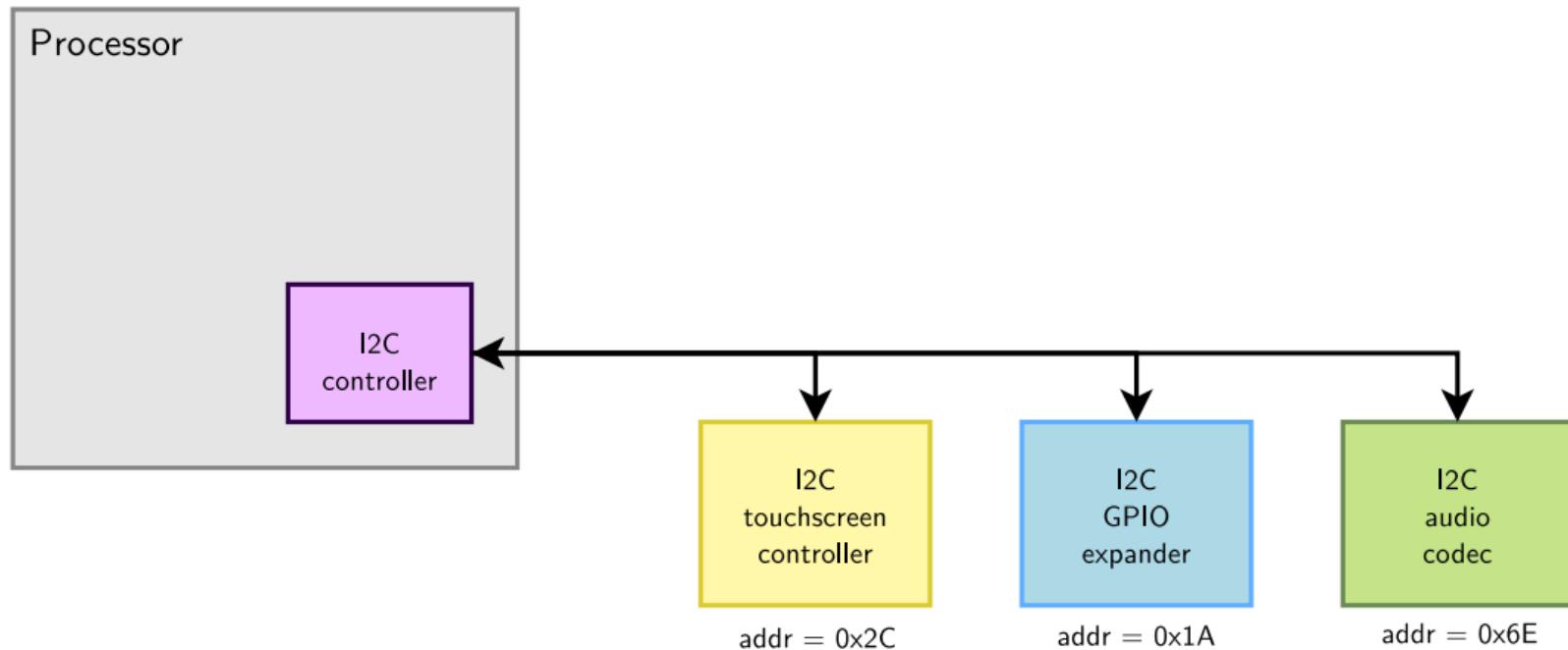


# What is I2C?

- ▶ A very commonly used low-speed bus to connect on-board and external devices to the processor.
- ▶ Uses only two wires: SDA for the data, SCL for the clock.
- ▶ It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- ▶ In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- ▶ Each slave device is identified by a unique I2C address. 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 subsystem

- ▶ Like all bus subsystems, the I2C subsystem 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 subsystem 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 type of device (ex: `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` field that must point to a list of *device IDs* (which is a list of tuples containing a string and some private driver data). It is used for non-DT based probing of I2C devices.
- ▶ The `i2c_add_driver()` and `i2c_del_driver()` functions are used to register/unregister the driver.
- ▶ If the driver doesn't do anything else in its `init()`/`exit()` functions, it is advised to use the `module_i2c_driver()` macro instead.



# Registering an I2C device driver: example

```
static const struct i2c_device_id <driver>_id[] = {
    { "<device-name>", 0 },
    { }
};

MODULE_DEVICE_TABLE(i2c, <driver>_id);

#ifndef CONFIG_OF
static const struct _device_id <driver>_dt_ids[] = {
    { .compatible = "<vendor>,<device-name>", },
    { }
};
MODULE_DEVICE_TABLE(of, <driver>_dt_ids);
#endif

static struct i2c_driver <driver>_driver = {
    .probe        = <driver>_probe,
    .remove       = <driver>_remove,
    .id_table     = <driver>_id,
    .driver = {
        .name      = "<driver-name>",
        .owner     = THIS_MODULE,
        .of_match_table = of_match_ptr(<driver>_dt_ids),
    },
};

module_i2c_driver(<driver>_driver);
```



## 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 <board>_i2c_devices[] __initdata = {
{
    I2C_BOARD_INFO("cs42l51", 0x4a),
},
};

void board_init(void)
{
    /*
     * Here should be the registration of all devices, including
     * the I2C controller device.
    */

    i2c_register_board_info(0, <board>_i2c_devices,
                           ARRAY_SIZE(<board>_i2c_devices));

    /* More devices registered here */
}
```



# 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.



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

### Definition of the I2C controller, sun7i-a20.dtss file

```
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>;
};
```



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

Definition of the I2C device, sun7i-a20-olinuxino-micro.dts file

```
&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>;  
    };  
};
```



## probe() and remove()

- ▶ The `->probe()` 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`.
  - ▶ 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()`



## Probe/remove example

```
static int <driver>_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
    /* initialize device */
    /* register to a kernel framework */

    i2c_set_clientdata(client, <private data>);
    return 0;
}

static int <driver>_remove(struct i2c_client *client)
{
    <private data> = i2c_get_clientdata(client);
    /* unregister device from kernel framework */
    /* shut down the device */
    return 0;
}
```



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



## 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 *msg, 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

```
struct i2c_msg msg[2];
int error;
u8 start_reg;
u8 buf[10];

msg[0].addr = client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = &start_reg;
start_reg = 0x10;

msg[1].addr = client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = sizeof(buf);
msg[1].buf = buf;

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



## 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 Documentation/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

- ▶ <http://en.wikipedia.org/wiki/I2C>, general presentation of the I2C protocol
- ▶ Documentation/i2c/, details about Linux support for I2C
  - ▶ Documentation/i2c/writing-clients  
How to write I2C kernel device drivers
  - ▶ Documentation/i2c/dev-interface  
How to write I2C user-space device drivers
  - ▶ Documentation/i2c/instantiating-devices  
How to instantiate devices
  - ▶ Documentation/i2c/smbus-protocol  
Details on the SMBus functions
  - ▶ Documentation/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

## Introduction to pin muxing

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



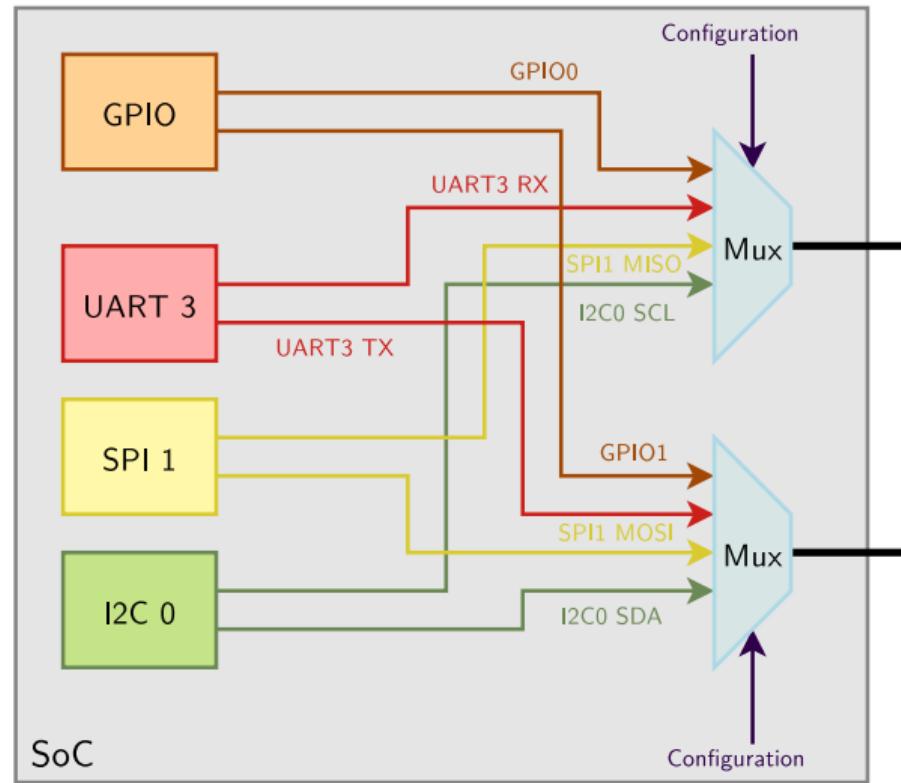


# What is pin muxing?

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



# Pin muxing diagram



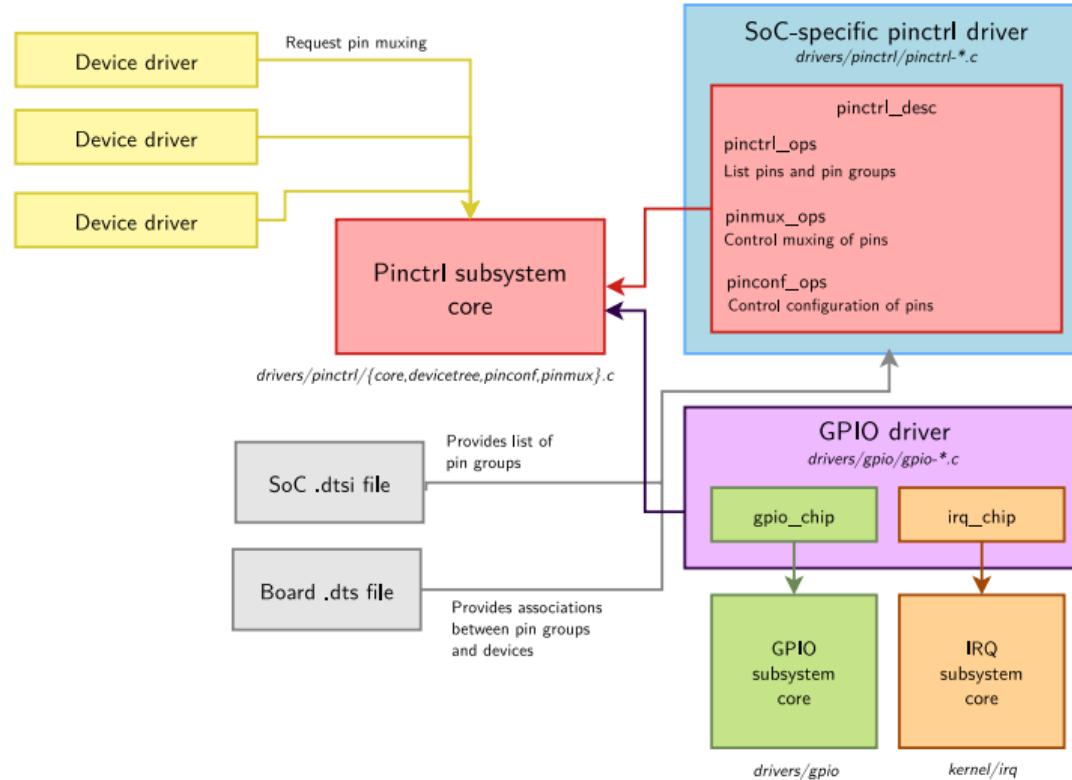


# Pin muxing in the Linux kernel

- ▶ Since Linux 3.2, a `pinctrl` subsystem has been added.
- ▶ This subsystem, located in `drivers/pinctrl/` provides a generic subsystem to handle pin muxing. It offers:
  - ▶ A pin muxing driver interface, to implement the system-on-chip specific drivers that configure the muxing.
  - ▶ A pin muxing consumer interface, for device drivers.
- ▶ Most `pinctrl` drivers provide a Device Tree binding, and the pin muxing must be described in the Device Tree.
  - ▶ The exact Device Tree binding depends on each driver. Each binding is documented in `Documentation/devicetree/bindings/pinctrl`.



# pinctrl subsystem diagram





# Device Tree binding for consumer devices

- ▶ The devices that require certain pins to be muxed will use the `pinctrl-<x>` and `pinctrl-names` Device Tree properties.
- ▶ The `pinctrl-0`, `pinctrl-1`, `pinctrl-<x>` properties link to a pin configuration for a given state of the device.
- ▶ The `pinctrl-names` property associates a name to each state. The name `default` is special, and is automatically selected by a device driver, without having to make an explicit `pinctrl` function call.
- ▶ In most cases, the following is sufficient:

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

- ▶ See Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt for details.



# 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 DT bindings documentation.



# 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: pim mux_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 Allwinner SoC

SoC level

```
/ {
    soc@01c00000 {
        pio: pinctrl@01c20800 (
            compatible = "allwinner,sun7i-a20-pinctrl";
            reg = <0x01c20800 0x400>;
            interrupts = <0 28 1>;

            uart0_pins_a: uart0@00 {
                allwinner,pins = "PB22", "PB23";
                allwinner,function = "uart0";
                allwinner,drive = <0>;
                allwinner,pull = <0>;
            };
            ...
        );
    };
};
```

Board level

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

    green {
        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_pins_a>;
        status = "okay";
    };
};

Enable UART0
and associate
pin mux
config
```



## 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

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



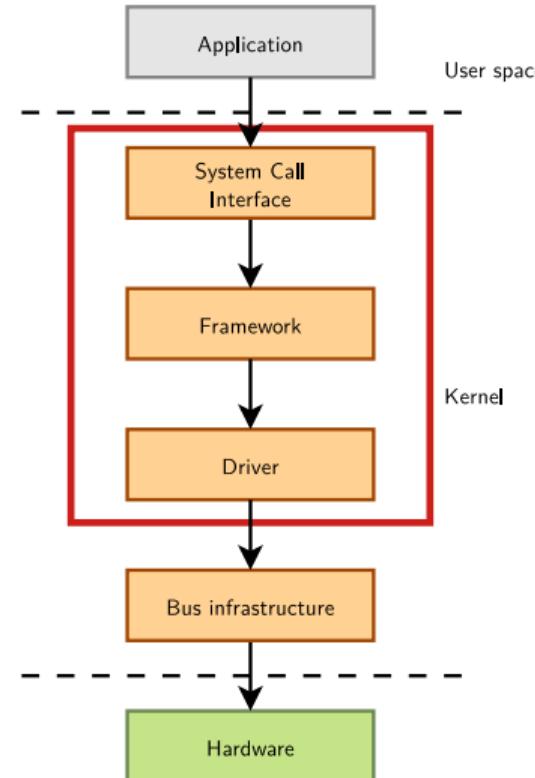


# Kernel and Device Drivers

In Linux, a driver is always interfacing with:

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

This section focuses on the *kernel frameworks*, while the *device model* 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 `ifconfig`.
- ▶ **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* typically indicates the number of the device (when there are for example several serial ports)
- ▶ 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/sda1 /dev/sda2 /dev/zero  
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  
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 known to the kernel. The `CONFIG_DEVTMPFS_MOUNT` kernel configuration option makes the kernel mount it automatically at boot time, except when booting on an initramfs.



## Character drivers

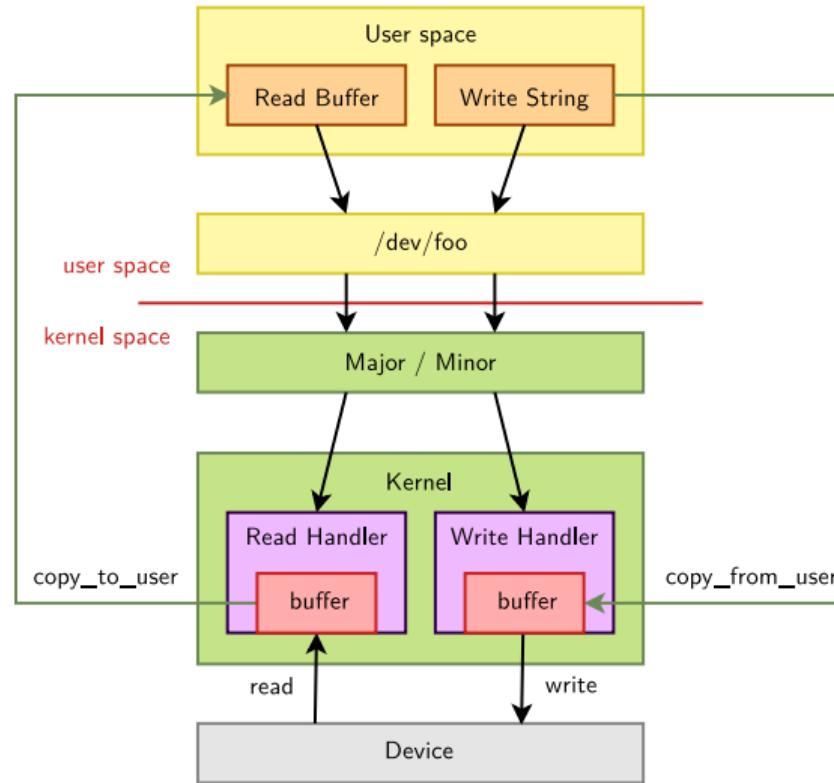


# A character driver in the kernel

- ▶ 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. All of them are optional.

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

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 *);
};
```



## 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 system (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()

- ▶ `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



## write()

- ▶ 

```
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.

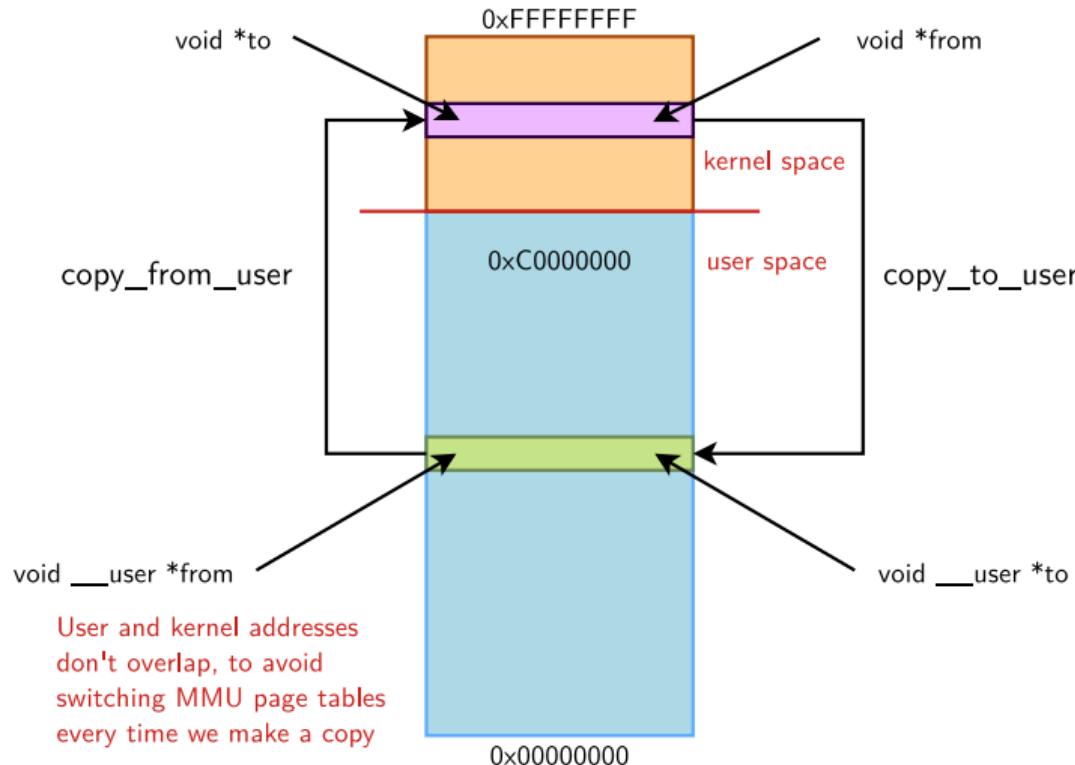


# Exchanging data with user space 2/3

- ▶ 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





## 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...
  - ▶ `cmd` is a number identifying the operation to perform
  - ▶ `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

```
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

```
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

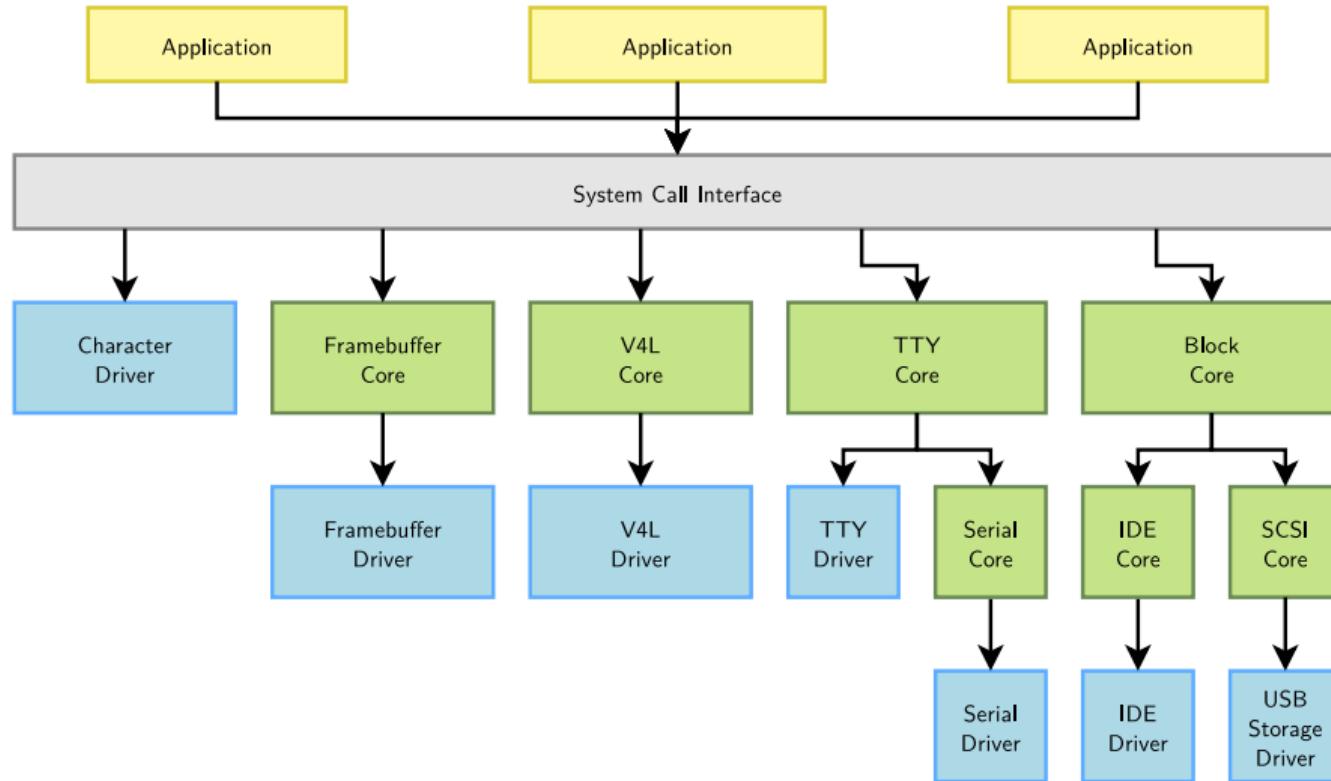


## Beyond character drivers: kernel frameworks

- ▶ 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
- ▶ 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

```
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 the character device 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 `Documentation/driver-model/devres.txt` 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)
{
    foo_t *foo;

    foo = kmalloc(sizeof(struct foo_t),
                 GFP_KERNEL);

    if (failure) {
        kfree(foo);
        return -EBUSY;
    }

    platform_set_drvdata(pdev, foo);
    pm_runtime_enable(&pdev->dev);
    pm_runtime_get_sync(&pdev->dev);
    return 0;
}

void foo_remove(struct platform_device *pdev)
{
    pm_runtime_disable(&pdev->dev);
    foo_t *foo = platform_get_drvdata(pdev);
    kfree(foo);
}
```

## With devm functions

```
int foo_probe(struct platform_device *pdev)
{
    foo_t *foo;

    foo = devm_kmalloc(&pdev->dev,
                      sizeof(struct foo_t),
                      GFP_KERNEL);

    if (failure)
        return -EBUSY;

    platform_set_drvdata(pdev, foo);
    pm_runtime_enable(&pdev->dev);
    pm_runtime_get_sync(&pdev->dev);
    return 0;
}

void foo_remove(struct platform_device *pdev)
{
    pm_runtime_disable(&pdev->dev);
}
```



## 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 its 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

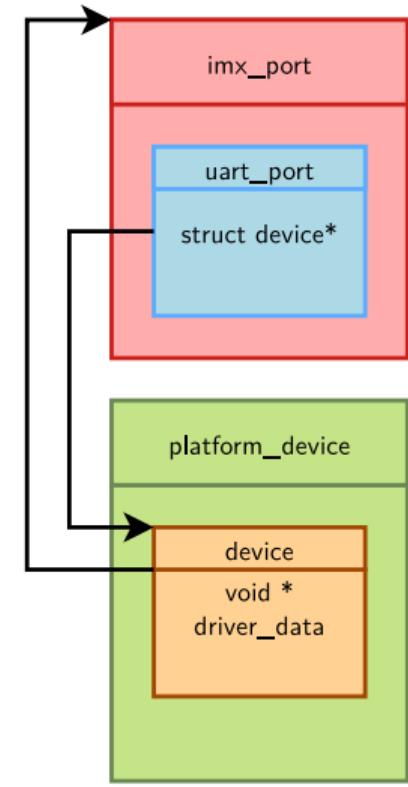
- ▶ The framework typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
  - ▶ It's the relation between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- ▶ The device structure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



# Links between structures 2/4

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    [...]
    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;
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, sport);
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

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

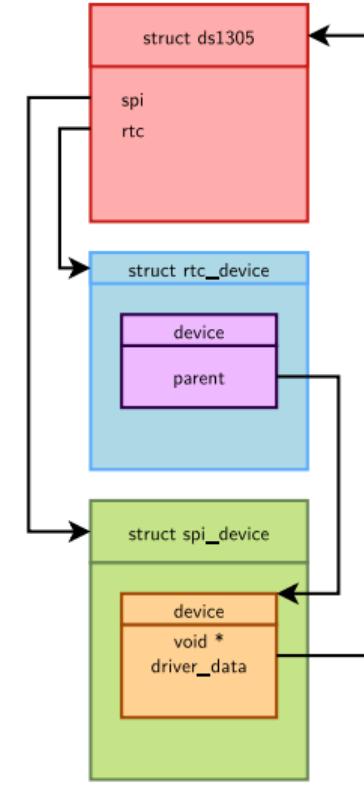




# Links between structures 3/4

```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305           *ds1305;
    [...]
    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
        return -ENOMEM;
    ds1305->spi = spi;
    spi_set_drvdata(spi, ds1305);
    [...]
    /* register RTC ... from here on, ds1305->ctrl needs locking */
    ds1305->rtc = devm_rtc_device_register(&spi->dev, "ds1305",
                                           &ds1305_ops, THIS_MODULE);
    [...]
}

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;
    dev->netdev = netdev;

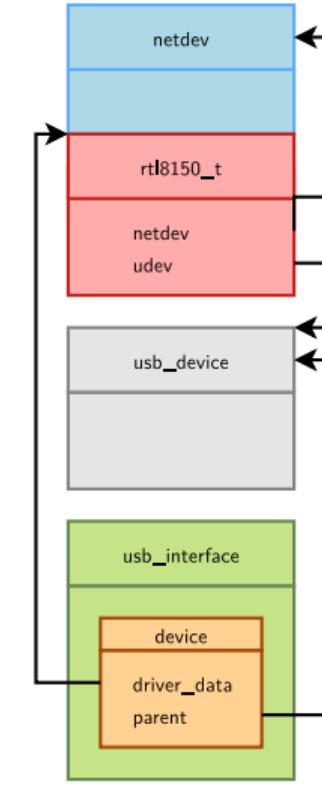
    [...]

    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);

    [...]
}

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

    [...]
}
```





# The input subsystem

# The input subsystem

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.  
Corrections, suggestions, contributions and translations are welcome!



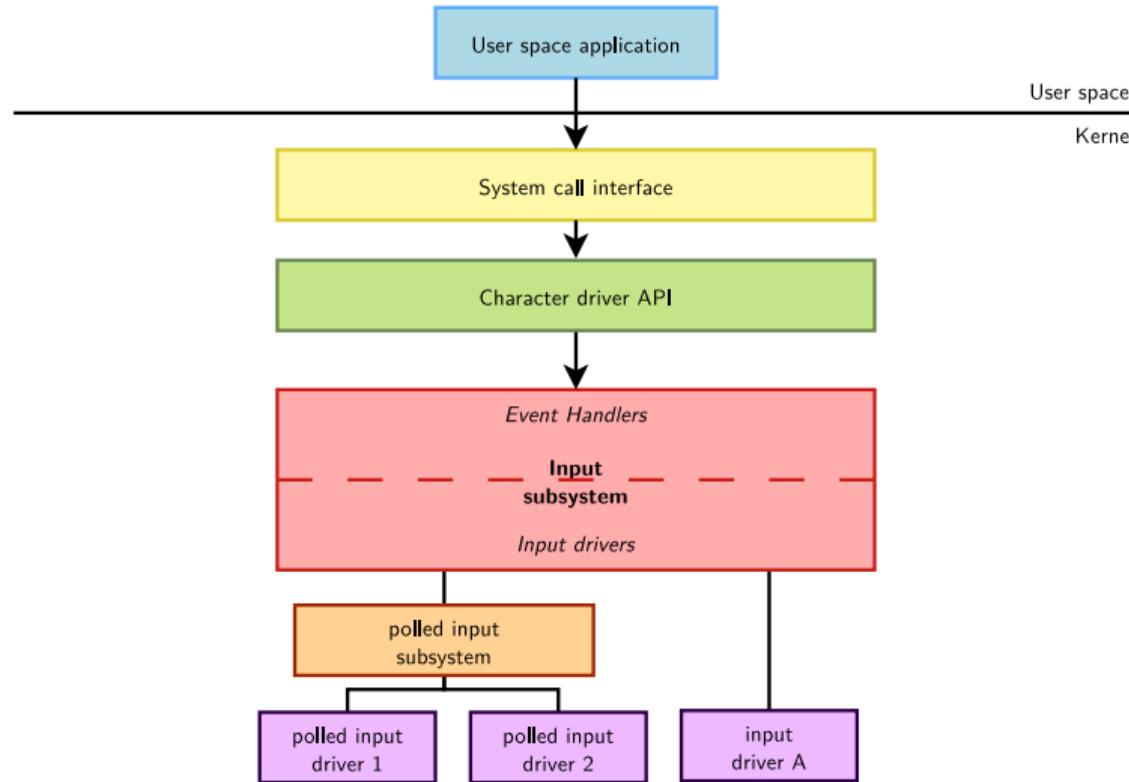


# What is the input subsystem?

- ▶ The input subsystem takes care of all the input events coming from the human user.
- ▶ Initially written to support the USB *HID* (Human Interface Device) devices, it quickly grew up to handle all kind 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, evbug.c
- ▶ Implements a single character driver and defines the user/kernel API
  - ▶ include/uapi/linux/input.h
- ▶ Defines the set of operations a 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;  
    [...]  
    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 it, 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 *);`
- ▶ When the driver is unloaded, the *input device* will be unregistered using:  
`void input_unregister_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 `Documentation/input/event-codes.txt`
  - ▶ 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()`, ...



## Polled input subclass

- ▶ The input subsystem provides a subclass supporting simple input devices that *do not raise interrupts* but have to be *periodically scanned or polled* to detect changes in their state.
- ▶ A *polled input device* is described by a `struct input_polled_dev` structure:

```
struct input_polled_dev {  
    void *private;  
    void (*open)(struct input_polled_dev *dev);  
    void (*close)(struct input_polled_dev *dev);  
    void (*poll)(struct input_polled_dev *dev);  
    unsigned int poll_interval; /* msec */  
    unsigned int poll_interval_max; /* msec */  
    unsigned int poll_interval_min; /* msec */  
    struct input_dev *input;  
/* private: */  
    struct delayed_work work;  
}
```



## Polled input subsystem API

- ▶ Allocating the `struct input_polled_dev` structure is done using `devm_input_allocate_polled_device()`
- ▶ Among the handlers of the `struct input_polled_dev` only the `poll()` method is mandatory, this function polls the device and posts input events.
- ▶ The fields `id`, `name`, `evkey` and `keybit` of the `input` field must be initialized too.
- ▶ If none of the `poll_interval` fields are filled then the default poll interval is 500ms.
- ▶ The device registration/unregistration is done with:
  - ▶ `input_register_polled_device(struct input_polled_dev *dev)`.
  - ▶ Unregistration is automatic after using `devm_input_allocate_polled_device()`!



## evdev user space interface

- ▶ 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 <http://cgit.freedesktop.org/evtest/>



## Practical lab - Expose the Nunchuk to user space



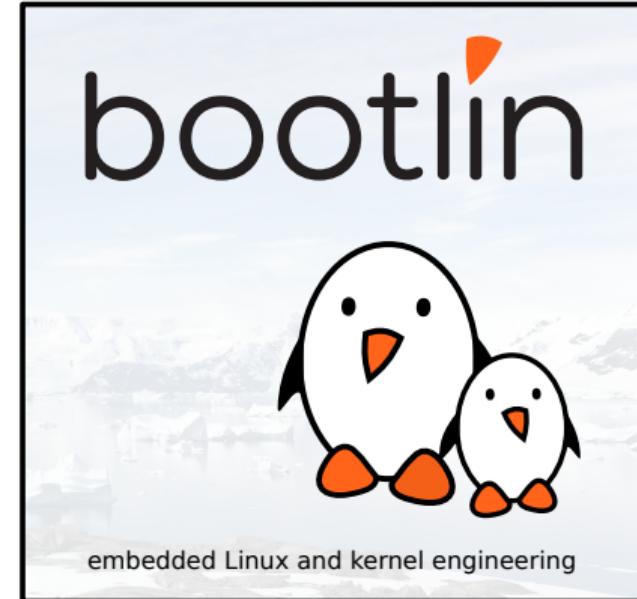
- ▶ Extend the Nunchuk driver to expose the Nunchuk features to user space applications, as an *input* device.
- ▶ Test the operation of the Nunchuk using sample user space applications.



# Memory Management

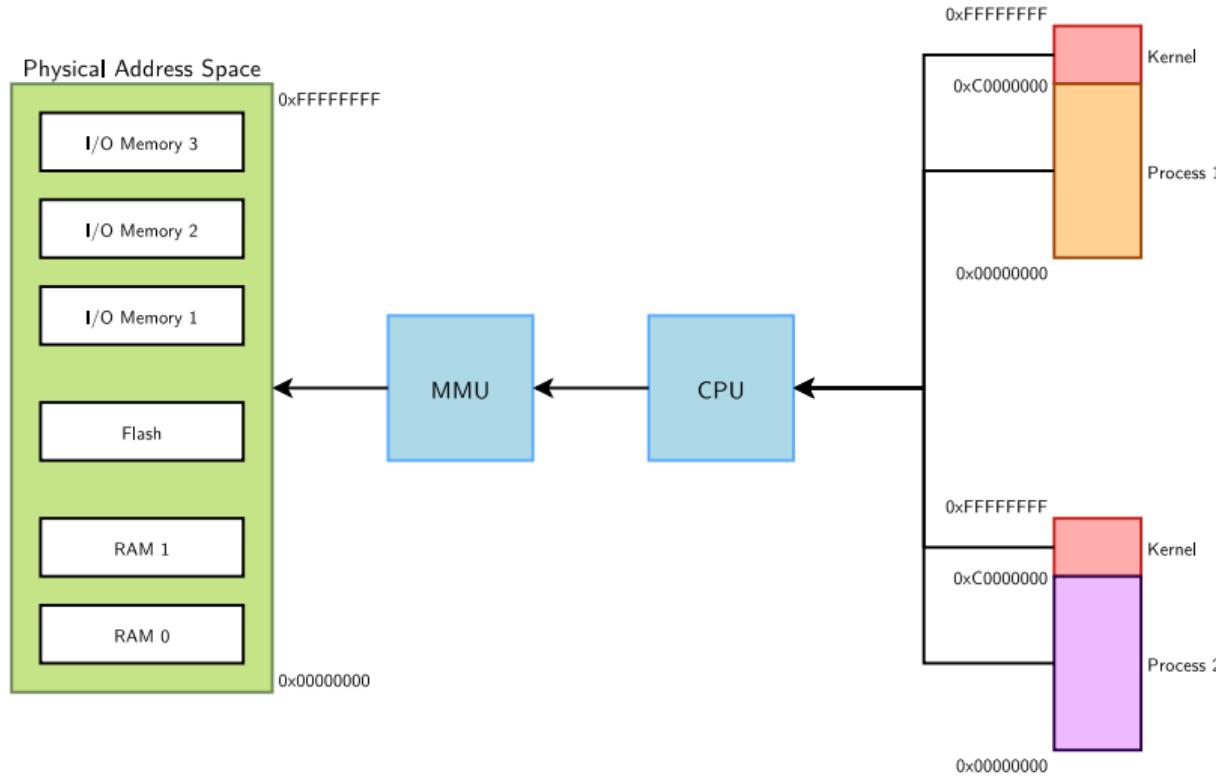
© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



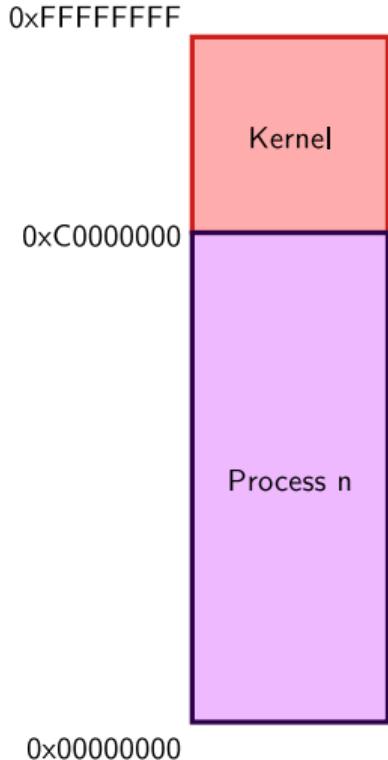


# Physical and Virtual Memory





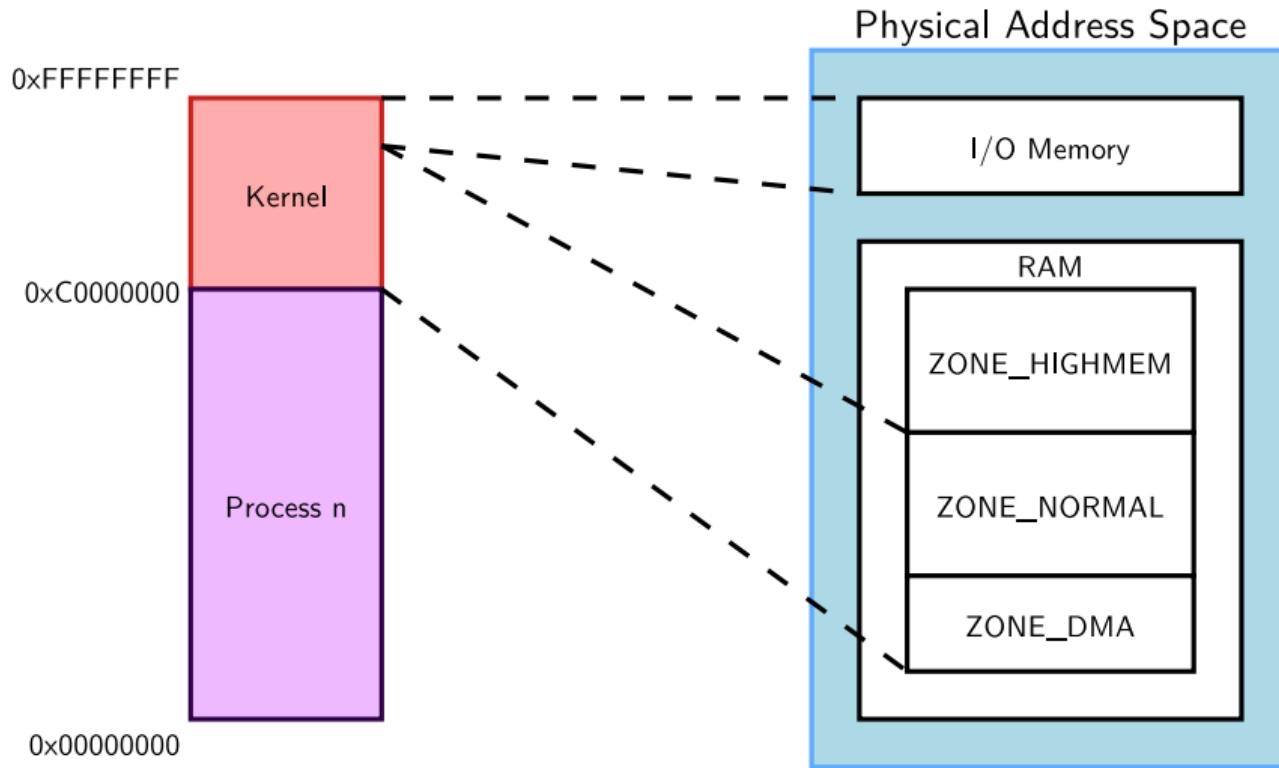
# Virtual Memory Organization



- ▶ 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





# Accessing more physical memory

- ▶ Only less than 1GB memory addressable directly through kernel virtual address space
- ▶ 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 1GB/3GB memory split to 2GB/2GB or 1GB/3GB (`CONFIG_VMSPLIT_2G` or `CONFIG_VMSPLIT_1G`) ⇒ reduce total user memory available for each process
  - ▶ Change for a 64 bit architecture ;-)  
See `Documentation/x86/x86_64/mm.txt` for an example.
  - ▶ 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 <http://lwn.net/Articles/75174/> for useful explanations



## 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) ⇒ 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.

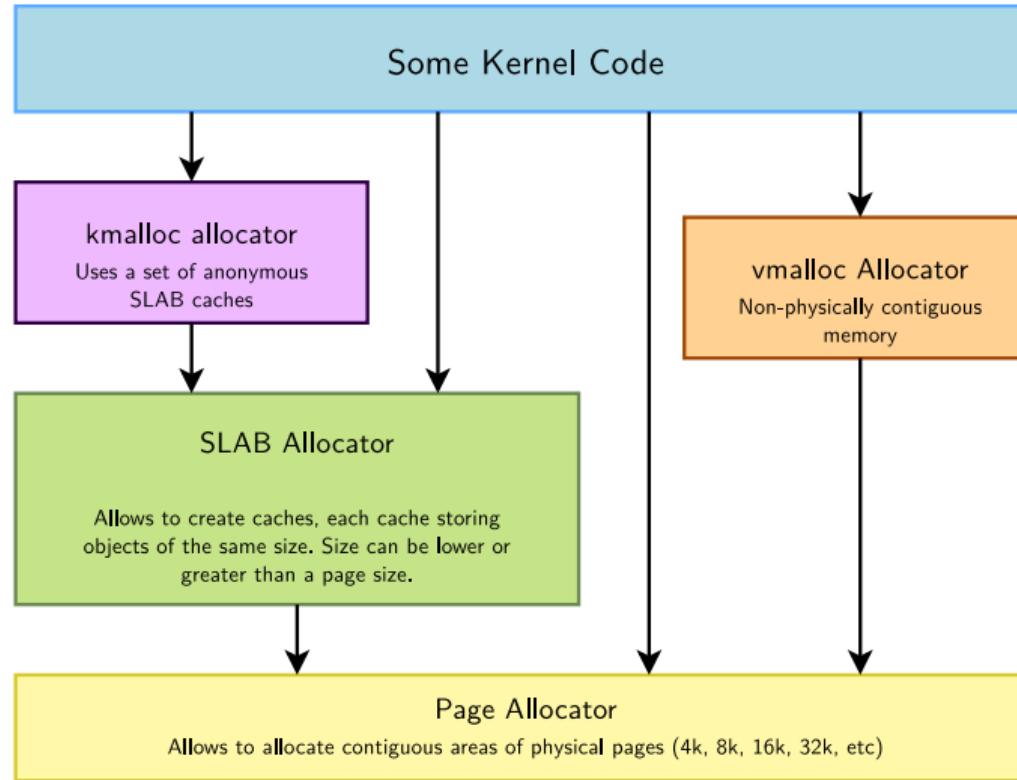


## Back to kernel memory

- ▶ Kernel memory allocators (see following slides) allocate physical pages, and kernel allocated memory cannot be swapped out, so no fault handling required for kernel memory.
- ▶ Most kernel memory allocation functions also return a kernel virtual address to be used within the kernel space.
- ▶ Kernel memory low-level allocator manages pages. This is the finest granularity (usually 4 KB, architecture dependent).
- ▶ However, the kernel memory management handles smaller memory allocations through its allocator (see *SLAB allocators* – used by `kmalloc()`).



# Allocators in the Kernel





- ▶ 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 virtually contiguous (of course), but also physically contiguous. 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.



# 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  $\log_2(\text{number\_of\_pages})$ . 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`

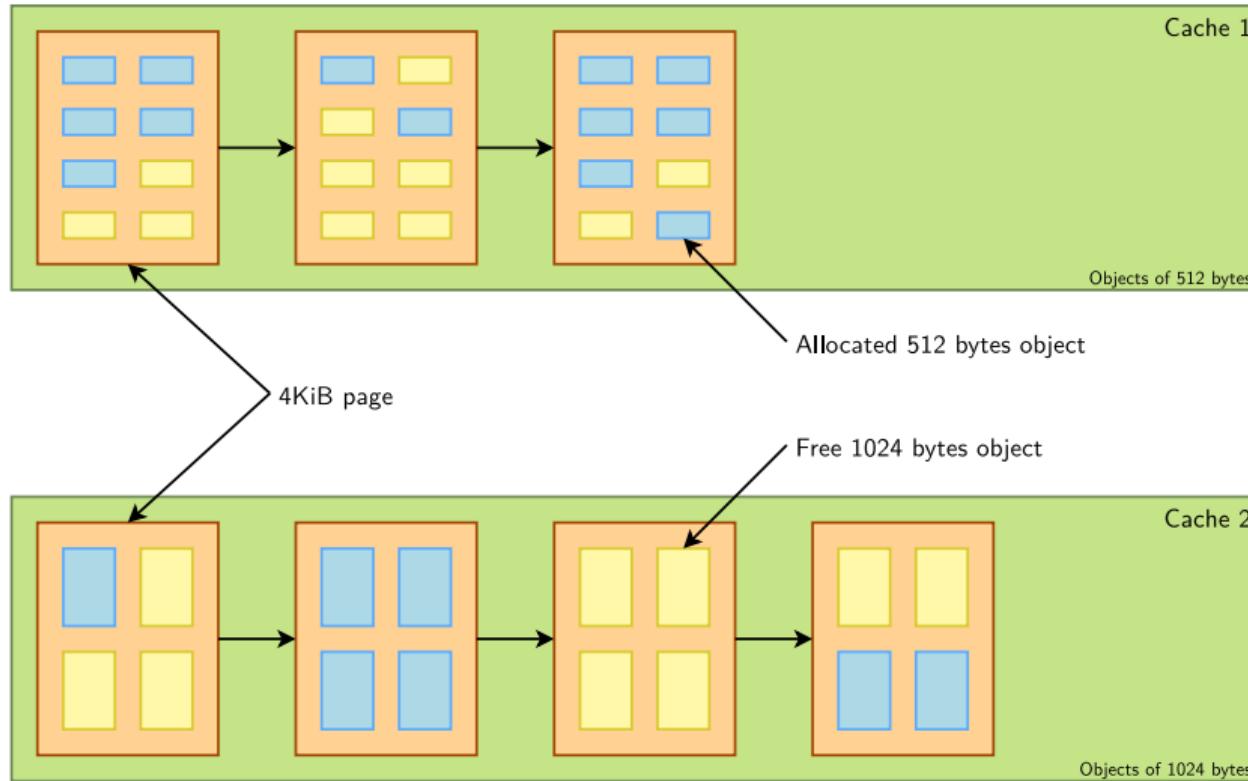


# SLAB Allocator 1/2

- ▶ The SLAB allocator allows to create *caches*, which contain a set of objects of the same size
- ▶ 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 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 4.20 on ARM: used in 48 defconfig files
- ▶ SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EXPERT`).  
Linux 4.20 on ARM: used in 7 defconfig files
- ▶ SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.  
Linux 4.20 on ARM: used in 0 defconfig files

## Choose SLAB allocator

SLAB

SLUB (Unqueued Allocator)  
SLOB (Simple Allocator)



# kmalloc Allocator

- ▶ The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- ▶ For small sizes, it relies on generic SLAB caches, named kmalloc-XXX in /proc/slabinfo
- ▶ For larger sizes, it relies on the page allocator
- ▶ The allocated area is guaranteed to be physically contiguous
- ▶ The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)
- ▶ It uses the same flags as the page allocator (GFP\_KERNEL, GFP\_ATOMIC, GFP\_DMA, etc.) with the same semantics.
- ▶ Maximum sizes, on x86 and arm (see <http://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.



## vmalloc Allocator

- ▶ The `vmalloc()` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. 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 <http://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
  - ▶ `vfree(void *addr);`



# Kernel memory debugging

- ▶ KASAN
  - ▶ Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.
  - ▶ Only available on `x86_64`, `arm64`, `s390` and `xtensa` so far (Linux 4.20 status), but will help to improve architecture independent code anyway.
  - ▶ See `dev-tools/kasan` for details.
- ▶ Kmemleak
  - ▶ Dynamic checker for memory leaks
  - ▶ This feature is available for all architectures.
  - ▶ See `dev-tools/kmemleak` for details.

Both 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  
<http://bit.ly/2Af1G2i> (video: <http://bit.ly/2Bwwv0C>)

The screenshot shows a presentation slide titled "Kernel Virtual Addresses (Small Mem)". The slide illustrates the memory hierarchy:

- Virtual Address Space:**
  - Kernel Virtual Addresses (0xFFFFFFFF (4GB))
  - Kernel Logical Addresses
  - Userspace Addresses
- Physical Address Space:**
  - Physical RAM (0x00000000)

A diagram shows the mapping from Kernel Virtual Addresses to Physical RAM, indicating the PAGE\_OFFSET. The presentation is part of a video titled "Virtual memory and Linux" by Alan Ott and Matt Porter, 2016, with a link to <http://bit.ly/2Af1G2i>. The video player interface includes a progress bar at 22:29 / 51:18 and various control icons.



# I/O Memory and Ports

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



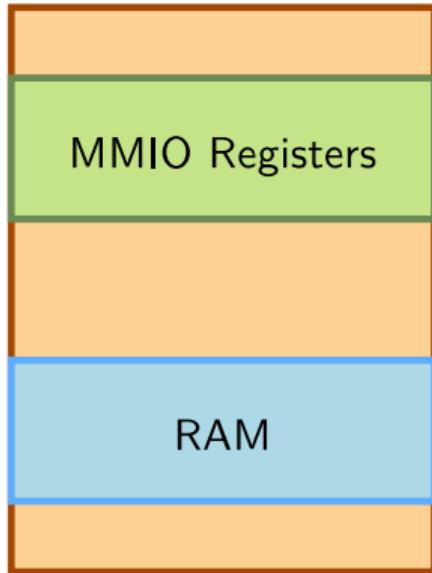


# Port I/O vs. Memory-Mapped I/O

- ▶ MMIO
  - ▶ Same address bus to address memory and I/O devices
  - ▶ Access to the I/O devices using regular instructions
  - ▶ Most widely used I/O method across the different architectures supported by Linux
- ▶ PIO
  - ▶ Different address spaces for memory and I/O devices
  - ▶ Uses a special class of CPU instructions to access I/O devices
  - ▶ Example on x86: IN and OUT instructions



# MMIO vs PIO



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



Separate I/O address space,  
accessed with specific instructions



## Requesting I/O ports

- ▶ Tells the kernel which driver is using which I/O ports
- ▶ Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.
- ▶ 

```
struct resource *request_region(
    unsigned long start,
    unsigned long len,
    char *name);
```
- ▶ Tries to reserve the given region and returns NULL if unsuccessful.
- ▶ `request_region(0x0170, 8, "ide1");`
- ▶ 

```
void release_region(
    unsigned long start,
    unsigned long len);
```



## /proc/ioports example (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
...

```



# Accessing I/O ports

- ▶ Functions to read/write bytes (b), word (w) and longs (l) to I/O ports:
  - ▶ `unsigned in[bwl](unsigned long port)`
  - ▶ `void out[bwl](value, unsigned long port)`
- ▶ And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!
  - ▶ `void ins[bwl](unsigned port, void *addr, unsigned long count)`
  - ▶ `void outs[bwl](unsigned port, void *addr, unsigned long count)`
- ▶ Examples
  - ▶ read 8 bits
    - ▶ `oldlcr = inb(baseio + UART_LCR)`
  - ▶ write 8 bits
    - ▶ `outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR)`



## Requesting I/O memory

- ▶ Functions equivalent to `request_region()` and `release_region()`, but for I/O memory.
- ▶ 

```
struct resource *request_mem_region(
    unsigned long start,
    unsigned long len,
    char *name);
```
- ▶ 

```
void release_mem_region(
    unsigned long start,
    unsigned long len);
```



## /proc/iomem example - ARM (Raspberry Pi, Linux 4.14)

```
00000000-3b3fffff : System RAM
00008000-00afffff : Kernel code
00c00000-00d468af : Kernel data
3f006000-3f006fff : dwc_otg
3f007000-3f007eff : /soc/dma@7e007000
3f00b840-3f00b84e : /soc/vchiq
3f00b880-3f00b8bf : /soc/mailbox@7e00b880
3f100000-3f100027 : /soc/watchdog@7e100000
3f101000-3f102fff : /soc/cprman@7e101000
3f200000-3f2000b3 : /soc/gpio@7e200000
3f201000-3f201fff : /soc/serial@7e201000
    3f201000-3f201fff : /soc/serial@7e201000
3f202000-3f2020ff : /soc/mmc@7e202000
3f212000-3f212007 : /soc/thermal@7e212000
3f215000-3f215007 : /soc/aux@0x7e215000
3f980000-3f98ffff : dwc_otg
```



# 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:

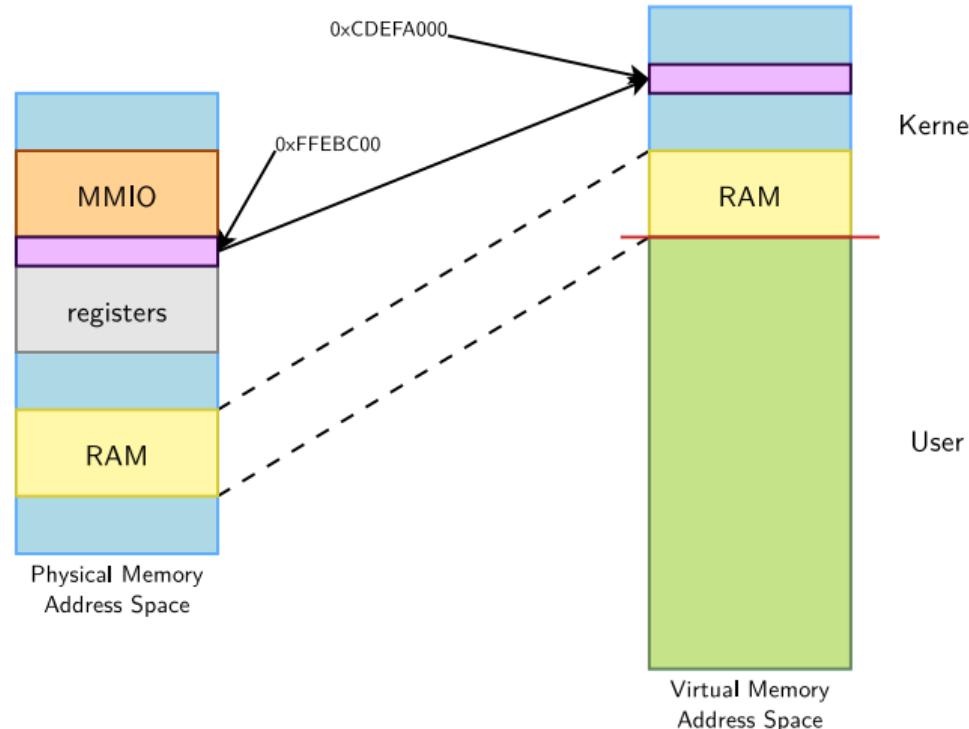
```
#include <asm/io.h>
```

```
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(0xFFEB00, 4096) = 0xCDEFA000`



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

- ▶ `devm_ioremap()`
- ▶ `devm_iounmap()`
- ▶ `devm_ioremap_resource()`
  - ▶ Takes care of both the request and remapping operations!



# 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, conversion being done automatically

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

- ▶ To do raw access, without endianness conversion

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

- ▶ Example

- ▶ 32 bits write

```
__raw_writel(1 << KS8695_IRQ_UART_TX,  
            membase + KS8695_INTST);
```



# Avoiding I/O access issues

- ▶ Caching on I/O ports or memory already disabled
- ▶ Use the macros, they do the right thing for your architecture
- ▶ The compiler and/or CPU can reorder memory accesses, which might cause troubles for your devices if 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 SMP.
- ▶ See `Documentation/memory-barriers.txt` for details



- ▶ 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, powerpc, s390 and unicore32: CONFIG\_STRICT\_DEVMEM option to restrict `/dev/mem` to non-RAM addresses, for security reasons (Linux 4.20 status).



# Practical lab - I/O Memory and Ports



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



## The misc subsystem

# The misc subsystem

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



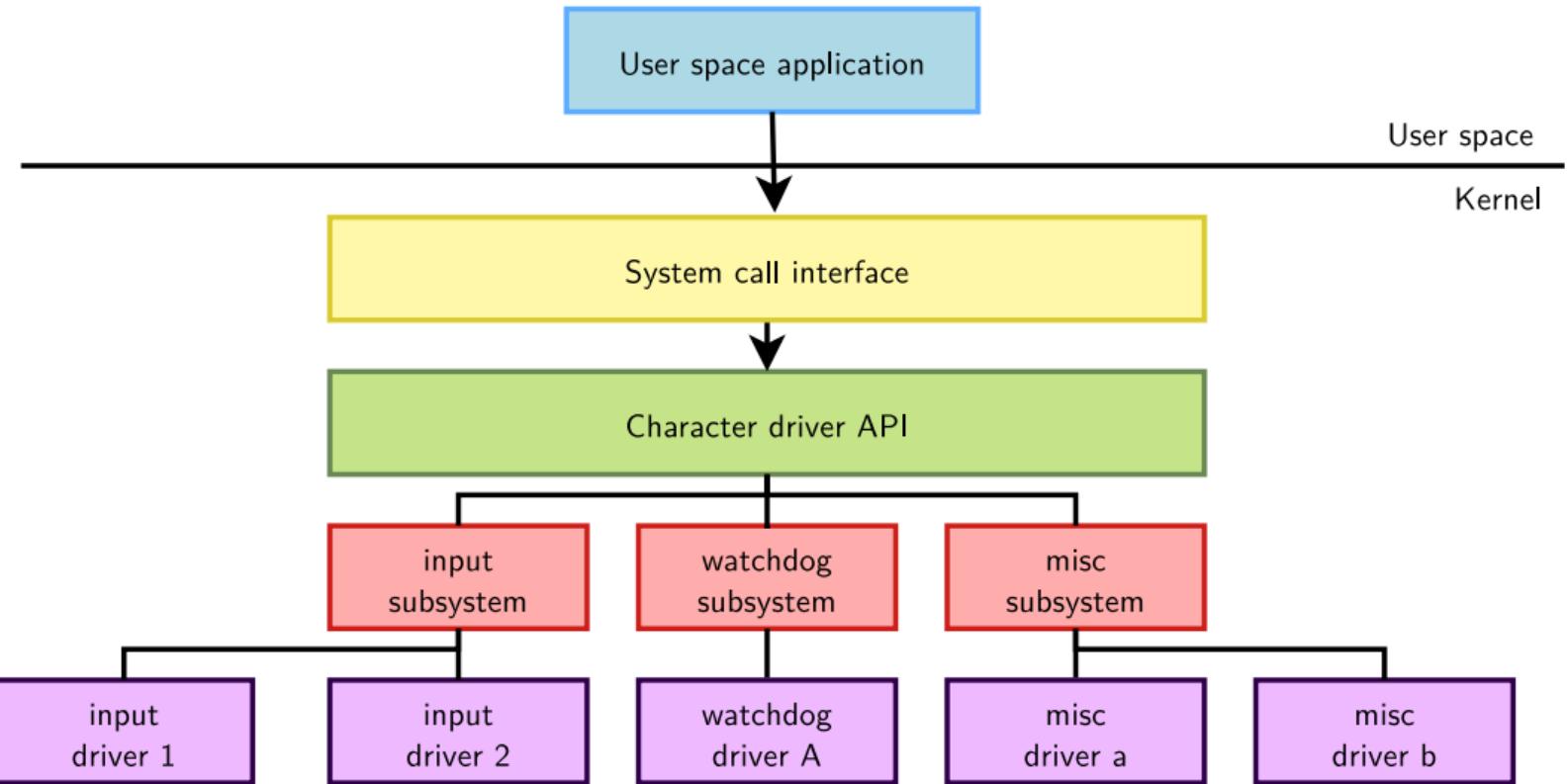


## Why a *misc* subsystem?

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



# Misc subsystem diagram





## Misc subsystem API (1/2)

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

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



## Misc subsystem API (2/2)

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

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



## 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

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





## Processes and scheduling



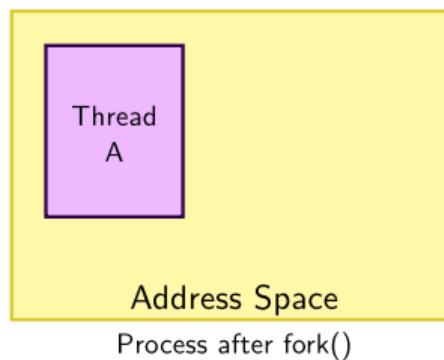
# Process, thread?

- ▶ 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.
  - ▶ One thread, entity known by the scheduler.
  - ▶ Upon creation, a process contains one thread
- ▶ 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()`

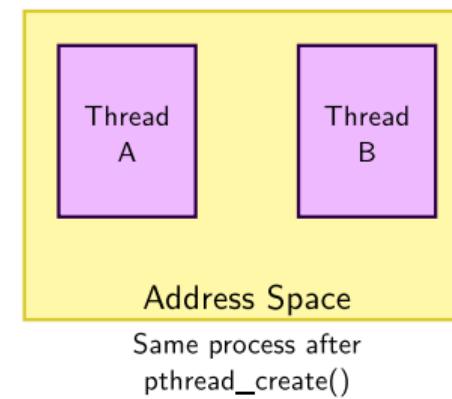


## Process, thread: kernel point of view

- ▶ The kernel represents each thread running in the system 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()`



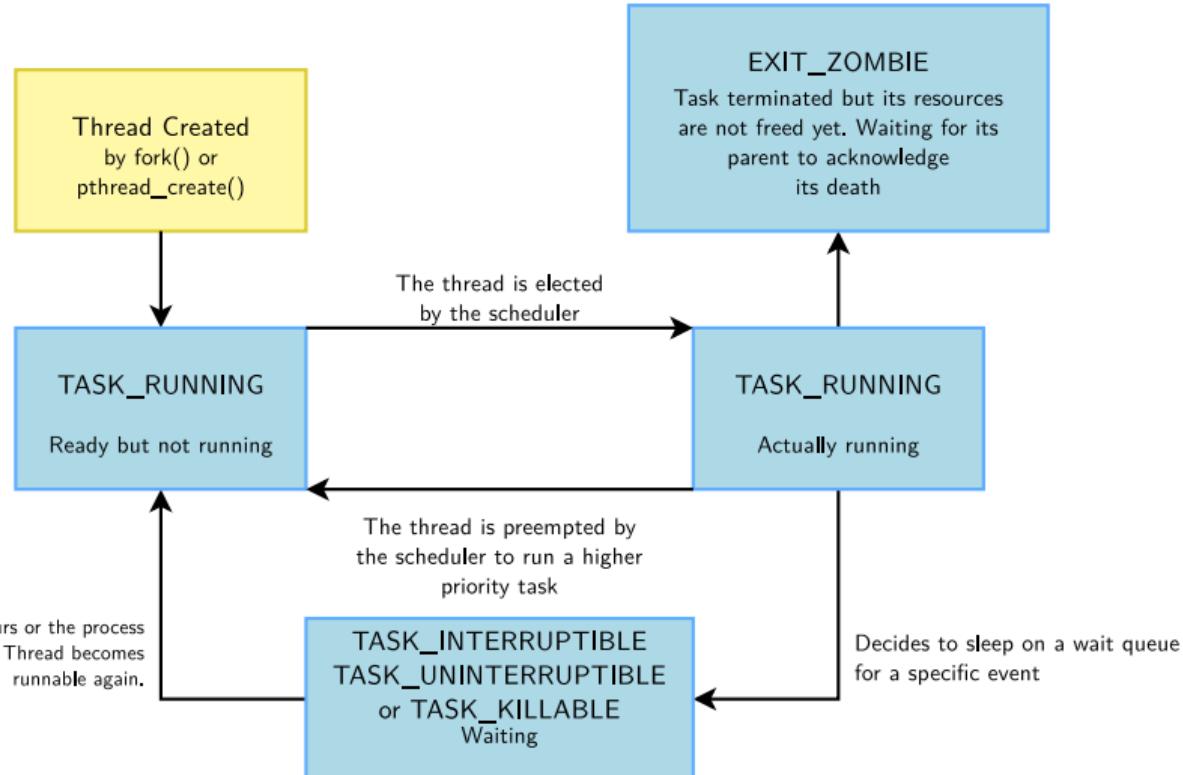
Process after `fork()`



Same process after  
`pthread_create()`

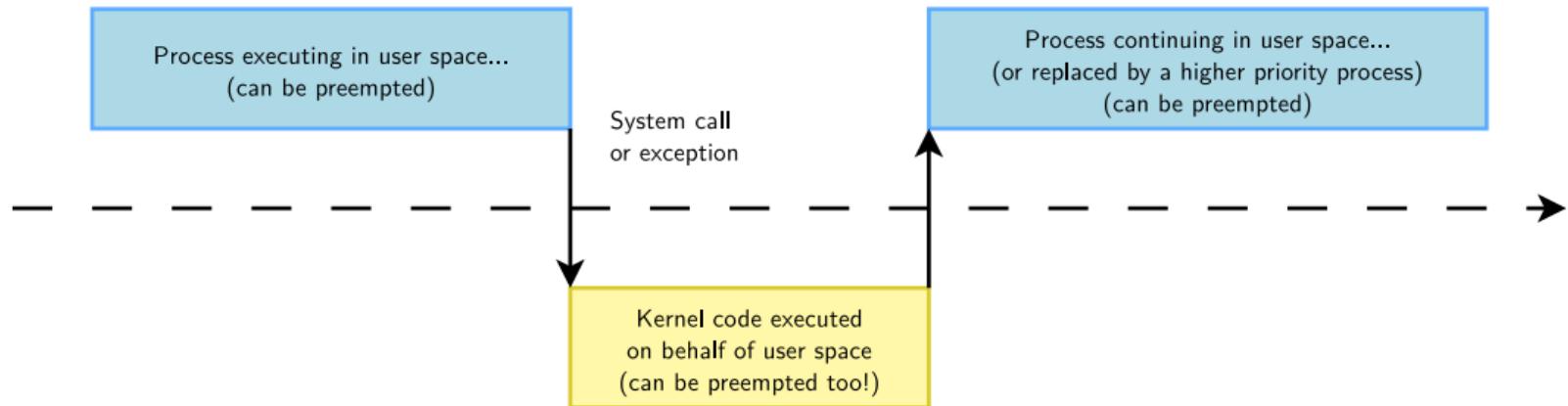


# A thread life





# Execution of system calls



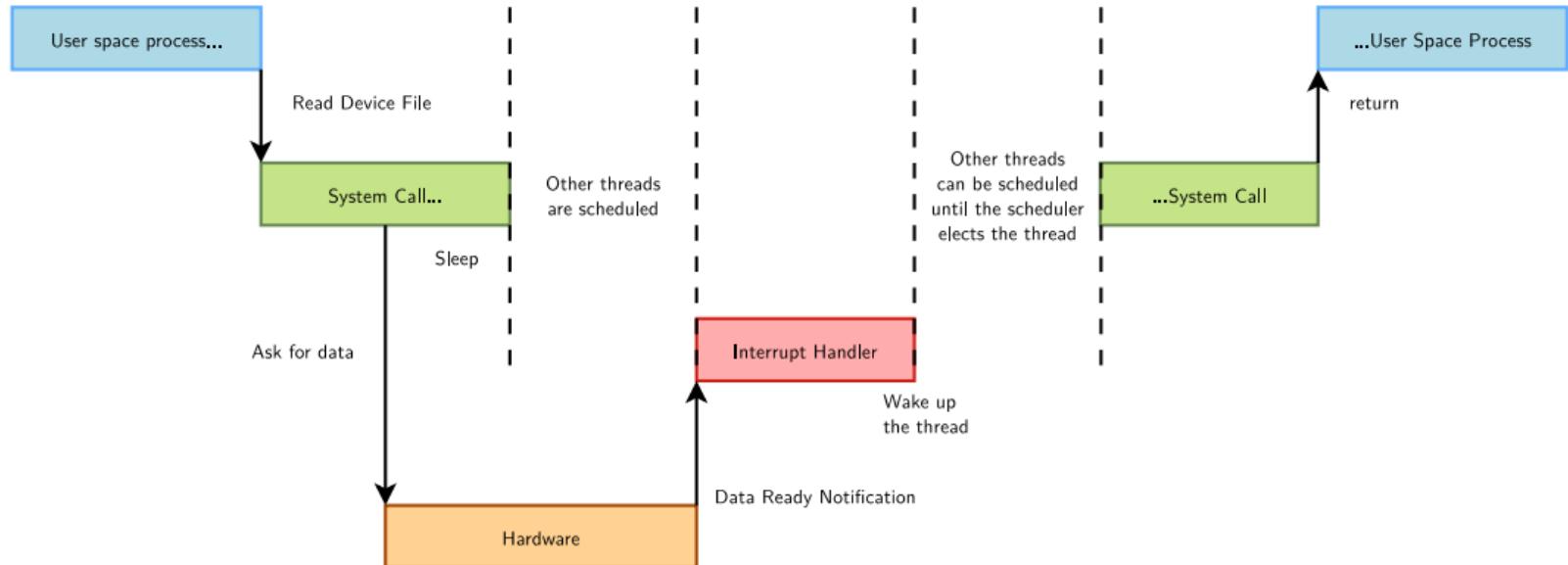
The execution of system calls takes place in the context of the thread requesting them.



## Sleeping



# Sleeping



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



## How to sleep 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 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);`
  - ▶ Can be interrupted by any signal. Returns -ERESTARTSYS if interrupted.



## How to sleep 3/3

- ▶ `int wait_event_timeout(queue, condition, timeout);`
  - ▶ Also stops sleeping when the task is woken up and the timeout expired. 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 - Example

```
ret = wait_event_interruptible
    (sonypi_device.fifo_proc_list,
     kfifo_len(sonypi_device.fifo) != 0);

if (ret)
    return ret;
```



# 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

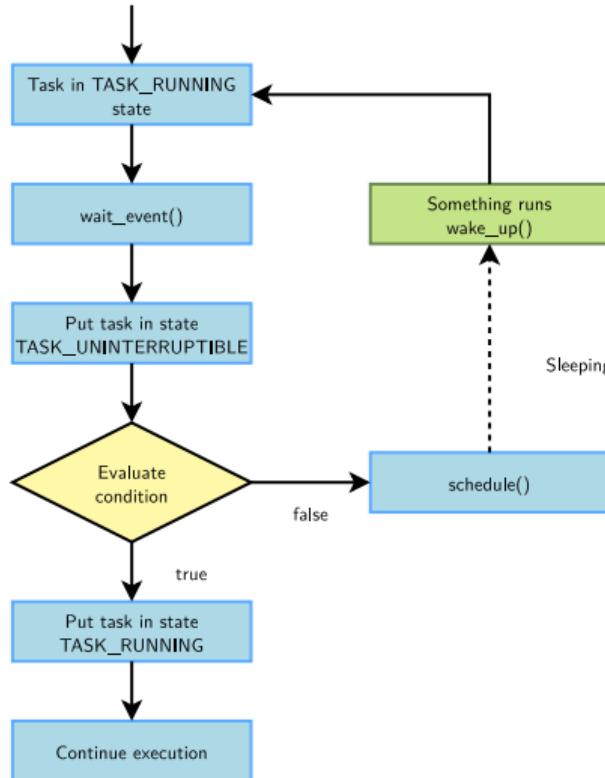


## Exclusive vs. non-exclusive

- ▶ `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.



## 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
- ▶ irq\_flags are option masks (see next slide)
- ▶ devname is the registered name (for /proc/interrupts)
- ▶ 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

Main `irq_flags` bit values

(can be combined, 0 when no flags are needed):

- ▶ `IRQF_SHARED`
  - ▶ The 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.



## 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 <http://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 4.14)

	CPU0	CPU1	CPU2	CPU3			
16:	0	0	0	0	bcm2836-timer	0 Edge	arch_timer
17:	34723454	46066453	21374961	21330046	bcm2836-timer	1 Edge	arch_timer
21:	0	0	0	0	bcm2836-pmu	9 Edge	arm-pmu
23:	2039429	0	0	0	ARMCTRL-level	1 Edge	3f00b880.mailbox
24:	2	0	0	0	ARMCTRL-level	2 Edge	VCHIQ doorbell
46:	0	0	0	0	ARMCTRL-level	48 Edge	bcm2708_fb dma
48:	0	0	0	0	ARMCTRL-level	50 Edge	DMA IRQ
50:	644766	0	0	0	ARMCTRL-level	52 Edge	DMA IRQ
59:	0	0	0	0	ARMCTRL-level	61 Edge	bcm2835-auxirq
62:	1157888875	0	0	0	ARMCTRL-level	64 Edge	dwc_otg, dwc_otg_pcd, ...
86:	641384	0	0	0	ARMCTRL-level	88 Edge	mmc0
87:	3	0	0	0	ARMCTRL-level	89 Edge	uart-pl011
FIQ:	usb_fiq						
IPI0:	0	0	0	0	CPU wakeup interrupts		
IPI1:	0	0	0	0	Timer broadcast interrupts		
IPI2:	3648739	13019827	4881211	4703599	Rescheduling interrupts		
IPI3:	5	10	11	8	Function call interrupts		
IPI4:	0	0	0	0	CPU stop interrupts		
IPI5:	7601406	13651564	2755152	2939328	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 real physical interrupt numbers can be seen in /sys/kernel/debug/irq\_domain\_mapping.



# 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.



## 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 to communicate with them.
- ▶ Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux

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

- ▶ handler, “hard IRQ” handler
- ▶ thread\_fn, executed in a thread



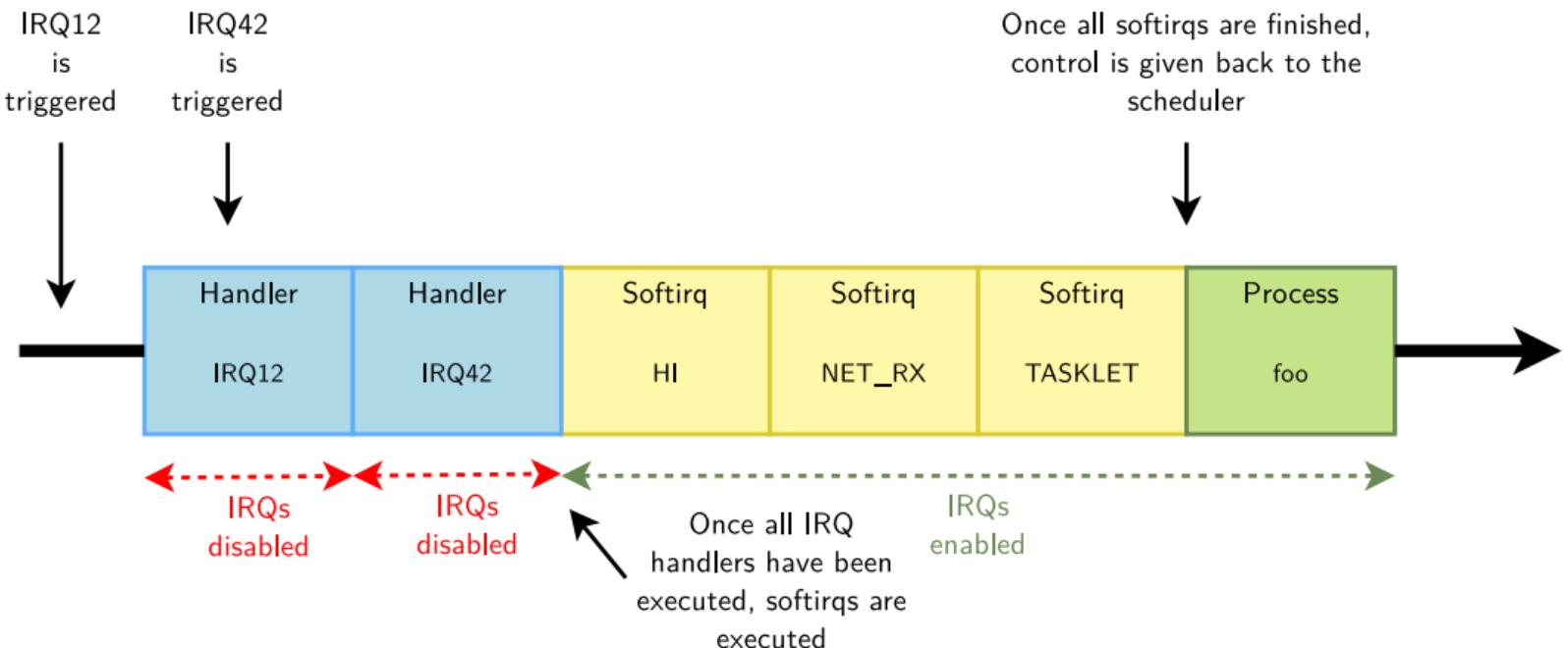
# Top half and bottom half processing

Splitting the execution of interrupt handlers in 2 parts

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



# Top half and bottom half diagram





- ▶ Softirqs are a form of bottom half processing
- ▶ The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- ▶ They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- ▶ The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by complete kernel subsystems (network, etc.)
- ▶ The list of softirqs is defined in `include/linux/interrupt.h`: HI, TIMER, NET\_TX, NET\_RX, BLOCK, BLOCK\_IOPOLL, TASKLET, SCHED, HRTIMER, RCU
- ▶ The HI and TASKLET softirqs are used to execute tasklets



# Tasklets

- ▶ Tasklets are executed within the HI and TASKLET 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 the TASKLET softirq
  - ▶ `tasklet_hi_schedule()` to get it executed in the HI softirq (higher 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;
    [...]
    platform_set_drvdata(pdev, sha_dd);
    [...]
    tasklet_init(&sha_dd->done_task, atmel_sha_done_task,
                (unsigned long)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.
- ▶ The function registered as workqueue is executed in a thread, which means:
  - ▶ All interrupts are enabled
  - ▶ Sleeping is allowed
- ▶ A workqueue 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.)



# 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

# Concurrent Access to Resources: Locking

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!



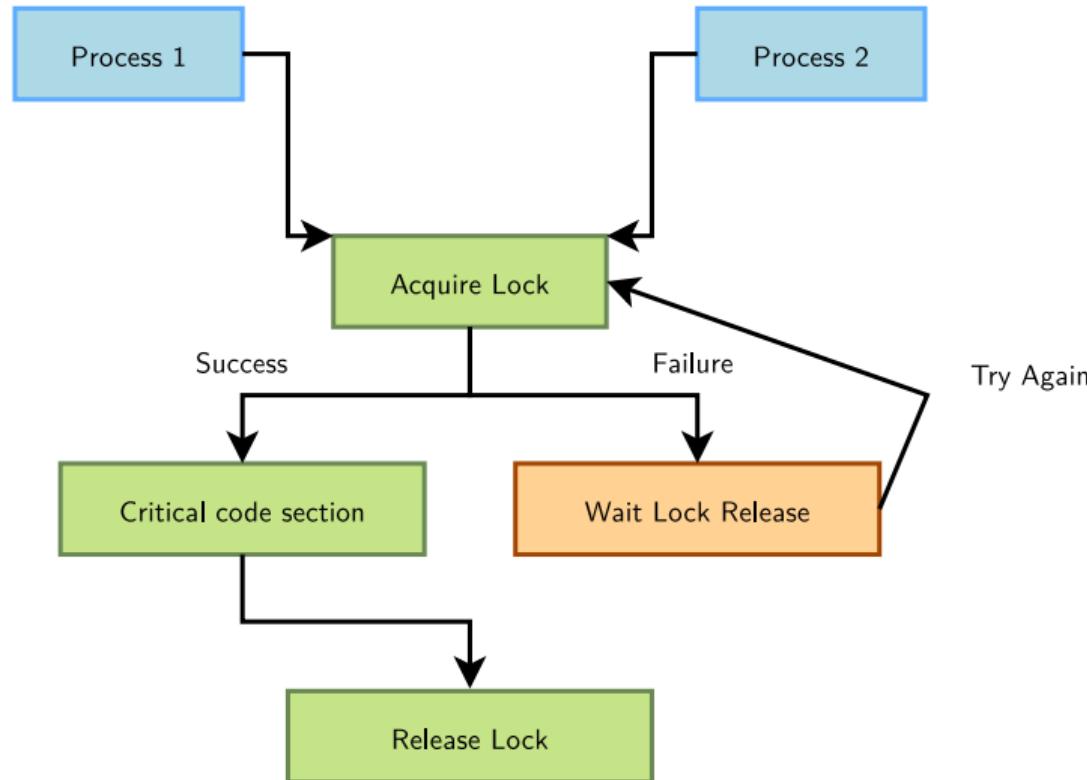


# Sources of concurrency issues

- ▶ In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- ▶ Concurrency arises because of
  - ▶ *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources (memory addresses, hardware registers...)
  - ▶ *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one system call 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

- ▶ The kernel's main locking primitive
- ▶ 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:
  - ▶ `DEFINE_MUTEX(name);`
- ▶ Or initializing a mutex dynamically:
  - ▶ `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.



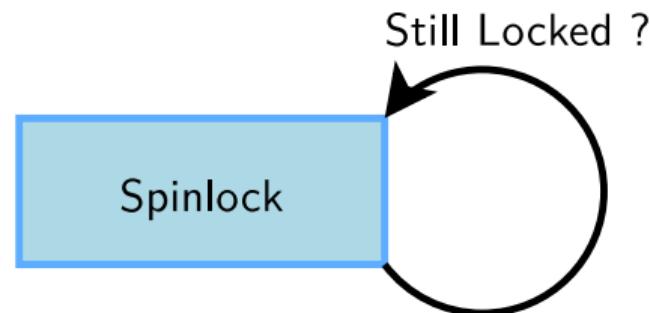
## Locking and Unlocking Mutexes 2/2

- ▶ `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.
- ▶ Spinlocks cause kernel preemption to be disabled on the CPU executing them.
- ▶ The critical section protected by a spinlock is not allowed to sleep.





# Initializing Spinlocks

- ▶ Statically
  - ▶ `DEFINE_SPINLOCK(my_lock);`
- ▶ Dynamically
  - ▶ `void spin_lock_init(spinlock_t *lock);`



# Using Spinlocks 1/2

- ▶ Several variants, depending on where the spinlock is called:
  - ▶ `void spin_lock(spinlock_t *lock);`
  - ▶ `void spin_unlock(spinlock_t *lock);`
    - ▶ Doesn't disable interrupts. Used for locking in process context (critical sections in which you do not want to sleep).
  - ▶ `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, to prevent preemption by interrupts.



## Using Spinlocks 2/2

- ▶ `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

- ▶ Spinlock structure embedded into struct uart\_port

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

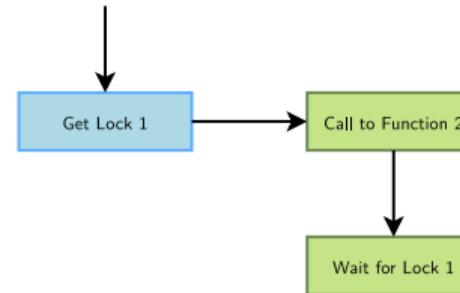
- ▶ Spinlock taken/released with protection against interrupts

```
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);  
}
```

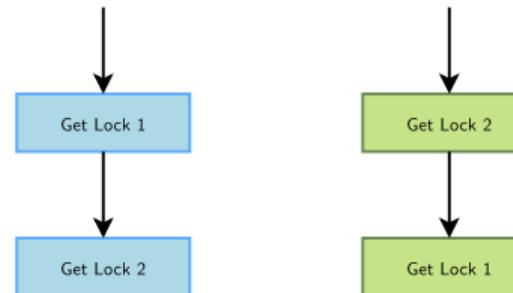


# Deadlock Situations

- ▶ They can lock up your system. Make sure they never happen!
- ▶ Don't call a function that can try to get access to the same lock



- ▶ Holding multiple locks is risky!





## Kernel lock validator

- ▶ From Ingo Molnar and Arjan van de Ven
  - ▶ 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 Documentation/locking/lockdep-design.txt 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 <http://en.wikipedia.org/wiki/RCU>).
  - ▶ 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.



## Practical lab - Locking



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



# Kernel debugging

# Kernel debugging

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# 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);
```

```
[ 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/device.h`
  - ▶ To be used in drivers integrated with the Linux device model
  - ▶ Example:

```
dev_info(&pdev->dev, "in probe\n");  
[ 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 above which messages are displayed on the console. Details in `admin-guide/kernel-parameters`.
  - ▶ 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 `Documentation/sysctl/kernel.txt`



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 <http://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

- ▶ 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

- ▶ 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

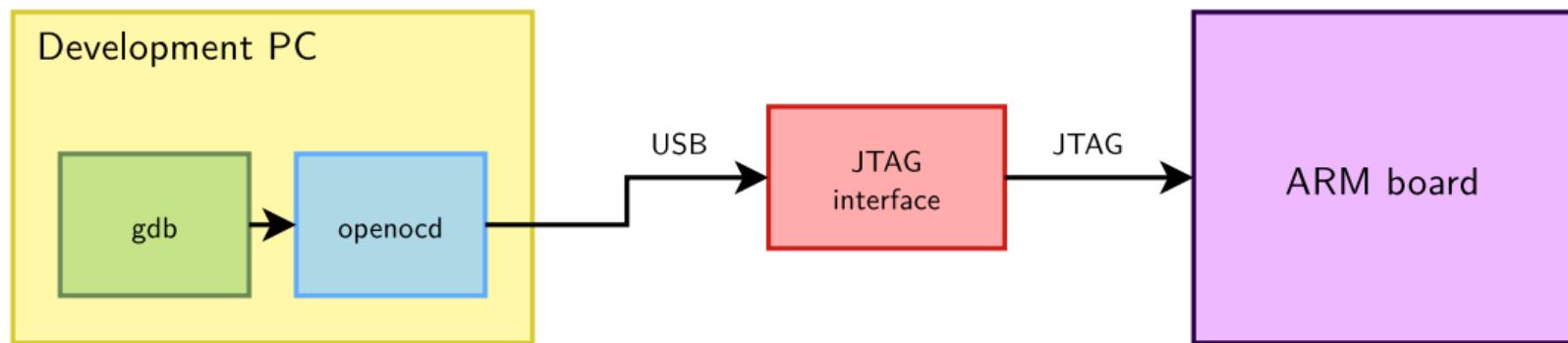
- ▶ Then also pass `kgdbwait` to the kernel: it makes kgdb wait for a debugger connection.
- ▶ Boot your kernel, and when the console is initialized, interrupt the kernel with a break character and then `g` in the serial console (see our *Magic SysRq* explanations).
- ▶ On your workstation, start gdb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



# Debugging with a JTAG interface

Two types of JTAG dongles

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





## 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

## Porting the Linux Kernel to an ARM Board

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# 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 focused on the ARM architecture only



# Architecture, CPU and Machine

- ▶ In the source tree, each architecture has its own directory
  - ▶ arch/arm/ for the ARM architecture
- ▶ This directory contains generic ARM code
  - ▶ boot, common, configs, kernel, lib, mm, nwfpe, vfp, oprofile, tools
- ▶ And many directories for different SoC families
  - ▶ mach-\* directories: mach-pxa for PXA CPUs, mach-imx for Freescale iMX CPUs, etc.
    - ▶ Before the ARM cleanup, these directories contained support for the SoC family (GPIO, clocks, pinmux, power management, interrupt controller, etc.) and for the various boards.
    - ▶ Nowadays, they contain a lot less code, essentially a small SoC description file, power management and SMP code.
- ▶ Some CPU types share some code, in directories named plat-\*
- ▶ Device Tree source files 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-<foo>`.
- ▶ Each board supported by the kernel was associated to an unique *machine ID*.
- ▶ The entire list of *machine ID* can be downloaded at  
<http://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.



# 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-<foo>`:
  - ▶ 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-<foo>` 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. 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





# Crystalfontz CFA-10036 Device Tree, header

- ▶ Mandatory Device Tree language definition

```
/dts-v1/
```

- ▶ Include the .dtsi file describing the SoC

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

- ▶ Start the root of the tree

```
/ {
```

- ▶ A human-readable string to describe the machine

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

- ▶ A list of *compatible* strings, from the most specific one to the most general one.  
Can be used by kernel code to do a SoC or board-specific check.

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



# Crystalfontz CFA-10036 Device Tree, chosen/memory

- ▶ Definition of the default *kernel command line*. Some additional operating-system specific entries can be added in chosen:

```
chosen {  
    bootargs = "console=ttyS0,115200 earlyprintk";  
};
```

- ▶ Definition of the size and location of the RAM:

```
memory {  
    device_type = "memory";  
    reg = <0x40000000 0x8000000>; /* 128 MB */  
};
```



- ▶ Start of the internal SoC peripherals.

```
apb@80000000 {  
    apbh@80000000 {  
        apbx@80040000 {
```

- ▶ The CFA-10036 has one debug UART, so the corresponding controller is enabled:

```
duart: serial@80074000 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&duart_pins_b>;  
    status = "okay";  
};
```



# Crystalfontz CFA-10036 Device Tree, Muxing

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

```
pinctrl@80018000 {
    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, LED

- ▶ One LED is connected to this platform. Note the reference to the `led_pins_cfa10036` muxing configuration.

```
leds {  
    compatible = "gpio-leds";  
    pinctrl-names = "default";  
    pinctrl-0 = <&led_pins_cfa10036>;  
  
    power {  
        gpios = <&gpio3 4 1>;  
        default-state = "on";  
    };  
};
```



# Crystalfontz CFA-10036 Device Tree, SD Card/USB

- ▶ The platform also has a USB port

```
usb0: usb@80080000 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&usb0_otg_cfa10036>;  
    status = "okay";  
};
```

- ▶ and an SD Card slot:

```
ssp0: ssp@80010000 {  
    compatible = "fsl,imx28-mmc";  
    pinctrl-names = "default";  
    pinctrl-0 = <&mmc0_4bit_pins_a  
                &mmc0_cd_cfg &mmc0_sck_cfg>;  
    bus-width = <4>;  
    status = "okay";  
};
```



# Crystalfontz CFA-10036 Device Tree, I2C bus

- ▶ An I2C bus, with a Solomon SSD1306 OLED display connected on it:

```
i2c0: i2c@80058000 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c0_pins_b>;
    clock-frequency = <400000>;
    status = "okay";

    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>;
    };
};
```



# 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, 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, which calls `of_platform_populate()`. This function travels through the Device Tree and instantiates all the devices.



# arch/arm/mach-mvebu/board-v7.c

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

    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
}

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,
    .smp             = smp_ops(armada_xp_smp_ops),
    .init_machine    = mvebu_dt_init,
    .init_irq         = mvebu_init_irq,
    .restart          = mvebu_restart,
    .reserve          = mvebu_memblock_reserve,
    .dt_compat        = armada_370_xp_dt_compat,
MACHINE_END
```



# Components of the minimal SoC support

The minimal SoC support consists in

- ▶ 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/`
- ▶ A interrupt controller driver, `drivers/irqchip/irq-armada-370-xp.c`
- ▶ A timer driver, `drivers/clocksource/time-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 device 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 driver, in `drivers/net/ethernet/marvell/mvneta.c`
- ▶ SATA driver, in `drivers/ata/sata_mv.c`
- ▶ I2C driver, in `drivers/i2c/busses/i2c-mv64xxx.c`
- ▶ SPI driver, in `drivers/spi/spi-orion.c`
- ▶ PCIe driver, in `drivers/pci/controller/pci-mvebu.c`
- ▶ USB 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!  
<http://bit.ly/2r8lHnE>
- ▶ Thomas Petazzoni, Your new ARM SoC Linux support check-list!  
<http://bit.ly/2ivqtDD>
- ▶ Our technical presentations on various kernel subsystems:  
<https://bootlin.com/docs/>



Embedded Linux Conference 2016

Your newer ARM64 SoC  
Linux check list!

Gregory CLEMENT  
*free electrons*  
gregory@free-electrons.com

© Copyright 2016 Free Electrons.  
Creative Commons BY-SA 3.0 license.  
Contributions, suggestions, corrections and translations are welcome.



Embedded Linux Conference 2013

Your new ARM SoC  
Linux support  
check-list!

Thomas Petazzoni  
*Free Electrons*  
thomas.petazzoni@free-electrons.com



Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <https://free-electrons.com> 1 / 10



# Power Management

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





# PM building blocks

- ▶ Several power management *building blocks*
  - ▶ Suspend and resume
  - ▶ CPUidle
  - ▶ Runtime power management
  - ▶ 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 get a reference to a clock
  - ▶ `clk_enable()` to start the clock
  - ▶ `clk_disable()` to stop the clock
  - ▶ `clk_put()` to free the clock source
  - ▶ `clk_get_rate()` to get the current rate



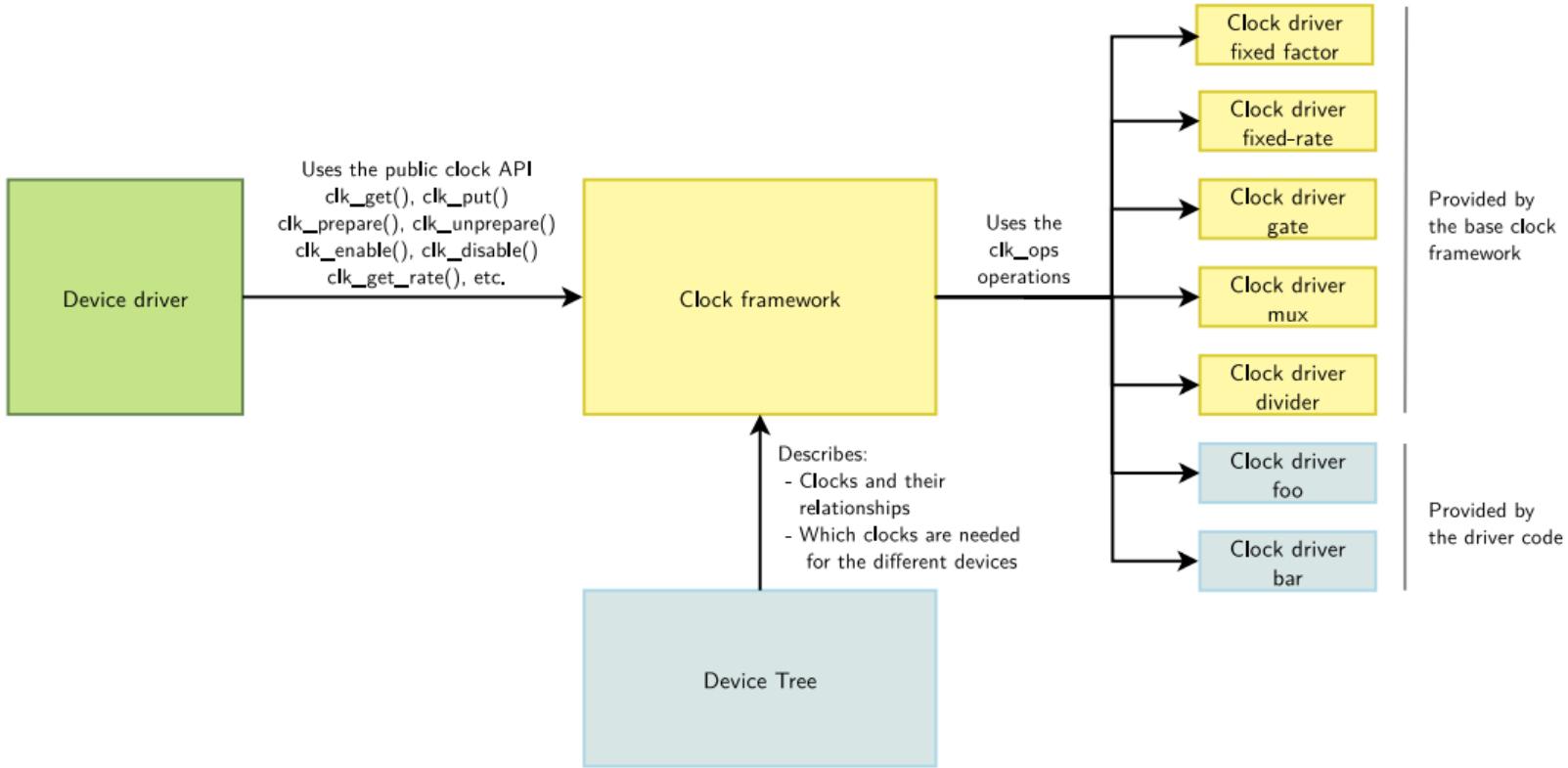
## Clock framework (2)

### The common clock framework

- ▶ Allows to declare the available clocks and their association to devices in the Device Tree (preferred) or statically in the source code (old method)
- ▶ 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



## 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 it with an **initialization** function using `CLK_OF_DECLARE()`
  - ▶ Example: `drivers/clk/at91/clk-pll.c`



## Suspend and resume

- ▶ Infrastructure in the kernel to support suspend and resume
- ▶ Platform hooks
  - ▶ `prepare()`, `enter()`, `finish()`, `valid()` in a `struct platform_suspend_ops` structure
  - ▶ Registered using the `suspend_set_ops()` function
  - ▶ See `arch/arm/mach-at91/pm.c`
- ▶ Device drivers
  - ▶ `suspend()` and `resume()` hooks in the `*_driver` structures (`struct platform_driver`, `struct usb_driver`, etc.)
  - ▶ See `drivers/net/ethernet/cadence/macb_main.c`



## Triggering suspend

- ▶ 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.
- ▶ The execution of this function can be triggered from user space. To suspend to RAM:
  - ▶ echo mem > /sys/power/state
- ▶ Can also use the s2ram program from <http://suspend.sourceforge.net/>
- ▶ Read kernel/power/suspend.c



## Runtime power management

- ▶ 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()`
- ▶ API and details on Documentation/power/runtime\_pm.txt
- ▶ See also Kevin Hilman's presentation at ELC Europe 2010:  
<http://elinux.org/images/c/cd/ELC-2010-khilman-Runtime-PM.odp>



# Saving power in the idle loop

- ▶ The idle loop is what you run when there's nothing left to run in the system.
- ▶ Implemented in all architectures in `arch/<arch>/kernel/process.c`
- ▶ Example to read: look for `cpu_idle` in `arch/arm/kernel/process.c`
- ▶ Each ARM cpu defines its own `arch_idle` function.
- ▶ The CPU can run power saving HLT instructions, enter NAP mode, and even disable the timers (tickless systems).
- ▶ See also [http://en.wikipedia.org/wiki/Idle\\_loop](http://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 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 (ladder and menu)
  - ▶ Available for x86/ACPI, not supported yet by all ARM cpus. (look for cpuidle\* files under arch/arm/)
  - ▶ See Documentation/cpuidle/ in kernel sources.



<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.



# 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 drivers 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/cpu-freq/` for useful explanations



# Regulator framework

- ▶ Modern embedded hardware 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 Documentation/power/regulator/ in kernel sources.



## 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.
- ▶ All other parts of the PM infrastructure should be already there: suspend / resume, cpuidle, cpu frequency and voltage scaling.



## Useful resources

- ▶ Documentation/power/ in the Linux kernel sources.
  - ▶ Will give you many useful details.
- ▶ <http://wiki.linaro.org/WorkingGroups/PowerManagement/>
  - ▶ Ongoing developments on the ARM platform.
- ▶ Introduction to kernel power management, Kevin Hilman, Linaro
  - ▶ [http://elinux.org/images/d/dd/Intro\\_Kernel\\_PM.svg](http://elinux.org/images/d/dd/Intro_Kernel_PM.svg)
  - ▶ <https://www.youtube.com/watch?v=Um0oRanCtzY>
- ▶ Tips and ideas for prolonging battery life (Amit Kucherla)
  - ▶ <http://j.mp/fVdxKh>



# The kernel development and contribution process

## The kernel development and contribution process

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





The kernel development and contribution process

# Linux versioning scheme and development process

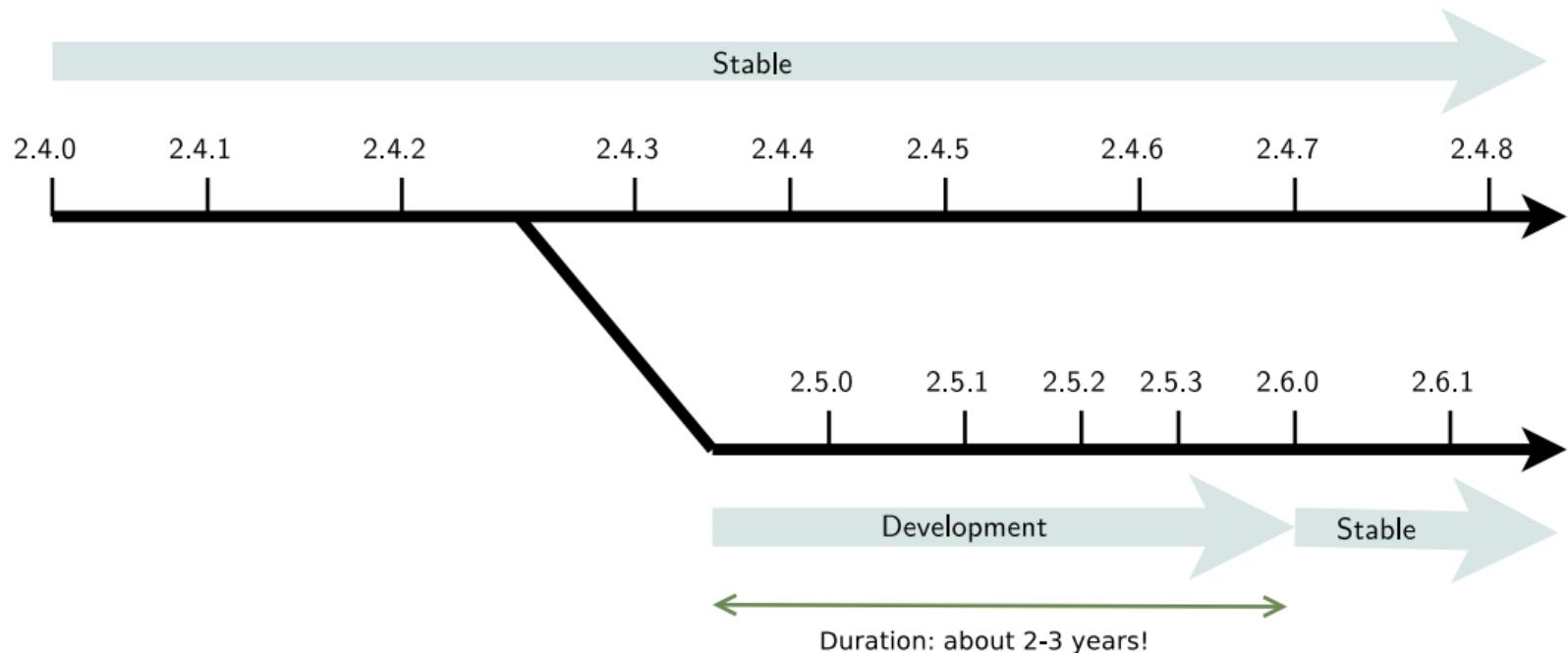


## Until 2.6 (1)

- ▶ One stable major branch every 2 or 3 years
  - ▶ Identified by an even middle number
  - ▶ Examples: 1.0.x, 2.0.x, 2.2.x, 2.4.x
- ▶ One development branch to integrate new functionalities and major changes
  - ▶ Identified by an odd middle number
  - ▶ Examples: 2.1.x, 2.3.x, 2.5.x
  - ▶ After some time, a development version becomes the new base version for the stable branch
- ▶ Minor releases once in while: 2.2.23, 2.5.12, etc.



## Until 2.6 (2)





## Changes since Linux 2.6

- ▶ Since 2.6.0 (Dec. 2003), kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make disruptive changes to existing subsystems.
- ▶ Since then, there has been no need to create a new development branch massively breaking compatibility with the stable branch.
- ▶ Thanks to this, **more features are released to users at a faster pace.**



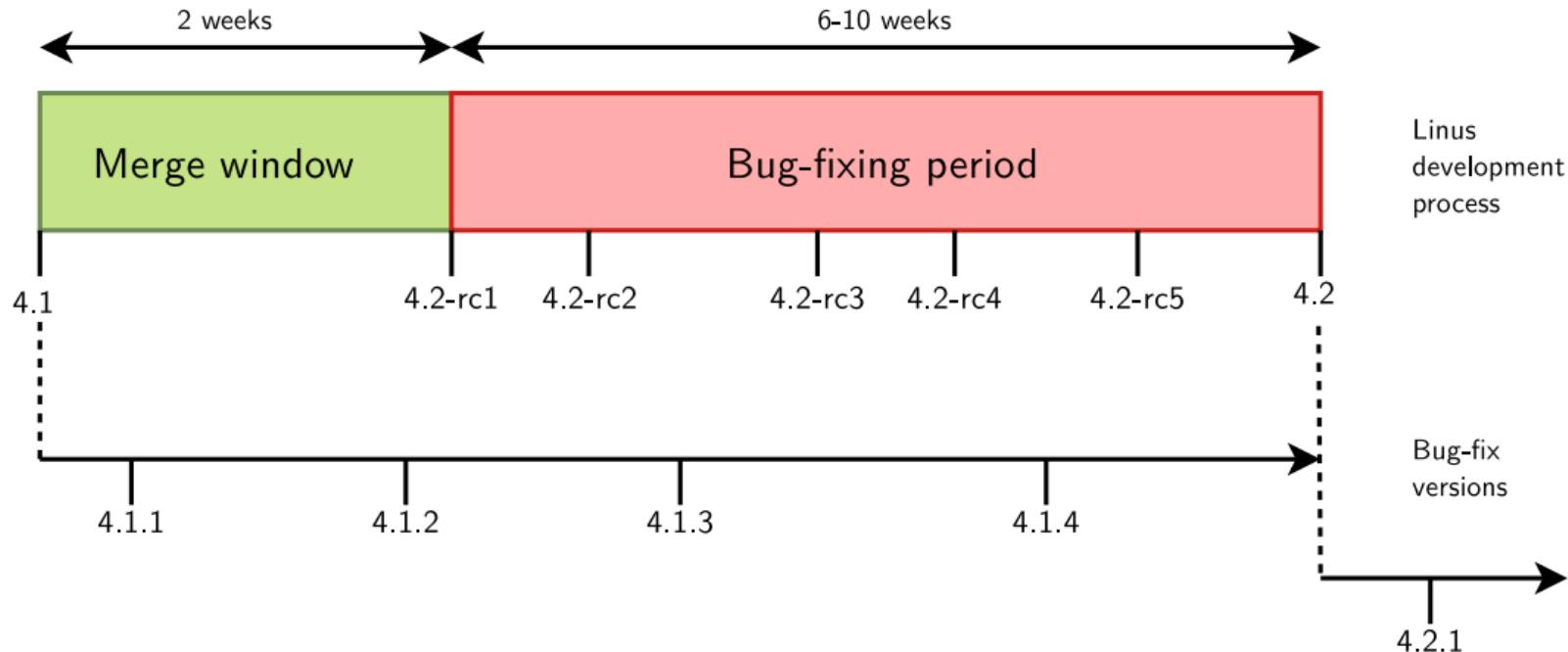
## Versions since 2.6.0

- ▶ From 2003 to 2011, the official kernel versions were named 2.6.x.
- ▶ Linux 3.0 was released in July 2011
- ▶ Linux 4.0 was released in April 2015
- ▶ This is only a change to the numbering scheme
  - ▶ Official kernel versions are now named x.y  
(3.0, 3.1, 3.2, ..., 3.19, 4.0, 4.1, etc.)
  - ▶ Stabilized versions are named x.y.z (3.0.2, 4.2.7, etc.)
  - ▶ It effectively only removes a digit compared to the previous numbering scheme



# New development model

## Using merge and bug fixing windows





# Need for long term support

- ▶ Issue: bug and security fixes only released for most recent stable kernel versions.  
Only LTS (*Long Term Support*) releases are supported for up to 6 years.
- ▶ Example at Google: starting from *Android O*, all new Android devices will have to run such an LTS kernel.
- ▶ You could also get long term support from a commercial embedded Linux provider.
- ▶ 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 <http://bit.ly/2hy1QYC>.

## Longterm release kernels

Source: follow the "Releases" link on <http://kernel.org/>

Version	Maintainer	Released	Projected EOL
4.9	Greg Kroah-Hartman	2016-12-11	Jan, 2019
4.4	Greg Kroah-Hartman	2016-01-10	Feb, 2022
4.1	Sasha Levin	2015-06-21	Sep, 2017
3.16	Ben Hutchings	2014-08-03	Apr, 2020
3.10	Willy Tarreau	2013-06-30	Oct, 2017
3.2	Ben Hutchings	2012-01-04	May, 2018



# 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

- ▶ <http://wiki.kernelnewbies.org/LinuxChanges>  
(some versions are missing)
- ▶ <http://lwn.net>
- ▶ <http://www.linux-arm.info>  
News about Linux on ARM, including kernel changes.
- ▶ <http://linuxfr.org>, for French readers



## 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 Documentation/SubmittingPatches for guidelines and <http://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: <http://www.youtube.com/watch?v=LLBrBBImJt4>
- ▶ How to Participate in the Linux Community (by Jonathan Corbet). A guide to the kernel development process <http://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 --to linux-arm-
 kernel@lists.infradead.org --to 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.



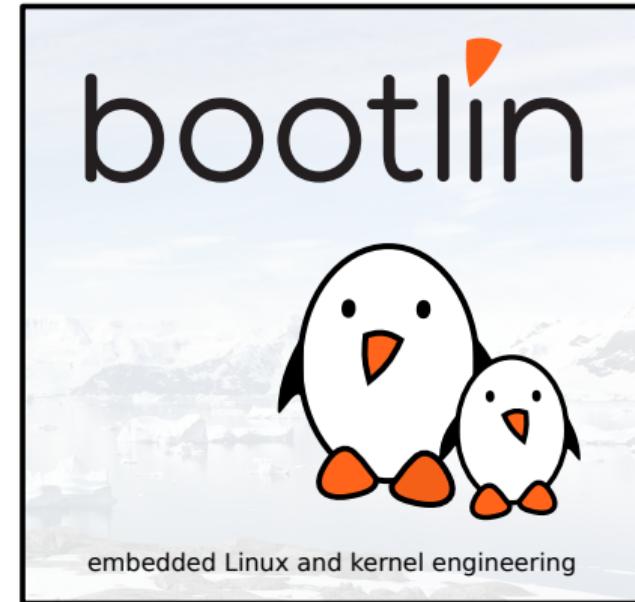
# Contribute to the Linux Kernel (6)

- ▶ 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

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.  
Corrections, suggestions, contributions and translations are welcome!





## Linux Weekly News

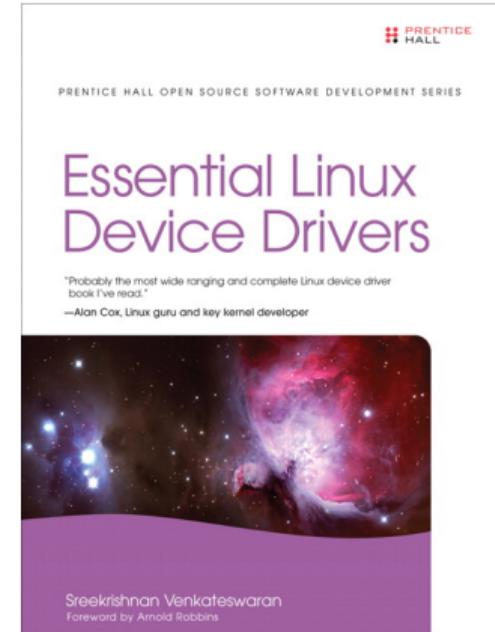
- ▶ <http://lwn.net/>
- ▶ The weekly digest off all Linux and free software information sources
- ▶ In depth technical discussions about the kernel
- ▶ Subscribe to finance the editors (\$7 / month)
- ▶ Articles available for non subscribers after 1 week.



# Useful Reading (1)

## Essential Linux Device Drivers, April 2008

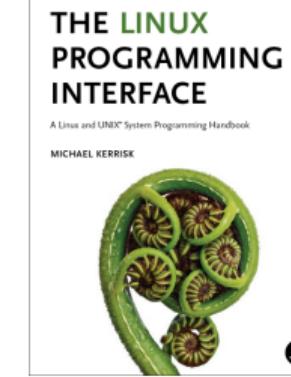
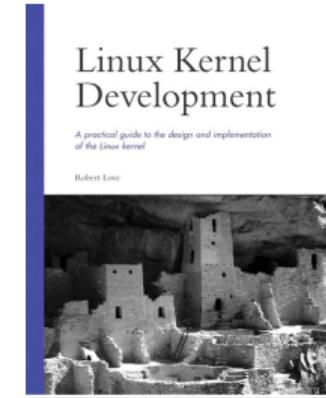
- ▶ <http://elinuxdd.com/>
- ▶ By Sreekrishnan Venkateswaran, an embedded IBM engineer with more than 10 years of experience
- ▶ Covers a wide range of topics not covered by LDD: serial drivers, input drivers, I2C, PCMCIA, PCI, USB, video drivers, audio drivers, block drivers, network drivers, Bluetooth, 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://bootlin.com/redir/lkd3-book.html>
  - ▶ 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
  - ▶ <http://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
  - ▶ <http://kernelnewbies.org/>
  - ▶ Glossary, articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.
- ▶ Kernel glossary
  - ▶ <http://kernelnewbies.org/KernelGlossary>



- ▶ Embedded Linux Conference:  
<http://embeddedlinuxconference.com/>
  - ▶ Organized by the Linux Foundation in the USA (February-April) and in Europe (October-November)
  - ▶ Very interesting kernel and user space topics for embedded systems developers.
  - ▶ Presentation slides and videos freely available
- ▶ Linux Plumbers: <http://linuxplumbersconf.org>
  - ▶ Conference on the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc.
- ▶ linux.conf.au: <http://linux.org.au/conf/>
  - ▶ In Australia / New Zealand
  - ▶ Features a few presentations by key kernel hackers.



## 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>
- ▶ Propose fixes for issues reported by the *Coccinelle* tool:  
make coccicheck
- ▶ Learn by reading the kernel code by yourself, ask questions and propose improvements.
- ▶ Implement and share drivers for your own hardware, of course!

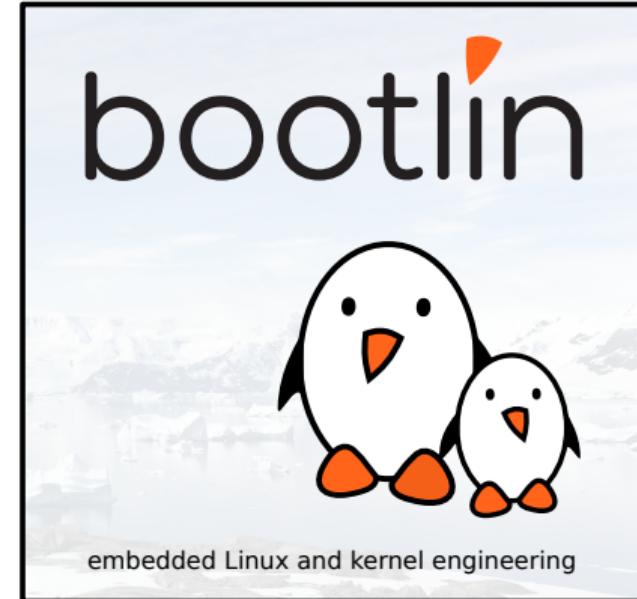


## Last slides

# Last slides

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Thank you!

And may the Source be with you

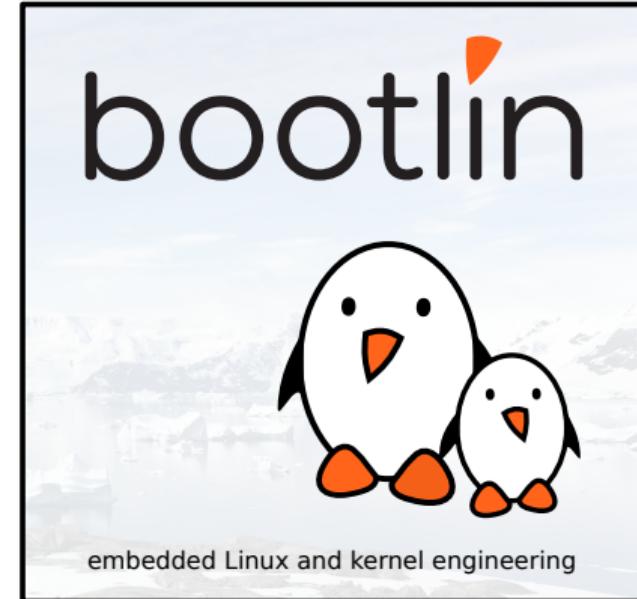


# Backup slides

# Backup slides

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!

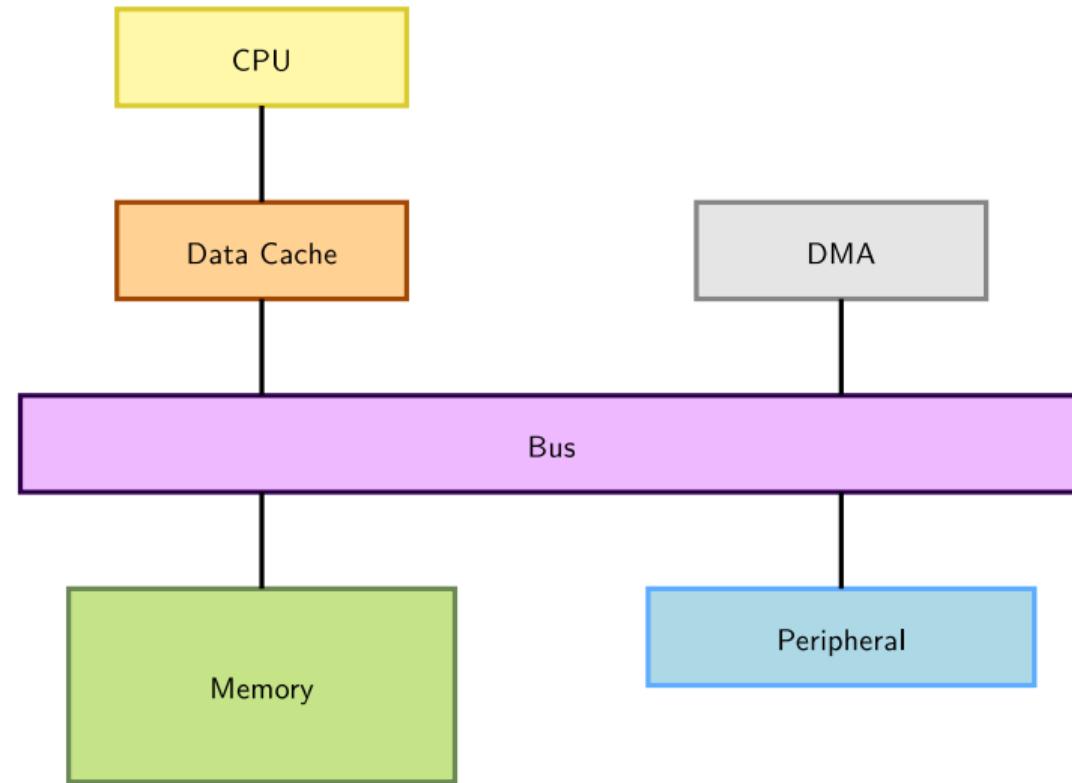




# DMA

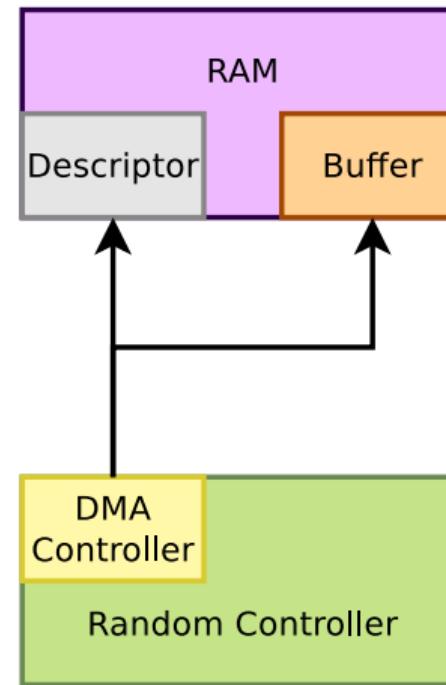


# DMA integration



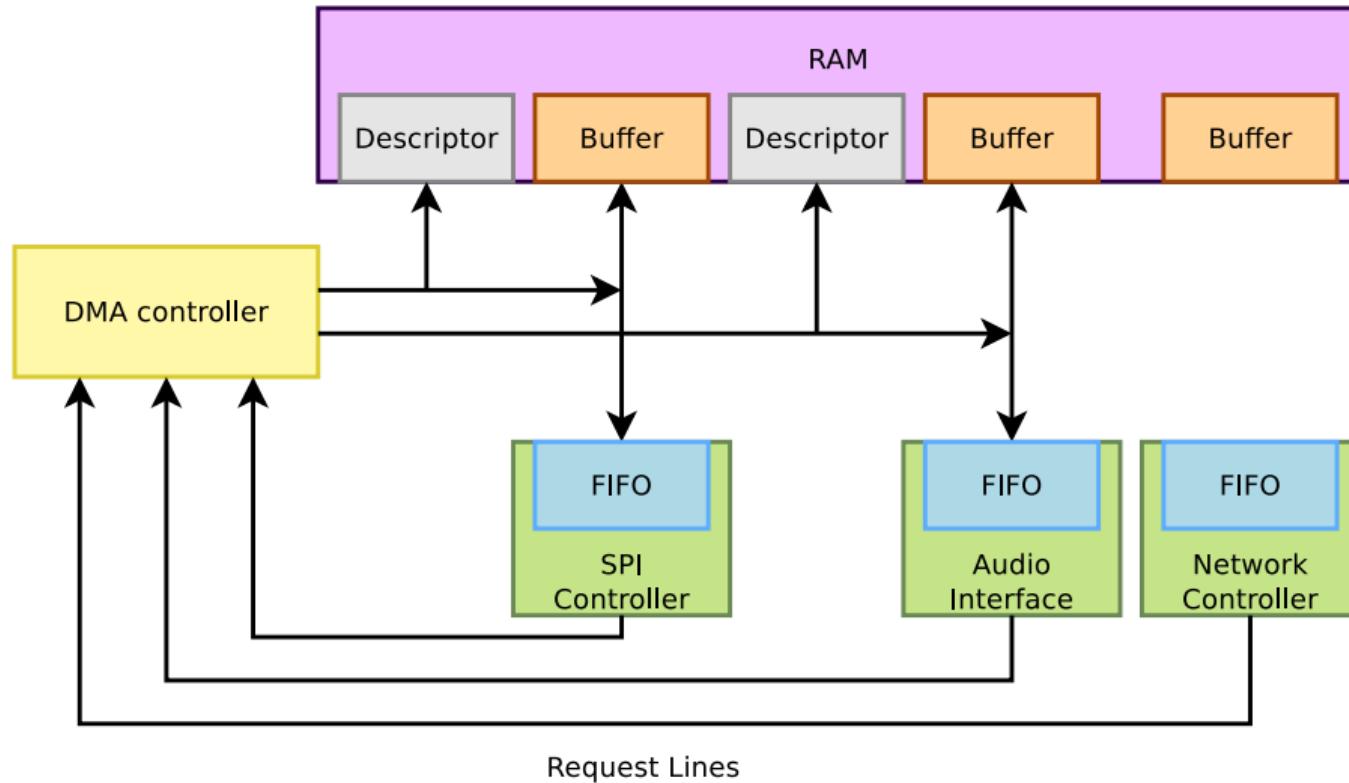


# Peripheral DMA



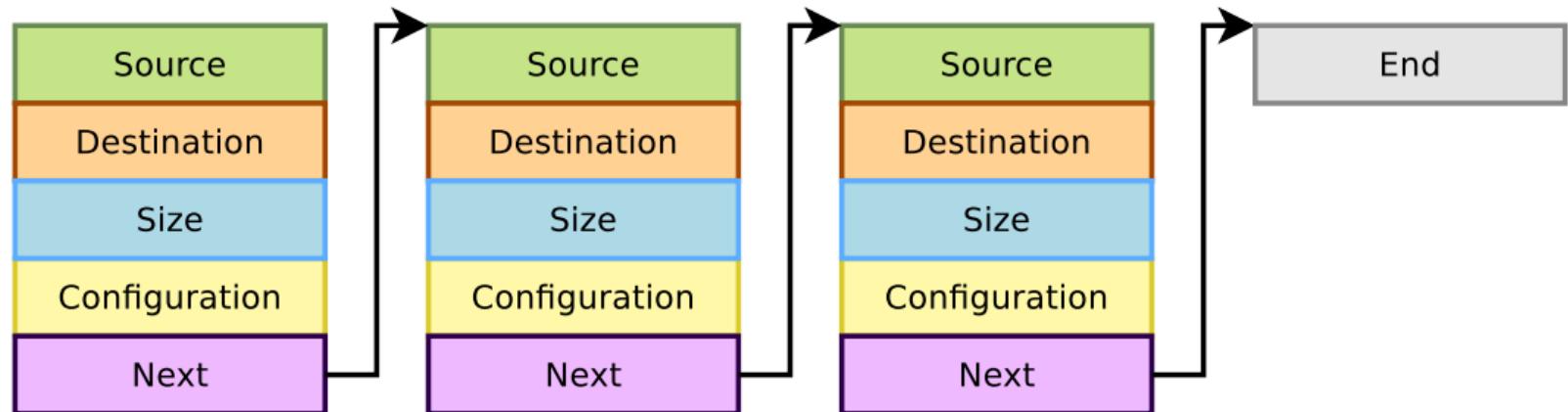


# DMA controllers





# DMA descriptors





## 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 to the CPU cache, so one need to take care of cache coherency (cache content vs memory content)
  - ▶ Either flush or invalidate the cache lines corresponding to the buffer accessed by DMA and processor at strategic 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 DMA buffer are committed.
- ▶ After DMA from device
  - ▶ Before drivers read from DMA buffer, need to make sure that memory caches are flushed.
- ▶ Bidirectional DMA
  - ▶ Need to flush caches before and after the DMA transfer.



The kernel DMA utilities can take care of:

- ▶ Either allocating a buffer in a cache coherent area,
- ▶ Or making sure caches are flushed when required,
- ▶ Managing the DMA mappings and IOMMU (if any).
- ▶ See `Documentation/DMA-API.txt` for details about 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 a buffer already allocated by the driver.
  - ▶ Mapping set up for each transfer. Keeps DMA registers free on the hardware.
  - ▶ The recommended solution.



# Allocating coherent mappings

The kernel takes care of both buffer allocation and mapping

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

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

void dma_free_coherent(struct device *dev,
    size_t size, void *cpu_addr, dma_addr_t handle);
```



## Setting up streaming mappings

Works on buffers already allocated by the driver

```
#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 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.



## 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)

In order to start a DMA transfer, 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 addresses, 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)

- ▶ 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*

Details in kernel documentation: [driver-api/dmaengine/client](#)



# Backup slides

# 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>`

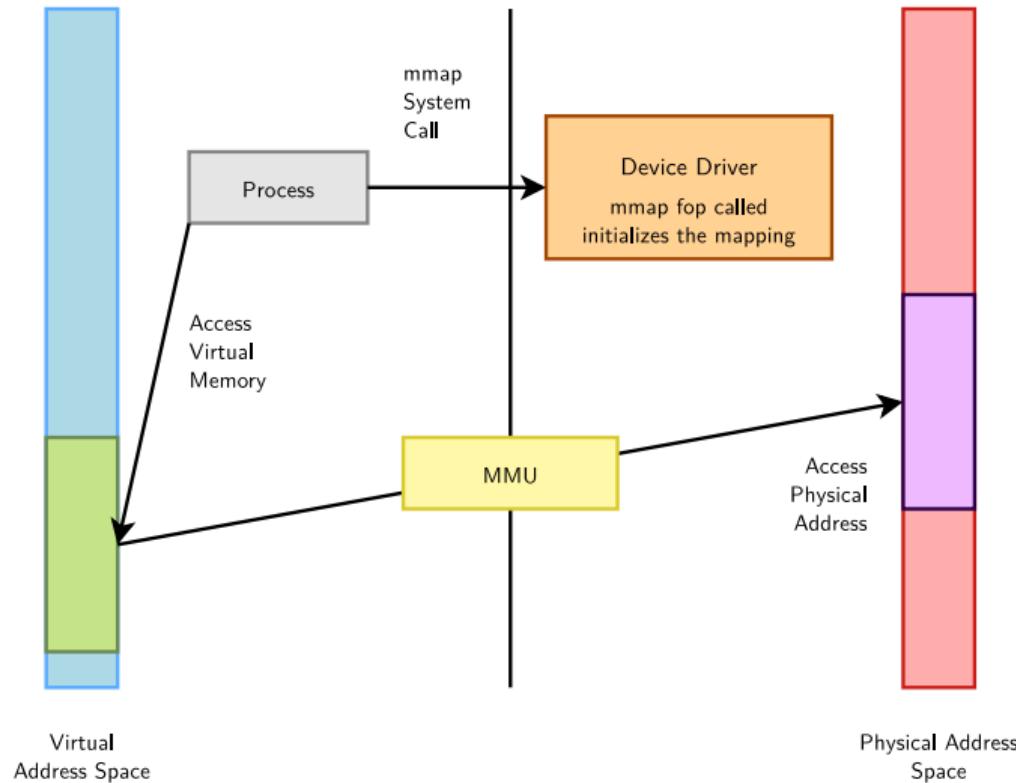


## /proc/<pid>/maps

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

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

    if (size > ACME_SIZE)
        return -EINVAL;

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

    return 0;
}
```



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



## 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



## Introduction to Git



# What is Git?

- ▶ A version control system, like CVS, SVN, Perforce or ClearCase
- ▶ Originally developed for the Linux kernel development, now used by a large number of projects, including U-Boot, GNOME, Buildroot, uClibc and many more
- ▶ Contrary to CVS or SVN, Git is a distributed version control system
  - ▶ No central repository
  - ▶ Everybody has a local repository
  - ▶ Local branches are possible, and very important
  - ▶ Easy exchange of code between developers
  - ▶ Well-suited to the collaborative development model used in open-source projects



# Install and Setup

- ▶ Git is available as a package in your distribution
  - ▶ `sudo apt install git`
- ▶ Everything is available through the git command
  - ▶ git has many commands, called using `git <command>`, where `<command>` can be `clone`, `checkout`, `branch`, etc.
  - ▶ Help can be found for a given command using `git help <command>`
- ▶ Set up your name and e-mail address
  - ▶ They will be referenced in each of your commits
  - ▶ `git config --global user.name 'My Name'`
  - ▶ `git config --global user.email me@mydomain.net`



## Clone a Repository

- ▶ To start working on a project, you use Git's clone operation.
- ▶ With CVS or SVN, you would have used the checkout operation, to get a working copy of the project (latest version)
- ▶ With Git, you get a full copy of the repository, including the history, which allows to perform most of the operations offline.
- ▶ Cloning Linus Torvalds' Linux kernel repository `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- ▶ `git://` is a special Git protocol. Most repositories can also be accessed using `http://`, but this is slower.
- ▶ After cloning, in `linux/`, you have the repository and a working copy of the master branch.



# Explore the History

- ▶ git log will list all the commits. The latest commit is the first.

```
commit 4371ee353c3fc41aad9458b8e8e627eb508bc9a3
Author: Florian Fainelli <florian@openwrt.org>
Date: Mon Jun 1 02:43:17 2009 -0700
```

MAINTAINERS: take maintainership of the cpmac Ethernet driver

This patch adds me as the maintainer of the CPMAC (AR7) Ethernet driver.

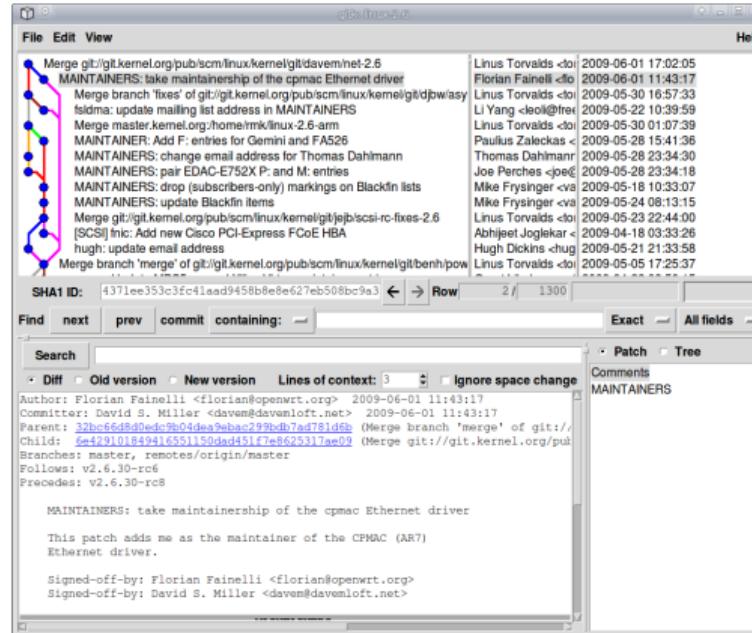
Signed-off-by: Florian Fainelli <florian@openwrt.org>  
Signed-off-by: David S. Miller <davem@davemloft.net>

- ▶ git log -p will list the commits with the corresponding diff
- ▶ The history in Git is not linear like in CVS or SVN, but it is a graph of commits
  - ▶ Makes it a little bit more complicated to understand at the beginning
  - ▶ But this is what allows the powerful features of Git (distributed, branching, merging)



# Visualize the History: gitk

- ▶ gitk is a graphical tool that represents the history of the current Git repository
- ▶ Can be installed from the gitk package





## Visualize the History: cgit

- ▶ Another great tool is cgit, a web interface to Git. For the kernel sources, it is used on <http://git.kernel.org/>

index : kernel/git/torvalds/linux.git

Linux kernel source tree

master switch Linus Torvalds

summary refs log tree commit diff stats log msg search

author Ezequiel Garcia <ezequiel.garcia@free-electrons.com> 2013-11-14 21:25:28 (GMT)  
committer Brian Norris <computersforpeace@gmail.com> 2014-01-03 19:22:12 (GMT)  
commit 776f265e279744e3f327cc3e7eb378046311373 (patch)  
tree 545d78d80e8c225fe0bd8c70f5e9fb66ab56760  
parent 56704857ac9a1044ec3cfe0ff95dc0bdbd9d58 (diff)

**mtd: nand: pxa3xx: Add bad block handling**

Add support for flash-based bad block table using Marvell's custom in-flash bad block table layout. The support is enabled a 'flash\_bbt' platform data or device tree parameter.

Signed-off-by: Ezequiel Garcia <ezequiel.garcia@free-electrons.com>  
Tested-by: Daniel Mack <zongque@gmail.com>  
Signed-off-by: Brian Norris <computersforpeace@gmail.com>

**Diffstat**

rw-r--r-- Documentation/devicetree/bindings/mtd/pxa3xx-nand.txt	2
rw-r--r-- drivers/mtd/nand/pxa3xx_nand.c	37
rw-r--r-- include/linux/platform_data/mtd-nand-pxa3xx.h	3

3 files changed, 42 insertions, 0 deletions

diff --git a/Documentation/devicetree/bindings/mtd/pxa3xx-nand.txt b/Documentation/devicetree/bindings/mtd/pxa3xx-nand.txt  
index bed8390..86ea56 100644  
++ a/Documentation/devicetree/bindings/mtd/pxa3xx-nand.txt  
++ b/Documentation/devicetree/bindings/mtd/pxa3xx-nand.txt  
@@ -15,6 +15,8 @@ Optional properties:  
- marvell,nand-keep-config: Set to keep the NAND controller config as set  
by the bootloader  
- num-CS: Number of chipselect lines to use  
+ hand-on-flash-bbt: boolean to enable on flash bbt option if  
+ not present false

diff options context: 3 space: include mode: unified



## Update your Repository

- ▶ The repository that has been cloned at the beginning will change over time
- ▶ Updating your local repository to reflect the changes of the remote repository will be necessary from time to time
- ▶ `git pull`
- ▶ Internally, does two things
  - ▶ Fetch the new changes from the remote repository (`git fetch`)
  - ▶ Merge them in the current branch (`git merge`)



# Tags

- ▶ The list of existing tags can be found using
  - ▶ `git tag -l`
- ▶ To check out a working copy of the repository at a given tag
  - ▶ `git checkout <tagname>`
- ▶ To get the list of changes between a given tag and the latest available version
  - ▶ `git log v4.19..master`
- ▶ List of changes with diff on a given file between two tags
  - ▶ `git log -p v4.18..v4.19 MAINTAINERS`
- ▶ With gitk
  - ▶ `gitk v4.19..master`



# Branches

- ▶ To start working on something, the best is to make a branch
  - ▶ It is local-only, nobody except you sees the branch
  - ▶ It is fast
  - ▶ It allows to split your work on different topics, try something and throw it away
  - ▶ It is cheap, so even if you think you're doing something small and quick, do a branch
- ▶ Unlike other version control systems, Git encourages the use of branches. Don't hesitate to use them.



# Branches

- ▶ Create a branch
  - ▶ `git branch <branchname>`
- ▶ Move to this branch
  - ▶ `git checkout <branchname>`
- ▶ Both at once (create and switch to branch)
  - ▶ `git checkout -b <branchname>`
- ▶ List of local branches
  - ▶ `git branch`
- ▶ List of all branches, including remote branches
  - ▶ `git branch -a`



# Making Changes

- ▶ Edit a file with your favorite text editor
- ▶ Get the status of your working copy
  - ▶ `git status`
- ▶ Git has a feature called the index, which allows you to stage your commits before committing them. It allows to commit only part of your modifications, by file or even by chunk.
- ▶ On each modified file
  - ▶ `git add <filename>`
- ▶ Then commit. No need to be on-line or connected to commit
  - ▶ Linux requires the `-s` option to sign your changes
  - ▶ `git commit -s`
- ▶ If all modified files should be part of the commit
  - ▶ `git commit -as`



## Sharing Changes: E-mail

- ▶ The simplest way of sharing a few changes is to send patches by e-mail
- ▶ The first step is to generate the patches
  - ▶ `git format-patch master..<yourbranch>`
  - ▶ Will generate one patch for each of the commits done on `<yourbranch>`
  - ▶ The patch files will be `0001-....`, `0002-....`, etc.
- ▶ The second step is to send these patches by e-mail
  - ▶ `git send-email --compose --to email@domain.com 00*.patch`
  - ▶ Required Ubuntu package: `git-email`
  - ▶ In a later slide, we will see how to use `git config` to set the SMTP server, port, user and password.



# Sharing Changes: Your Own Repository

- ▶ If you do a lot of changes and want to ease collaboration with others, the best is to have your own public repository
- ▶ Use a git hosting service on the Internet:
  - ▶ GitLab (<http://gitlab.com/>)
    - ▶ Open Source server. Proprietary and commercial extensions available.
  - ▶ GitHub (<https://github.com/>)
    - ▶ For public repositories. Need to pay for private repositories.
- ▶ Publish on your own web server
  - ▶ Easy to implement.
  - ▶ Just needs git software on the server and ssh access.
  - ▶ Drawback: only supports http cloning (less efficient)
- ▶ Set up your own git server
  - ▶ Most flexible solution.
  - ▶ Today's best solutions are gitolite (<https://github.com/sitaramc/gitolite>) for the server and cgit for the web interface (<http://git.zx2c4.com/cgit/about/>).



# Sharing changes: HTTP Hosting

- ▶ Create a bare version of your repository
  - ▶ cd /tmp
  - ▶ git clone --bare ~/project project.git
  - ▶ touch project.git/git-daemon-export-ok
- ▶ Transfer the contents of project.git to a publicly-visible place (reachable read-only by HTTP for everybody, and read-write by you through SSH)
- ▶ Tell people to clone `http://yourhost.com/path/to/project.git`
- ▶ Push your changes using
  - ▶ `git push ssh://yourhost.com/path/toproject.git srcbranch:destbranch`



# Tracking Remote Trees

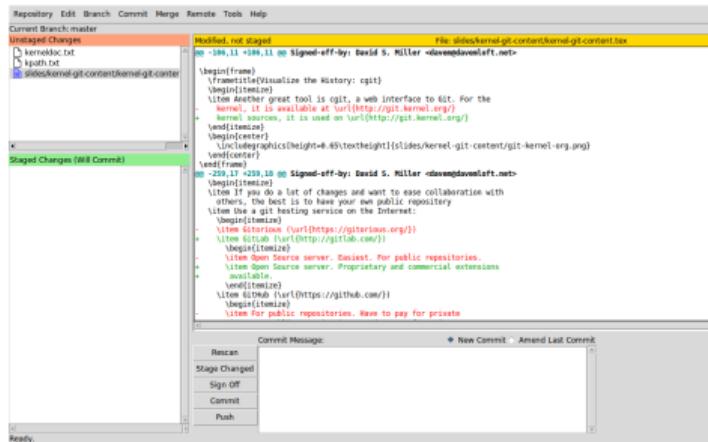
- ▶ In addition to the official Linus Torvalds tree, you might want to use other development or experimental trees
  - ▶ The OMAP tree at  
`git://git.kernel.org/pub/scm/linux/kernel/git/tmlind/linux-omap.git`
  - ▶ The stable realtime tree at  
`git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git`
- ▶ The git remote command allows to manage remote trees
  - ▶ `git remote add rt git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git`
- ▶ Get the contents of the tree
  - ▶ `git fetch rt`
- ▶ Switch to one of the branches
  - ▶ `git checkout rt/master`



## git-gui

<http://www.git-scm.com/docs/git-gui>

- ▶ A graphical interface to create and manipulate commits, replacing multiple git command-line commands.
  - ▶ Not meant for history browsing (opens gitk when needed).



- ▶ Example usage on Ubuntu/Debian:  
sudo apt install git-gui  
git gui blame Makefile



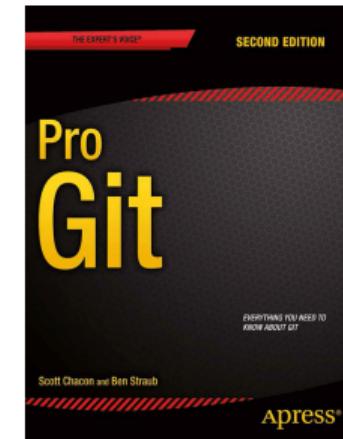
# About Git

We have just seen the very basic features of Git.

Many more interesting features are available (rebasing, bisection, merging and more).

For more details:

- ▶ Git Manual  
<http://schacon.github.com/git/user-manual.html>
- ▶ Git Book (freely available on-line, or in print form)  
<http://git-scm.com/book>
- ▶ Git official website  
<http://git-scm.com/>
- ▶ Video: James Bottomley's tutorial on using Git  
<http://bit.ly/2fZJxLZ>





## Practical lab - Going further: git



- ▶ Get familiar with git by contributing to a real project: the Linux kernel
- ▶ Send your patches to the maintainers and mailing lists.